ETH Lecture 401-0674-00L Numerical Methods for (Partial) Differential Equations

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Spring Term 2025, August 9, 2025
(C) Seminar für Angewandte Mathematik, ETH Zürich

Link to the current version of this homework problem collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single ".pdf" file. Every one or two weeks, on the course Moodle page we publish a list of problems that should be solved.

0.1.2 Importance of Homework

Homework assignments are **not** mandatory. However, it is **very important** that you constantly exercise with the material you learn. "Solving" the homework assignments one week before the main exam by looking at the solutions will likely result in failure.

We provide hints and solutions from the beginning. It is your responsibility to look at the solutions only if you are stuck in a difficult problem and you tried to find a solution for a sufficient amount of time.

0.1.3 Corrections and Grading of Assignments

The assistants are happy to look at your solution **upon request** and you can simply send a PDF scan to your tutor or give him your attempted solution on paper during the tutorial classes. You can submit your codes to the assistants using the submission interface, see the course Moodle page for details. Do not hesitate to hand in your solutions even if those are incomplete and/or incorrect, because feedback from the assistants is most valuable in case of incomplete or incorrect solutions.

You will get feedback about your submissions usually during the tutorial class or in a private (online) meeting with the tutor.

0.1.4 Codes and Templates

The codes for each homework problem will reside in a sub-directory of the `homeworks` folder in the GitLab homework codes repository. The folder structure within these sub-directories is as follows:

```
mastersolution
  test
mysolution
  test
templates
  test
CmakeLists.txt.
```

Optional sub-directories may be provided and may contain some data.

Running `cmake ..` in the `build` folder residing in the top level directory will automatically create a mirror of the directory hierarchy in the `build` folder. You can then navigate to the corresponding folder for the current homework problem, which contains the same sub-directories as listed above. In that folder you can issue the following make commands:

- make: compiles everything
- make <problemdirname>_mysolution: compiles your solution
- make <problemdirname>_mastersolution: compiles the master solution
- make <problemdirnane>_test_mysolution: runs the unit tests on your solution
- make <problemdirname>_test_master: runs the unit tests on the master solution

For additional information see [Lecture → Section 0.2.1].

0.1.5 Hints and Solutions

Hints and solutions are accessible online in separate ".pdf" files to which links are given in this document. They can also be found at the end of this homework problem collection.

Contents

0.1 General Information . . . 1
0.1.1 Weekly Homework Assignments . . . 1
0.1.2 Importance of Homework . . . 1
0.1.3 Corrections and Grading of Assignments . . . 1
0.1.4 Codes and Templates . . . 2
0.1.5 Hints and Solutions . . . 2

0 Introduction **7**
Problem 0-1: Setting up implementation environment for NumPDE course . . . 7
Problem 0-2: Using the GDB Debugger . . . 10

1 Second-Order Scalar Elliptic Boundary Value Problems **13**
Problem 1-1: Quadratic Functionals . . . 13
Problem 1-2: Linear functionals on Sobolev spaces . . . 16
Problem 1-3: L^∞-Norms *Are* Bounded by H^1-Seminorms in 1D . . . 18
Problem 1-4: A Poincaré-type inequality . . . 20
Problem 1-5: A second-order elliptic transmission problem in 1D . . . 22
Problem 1-6: Heat conduction with non-local boundary conditions . . . 25
Problem 1-7: A second-order boundary value problem for vector fields . . . 27
Problem 1-8: A coupled reaction-diffusion problem . . . 30
Problem 1-9: Second-order Elliptic BVP from weak formulations . . . 32
Problem 1-10: From Boundary Value Problems to Variational Problems . . . 33
Problem 1-11: Strong and Weak Form of Elliptic BVPs . . . 35

2 Finite Element Methods (FEM) **37**
Problem 2-1: Properties of Galerkin solutions . . . 37
Problem 2-2: Transformation of Galerkin Matrices . . . 39
Problem 2-3: Pointwise "Exact" Galerkin Solution . . . 42
Problem 2-4: Linear finite elements for two-point boundary value problems . . . 44
Problem 2-5: Triangular linear FEM for 2D reaction-diffusion BVP . . . 48
Problem 2-6: Incidence matrices of a hybrid 2D mesh . . . 52
Problem 2-7: Computing the length of the boundary in LEHRFEM++ . . . 56
Problem 2-8: Introduction to local assembly in LEHRFEM++ . . . 58
Problem 2-9: Handling degrees of freedom (DOFs) in LEHRFEM++ . . . 62
Problem 2-10: Projection onto Gradients . . . 65
Problem 2-11: Hybrid-mesh Galerkin matrices and right-hand-side vectors . . . 69
Problem 2-12: Testing built-in quadrature rules of LEHRFEM++ . . . 71
Problem 2-13: Local computations for parametric Lagrangian finite elements . . . 73
Problem 2-14: Non-conforming Crouzeix-Raviart finite element method . . . 77
Problem 2-15: Regularized Neumann Problem . . . 86
Problem 2-16: Rigidity of Piecewise Polynomial Continuous Functions . . . 90
Problem 2-17: Local Computations for Convection Bilinear Form . . . 92

Problem 2-18: Nitsche's Method for Elliptic BVPs . . . 96
Problem 2-19: Lagrangian Finite Elements on Criss-Cross Meshes . . . 102
Problem 2-20: `DofHandler` and Assembly . . . 104
Problem 2-21: Blended Parametric Representation of Curvi-Linear triangles . . . 106
Problem 2-22: Constrained Neumann Problem . . . 108
Problem 2-23: Galerkin Matrices . . . 111
Problem 2-24: Parametric FEM: Local Computations . . . 113

3 FEM: Convergence and Accuracy 115
Problem 3-1: Computing Averages over the Boundary . . . 115
Problem 3-2: Debugging Finite Element Codes . . . 119
Problem 3-3: Dirichlet BVP with point-evaluation right-hand-side functional . . . 122
Problem 3-4: A boundary value problem modelling stationary heat conduction . . . 126
Problem 3-5: Error estimates for traces . . . 128
Problem 3-6: Projection Onto Constants . . . 131
Problem 3-7: Maximum principle for reaction-diffusion boundary value problems . . . 133
Problem 3-8: Output Functionals for a 2nd-Order Elliptic Boundary Value Problem with Impedance Boundary C
Problem 3-9: Zienkiewicz-Zhu A-Posteriori Error Estimator . . . 139
Problem 3-10: Parametric Finite Elements . . . 143
Problem 3-11: Stable Evaluation at a Point . . . 148
Problem 3-12: Computation of Electrostatic Forces . . . 153
Problem 3-13: Computation of Stationary Currents . . . 158
Problem 3-14: A Local Quasi-Interpolation Operator . . . 164
Problem 3-15: Asymptotic Convergence of Finite-Element Discretization Errors and Interpolation Errors 168
Problem 3-16: Non-conforming Crouzeix-Raviart FEM: Theoretical aspects . . . 169
Problem 3-17: Residual-Based A-Posteriori Error Estimator . . . 173
Problem 3-18: Hierarchical Local A-Posteriori Error Estimator . . . 176
Problem 3-19: Convergence of Finite-Element Solutions . . . 179
Problem 3-20: First-order System Least-Squares Variational Formulation . . . 182
Problem 3-21: Asymptotic Convergence of Finite-Element Solutions . . . 184
Problem 3-22: Coupled Elliptic Boundary Value Problems . . . 190
Problem 3-23: Neumann Data by Projection . . . 193

4 Beyond FEM: Alternative Discretizations 198
Problem 4-1: Mehrstellenverfahren for Poisson Equation . . . 198
Problem 4-2: Shifted Finite-Differences for Neumann BVP . . . 202

5 Non-Linear Elliptic Boundary Value Problems 205
Problem 5-1: A curve with tension . . . 205
Problem 5-2: The Brachistochrone Problem . . . 207
Problem 5-3: Minimal Surface Problem for Graphs . . . 214
Problem 5-4: Iterative Methods for Non-Linear Variational Problems . . . 219
Problem 5-5: Solving a Semi-Linear Elliptic Boundary Value Problem . . . 221
Problem 5-6: The P-Laplacian . . . 227

6 Numerical Integration – Single Step Methods 229
Problem 6-1: Linear ODE in spaces of matrices . . . 229
Problem 6-2: Explicit Runge-Kutta Methods . . . 231
Problem 6-3: Extrapolation of evolution operators . . . 233
Problem 6-4: System of Second-Order ODEs . . . 236
Problem 6-5: Non-linear Evolutions in Spaces of Matrices . . . 238
Problem 6-6: Order is not everything . . . 241

Problem 6-7: Initial Condition for Lotka-Volterra ODE . . . 243
Problem 6-8: Integrating ODEs using the Taylor expansion method . . . 247
Problem 6-9: Drawing Isolines of a Function . . . 249
Problem 6-10: Symplectic Timestepping for Equations of Motion . . . 253

7 Single Step Methods for Stiff Initial Value Problems 256
Problem 7-1: Implicit Runge-Kutta method . . . 256
Problem 7-2: Damped precession of a magnetic needle . . . 259
Problem 7-3: Singly Diagonally Implicit Runge-Kutta Method . . . 262
Problem 7-4: Semi-implicit Runge-Kutta SSM . . . 264
Problem 7-5: Exponential integrator . . . 267
Problem 7-6: Mono-implicit Runge-Kutta single step method . . . 269
Problem 7-7: Stability of a Runge-Kutta method . . . 271
Problem 7-8: Implicit Runge-Kutta Method for Gradient-Flow ODEs . . . 272

8 Structure-Preserving Numerical Integration 275

9 Second-Order Linear Evolution Problems 276
Problem 9-1: Implicit Two-Stage Radau Runge-Kutta Single-Step Method for Parabolic IBVPs . . . 276
Problem 9-2: Implicit Timestepping for Parabolic IBVP . . . 282
Problem 9-3: Decaying Method-of-Lines Solution with Implicit-Euler Timestepping . . . 287
Problem 9-4: The One-Dimensional Wave Equation with Absorbing Boundary Condition . . . 289
Problem 9-5: Symplectic Timestepping for Wave Equations . . . 293
Problem 9-6: A mixed elliptic-hyperbolic linear evolution problem . . . 298
Problem 9-7: Absorbing Boundary Conditions (ABCs) for the 2D Wave Equation . . . 301
Problem 9-8: Non-linear Schrödinger Equation with Cubic Non-Linearity . . . 306
Problem 9-9: The Fisher KPP Equation for Migration Modeling . . . 312
Problem 9-10: Mixed-Hybrid Finite-Element Method for the Wave Equation . . . 318
Problem 9-11: Gauss-Lobatto IIIC Timestepping . . . 323
Problem 9-12: The Sobolev Evolution Initial-Boundary Value Problem . . . 327
Problem 9-13: 1D Wave Equation with Perfectly Matched Layers . . . 329
Problem 9-14: Two-Step Radau Runge-Kutta Single-Step Method for the Heat Equation . . . 337
Problem 9-15: Implicit-Explicit Runge-Kutta Single-Step Methods . . . 339
Problem 9-16: Finite-Element Discretization of the Guyer-Krumhansl Heat Equation . . . 344
Problem 9-17: Scalar Parabolic Evolution Problems: Continuous and Discrete . . . 349
Problem 9-18: Leapfrog Timestepping for Dissipative Wave Equation . . . 351
Problem 9-19: Implicit Runge-Kutta Timestepping for a Degenerate Parabolic Evolution Problem 355
Problem 9-20: Method-of-Lines with SDIRK timestepping . . . 360
Problem 9-21: Magnetic Diffusion in a Wire . . . 362

10 Convection-Diffusion Problems 366
Problem 10-1: Exponentially fitted upwind scheme . . . 366
Problem 10-2: Upwind Quadrature Method . . . 372
Problem 10-3: Upwind Finite Volume Method . . . 375
Problem 10-4: Method of Characteristics and Semi-Lagrangian Discretization . . . 379
Problem 10-5: Semi-Lagrangian Discretization of Transport Problem . . . 384
Problem 10-6: Streamline Upwind Method for Pure Advection Problem . . . 389
Problem 10-7: Streamline-Upwind Stabilized Finite Element Method . . . 394
Problem 10-8: Upwind Quadrature Scheme for Transport . . . 398
Problem 10-9: Potential Flow . . . 401
Problem 10-10: Least-squares Galerkin FEM for Scalar Advection-Reaction BVP . . . 405

11 Numerical Methods for Conservation Laws **410**
Problem 11-1: Burgers Equation 410
Problem 11-2: Conservative finite-volume discretization based on Engquist-Osher numerical flux 413
Problem 11-3: Conservative finite-volume discretization based on Godunov numerical flux . . . 416
Problem 11-4: Lax-Wendroff Scheme 419
Problem 11-5: Discontinous Galerkin Discretization for 1D Conservation Laws 423
Problem 11-6: One-Dimensional Scalar Conservation Law with Empiric Flux 427
Problem 11-7: Higher-Order Finite Volume Method 431
Problem 11-8: 2D Finite Volume Method for Advection 438
Problem 11-9: Non-Linear Conservation Laws with Source Terms 444
Problem 11-10: Flux-Limited Finite-Volume Methods for Scalar Conservation Laws 447
Problem 11-11: Conservation laws with non-linear density 451
Problem 11-12: Finite Volumme Methods for 1D Damped Wave Equation 455
Problem 11-13: The p-System and its Finite Volume Discretization 459
Problem 11-14: Finite Volume Method for the Shallow Water Equations 462
Problem 11-15: Finite Volume Methods for Isentropic Euler Equations 466

12 Finite Elements for the Stokes Equation **471**
Problem 12-1: Viscous Flow Simulation with Taylor-Hood FEM 471
Problem 12-2: The MINI Element for Stokes 482
Problem 12-3: Taylor-Hood FEM for Stokes BVP 491
Problem 12-4: FEM for Stokes BVPs 498
Problem 12-5: Stabilized P1-FEM for Stokes 501

13 Finite-Element Exterior Calculus (FEEC) **506**
Problem 13-1: Incidence Matrices of a 2D Finite Element Mesh 506
Problem 13-2: Mixed FEM for the 2D Hodge-Laplacian 511
Problem 13-3: Whitney Finite Elements for Magnetostatic BVP in Two Dimensions 523

Chapter 0

Introduction

Problem 0-1: Setting up implementation environment for NumPDE course

During the course you will be asked to implement many algorithms in C++. We will also rely on the finite element library LEHRFEM++, see [Lecture → Rem. 2.7.0.1]. These instructions will guide you through the setup of a coding environment for the course "Numerical Methods for Partial Differential Equations".

The Microsoft Windows OS is not supported in this course. Use UNIX, either Linux or Mac OS X.
▷ problem name/problem code folder: NumPDESetup

(0-1.a) ⊡ (10 min.)

Please create bookmarks to the following online resources and store them in a folder "NumPDE":

- Online C++ reference pages ➢ link,
- LEHRFEM++ Github repository ➢ link,
- LEHRFEM++ Doxygen documentation ➢ link
- EIGEN documentation ➢ link
- Lecture document ➢ link
- Collection of homework problems ➢ link
- Lecture and homework code repository Link

▲

(0-1.b) ⊡ (10 min.)

Please visit the homepage of the GIT repository for this course

`https://gitlab.math.ethz.ch/ralfh/NPDERepo`

Browse the information contained in the top-level `README.md` file ("`.md`" stands for "markdown", see primer). ▲

(0-1.c) ⊡ (10 min.)

Retrieving files required for coding projects:

1. Open a UNIX terminal and go to a dedicated directory for this course. Then copy the homework codes by issuing the command

```
> git clone https://gitlab.math.ethz.ch/ralfh/NPDERepo.git
```

This will create a subdirectory `NPDECODES` containing codes and other data needed for homework problems.

2. Then create a build directory and initialize the build system

```
> cd NPDERepo
> mkdir build
> cd build
> cmake ..
```

be patient (can take a while) and keep your fingers crossed that everything will run smoothly.

3. Compile and run a small demonstration code:

```
build> make NumPDESetup_mastersolution
build> ./homeworks/NumPDESetup/NumPDESetup_mastersolution
build> make NumPDESetup_mysolution
build> ./homeworks/NumPDESetup/NumPDESetup_mysolution
```

SOLUTION for (0-1.c) → 0-1-3-0:s2.pdf ▲

(0-1.d) ⊡ (1 min.) All during the course, the master solution might be updated and new problem might be added. To keep track of these changes regularly execute

```
NPDECODES> git pull
```

At this point nothing will probably happen, but later the command will tell you, if new versions of files are copied to your local directories.

A problem could be that some of the updates might affect files in the `homeworks/*/mysolution` folders that you have changed. To avoid your solutions being overwritten, you have to give the following extended sequence of GIT commands:

```
NPDECODES> git stash; git pull; git stash apply
```

▲

(0-1.e) ⊡ (10 min.) The codes we study in this "project" partly rely on the EIGEN C++ library for efficient elementary linear algebra, see EIGEN documentation.

Eigen was also used in the course 401-2673-00L Numerical Methods for CSE, taught at ETH Zurich in autumn term. [NumCSE course → Section 1.2.1] gives an introduction, [NumCSE course → § 2.5.0.8] discusses direct elimination solvers, and [NumCSE course → Section 2.7.2] treats the use of *sparse matrix* formats in EIGEN, EIGEN documentation.

Consult EIGEN documentation to learn how to solve large linear systems with sparse coefficient matrices in EIGEN.

SOLUTION for (0-1.e) → 0-1-5-0:.pdf ▲

(0-1.f) ⊡ (20 min.) Some homework problems come with test making use of the GoogleTest framework. Issue the commands

```
build> make NumPDESetup_test_mysolution
build> ./homeworks/NumPDESetup/NumPDESetup_test_mysolution
```

What output do you get and why?

SOLUTION for (0-1.f) → 0-1-6-0:s40h1.pdf ▲

End Problem 0-1 , 61 min.

Problem 0-2: Using the GDB Debugger

Debugging is an important aspect of implementation in particular in the case of complex numerical codes. In this problem we practice the use of the debugger **GDB**.

The problem requires familiarity with C++ and some experience in installing software on Linux or Mac OS X. The learning objective is to become familiar with using a source-code debugger for examining a complex code.
▷ problem name/problem code folder: DebuggingWithGDB

A debugger is a tool that lets a developer watch the execution of a code and inspect variables, while it is running. Key concepts are

- **breakpoints**, positions in the code where executions stops to allow user interaction,
- **stepping**, the execution of the code line-by-line,
- **inspection** of call stacks and of the contents variables and objects.

Some introductory remarks on this problem:

- If you are proficient at using an IDE with an integrated debugger, you may also use it to work on the task of this problem.
- If you have big difficulties with installing some of the tools on your computer, please work from remote on UNIX machines of ETH Zürich, for example a node of the Euler Linux cluster.
- Instead of GDB you may use LLDB.

(0-2.a) ⚀ (10 min.) Install the `g++` C++ compiler on your system in case it is not already available. Usually, `g++` can be downloaded and installed as a package from a software repository for your OS.

SOLUTION for (0-2.a) → `0-2-1-0:s1.pdf` ▲

(0-2.b) ⚀ (10 min.) Make sure that the GDB debugger application is available on your machine.

SOLUTION for (0-2.b) → `0-2-2-0:s2.pdf` ▲

(0-2.c) ⚀ (15 min.) Navigate to the top level of your local copy of the `NPDECODES` repository, create a sub-directory `build_gcc`, go there, issue the command `cmake ...` This will generate the makefiles to build binaries. The **cmake** configurations are cached in `build_gcc/CMakeCache.txt`. There, it is recommended to change the following settings:

- set the `BUILD_TYPE` to `Debug`,
- and the `CMAKE_CXX_COMPILER` to `g++`.

These settings will allow your debugger to be aware of the source code. You may also enforce these settings through command-line arguments when invoking `cmake ..`, but then they will not persist.

SOLUTION for (0-2.c) → `0-2-3-0:s4.pdf` ▲

(0-2.d) ⚀ (10 min.) Install a GDB plugin for printing EIGEN data types from

`https://github.com/dmillard/eigengdb.git`

If you are folowing this Problem with LLDB, an equialent plugin can be found under

`https://github.com/tehrengruber/LLDB-Eigen-Data-Formatter.git`

Installation instructions can be found in the repository. The LLDB extension was written by Till Ehrengruber, former CSE student at ETH Zurich, and now employed with CSCS.

SOLUTION for (0-2.d) → 0-2-4-0:s4.pdf ▲

The C++ code we want to debug is listed next. It is based on the LEHRFEM++ library. It reads a so-called triangular finite element mesh [Lecture → Section 2.4.1], loops over its triangles, edges, and nodes, and computes a "center of gravity" of the nodes for every type of "entity".

C++ code 0.2.1: Some function to be analyzed with GDB ⇝ GitLab

```
2  void ReadAndOutputMesh(const char *filename) {
3    if (filename != nullptr) {
4      // Build full path to the mesh file
5      auto gmshfile_path = filename;
6      // Load the mesh from a file produced by Gmsh
7      auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
8      lf::io::GmshReader reader(std::move(mesh_factory), gmshfile_path);
9      // Obtain pointer to read mesh
10     std::shared_ptr<const lf::mesh::Mesh> mesh_p = reader.mesh();
11     const lf::mesh::Mesh &mesh{*mesh_p};
12     // Run through all entities of the mesh and print entity information
13     for (int codim = 0; codim <= mesh.DimMesh(); ++codim) {
14       int cnt = 0;
15       Eigen::VectorXd c{Eigen::VectorXd::Zero(mesh.DimWorld())};
16       for (const lf::mesh::Entity *entity : mesh.Entities(codim)) {
17         // Number of vertices
18         const int no_vertices = (entity->RefEl()).NumNodes();
19         // Obtain "convex hull" of an entity
20         const Eigen::MatrixXd vertices{
21             lf::geometry::Corners(*entity->Geometry())};
22         // Compute center of gravity
23         c += vertices.rowwise().sum() / no_vertices;
24         cnt++;
25       }
26       if (cnt != mesh.NumEntities(codim)) {
27         std::cerr << "Count mismatch for entities of codim = " << codim << ": "
28                   << cnt << " <-> " << mesh.NumEntities(codim) << std::endl;
29       }
30       c /= cnt;
31       std::cout << "Center of codim-" << codim
32                 << " entities = " << c.transpose() << std::endl;
33     }
34     // Wite mesh data to file for visualization with Python script
35     lf::io::writeMatplotlib(mesh, "ljoint.csv");
36     std::cout << "Wrote ljoint.csv" << std::endl;
37     systemcall::execute("python3 scripts/plot_mesh.py ljoint.csv mesh.eps");
38   }
39 }
```

C++ code 0.2.2: Main function for test code ⇝ GitLab

```
2  int main() {
3    // A function using some of LehrFEM++'s facilities
4    DebuggingWithGDB::ReadAndOutputMesh("meshes/ljoint.msh");
5
6    return 0;
7  }
```

You can find the C++ source files in the directory NPDERepo/homeworks/DebuggingWithGDB/mysoluti
.

(0-2.e) ⚀ (10 min.) In the directory `build_gcc` issue the commands

```
build_gcc> make DebuggingWithGDB_mysolution
build_gcc> ./homeworks/DebuggingWithGDB/DebuggingWithGDB_mysolution
```

and watch the output.

SOLUTION for (0-2.e) → 0-2-5-0:.pdf ▲

(0-2.f) ⚁ (10 min.) Now set breakpoints

- At the beginning of the function `ReadAndOutputMesh()`
- at the code line **int** `cnt = 0;`
- at the code line `c += vertices.rowwise().sum()/ no_vertices;` and with the *condition* `mesh.Index(*entity)== 10`.

Check the C++ source files at `NPDERepo/homeworks/DebuggingWithGDB/mysolution/` for the line numbers (the ones at Code 0.2.1 are not the same).

HIDDEN HINT 1 for (0-2.f) → 0-2-6-0:s6h1.pdf

SOLUTION for (0-2.f) → 0-2-6-1:s6.pdf ▲

(0-2.g) ⚂ (15 min.) Examine the matrix `vertices` for `codim == 1` (outer loop) and the 10th entity (inner loop).

SOLUTION for (0-2.g) → 0-2-7-0:s7.pdf ▲

End Problem 0-2 , 80 min.

Chapter 1

Second-Order Scalar Elliptic Boundary Value Problems

Problem 1-1: Quadratic Functionals

In [Lecture → Section 1.2] we learned about mathematical models whose core element turned out to be a quadratic minimization problem over some function space. Quadratic minimization problems involve quadratic functionals as defined in [Lecture → Def. 1.2.3.2] and this problem practices indentifying quadratic functionals and their components.

This is a purely theoretical problem based on the contents of [**Lecture → Section 1.2.3**], which has to be studied thoroughly beforehand. You will practice computations related to concrete quadratic functionals and to find their constituent parts. You will also be asked to determine properties of bilinear forms.

Each of the following sub-problems, except the last one, presents a functional $J : V_0 \to \mathbb{R}$ defined on some vector space V_0. In each case, the task is

(i) to decide whether J is a **quadratic functional** according to [Lecture → Def. 1.2.3.2] that is, whether J can be written in the form

$$J(v) = \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v) + c\,, \quad v \in V_0\,, \tag{1.1.1}$$

with a *symmetric* bilinear form a [Lecture → Def. 0.3.1.4], a linear form ℓ [Lecture → Def. 0.3.1.3], and $c \in \mathbb{R}$, and,

(ii) if so, to state its components, the bilinear form a, and the linear form ℓ, explicitly, and,

(iii) to tell, in the case of a quadratic functional, whether a is positive semi-definite [Lecture → Def. 0.3.1.16], [Lecture → Def. 1.2.3.24] or even positive definite (s.p.d.) [Lecture → Def. 1.2.3.27].

Throughout, the symbol Ω designates a bounded polygonal domain $\subset \mathbb{R}^2$. For notations concerning function spaces see [Lecture → Section 0.3.2.3].

For your convenience, we collect the relevant definitions:

Definition [Lecture → Def. 0.3.1.3]. Linear forms

Given a vector space V over $\mathbb{R}$, a linear form/linear functional (LF) ℓ is a mapping $\ell : V \mapsto \mathbb{R}$ that satisfies

$$\ell(\alpha u + \beta v) = \alpha\ell(u) + \beta\ell(v) \quad \forall u, v \in V\,, \forall \alpha, \beta \in \mathbb{R}\,.$$

Definition [Lecture → Def. 0.3.1.4]. (Bi-)linear forms

Given an $\mathbb{R}$-vector space V, a A bilinear form (BLF) a on V is a mapping $\mathsf{a} : V \times V \mapsto \mathbb{R}$, for which

$$\mathsf{a}(\alpha_1 v_1 + \beta_1 u_1, \alpha_2 v_2 + \beta_2 u_2) = \alpha_1\alpha_2\,\mathsf{a}(v_1, v_2) + \alpha_1\beta_2\,\mathsf{a}(v_1, u_2) + \beta_1\alpha_2\,\mathsf{a}(u_1, v_2) + \beta_1\beta_2\,\mathsf{a}(u_1, u_2)$$

for all $u_i, v_i \in V$, $\alpha_i, \beta_i \in \mathbb{R}$, $i = 1, 2$.

Definition [Lecture → Def. 0.3.1.15]. Symmetric bilinear form

A bilinear form (→ [Lecture → Def. 0.3.1.4]) a on a vector space V is symmetric, if and only if

$$\mathsf{a}(u, v) = \mathsf{a}(v, u) \quad \forall u, v \in V\,.$$

Definition [Lecture → Def. 0.3.1.16]. Positive (semi-)definite bilinear form

A bilinear form $a : V \times V \to \mathbb{R}$ on a real vector space V is called

$$\begin{aligned} \text{positive semi-definite} &:\Leftrightarrow \quad \mathsf{a}(v, v) \geq 0 \quad \forall v \in V\,, \\ \text{positive definite} &:\Leftrightarrow \quad \mathsf{a}(v, v) > 0 \quad \forall v \in V \setminus \{\mathbf{0}\}\,. \end{aligned}$$

(1-1.a) ⚀ (15 min.)

$$J(v) = \int_{-\pi}^{\pi} x^2 |v(x) + 1|^2 \mathrm{d}x\,, \quad v \in C^0_{\mathrm{pw}}([-\pi, \pi])\,. \tag{1.1.2}$$

SOLUTION for (1-1.a) → 1-1-1-0:I.pdf ▲

(1-1.b) ⚁ (15 min.)

$$J(v) = \int_{-1}^{1} \frac{d^2 v}{dx^2}(x)\, \frac{dv}{dx}(x)\, \mathrm{d}x\,, \quad v \in C^2([-1, 1])\,. \tag{1.1.4}$$

HIDDEN HINT 1 for (1-1.b) → 1-1-2-0:s2h0.pdf

HIDDEN HINT 2 for (1-1.b) → 1-1-2-1:s2h1.pdf

HIDDEN HINT 3 for (1-1.b) → 1-1-2-2:s2h3.pdf

SOLUTION for (1-1.b) → 1-1-2-3:II.pdf ▲

(1-1.c) ⚂ (20 min.)

$$J(v) := \sum_{j=1}^{n} (v_j - v_{j-1})^2\,, \quad \boldsymbol{v} = [v_1, \ldots, v_n]^\top \in \mathbb{R}^n\,, \quad n \in \mathbb{N}\,, \tag{1.1.5}$$

with the convention $v_0 := 0$. If you think that this J is a quadratic functional on $\mathbb{R}^n$, please also write it in the "canonical" form [Lecture → (1.2.3.6)]

$$J(\boldsymbol{v}) = \tfrac{1}{2}\boldsymbol{v}^\top \mathbf{A}\boldsymbol{v} + \boldsymbol{b} \cdot \boldsymbol{v} + c\,, \quad \mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{n,n}\,, \quad \boldsymbol{b} \in \mathbb{R}^n\,, \quad c \in \mathbb{R}\,, \tag{1.1.6}$$

for quadratic functionals on $\mathbb{R}^n$.

HIDDEN HINT 1 for (1-1.c) → 1-1-3-0:s3h1.pdf

SOLUTION for (1-1.c) → 1-1-3-1:III.pdf ▲

(1-1.d) ⚂ (10 min.)

$$J(v) := \int_\Omega |\mathbf{f}(\boldsymbol{x}) \cdot \mathbf{grad}\, v - 1|^2 \,\mathrm{d}\boldsymbol{x}\,, \quad v \in C^1_{\mathrm{pw}}(\Omega)\,, \tag{1.1.7}$$

where $\mathbf{f} : \Omega \to \mathbb{R}^2$, $\mathbf{f} \in (C^0(\overline{\Omega}))^2$ is a given continuous vectorfield.

HIDDEN HINT 1 for (1-1.d) → 1-1-4-0:s4h1.pdf

SOLUTION for (1-1.d) → 1-1-4-1:IV.pdf ▲

(1-1.e) ⚂ (10 min.)

$$J(v) := \int_\Omega \left| v \frac{\partial v}{\partial x_1} \right|^2 \mathrm{d}\boldsymbol{x}\,, \quad v \in C^1_{\mathrm{pw}}(\Omega)\,. \tag{1.1.8}$$

HIDDEN HINT 1 for (1-1.e) → 1-1-5-0:5h1.pdf

SOLUTION for (1-1.e) → 1-1-5-1:V.pdf ▲

(1-1.f) ⚂ (10 min.)

$$J(v) := \left(\int_\Omega v(\boldsymbol{x}) + 1 \,\mathrm{d}\boldsymbol{x} \right)^2, \quad v \in C^0_{\mathrm{pw}}(\Omega)\,. \tag{1.1.9}$$

HIDDEN HINT 1 for (1-1.f) → 1-1-6-0:s6h1.pdf

SOLUTION for (1-1.f) → 1-1-6-1:VI.pdf ▲

(1-1.g) ⚄ (20 min.) Finally show the following result, which gives a general formula for extracting the bilinear form from a quadratic functional. This formula is also known as **polarization identity**.

Lemma 1.1.10. Extracting the bilinear form from a quadratic functional

Let J be a quadratic functional on a real vector space V_0 according to [*Lecture → Def. 1.2.3.2*],

$$J(v) = \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v) + c\,, \quad v \in V_0\,, \tag{1.1.11}$$

with a symmetric bilinear form a, a linear form ℓ, and $c \in \mathbb{R}$. Then

$$\mathsf{a}(u,v) = J(u+v) - J(u) - J(v) + J(0)\,. \tag{1.1.12}$$

SOLUTION for (1-1.g) → 1-1-7-0:X.pdf ▲

End Problem 1-1, 100 min.

Problem 1-2: Linear functionals on Sobolev spaces

In [Lecture → Section 1.2.3] we have seen that continuity (→ [Lecture → Def. 1.2.3.42]) of linear forms with respect to energy norms (→ [Lecture → Def. 1.2.3.35]) induced by symmetric positive definite bilinear forms (→ [Lecture → Def. 1.2.3.27]) is a key property, *cf.* [Lecture → Lemma 1.2.3.39]. Thus, for elliptic boundary value problems, continuity of linear forms in Sobolev spaces is crucial. The reassuring advice that what matters most about Sobolev spaces are their (rather simple) norms will be confirmed.

This theoretical problems requires knowledge covered in [**Lecture → Section 1.2.3**] and [**Lecture → Section 1.3**]. From [Lecture → Section 1.9] we need [**Lecture → Thm. 1.9.0.10**].

For the point evaluation functional, we investigated its continuity on the Sobolev space $H^1(\Omega)$ in [Lecture → Ex. 1.2.3.45], for the source functional $v \to \int_\Omega fv\,\mathrm{d}\boldsymbol{x}$ continuity on $H^1(\Omega)$ was studied in [Lecture → Section 1.3.4], whereas boundary functionals arising from non-homogeneous Neumann problems were examined in [Lecture → § 1.9.0.9].

In this problem we consider the four linear functionals

$$\ell_1(v) := \int_\Omega \mathbf{c}\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad \mathbf{c}\in\mathbb{R}^2\,, \tag{1.2.1}$$

$$\ell_2(v) := \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \tag{1.2.2}$$

$$\ell_3(v) := \int_{\partial\Omega} \mathbf{grad}\, v(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \tag{1.2.3}$$

$$\ell_4(v) := \int_\Omega v\left(\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}\right)\mathrm{d}\boldsymbol{x}\,. \tag{1.2.4}$$

on the unit disk $\Omega = \{\boldsymbol{x}\in\mathbb{R}^2 : \|\boldsymbol{x}\| < 1\}$. These functionals all make sense when we apply them to smooth functions $v : \overline{\Omega} \to \mathbb{R}$. Recall [Lecture → Def. 0.3.1.3] and make sure that you "see" the linearity of all the functionals.

The standard tools for proving continuity of linear functionals on Sobolev spaces are the **Cauchy-Schwarz inequality** [Lecture → (1.3.4.15)] and, in $H^1(\Omega)$, **trace theorems** like the following.

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

For every bounded domain $\Omega\subset\mathbb{R}^d$ there exists a constant $C>0$ such that

$$\|u\|^2_{L^2(\partial\Omega)} \le C\|u\|_{L^2(\Omega)}\cdot\|u\|_{H^1(\Omega)} \quad \forall u\in H^1(\Omega)\,.$$

Remark 1.2.5 (Debunking the continuity of linear functionals on function spaces) You may appeal to the following "sloppy" argument, which can be made more rigorous by examining sequences as in [Lecture → Ex. 1.2.3.45]:

In order to show that a functional is unbounded on $L^2(\Omega)$, one tries to find a function $v\in L^2(\Omega)$, which can be verified by showing $\|v\|_{L^2(\Omega)} < \infty$, for which "$|\ell(v)| = \infty$", which often manifests itself as the divergence of an improper integral.

Be warned that, in the case of lack of continuity, finding such "counterexample" functions is by no means easy and requires intuition. Often those are logarithms or functions derived from them.

⌟

HIDDEN HINT 1 for (1-2.) → 1-2-0-0:lof11.pdf

(1-2.a) ⚀ (15 min.) Is the functional ℓ_1, $\ell_1(v) := \int_\Omega \mathbf{c} \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$, continous on $L^2(\Omega)$?

HIDDEN HINT 1 for (1-2.a) → 1-2-1-0:s1h1.pdf

SOLUTION for (1-2.a) → 1-2-1-1:s1.pdf ▲

(1-2.b) ⚁ (10 min.) Is the functional ℓ_2, $\ell_2(v) := \int_\Omega v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$, continous on $L^2(\Omega)$?

HIDDEN HINT 1 for (1-2.b) → 1-2-2-0:s2h1.pdf

SOLUTION for (1-2.b) → 1-2-2-1:s2.pdf ▲

(1-2.c) ⚂ (15 min.) Is the functional ℓ_3, $\ell_3(v) := \int_{\partial\Omega} \mathbf{grad}\, v(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})$, continuous on $L^2(\Omega)$?

HIDDEN HINT 1 for (1-2.c) → 1-2-3-0:s3h1.pdf

SOLUTION for (1-2.c) → 1-2-3-1:s3.pdf ▲

(1-2.d) ⚂ (15 min.) Is the functional ℓ_4, $\ell_4(v) := \int_\Omega v\left(\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}\right) \mathrm{d}\boldsymbol{x}$, continuous on $L^2(\Omega)$?

HIDDEN HINT 1 for (1-2.d) → 1-2-4-0:lof12.pdf

HIDDEN HINT 2 for (1-2.d) → 1-2-4-1:lofx.pdf

SOLUTION for (1-2.d) → 1-2-4-2:s45.pdf ▲

(1-2.e) ⚁ (10 min.) Is ℓ_1, $\ell_1(v) := \int_\Omega \mathbf{c} \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$, continuous on $H^1(\Omega)$?

HIDDEN HINT 1 for (1-2.e) → 1-2-5-0:s5h1.pdf

SOLUTION for (1-2.e) → 1-2-5-1:s5.pdf ▲

(1-2.f) ⚁ (10 min.) Is ℓ_2, $\ell_2(v) := \int_\Omega v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$, continuous on $H^1(\Omega)$?

HIDDEN HINT 1 for (1-2.f) → 1-2-6-0:s6h1.pdf

SOLUTION for (1-2.f) → 1-2-6-1:s6.pdf ▲

(1-2.g) ⚂ (15 min.) Is ℓ_3, $\ell_3(v) := \int_{\partial\Omega} \mathbf{grad}\, v(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})$, continuous on $H^1(\Omega)$?

HIDDEN HINT 1 for (1-2.g) → 1-2-7-0:s7h1.pdf

HIDDEN HINT 2 for (1-2.g) → 1-2-7-1:s7h2.pdf

SOLUTION for (1-2.g) → 1-2-7-2:s7.pdf ▲

(1-2.h) ⚂ (10 min.) Is ℓ_4, $\ell_4(v) := \int_\Omega v\left(\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}\right) \mathrm{d}\boldsymbol{x}$, continuous on $H^1(\Omega)$?

HIDDEN HINT 1 for (1-2.h) → 1-2-8-0:s8h1.pdf

SOLUTION for (1-2.h) → 1-2-8-1:s8.pdf ▲

End Problem 1-2, 100 min.

Problem 1-3: L^∞-Norms *Are* Bounded by H^1-Seminorms in 1D

The boundedness of right-hand side functionals in linear variational problems with respect to the associated energy norm is necessary for the existence of solutions, see [Lecture → Lemma 1.2.3.39]. To verify it, we often have to rely on estimates involving different (Sobolev) norms on function spaces, like the two inequalities of [Lecture → Thm. 1.3.4.17] and [Lecture → Thm. 1.8.0.20]. In this exercise we bound the maximum norm for functions on an interval I by the Sobolev seminorm of $H^1(I)$.

This is a purely theoretical problem mainly based on [**Lecture → Section 1.3**] (and also a result from [Lecture → Section 1.9], which will be provided).

(1-3.a) ⚄ (25 min.) Show that

$$\|u\|_{L^\infty(]0,1[)} \le |u|_{H^1(]0,1[)} \quad \forall u \in C_0^1([0,1])) \,, \tag{1.3.1}$$

see [Lecture → (0.3.2.26)] and [Lecture → Def. 1.3.4.3] for the definitions of the norms (here for a generic domain Ω):

$$\begin{aligned} \|u\|_\infty \left(= \|u\|_{L^\infty(\Omega)} \right) &:= \sup_{x\in\Omega} \|u(x)\| \,, \quad \mathbf{u} \in L^\infty(\Omega) \,, \\ |v|^2_{H^1(\Omega)} &:= \int_\Omega \|\mathbf{grad}\, v(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \,, \\ \|v\|^2_{H^1(\Omega)} &:= \int_\Omega \|\mathbf{grad}\, v(\boldsymbol{x})\|^2 + v(\boldsymbol{x})^2 \,\mathrm{d}\boldsymbol{x} \,. \end{aligned}$$

HIDDEN HINT 1 for (1-3.a) → 1-3-1-0:emb1.pdf

SOLUTION for (1-3.a) → 1-3-1-1:emb.pdf ▲

Remark 1.3.2 (Extending estimate by density argument) By a density argument, see the explanations in the proof of [Lecture → Thm. 1.3.4.17], we can even conclude that the estimate (1.3.1) holds for all functions in $H^1_0(]0,1[)$. ⌟

Remark 1.3.3 (Point evaluation is continuous in 1D) The above estimate (1.3.1) confirms that point evaluation $v \mapsto v(x)$, $0 < x < 1$, is a *continuous* functional on $H^1_0(]0,1[)$. Therefore, this is a valid right hand side for a variational problem posed on $H^1_0(]0,1[)$.

Note that $\|u\|_{L^\infty(\Omega)}$ **cannot** be bounded by $\|u\|_{H^1(\Omega)}$ for any open domain in any higher dimension $d > 1$. This is strikingly demonstrated in [Lecture → Ex. 1.2.3.45]. ⌟

(1-3.b) ⚄ (20 min.) Show that the, compared to (1.3.1) slightly weaker, assertion

$$\exists C > 0\!: \quad \|u\|_{L^\infty(]0,1[)} \le C|u|_{H^1(]0,1[)} \quad \forall u \in C_0^1(]0,1[)) \,, \tag{1.3.4}$$

is an "immediate" consequence of the multiplicative trace inequality.

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

For every bounded domain $\Omega \subset \mathbb{R}^d$ there exists a constant $C > 0$ such that

$$\|u\|^2_{L^2(\partial\Omega)} \le C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega) \,.$$

HIDDEN HINT 1 for (1-3.b) → 1-3-2-0:embh21.pdf

HIDDEN HINT 2 for (1-3.b) → 1-3-2-1:embh2.pdf

SOLUTION for (1-3.b) → 1-3-2-2:emb2.pdf ▲

End Problem 1-3 , 45 min.

Problem 1-4: A Poincaré-type inequality ⚄

Most important results about Sobolev spaces that we are going to use in the design and analysis of numerical methods for partial differential equations concern *inequalities connecting relevant Sobolev norms*; recall the advice "Do not be afraid of Sobolev spaces! It is only the norms that matter for us, the spaces are irrelvant!" from [Lecture → Section 1.3]. An important example of such a fundamental relationship between norms of Sobolev spaces are the **Poincaré-Friedrichs inequalities** of Theorems [Lecture → Thm. 1.3.4.17] and [Lecture → Thm. 1.8.0.20]. In this problem we will prove an estimate related to a Poincaré-Friedrichs inequality in 1D.

This is a theoretical problem related to [**Lecture → Section 1.3**] and [Lecture → Section 1.8], in particular [Lecture → Thm. 1.3.4.17] and [Lecture → Thm. 1.8.0.20]. There are also close connections with Problem 1-3. For preparation please study the proofs of [Lecture → Thm. 1.3.4.17] and [Lecture → Thm. 1.8.0.20] for the one-dimensional case $d = 1$.

Recall that the Poincaré-Friedrichs inequalities introduced in the lecture relate $L^2(\Omega)$-norms and $H^1(\Omega)$-semi-norms:

Theorem [Lecture → Thm. 1.3.4.17]. First Poincaré-Friedrichs inequality

If $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$, *is* bounded, *then*

$$\|u\|_0 \leq \operatorname{diam}(\Omega)\, \|\mathbf{grad}\, u\|_0 \quad \forall u \in H^1_0(\Omega)\ .$$

Theorem [Lecture → Thm. 1.8.0.20]. Second Poincaré-Friedrichs inequality

If $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$, *is bounded, then*

$$\exists C = C(\Omega) > 0\colon \quad \|u\|_0 \leq C\, \operatorname{diam}(\Omega) \|\mathbf{grad}\, u\|_0 \quad \forall u \in H^1_*(\Omega)\ ,$$

where

$$H^1_*(\Omega) := \{v \in H^1(\Omega)\colon \int_\Omega v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0\}\ . \qquad \text{[Lecture → (1.8.0.15)]}$$

In this problem the focus is on $d = 1$, that is to say, Ω is a simple interval.

For $u \in C^1([a,b])$ prove the estimate

$$\int_a^b |u(x)|^2 \mathrm{d}x \leq \frac{2}{b-a}\left|\int_a^b u(y)\mathrm{d}y\right|^2 + 2(b-a)^2 \int_a^b |u'(t)|^2 \mathrm{d}t\ . \tag{1.4.1}$$

Remark. In the theory of Sobolev spaces, usually it is sufficient to prove an inequality between norms for smooth functions only, using classical calculus. The reason is that those functions are dense in the Sobolev spaces, *cf.* the remark in the proof of [Lecture → Thm. 1.3.4.17].

Remark. Note that (1.4.1) implies the second Poincaré-Friedrichs inequality [Lecture → Thm. 1.8.0.20] in one dimension: the first summand on the right-hand side of (1.4.1) vanishes for $u \in H^1_*(]a,b[)$ (→ [Lecture → (1.8.0.15)]).

HIDDEN HINT 3 for (1-4.) → `1-4-0-0:poin1.pdf`

SOLUTION for (1-4.) → `1-4-0-1:poin.pdf`

End Problem 1-4 , (45 min.)

Problem 1-5: A second-order elliptic transmission problem in 1D

This problem studies the transition from variational formulation to the strong form of a boundary value problem. It does so for a non-standard second-order elliptic variational problem in 1D, which will lead to a so-called **transmission problem**.

This theoretical problem relies on the techniques and arguments introduced in [**Lecture → Section 1.5.1**], which should be studied before tackling this problems.

In [Lecture → Section 1.5.1] we learned how to find the boundary value problem in strong form spawned by a linear variational problem in one space dimension. The key tool was the integration by parts formula

$$\int_a^b f(\xi)v'(\xi)\,\mathrm{d}\xi = -\int_a^b f'(\xi)v(\xi)\,\mathrm{d}\xi + \underbrace{(f(b)v(b)-f(a)v(a))}_{\text{boundary terms}} \quad \forall f,v \in C^1_{\mathrm{pw}}([a,b])\,.$$

[Lecture → (1.5.1.8)]

Now we apply the approach of [Lecture → § 1.5.1.9] to the linear variational problem:

$$u \in V_0\colon \quad \underbrace{\int_0^1 \frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x + u(1)v(1)}_{=:\mathsf{a}(u,v)} = \underbrace{v(1/2)}_{=:\ell(v)} \quad \forall v \in V_0\,, \tag{1.5.1}$$

posed on the Sobolev space (→ [Lecture → Section 1.3.4])

$$V_0 := \{v \in H^1(]0,1[)\,:\; v(0)=0\}\,. \tag{1.5.2}$$

This space is equipped with the norm of $H^1(]0,1[)$ and, thus, becomes a Hilbert space. Examining the proof of the first Poincaré inequality in 1D, see [Lecture → Thm. 1.3.4.17], we conclude that

$$\|v\|_{L^2(]0,1[)} \le |v|_{H^1(]0,1[)} \quad \forall v \in V_0\,. \tag{1.5.3}$$

This estimate will be useful below.

Remark. A fundamental estimate for this problem will be an inequality established in Problem 1-3, namely

$$|v(x)| \le |v|_{H^1(]0,1[)} \quad \forall x \in [0,1]\,, \quad \forall v \in H^1_0(]0,1[)\,, \tag{1.5.4}$$

which can easily be generalized to V_0:

$$|v(x)| \le |v|_{H^1(]0,1[)} \quad \forall x \in [0,1]\,, \quad \forall v \in V_0\,. \tag{1.5.5}$$

This estimate can be used without proof. It can be obtained as in Hint (1-3.a.arabichintcnt),

$$|u(x)| = \Big|\int_0^x u'(\tau)\,\mathrm{d}\tau + \underbrace{u(0)}_{=0}\Big| \overset{\text{(CSI)}}{\le} \left(\int_0^x 1\,\mathrm{d}\tau\right)^{\frac12}\left(\int_0^x |u'(\tau)|^2\,\mathrm{d}\tau\right)^{\frac12}$$

$$\le \left(\int_0^1 |u'(\tau)|^2\,\mathrm{d}\tau\right)^{\frac12} = |u|_{H^1(]0,1[)} \quad \forall u \in C^1([0,1]),\; u(0)=0\,,$$

and can also be deduced from the multiplicative trace inequality of [Lecture → Thm. 1.9.0.10] for $d=1$. To begin with, (1.5.4) ensures that the variational problem (1.5.1) is well-posed, that is, has a meaningful solution, which will elaborated in more detail in Sub-problem (1-5.a) and Sub-problem (1-5.c).

(1-5.a) ⚁ (15 min.) Show that the bilinear form a on the left-hand side of (1.5.1) is continuous on V_0 (, which, of course, is equipped with the norm of $H^1(]0,1[)$).

HIDDEN HINT 1 for (1-5.a) → 1-5-1-0:h1.pdf

HIDDEN HINT 2 for (1-5.a) → 1-5-1-1:h12.pdf

SOLUTION for (1-5.a) → 1-5-1-2:s1.pdf ▲

(1-5.b) ⚁ (15 min.) Show that the bilinear form a is symmetric and positive definite in the sense of [Lecture → Def. 1.2.3.27].

HIDDEN HINT 1 for (1-5.b) → 1-5-2-0:h21.pdf

SOLUTION for (1-5.b) → 1-5-2-1:s2.pdf ▲

(1-5.c) ⚁ (15 min.) Show that the linear form ℓ on the right-hand side of (1.5.1) is continuous on V_0 with respect to the energy norm induced by a.

HIDDEN HINT 1 for (1-5.c) → 1-5-3-0:h31.pdf

SOLUTION for (1-5.c) → 1-5-3-1:s3.pdf ▲

The next sub-problems are devoted to a step-by-step derivation of the differential equation hidden in (1.5.1). We remind that these consierations hinge on extra smoothness requirements for both solutions and data, compare [Lecture → § 1.5.1.1], [Lecture → Ass. 1.5.1.2]. Obviously, the right-hand-side functional of (1.5.1) does not meet these smoothness requirements in the whole interval $]0,1[$, *only on parts* of it. Therefore the approach of [Lecture → Section 1.5.1] can be pursued only for sub-intervals of $]0,1[$, where [Lecture → Ass. 1.5.1.2] holds. This is the rationale behind the following steps.

Remark. What does it mean that the right-hand side of (1.5.1) "fails to meet the smoothness requirements" of [Lecture → Section 1.5.1]? In that section the focus is on

$$u_* \in H^1(]a,b[)\,, \quad u(a)=u_a,\ u(b)=u_b \quad : \quad \int_a^b \sigma(x)\frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x = \int_a^b f(x)v(x)\,\mathrm{d}x \quad \forall v \in H_0^1(]a,b[)\,,$$

[Lecture → (1.4.2.1)]

with a uniformly positive stiffness coefficient function $\sigma \in C^0_{\mathrm{pw}}([a,b])$ and $f \in L^2(]a,b[)$. The above-mentioned smoothness requirements are stated in

Assumption [Lecture → Ass. 1.5.1.2]. Smoothness required for two-point boundary value problems

The following smoothness requirements have to be satisfied

$$u \in C^2([a,b]) \quad , \quad \sigma \in C^1([a,b]) \quad , \quad f \in C^0([a,b])\,. \tag{1.5.10}$$

It is important to be aware that [Lecture → Ass. 1.5.1.2] is connected with [Lecture → (1.4.2.1)] and assumes a right-hand side linear functional of the form $\ell(v) = \int_a^b f(x)v(x)\,\mathrm{d}x$. Hence, the right-hand side linear functional of (1.5.1) does not comply with [Lecture → Ass. 1.5.1.2] in the sense that

$$\text{there is } \textbf{no} \quad f \in C^0([a,b]) \quad \text{such that} \quad v(1/2) = \int_0^1 f(x)v(x)\,\mathrm{d}x \quad \forall v \in C^\infty([0,1])\,.$$

(1-5.d) ⚂ (20 min.) Find the ordinary differential equation (ODE) satisfied by the solution u of (1.5.1) in *both intervals* $]0,1/2[$ and $]1/2,1[$.

HIDDEN HINT 1 for (1-5.d) → 1-5-4-0:h41.pdf

SOLUTION for (1-5.d) → 1-5-4-1:s4.pdf ▲

(1-5.e) ⚂ (15 min.) [depends on Sub-problem (1-5.d)]

Which boundary condition does the (strong) solution u of (1.5.1) satisfy in $x = 1$?

HIDDEN HINT 1 for (1-5.e) → 1-5-5-0:h51.pdf

SOLUTION for (1-5.e) → 1-5-5-1:s5.pdf ▲

(1-5.f) ⚃ (15 min.) [depends on Sub-problem (1-5.d)]

Which conditions does (1.5.1) imply for its solution u at $x = 1/2$.

HIDDEN HINT 1 for (1-5.f) → 1-5-6-0:h61.pdf

HIDDEN HINT 2 for (1-5.f) → 1-5-6-1:h62.pdf

SOLUTION for (1-5.f) → 1-5-6-2:s6.pdf ▲

(1-5.g) ⚄ (15 min.) [depends on Sub-problem (1-5.d), Sub-problem (1-5.e), Sub-problem (1-5.f)]

State an explicit formula for the solution of (1.5.1).

HIDDEN HINT 1 for (1-5.g) → 1-5-7-0:h7.pdf

SOLUTION for (1-5.g) → 1-5-7-1:s7.pdf ▲

End Problem 1-5 , 110 min.

Problem 1-6: Heat conduction with non-local boundary conditions

This problem is meant to practice the conversion of a variational problem into a boundary value problem for a partial differential equation, see [Lecture → Section 1.5] and the extraction of boundary conditions hidden in the variational formulation as in [Lecture → Ex. 1.5.3.11].

A purely theoretical problem related to the material presented in [**Lecture → Section 1.8**]. We practice the analysis of second-order elliptic linear variational problems and, in particular, the extraction of a boundary value problem from its associated variational problem. In a sense, this task is also linked to [**Lecture → Section 1.5.3**].

Concretely, we consider the modeling of a two-dimensional cross-section of a submerged insulated wire, see Fig. 2. The wire has a central core of conducting material, say copper, which carries a current. Ohmic losses lead to heat generation in the copper. Copper conducts heat very well and, thus, the copper core can be assumed to have a uniform but unknown temperature.

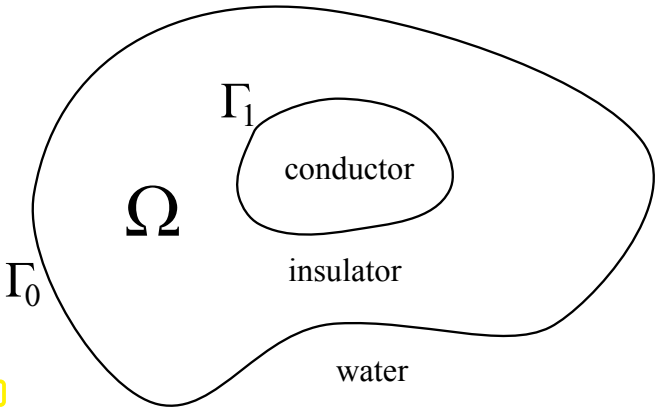


Fig. 2

The copper is surrounded by an annulus of insulator, some plastic for example, which is submerged in water. Furthermore, we assume the water to be at a constant temperature of 0 (w.r.t. to some reference temperature in arbitrary units). We seek a mathematical model providing us with the temperature distribution within the insulator. Such a model is given by the linear variational problem

$$u \in V_0 : \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = \int_{\Gamma_1} v(\boldsymbol{x}) \mathrm{d}S\,, \quad \forall v \in V_0\,, \tag{1.6.1}$$

where the heat conductivity $\kappa \in L^\infty(\Omega)$ is uniformly positive (→ [Lecture → Def. 1.2.2.9]) and bounded, and which is posed on the Sobolev space

$$V_0 := \left\{ v \in H^1(\Omega) : \; v|_{\Gamma_0} = 0,\; v|_{\Gamma_1} = \mathrm{const} \right\}. \tag{1.6.2}$$

The computational domain $\Omega \subset \mathbb{R}^2$ is the region of the plane occupied by the cross-section of the insulator.

(1-6.a) ⚀ (5 min.) Determine a bilinear form a and linear form ℓ so that (1.6.1) becomes an abstract linear variational problem $\mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0$.

SOLUTION for (1-6.a) → `1-6-1-0:heatnonloc1.pdf` ▲

(1-6.b) ⚁ (5 min.) Argue, why the bilinear form a found in (1-6.a) is positive definite (→ [Lecture → Def. 1.2.3.24]) on the Sobolev space V_0.

HIDDEN HINT 1 for (1-6.b) → `1-6-2-0:hcw1ah1.pdf`

SOLUTION for (1-6.b) → `1-6-2-1:heatnonloc1a.pdf` ▲

(1-6.c) ⚂ (10 min.) Show that ℓ is continuous (bounded) with respect to the energy norm induced by a, *cf.* [Lecture → Def. 1.2.3.42]. In the lecture we found this to be an essential condition for the well-posedness of a linear variational problem, see [Lecture → Lemma 1.2.3.39].

Without proof you may use a consequence of a slightly generalized Poincaré-Friedrichs inequality

[Lecture → Thm. 1.3.4.17], namely

$$\exists C = C(\Omega): \quad \|v\|_{L^2(\Omega)} \leq |v|_{H^1(\Omega)} \quad \forall v \in V_0 \ . \tag{1.6.3}$$

HIDDEN HINT 1 for (1-6.c) → 1-6-3-0:heatnonloc21.pdf

SOLUTION for (1-6.c) → 1-6-3-1:heatnonloc2.pdf ▲

(1-6.d) ⚂ (30 min.) If u solves (1.6.1) and is sufficiently smooth, it also satisfies a partial differential equation on Ω pointwise (i.e. in the sense of classical calculus). Find this equation.

HIDDEN HINT 1 for (1-6.d) → 1-6-4-0:heatnonloc31.pdf

SOLUTION for (1-6.d) → 1-6-4-1:heatnonloc3.pdf ▲

(1-6.e) ⚄ (15 min.) [depends on Sub-problem (1-6.d)]

The function u from problem (1-6.d) must also satisfy a certain *non-local* boundary condition implied by (1.6.1). Find this boundary condition.

HIDDEN HINT 1 for (1-6.e) → 1-6-5-0:heatnonloc41.pdf

SOLUTION for (1-6.e) → 1-6-5-1:heatnonloc4.pdf ▲

(1-6.f) ⚂ (10 min.) [depends on Sub-problem (1-6.e)]

What is the physical interpretation of the boundary condition from (1-6.e) in terms of heat conduction?

SOLUTION for (1-6.f) → 1-6-6-0:heatnonloc5.pdf ▲

End Problem 1-6 , 75 min.

Problem 1-7: A second-order boundary value problem for vector fields

This problem studies a somewhat exotic linear "elliptic" variational problem for vector fields $\Omega \subset \mathbb{R}^d \mapsto \mathbb{R}^d$. It revisits the derivation of a boundary value problem in strong form from a variational formulation, as presented in [Lecture → Section 1.5]. Integration by parts by means of Green's formula [Lecture → Thm. 1.5.2.7] will play a crucial role.

This is a theoretical problem connected with [**Lecture → Section 1.5**] and [**Lecture → Section 1.3**]. It is an excellent test of your grasp of Sobolev spaces, linear variational problems, and the manipulations converting the weak and strong form of boundary value problems into each other.

Given a bounded computational domain $\Omega \subset \mathbb{R}^2$, we consider the linear variational problem:
seek a continuously differentiable vector field $\mathbf{u} \in (C^1(\overline{\Omega}))^2$, that is, a continuously differentiable function $\mathbf{u} : \overline{\Omega} \to \mathbb{R}^2$, such that

$$\int_\Omega \alpha(\boldsymbol{x}) \operatorname{div} \mathbf{u}(\boldsymbol{x}) \operatorname{div} \mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \ell(\mathbf{v}) \quad \forall \mathbf{v} \in (C^1(\overline{\Omega}))^2 \tag{1.7.1}$$

with a suitable linear form $\ell : (C^1(\overline{\Omega}))^2 \to \mathbb{R}$. The divergence of a vector field $\mathbf{v} : \Omega \to \mathbb{R}^2$ is the function [Lecture → § 0.3.2.19]

$$\operatorname{div} \mathbf{v}(\boldsymbol{x}) := \frac{\partial v_1}{\partial x_1}(\boldsymbol{x}) + \frac{\partial v_2}{\partial x_2}(\boldsymbol{x}) \,, \quad \mathbf{v}(\boldsymbol{x}) = \begin{bmatrix} v_1(\boldsymbol{x}) \\ v_2(\boldsymbol{x}) \end{bmatrix} \,, \quad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \,.$$

We assume that the coefficient function $\alpha : \overline{\Omega} \to \mathbb{R}$ is continuous and uniformly positive (UP) [Lecture → (1.2.1.20)],

$$\alpha \in C^0(\overline{\Omega}) \quad \text{and} \quad \exists \alpha_0 > 0 : \quad \alpha(\boldsymbol{x}) \geq \alpha_0 \quad \forall \boldsymbol{x} \in \overline{\Omega} \,. \tag{1.7.2}$$

Moreover, for the space of continuously differentiable vector fields on $\overline{\Omega}$, we have written [Lecture → Section 0.3.2.3]

$$(C^1(\overline{\Omega}))^2 := \{\mathbf{v} : \overline{\Omega} \to \mathbb{R}^2 \text{ continuously differentiable}\} \,. \tag{1.7.3}$$

We also remark that the $L^2(\Omega)$-norm of a vectorfield $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \in (L^2(\Omega))^2$ is defined component-wise:

$$\|\mathbf{v}\|_0^2 := \|\mathbf{v}\|_{L^2(\Omega)}^2 := \int_\Omega v_1^2(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} + \int_\Omega v_2^2(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \,. \tag{1.7.4}$$

(1-7.a) ⚀ (10 min.) Give the formula for the energy norm $\|\cdot\|_a$ (→ [Lecture → Def. 1.2.3.35]) associated with the linear variational problem (1.7.1).

SOLUTION for (1-7.a) → 1-7-1-0:s0.pdf ▲

(1-7.b) ⚁ (10 min.) [depends on Sub-problem (1-7.a)]

How could one define the "energy space" V_0 induced by the left-hand-side bilinear form of (1.7.1)?

SOLUTION for (1-7.b) → 1-7-2-0:s0a.pdf ▲

(1-7.c) ⚂ (10 min.) [depends on Sub-problem (1-7.a)]

Show that for $\mathbf{f} \in (L^2(\Omega))^2$ the concrete right-hand side functional

$$\ell(\mathbf{v}) := \int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \,, \quad \mathbf{v} \in (C^1(\overline{\Omega}))^2 \,, \tag{1.7.6}$$

is continuous with respect to the energy norm $\|\cdot\|_a$ found in Sub-problem (1-7.a).

SOLUTION for (1-7.c) → 1-7-3-0:g1s.pdf ▲

(1-7.d) ⚄ (30 min.) [depends on Sub-problem (1-7.a)]

Another possibility for a right-hand side functional in (1.7.1) is

$$\ell(\mathbf{v}) := \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) \,, \quad \mathbf{v} \in (C^1(\overline{\Omega}))^2 \,, \tag{1.7.8}$$

where $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ is the exterior unit normal vector field.

Show that also this functional is continuous with respect to the energy norm $\|\cdot\|_a$ found in Sub-problem (1-7.a).

HIDDEN HINT 1 for (1-7.d) → 1-7-4-0:g2h.pdf

SOLUTION for (1-7.d) → 1-7-4-1:g2s.pdf ▲

(1-7.e) ⚃ (30 min.) Assuming that $\alpha \in C^1(\Omega)$ and $\mathbf{u} \in (C^2(\overline{\Omega}))^2$, state the PDE ("strong" form) of the boundary value problem solved by the solution $\mathbf{u}$ of the variational problem (1.7.1) with the right-hand side functional (1.7.6).

HIDDEN HINT 1 for (1-7.e) → 1-7-5-0:gpdeh.pdf

SOLUTION for (1-7.e) → 1-7-5-1:gpde.pdf ▲

(1-7.f) ⚃ (20 min.) [depends on Sub-problem (1-7.e)]

Answer the question of the previous sub-problem ("state the PDE form/strong form of the boundary value problem") for the choice (1.7.8) of right-hand side functional in (1.7.1), that is, for the variational problem

$$\int_\Omega \alpha(\boldsymbol{x}) \, \operatorname{div} \mathbf{u}(\boldsymbol{x}) \, \operatorname{div} \mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \ell(\mathbf{v}) := \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) \quad \forall \mathbf{v} \in (C^1(\overline{\Omega}))^2$$

HIDDEN HINT 1 for (1-7.f) → 1-7-6-0:gpde2h.pdf

SOLUTION for (1-7.f) → 1-7-6-1:spde2.pdf ▲

(1-7.g) ⚀ (15 min.) State the quadratic minimization problem that is equivalent to (1.7.1) for ℓ given by (1.7.8).

SOLUTION for (1-7.g) → 1-7-7-0:qmps.pdf ▲

(1-7.h) ⚃ (20 min.) A student wants to use the method of manufactured solutions [Lecture → § 3.8.0.9] for the validation of his implementation of a finite element method for variational problems of the type (1.7.1). She chooses $\alpha \equiv 1$ and wants to impose the exact solution

$$\mathbf{u}(\boldsymbol{x}) := \begin{bmatrix} \sin(x_1) \\ \cos(x_2) \end{bmatrix} \,, \quad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \,. \tag{1.7.17}$$

Specify a right-hand side linear form ℓ for (1.7.1) so that this $\mathbf{u}$ will be the solution.

HIDDEN HINT 1 for (1-7.h) → 1-7-8-0:mfph.pdf

SOLUTION for (1-7.h) → 1-7-8-1:mfps.pdf ▲

(1-7.i) ⚃ (20 min.) [depends on Sub-problem (1-7.a)]

The student employs a finite element Galerkin discretization of (1.7.1) using the trial and test spaces $V_{0,h} = (\mathcal{S}_2^0(\mathcal{M}))^2$, where $\mathcal{M}$ stands for a triangular mesh of Ω.

She studies convergence of the finite element Galerkin solutions in the energy norm $\|\cdot\|_a$ from Sub-problem (1-7.a) and on sequences of triangular meshes obtained by successive regular refinement of

an initial coarse mesh. The underlying variational problem is the one derived in Sub-problem (1-7.h) with exact solution $\mathbf{u}$ given in (1.7.17).

Writing $\mathbf{u}_h$ for the finite element solution, describe the asymptotic convergence of $\|u - u_h\|_a$ *in terms of the number* L *of refinements* for $L \to \infty$.

SOLUTION for (1-7.i) $\rightarrow$ 1-7-9-0:fes.pdf ▲

End Problem 1-7 , 165 min.

Problem 1-8: A coupled reaction-diffusion problem

This problems starts from a spatial equilibrium model for complementary biochemical reactions. One compound promotes the production of the other, whereas that one inhibits the production of the former. In addition, one compound attracts the inflow of the other. This situation can be described by two coupled second-order elliptic boundary value problems. We want to derive the weak formulation for this coupled system.

This theoretical problem mainly relies on the techniques introduced in [**Lecture** → **Section 1.8**].

Let $\Omega \subset \mathbb{R}^3$ stand for a compartment of a cell, and $u, w : \Omega \to \mathbb{R}^+$ denote the (stationary $\hat{=}$ time-independent) concentrations of two compounds U and W, units $[u] = [w] = \frac{\text{mol}}{\text{l}}$. In the sequel, however, we assume appropriate scaling and start from a non-dimensional model.

The two compounds spread by diffusion and react with each other in the following way:

- There is a steady production of U, described by a source function $f : \Omega \to \mathbb{R}$.
- W acts as a catalyst for the production of U.
- U promotes the breakdown of W.
- U enables the inflow of W into Ω.

These relationships are captured by the following model in the form of two coupled scalar second-order elliptic boundary value problems:

$$\begin{aligned} -\Delta u &= c(\boldsymbol{x})w + f \\ -\Delta w &= -u \end{aligned} \quad \text{in} \quad \Omega \quad , \quad \begin{aligned} \mathbf{grad}\, u \cdot \boldsymbol{n} &= 0 \\ \mathbf{grad}\, w \cdot \boldsymbol{n} &= -u \end{aligned} \quad \text{on} \quad \partial\Omega \, . \tag{1.8.1}$$

Here, $f \in L^2(\Omega)$ and $c : \Omega \to \mathbb{R}^+$ is a uniformly positive reaction coefficient.

(1-8.a) ⚄ (15 min.) Following the steps outlined in [Lecture → Section 1.8] derive the variational formulation of (1.8.1). Do not forget to specify the trial and test (Sobolev) spaces.

HIDDEN HINT 1 for (1-8.a) → 1-8-1-0:h1.pdf

SOLUTION for (1-8.a) → 1-8-1-1:s1.pdf ▲

(1-8.b) ⚄ (15 min.) [depends on Sub-problem (1-8.a)]

The variational formulation found in Sub-problem (1-8.a) is a *linear variational problem* of the form

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \, ,$$

posed on a vector space V_0, and with a suitable bilinear form $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$ and linear form $\ell : V_0 \to \mathbb{R}$. Identify V_0, a, and ℓ and give concrete formulas for the two latter.

SOLUTION for (1-8.b) → 1-8-2-0:s2.pdf ▲

(1-8.c) ⚄ (15 min.) [depends on Sub-problem (1-8.a)]

Show the *uniqueness* of solutions (assuming they exist) of the linear variational problem from Sub-problem (1-8.a).

HIDDEN HINT 1 for (1-8.c) → 1-8-3-0:h1.pdf

HIDDEN HINT 2 for (1-8.c) → 1-8-3-1:h32.pdf

SOLUTION for (1-8.c) → 1-8-3-2:s5.pdf ▲

(1-8.d) ⚄ (15 min.) [depends on Sub-problem (1-8.b)]

Assume constant reaction coefficient $c \equiv 1$. Is the bilinear form on the left-hand-side of the variational problem established in Sub-problem (1-8.b) positive definite?

HIDDEN HINT 1 for (1-8.d) → 1-8-4-0:h41.pdf

SOLUTION for (1-8.d) → 1-8-4-1:s4.pdf ▲

End Problem 1-8 , 60 min.

Problem 1-9: Second-order Elliptic BVP from weak formulations

Given the variational formulation of a boundary value problem in Sobolev spaces, integration by parts yields its strong form, comprising a partial differential equation plus boundary conditions.

This problem requires the techniques applied in [Lecture → Section 1.5.3]. The contents of [Lecture → Section 1.8] are useful.

As has been demonstrated in the lecture, the conversion from weak to strong form of a second-order elliptic boundary-value problem relies on Green's first formula:

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \tag{1.9.1}$$

(1-9.a) ⚂ (12 min.) Let $\Omega \subset \mathbb{R}^2$ be a bounded domain and $\boldsymbol{a} \in \mathbb{R}^2$ fixed. For the following linear variational problem

$$u \in H^1_0(\Omega)\colon \quad \int_\Omega (\boldsymbol{a} \cdot \mathbf{grad}\, u(\boldsymbol{x}))(\boldsymbol{a} \cdot \mathbf{grad}\, v(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} = \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_0(\Omega) \tag{1.9.2}$$

state the related second-order elliptic boundary value problem in strong form:

$$\square = \square \quad \text{in} \quad \Omega\,,$$
$$\square = \square \quad \text{on} \quad \partial\Omega\,.$$

SOLUTION for (1-9.a) → 1-9-1-0:.pdf

▲

(1-9.b) ⚂ (18 min.) We consider the linear variational problem: seek $u \in H^1(\Omega)$ such that

$$\int_\Omega (\boldsymbol{a} \cdot \boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})$$
$$= \int_{\partial\Omega} v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \tag{1.9.3}$$

for all $v \in H^1(\Omega)$, where Ω is a bounded domain and $\boldsymbol{a} \in \mathbb{R}^2$ is a given vector. State the corresponding boundary value problem:

$$\square = \square \quad \text{in} \quad \Omega\,,$$
$$\square = \square \quad \text{on} \quad \partial\Omega\,.$$

SOLUTION for (1-9.b) → 1-9-2-0:.pdf

▲

End Problem 1-9, 30 min.

Problem 1-10: From Boundary Value Problems to Variational Problems

[Lecture → Section 1.8] explains how to convert a linear second-order elliptic boundary value problem into an "equivalent" (∗) variational problem. In this problem we practise this conversion.
(∗): "Equivalent" is put in quotation marks because we know from [Lecture → Rem. 1.5.3.10] that, strictly speaking, these two formulations are not equivalent.

This is a purely theoretical problem and it requires familiarity with [Lecture → Section 1.8].
▷ problem name/problem code folder: VPtoBVP

In the following $\Omega \subset \mathbb{R}^2$ designates a two-dimensional bounded computational domain as introduced in [Lecture → § 1.2.1.14]. We also write $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ for its exterior unit normal vector field.

HIDDEN HINT 1 for (1-10.) → 1-10-0-0:vph1.pdf

(1-10.a) ⚂ (20 min.) State, in appropriate Sobolev spaces, the variational problem corresponding to the following second-order elliptic boundary value problem:

$$\frac{\partial}{\partial x_1}\left(e^{x_2}\frac{\partial u}{\partial x_1}(\boldsymbol{x})\right) + \frac{\partial}{\partial x_2}\left(e^{x_1}\frac{\partial u}{\partial x_2}(\boldsymbol{x})\right) = x_1^2 + x_2^2 \quad \text{in} \quad \Omega \quad , \quad u = 0 \quad \text{on} \quad \partial\Omega \, ? \tag{1.10.1}$$

$u \in$ [ ] : [ ] = [ ] $\forall v \in$ [ ] .

SOLUTION for (1-10.a) → 1-10-1-0:vpbvps1.pdf ▲

(1-10.b) ⚂ (20 min.) Write down the variational problem induced by the second-order elliptic boundary value problem

$$-\Delta u = 1 \quad \text{in} \quad \Omega \quad , \quad -\mathbf{grad}\, u \cdot \boldsymbol{n} = u \quad \text{on} \quad \partial\Omega \, . \tag{1.10.3}$$

in suitable Sobolev spaces:

$u \in$ [ ] : [ ] = [ ] $\forall v \in$ [ ] .

SOLUTION for (1-10.b) → 1-10-2-0:vpbvps2.pdf ▲

(1-10.c) ⚄ (20 min.) Given a uniformly positive [Lecture → Def. 1.2.2.9] continuous function $\alpha \in C^0(\overline{\Omega})$, convert the second-order *vectorial* boundary value problem

$$-\mathbf{grad}(\alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}(\boldsymbol{x})) + \mathbf{u}(\boldsymbol{x}) = 0 \quad \text{in} \quad \Omega \quad , \quad \alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}(\boldsymbol{x}) = 1 \quad \text{on} \quad \partial\Omega \, , \tag{1.10.4}$$

into a linear variational problem posed in a suitable Sobolev space

$\mathbf{u} \in V$: [ ] = [ ] $\forall v \in V$,

$V :=$ [ ] .

SOLUTION for (1-10.c) → 1-10-3-0:.pdf ▲

End Problem 1-10 , 60 min.

Problem 1-11: Strong and Weak Form of Elliptic BVPs

Elliptic boundary value problems as part of mathematical models come in various guises: some are stated as boundary value problems for PDEs in **strong form**, some are given in **weak form** as variational problems. Both forms can be converted into each other and this is the focus of this problem.

This problems is connected with [Lecture → Section 1.5] and [Lecture → Section 1.8].
▷ problem name/problem code folder: StrongWeakEllBVP

(1-11.a) ⚂ (20 min.) State the complete variational form (in appropriate Sobolev spaces) of the following second-order elliptic boundary value problem on a computational domain $\Omega \subset \mathbb{R}^2$:

$$-\left(\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial x_1 \partial x_2} + \frac{\partial^2 u}{\partial {x_2}^2}\right) + \|\boldsymbol{x}\|^2 u = 0 \quad \text{in} \quad \Omega\,,$$
$$u = 1 \quad \text{on} \quad \partial\Omega\,.$$

$u \in$ [ ] :

$$\int_\Omega \boxed{}\,\mathrm{d}\boldsymbol{x} = \boxed{}$$

$\forall v \in$ [ ] .

SOLUTION for (1-11.a) → 1-11-1-0:swes1.pdf ▲

(1-11.b) ⚂ (15 min.) Determine the boundary value problem in strong from, whose variational/weak form reads:

Seek $u \in H^1(\Omega)$ such that

$$\int_\Omega \frac{\partial u}{\partial x_1}(\boldsymbol{x})\frac{\partial v}{\partial x_1}(\boldsymbol{x}) + \alpha(\boldsymbol{x})\frac{\partial u}{\partial x_2}(\boldsymbol{x})\frac{\partial v}{\partial x_2}(\boldsymbol{x}) + \alpha(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,,$$

where $\alpha \in C^0(\overline{\Omega})$ is a given coefficient function.

$$\boxed{} = \boxed{} \quad \text{in} \quad \Omega\,,$$
$$\boxed{} = \boxed{} \quad \text{on} \quad \partial\Omega\,.$$

SOLUTION for (1-11.b) → 1-11-2-0:.pdf ▲

(1-11.c) ⚂ (15 min.) Let $\Omega \subset \mathbb{R}^2$ be a bounded domain. Give an alternative and simpler characterization of the following set of functions

$$\mathcal{A} := \left\{\alpha \in C^0(\overline{\Omega}) : \exists \gamma > 0 \text{ such that } \mathsf{a}(\alpha; v, v) \geq \gamma \|v\|^2_{H^1(\Omega)} \quad \forall v \in H^1(\Omega)\right\}\,,$$

with $\mathsf{a}(\alpha; u, v) := \int_\Omega \frac{\partial u}{\partial x_1}(\boldsymbol{x})\frac{\partial v}{\partial x_1}(\boldsymbol{x}) + \alpha(\boldsymbol{x})\frac{\partial u}{\partial x_2}(\boldsymbol{x})\frac{\partial v}{\partial x_2}(\boldsymbol{x}) + \alpha(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$

for $u, v \in H^1(\Omega)$.

$$\mathcal{A} = \left\{ \alpha \in C^0(\overline{\Omega}) : \quad \right\} .$$

SOLUTION for (1-11.c) → 1-11-3-0:swes3.pdf ▲

End Problem 1-11 , 50 min.

Chapter 2

Finite Element Methods (FEM)

Problem 2-1: Properties of Galerkin solutions

This problem studies the abstract Galerkin discretization of linear variational problems as introduced in [Lecture → Section 2.2]. The goal is to practice a few elementary manipulations and estimates and realize that even on the abstract level Galerkin solutions enjoy a few fundamental properties.

You should master [Lecture → Section 1.4] and [Lecture → Section 2.2.1] before tackling this problem. You will learn to handle bilinear and linear forms and to exploit the properties of a solution of a linear variational problem.

We consider a generic linear variational problem [Lecture → Def. 1.4.1.6]

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{2.1.1}$$

posed on a vector space V_0, and with a bilinear form $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$, and a linear form $\ell : V_0 \to \mathbb{R}$. Given a finite-dimensional *subspace* $V_{0,h} \subset V_0$ Galerkin discretization of (2.1.1) leads to the discrete variational problem:

$$u_h \in V_{0,h}: \quad \mathsf{a}(u_h,v_h) = \ell(v_h) \quad \forall v_h \in V_{0,h} \,. \tag{2.1.2}$$

(2-1.a) ⚄ (15 min.) For the space $V_0 := \mathbb{R}^2$ find a bilinear form $\mathsf{a} : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$, and a linear form $\ell : \mathbb{R}^2 \to \mathbb{R}$, and a non-trivial subspace $V_{0,h} \subset V_0$ such that

- (2.1.1) has a unique solution,
- but (2.1.2) fails to have a solution.

HIDDEN HINT 1 for (2-1.a) → `2-1-1-0:h1.pdf`

HIDDEN HINT 2 for (2-1.a) → `2-1-1-1:h12.pdf`

SOLUTION for (2-1.a) → `2-1-1-2:s1.pdf` ▲

For the remainder of the this problem we assume that

(i) the bilinear form a in (2.1.1) is *symmetric positive definite* (s.p.d.) on V_0, and

(ii) the linear form ℓ is *continuous* with respect to the energy norm $\|\cdot\|_{\mathsf{a}}$ induced by a, see [Lecture → Section 1.2.3.4].

(iii) the linear variational problem (2.1.1) has a unique solution $u \in V_0$.

The quadratic functional [Lecture → Def. 1.2.3.2] associated with (2.1.1) is

$$J(v) := \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v) \,, \quad v \in V_0 \,. \tag{2.1.5}$$

(2-1.b) ⚀ (5 min.) Argue, why the discrete variational problem (2.1.2) has a unique solution $u_h \in V_{0,h}$.

SOLUTION for (2-1.b) → 2-1-2-0:s2.pdf ▲

(2-1.c) ⚁ (10 min.) Show a property called **Galerkin orthogonality**:

$$\mathsf{a}(u - u_h, v_h) = 0 \quad \forall v_h \in V_{0,h} \;, \tag{2.1.6}$$

which will play a pivotal role in [Lecture → Chapter 3].

SOLUTION for (2-1.c) → 2-1-3-0:s3.pdf ▲

(2-1.d) ⚂ (15 min.) [depends on Sub-problem (2-1.c)]

Show that

$$J(u_h) - J(u) = \tfrac{1}{2}\|u - u_h\|_{\mathsf{a}}^2 \;. \tag{2.1.7}$$

HIDDEN HINT 1 for (2-1.d) → 2-1-4-0:h41.pdf

SOLUTION for (2-1.d) → 2-1-4-1:s4.pdf ▲

End Problem 2-1 , 45 min.

Problem 2-2: Transformation of Galerkin Matrices

The key result of [Lecture → Lemma 2.2.3.2] told us how Galerkin matrices arising from the Galerkin discretization of linear variational problems change as a consequence of a change of basis: the "new" Galerkin matrix $\widetilde{\mathbf{A}}$ is obtained from the "old" one $\mathbf{A}$ by a congruence transformation $\widetilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^\top$. Ths problem examines particular changes of bases and the related transformations of Galerkin matrices.

Familiarity with the principles of Galerkin discretization [Lecture → Section 2.2] is assumed. A small-scale implementation based on EIGEN will be requested.
▷ problem name/problem code folder: TransformationGalerkinMatrices

We consider a generic linear variational problem as introduced in [Lecture → (1.4.1.7)]

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{2.2.1}$$

posed on a vector space V_0, on which also the bilinear form $\mathsf{a}(\cdot,\cdot)$ and the linear form $\ell(\cdot)$ are defined.

We perform Galerkin discretization of (2.2.1) using the finite-dimensional trial/test space $V_{0,h} \subset V_0$, equipped with a basis

$$\mathfrak{B}_h := \{b_h^1, \ldots, b_h^N\} \,, \quad N := \dim V_{0,h} \,.$$

This results in the Galerkin matrix $\mathbf{A} \in \mathbb{R}^{N,N}$: $(\mathbf{A})_{i,j} = \mathsf{a}(b_h^j, b_h^i), \quad i,j \in \{1,\ldots,N\}$, see [Lecture → Section 2.2.2].

Assume that N is even, $N = 2M$, $M \in \mathbb{N}$, and consider the following change of basis: The new basis $\widetilde{\mathfrak{B}}_h = \left\{\widetilde{b}_h^1, \ldots, \widetilde{b}_h^N\right\}$ consists of the "vectors" (which may be functions!)

$$\widetilde{b}_h^j := b_h^{2j-1} + b_h^{2j} \quad \text{for} \quad j \in \{1,\ldots,M\} \,, \tag{2.2.2a}$$

$$\widetilde{b}_h^j := b_h^{2(j-M)-1} - b_h^{2(j-M)} \quad \text{for} \quad j \in \{M+1,\ldots,N\} \,. \tag{2.2.2b}$$

As in [Lecture → Lemma 2.2.3.2] we write $\widetilde{\mathbf{A}} \in \mathbb{R}^{N,N}$ for the Galerkin matrix spawned by the new basis $\widetilde{\mathfrak{B}}_h$:

Lemma [Lecture → Lemma 2.2.3.2]. Effect of change of basis on Galerkin matrix

Consider the ***discrete variational problem***

$$u_h \in V_{0,h}: \quad \mathsf{a}(u_h, v_h) = \ell(v_h) \quad \forall v_h \in V_{0,h} \,. \tag{2.2.3}$$

and two bases of $V_{0,h}$,

$$\mathfrak{B}_h := \{b_h^1, \ldots, b_h^N\} \quad , \quad \widetilde{\mathfrak{B}}_h := \{\widetilde{b}_h^1, \ldots, \widetilde{b}_h^N\} \,,$$

related by the basis transformation matrix $\mathbf{S}$ *according to*

$$\widetilde{b}_h^j = \sum_{k=1}^N s_{jk} b_h^k \quad \textit{with} \quad \mathbf{S} = \left[s_{jk}\right]_{j,k=1}^N \in \mathbb{R}^{N,N} \textit{ regular.} \tag{2.2.4}$$

Then the ***Galerkin matrices*** $\mathbf{A}, \widetilde{\mathbf{A}} \in \mathbb{R}^{N,N}$, *the* ***right hand side vectors*** $\vec{\boldsymbol{\varphi}}, \widetilde{\vec{\boldsymbol{\varphi}}} \in \mathbb{R}^N$, *and the coefficient vectors* $\vec{\boldsymbol{\mu}}, \widetilde{\vec{\boldsymbol{\mu}}} \in \mathbb{R}^N$, *respectively, satisfy*

$$\widetilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^\top \quad , \quad \widetilde{\vec{\boldsymbol{\varphi}}} = \mathbf{S}\vec{\boldsymbol{\varphi}} \quad , \quad \widetilde{\vec{\boldsymbol{\mu}}} = \mathbf{S}^{-\top}\vec{\boldsymbol{\mu}} \,. \tag{2.2.5}$$

(2-2.a) ⚀ (5 min.) For the case $N = 4$ write down the basis transformation matrix $\mathbf{S} = [s_{jk}] \in \mathbb{R}^{4,4}$ as in [Lecture → Lemma 2.2.3.2] defined through

$$\widetilde{b}_h^j = \sum_{k=1}^{4} s_{jk} b_h^k \,, \tag{2.2.6}$$

where the notations from (2.2.2) are used.

Also compute the matrix product $\mathbf{S}^\top \mathbf{S}$. What do you observe?

SOLUTION for (2-2.a) → 2-2-1-0:s1.pdf ▲

(2-2.b) ⚁ (10 min.) [depends on Sub-problem (2-2.a)]

Again we consider the change of basis $\mathfrak{B}_h \to \widetilde{\mathfrak{B}}_h$ defined in (2.2.2). Write down the basis transformation matrix $\mathbf{S} \in \mathbb{R}^{N,N}$ according to (2.2.6) for the general case $N = 2M$, $M \in \mathbb{N}$. Show that $\mathbf{S}$ is invertible and determine its inverse.

HIDDEN HINT 1 for (2-2.b) → 2-2-2-0:h2q.pdf

SOLUTION for (2-2.b) → 2-2-2-1:s2.pdf ▲

(2-2.c) ⚂ (20 min.) Looking at the formulas of [Lecture → Lemma 2.2.3.2] more closely, we see that every entry $\left(\widetilde{\mathbf{A}}\right)_{ij}$, $i,j \in \{1,\ldots,N\}$, of $\widetilde{\mathbf{A}}$ can be obtained by linearly combining four entries of $\mathbf{A}$. Give the concrete formulas in the form

$$\left(\widetilde{\mathbf{A}}\right)_{ij} = ? \cdot (\mathbf{A})_{?,?} + ? \cdot (\mathbf{A})_{?,?} + ? \cdot (\mathbf{A})_{?,?} + ? \cdot (\mathbf{A})_{?,?} \,, \quad i,j \in \{1,\ldots,N\} \,,$$

replacing the question marks with suitable numbers depending on the indices i and j.

HIDDEN HINT 1 for (2-2.c) → 2-2-3-0:h321.pdf

SOLUTION for (2-2.c) → 2-2-3-1:s3.pdf ▲

(2-2.d) ⚃ (20 min.) [depends on Sub-problem (2-2.c)]

We take a dual perspective compared to Sub-problem (2-2.c). For every entry $(\mathbf{A})_{k,\ell}$ of $\mathbf{A}$ determine to which four entries of $\widetilde{\mathbf{A}}$ it will contribute according to the formula $\widetilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^\top$, $\mathbf{S}$ the basis transformation matrix as in [Lecture → Lemma 2.2.3.2], and with what weight.

HIDDEN HINT 1 for (2-2.d) → 2-2-4-0:h41.pdf

SOLUTION for (2-2.d) → 2-2-4-1:s4.pdf ▲

Recall the **triplet format** or **COO-format** for a sparse matrix: It stores an $n \times m$-matrix $\mathbf{M}$ in a long vector of triplets corresponding to the non-zero entries of the matrix

$$[(i_\ell, j_\ell, a_\ell)]_{\ell=1}^{L} \,, \quad L \in \mathbb{N} \,, \quad i_\ell \in \{1,\ldots,n\} \,, \quad j_\ell \in \{1,\ldots,m\} \,, \quad a_\ell \in \mathbb{R} \,,$$

according to the convention

$$(\mathbf{M})_{k,q} = \sum_{\ell:\, \substack{i_\ell = k \\ j_\ell = q}} a_\ell \,, \quad k \in \{1,\ldots,n\} \,, \quad q \in \{1,\ldots,m\} \,.$$

EIGEN offers a particular data type for triplets: **Eigen::Triplet**, see ⇢ documentation, with member functions

- `Eigen::Triplet::col()` (↔ i_ℓ),
- `Eigen::Triplet::row()` (↔ j_ℓ),

- `Eigen::Triplet::value()` ($\leftrightarrow a_\ell$),

to access its data.

(2-2.e) ⚄ (45 min.) [depends on Sub-problem (2-2.d)]

Implement a C++ function

```
std::vector<Eigen::Triplet<double>> transformCOOmatrix(
  const std::vector<Eigen::Triplet<double>> &A);
```

that takes the Galerkin matrix $\mathbf{A}$ in triplet format and returns $\widetilde{\mathbf{A}}$ als in triplet format.

It can be taken for granted that the argument matrix has no rows or columns with all zero entries, which makes it possible to determine the matrix size from the triplet information.

SOLUTION for (2-2.e) → 2-2-5-0:s5.pdf ▲

(2-2.f) ⚄ (20 min.) Now we assume that the bilinear form $\mathsf{a}(\cdot,\cdot)$ is *symmetric* [Lecture → Def. 0.3.1.15]. Based on some of the theorems from [Lecture → Section 0.3.1.4] show the following result:

Theorem 2.2.13. Identity Galerkin matrices

Consider the Galerkin discretization of a symmetric positive-definite [Lecture → Def. 0.3.1.16] bilinear form $\mathsf{a}(\cdot,\cdot)$ using a finite-dimensional space $V_{0,h}$. Then there is a basis $\mathfrak{B}_h$ of $V_{0,h}$ such that the associated Galerkin matrix is the identity matrix.

This result is a striking reminder how the choice of a suitable basis of the Galerkin trial/test space can facilitate the solution of the resulting linear system.

HIDDEN HINT 1 for (2-2.f) → 2-2-6-0:s6h1.pdf

SOLUTION for (2-2.f) → 2-2-6-1:s6.pdf ▲

End Problem 2-2, 120 min.

Problem 2-3: Pointwise "Exact" Galerkin Solution

Usually the Galerkin finite element solution of a second-order elliptic variational problem need not agree with the exact solution in a single point. However, for Galerkin discretization by means of piecewise linear finite elements in one dimension [Lecture → Section 2.3] and assuming exact computation of the right-hand side vector, we can sometimes rely on agreement in all nodes of the mesh.
This will be proved in this problem based on the discrete variational formulation and elementary calculus.

This is a theoretical problem based on [Lecture → Section 2.2] and [Lecture → Section 2.3]. It is a good practice of integration-by-parts arguments in the context of variational equations.

For $a < b$ and $g \in C^0([a,b])$ we consider the linear variational problem

$$u \in H_0^1([a,b]): \quad \int_a^b \frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x = \int_a^b g(x)v(x)\,\mathrm{d}x \quad \forall v \in H_0^1([a,b])\,. \tag{2.3.1}$$

and its finite element Galerkin discretization based on the trial and test space $\mathcal{S}_{1,0}^0(\mathcal{M})$ [Lecture → § 2.3.1.4] of piecewise linear continuous functions on some mesh $\mathcal{M}$ of $[a,b]$,

$$\mathcal{M} := \{]x_{j-1}, x_j[:\ j \in \{1,\ldots,M\}\}\,, \quad a = x_0 < x_1 < \cdots < x_{M-1} < x_M := b\,, \quad M \in \mathbb{N}\,.$$

also see [Lecture → Section 2.3] for details and notations:

(2-3.a) (10 min.) Assuming $u \in C^2_{0,\mathrm{pw}}([a,b])$ solves (2.3.1), find the 2-point boundary value problem satisfied by u.

HIDDEN HINT 1 for (2-3.a) → 2-3-1-0:exact11.pdf

SOLUTION for (2-3.a) → 2-3-1-1:exact1.pdf ▲

(2-3.b) (40 min.) [depends on Sub-problem (2-3.a)]

Show that the Galerkin solution $u_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$ agrees with the exact solution $u \in H_0^1(]a,b[)$ of (2.3.1) *in the nodes of the mesh*, that is,

$$u_h(x_j) = u(x_j) \quad \text{for all} \quad j = 0,\ldots,M\,, \tag{2.3.2}$$

provided that all entries of the Galerkin matrix and the right-hand side vector are evaluated *exactly*.

HIDDEN HINT 1 for (2-3.b) → 2-3-2-0:exact21.pdf

HIDDEN HINT 2 for (2-3.b) → 2-3-2-1:exact22.pdf

HIDDEN HINT 3 for (2-3.b) → 2-3-2-2:exact23.pdf

There is no escaping "tedious computations" when solving this sub-problem. You have to master integration by parts in 1D as explained in [Lecture → § 1.5.1.5].

SOLUTION for (2-3.b) → 2-3-2-3:exact2.pdf ▲

(2-3.c) ⚄ (15 min.) [Lecture → § 2.3.3.6] presents the approximate evaluation of entries of the right-hand side vector by means of the composite trapezoidal rule

$$\int_a^b \psi(t)\,\mathrm{d}t \approx \sum_{l=1}^{M} \tfrac{1}{2}h_l(\psi(x_{l-1}) + \psi(x_l))\,. \qquad \text{[Lecture → (2.3.3.8)]}$$

Describe a reasonably large class of source functions g in (2.3.1) so that the statement proved in Subproblem (2-3.b) remains true though we use the composite trapezoidal rule for compute the right-hand side vector.

HIDDEN HINT 1 for (2-3.c) → 2-3-3-0:egs4h1.pdf

SOLUTION for (2-3.c) → 2-3-3-1:egs3.pdf ▲

End Problem 2-3 , 65 min.

Problem 2-4: Linear finite elements for two-point boundary value problems

This problem revisits the Galerkin finite element discretization of linear second-order two-point boundary value problems as introduced in [Lecture → Section 2.3]. You will practice the computation of Galerkin matrices and right hand side vectors both in terms of symbolic formulas and implementation.

The core of this exercise is an EIGEN-based implementation relying on the contents of [Lecture → Section 2.3]. Hence, it is good opportunity to get used to doing numerical linear algebra operations in C++ using the EIGEN template library.
▷ problem name/problem code folder: LinearFE1D

The ultimate goal of this exercise is to compute finite element solutions of the following linear variational problems posed on the one-dimensional domain $\Omega =]0,1[$:

(A) Seek $u \in H_0^1(]0,1[)$: such that

$$\int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x) + \gamma(x)u(x)v(x)\,\mathrm{d}x = \int_0^1 f(x)v(x)\,\mathrm{d}x \quad \forall v \in H_0^1(]0,1[)\,. \tag{A}$$

(B) Seek $u \in H^1(]0,1[)$ with $u(0) = u_0 \in \mathbb{R}$, $u(1) = u_1 \in \mathbb{R}$ (both given), such that

$$\int_0^1 \alpha(x)\frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x = \int_0^1 f(x)v(x)\,\mathrm{d}x \quad \forall v \in H_0^1(]0,1[)\,. \tag{B}$$

(C) Seek $u \in H^1(]0,1[)$ such that

$$\int_0^1 \alpha(x)\frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x + \int_0^1 \gamma(x)u(x)v(x)\,\mathrm{d}x = \int_0^1 v(x)\,\mathrm{d}x \quad \forall v \in H^1(]0,1[)\,. \tag{C}$$

Here,

- $f \in C^0([0,1])$ is a source function, given in procedural form through a "black-box" functor that supplies only point evaluations, *cf.* [Lecture → Rem. 2.1.2.5],
- $\alpha \in C^0_{\mathrm{pw}}([0,1])$ (piecewise continuous), $\gamma \in C^0([0,1])$ (globally continuous) are uniformly positive (→ [Lecture → (1.6.0.6)]) coefficient functions, also given in procedural form,
- $u_0, u_1 \in \mathbb{R}$ are prescribed Dirichlet boundary data.

For the following sub-problems you may assume that the cofficient functions α, γ, source functions f, and solutions u are smooth.

(2-4.a) ⚀ (10 min.) Derive the strong form of the two-point boundary value problem induced by (A).

HIDDEN HINT 1 for (2-4.a) → 2-4-1-0:1dfe1h1.pdf

SOLUTION for (2-4.a) → 2-4-1-1:1dfe1.pdf ▲

(2-4.b) ⚁ (10 min.) Describe the two-point boundary value problem (in strong form) related to (B)?

HIDDEN HINT 1 for (2-4.b) → 2-4-2-0:1dfe3h2.pdf

SOLUTION for (2-4.b) → 2-4-2-1:1dfe3.pdf ▲

(2-4.c) ⚄ (5 min.) State the two-point boundary value problem induced by (C)?

HIDDEN HINT 1 for (2-4.c) → 2-4-3-0:1dfe4h2.pdf

SOLUTION for (2-4.c) → 2-4-3-1:1dfe4.pdf ▲

As in [Lecture → § 2.3.1.3], for the domain $\Omega =]0,1[$ we denote by $\mathcal{M}$ a mesh of Ω with nodes

$$0 = x_0 < x_1 < \cdots < x_{M-1} < x_M = 1\,,$$

introduced as a "mesh" or a "grid" in [Lecture → § 2.3.1.3]. Based on $\mathcal{M}$ we define the two finite-dimensional spaces [Lecture → § 2.3.1.4], [Lecture → (2.4.2.2)]

$$\mathcal{S}^0_{1,0}(\mathcal{M}) := \left\{ \begin{array}{l} v \in C^0([0,1])\colon v_{|[x_{i-1},x_i]} \text{ linear,} \\ i = 1,\ldots,M,\, v(0) = v(1) = 0 \end{array} \right\}, \tag{2.4.4}$$

$$\mathcal{S}^0_{1}(\mathcal{M}) := \left\{ \begin{array}{l} v \in C^0([0,1])\colon v_{|[x_{i-1},x_i]} \text{ linear,} \\ i = 1,\ldots,M \end{array} \right\}. \tag{2.4.5}$$

Throughout we equip both spaces with the ordered "tent function" basis, defined by a cardinal basis property with respect to the nodes x_ℓ:

$$\mathcal{S}^0_{1,0}(\mathcal{M}) = \operatorname{span}\left\{b_h^j\right\}_{j=1}^{M-1}\,, \quad b_h^j(x_i) = \delta_{i,j}\,, \quad i,j \in \{1,\ldots,M-1\}\,,$$

$$\mathcal{S}^0_{1}(\mathcal{M}) = \operatorname{span}\left\{q_h^j\right\}_{j=0}^{M}\,, \quad q_h^j(x_i) = \delta_{i,j}\,, \quad i,j \in \{0,\ldots,M\}\,,.$$

For $\mathcal{S}^0_{1,0}(\mathcal{M})$ this *cardinal basis* has been presented in [Lecture → Section 2.3.2] and adding two cardinal basis functions for x_0 and x_M will give a suitable basis for $\mathcal{S}^0_{1}(\mathcal{M})$, [Lecture → (2.4.3.1)].

> Numerical quadrature necessary! Note that, since the source and coefficient functions are available only in procedural form we have to rely on numerical quadrature for the computation of Galerkin matrices and right-hand side vectors.

The use of numerical quadrature in the context of Galerkin finite elemenet discretization of second-order elliptic two-point boundary value problems is discussed in [Lecture → § 2.3.3.6] and [Lecture → § 2.3.3.10].

(2-4.d) ⚄ (20 min.) Compute the linear system arising from the Galerkin finite element discretization of (A) by means of $\mathcal{S}^0_{1,0}(\mathcal{M})$. Use the local trapezoidal rule on $\mathcal{M}$ [Lecture → (2.3.3.8)] for the approximate evaluation of all integrals involving coefficient functions.

HIDDEN HINT 1 for (2-4.d) → 2-4-4-0:1fe5h1.pdf

HIDDEN HINT 2 for (2-4.d) → 2-4-4-1:1fe5h2.pdf

SOLUTION for (2-4.d) → 2-4-4-2:1dfe5.pdf ▲

(2-4.e) ⚄ (30 min.) [depends on Sub-problem (2-4.d)]

Based on Sub-problem (2-4.d) in the file `linearfe1d.h` write a C++ function

```
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::VectorXd solveA(const Eigen::VectorXd& mesh,
                       FUNCTOR1&& gamma, FUNCTOR2&& f);
```

that computes the basis expansion coefficients of the $\mathcal{S}^0_{1,0}(\mathcal{M})$-Galerkin solution of (A) on the mesh passed by the `mesh` argument as a vector of length $M+1$ of sorted node positions x_j. The two other arguments contain functor objects of type `std::function<(double)double>` supplying the coefficient γ and the source function f in procedural form.

Use EIGEN's sparse matrix data format [NumCSE course → Section 2.7.2] for the Galerkin matrix and pay attention to its efficient initialization:

- You may opt for initialization via the COO (triplet-based) matrix format, see ↪ documentation.
- Alternatively, you may directly set the entries of the sparse matrix after a call to `Eigen::SparseMatrix<>::reserve`, because in this problem we know a tight bound for the maximal number of nonzero entries in each row and column of the matrices in advance, see [NumCSE course → Code 2.7.2.1].

Then use a suitable sparse Gaussian elimination solver supplied by EIGEN.

HIDDEN HINT 1 for (2-4.e) → `2-4-5-0:5Ah1.pdf`

SOLUTION for (2-4.e) → `2-4-5-1:s5A.pdf` ▲

(2-4.f) ⚄ (15 min.) [depends on Sub-problem (2-4.d)]

State the linear system of equations obtained through the Galerkin finite element discretization of (C) based on $\mathcal{S}_1^0(\mathcal{M})$. For the first integral on the left-hand side of the variational equation use numerical quadrature based on the composite midpoint rule [Lecture → (2.3.3.13)] on $\mathcal{M}$, for the second rely on the trapezoidal rule [Lecture → (2.3.3.8)] on $\mathcal{M}$.

HIDDEN HINT 1 for (2-4.f) → `2-4-6-0:h71.pdf`

HIDDEN HINT 2 for (2-4.f) → `2-4-6-1:h72.pdf`

SOLUTION for (2-4.f) → `2-4-6-2:1dlfe71.pdf` ▲

(2-4.g) ⚄ (30 min.) [depends on Sub-problem (2-4.f)]

Realize a C++ function

```
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::VectorXd solveC(const Eigen::VectorXd& mesh,
                       FUNCTOR1&& alpha,
                       FUNCTOR2&& gamma);
```

that computes the basis expansion coefficients of the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin solution of (C). The arguments provide the mesh $\mathcal{M}$ and the coefficient functions α and γ, see Sub-problem (2-4.e) for further explanations.

SOLUTION for (2-4.g) → `2-4-7-0:s6C.pdf` ▲

(2-4.h) ⚀ (15 min.) Study [Lecture → Rem. 2.3.3.15] to learn about the offset function technique for the incorporation of non-zero Dirichlet boundary conditions in Galerkin finite element method in one dimension.

▲

(2-4.i) ⚄ (15 min.) [depends on Sub-problem (2-4.d) and Sub-problem (2-4.f)]

The offset function technique for (B) according to [Lecture → Rem. 2.3.3.15] relies on the *offset function with minimal support*

$$\boxed{u_{0,h} = u_0 q_h^0 + u_1 q_h^M}\,, \tag{2.4.21}$$

where q_h^0 and q_h^M are the "half-tent" functions belonging to $x_0 = 0$ and $x_M = 1$. It then targets the modified discrete variational problem

$$w_h \in \mathcal{S}_{1,0}^0(\mathcal{M}): \quad \mathsf{a}(w_h, v_h) = \ell(v_h) - \mathsf{a}(u_{0,h}, v_h) \quad \forall v_h \in \mathcal{S}_{1,0}^0(\mathcal{M})\,. \tag{2.4.22}$$

We use the tent-function basis for $\mathcal{S}^0_{1,0}(\mathcal{M})$, the composite midpoint rule for the approximate evaluation of $a(\cdot,\cdot)$ and the composite trapezoidal rule for $\ell(\cdot)$.

With theses choices, write down the linear system of equations corresponding to (2.4.22).

SOLUTION for (2-4.i) → 2-4-9-0:9bs.pdf ▲

(2-4.j) ⚄ (45 min.) [depends on Sub-problem (2-4.i)]

In the file `lineafe1d.h` complete the implementation of a C++ function

```
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::VectorXd solveB(const Eigen::VectorXd& mesh,
                       FUNCTOR1&& alpha, FUNCTOR2&& f,
                       double u0, double u1)
```

whose return vector contains the values of the Galerkin solution of (B) at *all* the nodes of the mesh. The Galerkin solution should be computed based on the linear system of equations from Sub-problem (2-4.i). The functor arguments pass the coefficient function α and the source function f.

Concerning the implementation, heed the advice given in Sub-problem (2-4.e). Note that you have to set the first and last component of the solution vector to u_0 and u_1 respectively.

SOLUTION for (2-4.j) → 2-4-10-0:1dfe9.pdf ▲

End Problem 2-4 , 195 min.

Problem 2-5: Triangular linear FEM for 2D reaction-diffusion BVP

[Lecture → Section 2.4] gave a detailed introduction into the algorithmic aspects of the use of piecewise linear finite elements on triangular meshes for solving scalar second-order elliptic Neumann problems (in weak form). In this problem we will practice the actual implementation of such a scheme closely following the developments in that section of the course.

Familiarity with [Lecture → Section 2.4] is essential, if you want to benefit for tackling this problem. In this problem you will be asked to implement a few simple functions in C++ based on the numerical linear template library EIGEN. Information on EIGEN and further links can be found in [Lecture → § 2.7.3.2]. All C++ functions should be implemented in the file `simplelinearfiniteelements.cc`. Note that this problem is completely disconnected from LEHRFEM++.
▷ problem name/problem code folder: SimpleLinearFiniteElements

We consider the following linear variational problem on the unit square $\Omega = [0,1]^2$, *cf.* [Lecture → (2.1.2.2)]:

$$u \in H^1(\Omega): \quad \underbrace{\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v + u\, v \,\mathrm{d}\boldsymbol{x}}_{:=\mathsf{a}(u,v)} = \underbrace{\int_\Omega f v \,\mathrm{d}\boldsymbol{x}}_{:=\ell(v)} \quad \forall v \in H^1(\Omega)\,, \tag{2.5.1}$$

where $f \in L^2(\Omega)$ is a given source function.

(2-5.a) ⚀ (15 min.) Taking for granted sufficient smoothness of the solution u and the source function f, derive the strong (PDE) form of the second-order scalar elliptic boundary value problem represented by (2.5.1).

HIDDEN HINT 1 for (2-5.a) → `2-5-1-0:hm1.pdf`

SOLUTION for (2-5.a) → `2-5-1-1:sm.pdf` ▲

In [Lecture → Section 2.4] we have already discussed possible data structures and algorithms underlying an $\mathcal{S}_1^0(\mathcal{M})$-based finite element solver on triangular meshes for this problem. In this exercise, we develop a corresponding C++ code based on the class **TriaMesh2D** listed in [Lecture → Code 2.4.1.2] ➥ GitLab.

The mesh data structure contains the following fields, see also [Lecture → § 2.4.1.1]:

- `TriaMesh2d::Coordinates`: $N \times 2$ matrix, i-th row containing the coordinates of the i-th vertex $\boldsymbol{x}^i$, $i \in \{1,\ldots,N\}$;
- `TriaMesh2d::Elements`: $M \times 3$-matrix, j-th row $\hat{=}$ vertex indices in `TriaMesh2d::Coordinates` array.

(Watch out for clashes of different index conventions: mathematical vs. C++!)

(2-5.b) ⚁ (15 min.) Compute the **element matrix** $\mathbf{M}_K \in \mathbb{R}^{3,3}$ for the bilinear form of the $L^2(K)$ inner product

$$\mathsf{m}(u,v) := \int_\Omega u\, v \,\mathrm{d}\boldsymbol{x}, \quad u, v \in L^2(\Omega)\,, \tag{2.5.4}$$

the piecewise linear finte element space $\mathcal{S}_1^0(\mathcal{M})$ on a triangulation $\mathcal{M}$, and a triangle $K \in \mathcal{M}$ with vertices $\boldsymbol{a}^1$, $\boldsymbol{a}^2$, and $\boldsymbol{a}^3$. You can use the formula from [Lecture → Lemma 2.7.5.5] and the expression for $\mathbf{M}_K$ may also contain the area $|K|$ of K.

The concept of an element matrix has been introduced in [Lecture → Section 2.4.5.1]. Sloppily speaking, it boils down to the Galerkin matrix for a mesh consisting of a single triangle, see [Lecture → (2.4.5.6)].

HIDDEN HINT 1 for (2-5.b) → 2-5-2-0:s0hm.pdf

HIDDEN HINT 2 for (2-5.b) → 2-5-2-1:LFE11.pdf

SOLUTION for (2-5.b) → 2-5-2-2:s0.pdf ▲

(2-5.c) ⚄ (15 min.) [depends on Sub-problem (2-5.b)]

In the file `simplelinearfiniteelements.cc` implement the C++ function

```
Eigen::Matrix3d ElementMatrix_Mass_LFE(
const Eigen::Matrix<double, 2, 3>& triangle);
```

to compute the local element matrix associated to the bilinear form

$$\mathrm{m}(u,v) = \int_\Omega u\,v\,\mathrm{d}\boldsymbol{x}, \quad u,v \in L^2(\Omega)\,, \tag{2.5.8}$$

and linear Lagrangian finite elements on triangular elements. The parameter `triangle` passes a 2×3 matrix containing the coordinates of the vertices of the triangle in its column, *cf.* the type **TriGeo_t** used in [Lecture → Code 2.4.5.11] and [Lecture → Code 2.4.5.13].

SOLUTION for (2-5.c) → 2-5-3-0:LFE1.pdf ▲

(2-5.d) ⚄ (30 min.) Implement the function

```
template <typename FUNCTOR>
double L2Error(const TriaMesh2D &mesh, const Eigen::VectorXd &uFEM,
FUNCTOR &&exact);
```

to compute the error $\|u_h - u\|_{L^2(\Omega)}$, where u is the exact solution of (2.5.1), passed through the functor object `exact` of type `std::function<double(const Eigen::Vector2d &)>`, and $u_h \in \mathcal{S}_1^0(\mathcal{M})$ is the finite element Galerkin solution passed through its coefficient vector `uFEM` with respect to the customary basis of tent functions [Lecture → Section 2.4.3]. The argument `mesh` contains the mesh data structure, see the explanations concerning **TriaMesh2D** in [Lecture → § 2.4.1.1].

C++ code 2.5.11: Sub-problem (2-5.d): Mesh data structure ⇝ GitLab

```
2  using TriGeo_t = Eigen::Matrix<double, 2, 3>;
3  struct TriaMesh2D {
4    // Constructor:  reads mesh data from file
5    TriaMesh2D(std::string filename);
6    // Retrieve location of vertices of a triangular cell as rows
7    // of a fixed-size 3x2 matrix
8    TriGeo_t getVtCoords(int i) const;
9
10   void SaveMesh3D(std::string filename, const Eigen::VectorXd &z) const;
11
12   // Data members describing geometry and topolgy
13   Eigen::Matrix<double, Eigen::Dynamic, 2> _nodecoords;
14   Eigen::Matrix<int, Eigen::Dynamic, 3> _elements;
15 };
```

Mandatory mumerical quadrature

Of course, since u can be any function, the exact evaluation of $\|u - u_h\|_{L^2(\Omega)}$ is not possible in general. Thus, to compute the integrals, use the $\mathcal{M}$-based 2D local trapezoidal quadrature rule, as defined in [Lecture → (2.4.6.10)].

SOLUTION for (2-5.d) → 2-5-4-0:LFE2.pdf ▲

(2-5.e) ⚄ (30 min.) Implement the function

```
double H1Serror(const TriaMesh2D& mesh,
const Eigen::VectorXd& uFEM,
const std::function<Eigen::Vector2d(const Eigen::Vector2d&)>
   exact);
```

to compute the error norm $|u_h - u|_{H^1(\Omega)}$, where u is the exact solution of (2.5.1), whose *gradient* is passed in the function handle `exact` (that returns a column vector), and $u_h \in \mathcal{S}_1^0(\mathcal{M})$ is the finite element Galerkin solution, passed through the coefficient vector `uFEM`. The input argument `mesh` contains the mesh data structure, see also Sub-problem (2-5.d).

To compute the integrals, again rely on the 2D *local* trapezoidal quadrature rule, see [Lecture → (2.4.6.10)].

HIDDEN HINT 1 for (2-5.e) → 2-5-5-0:lferd:3h1.pdf

SOLUTION for (2-5.e) → 2-5-5-1:LFE3.pdf ▲

(2-5.f) ⚄ (35 min.) Implement a function

```
std::tuple<Eigen::VectorXd, double, double> Solve(
const SimpleLinearFiniteElements::TriaMesh2D &mesh);
```

that, given in input a mesh data structure `mesh`, computes the discrete solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ of (2.5.1) in the case that the exact solution is

$$u(\boldsymbol{x}) = \cos(2\pi x_1)\cos(2\pi x_2) \quad \Leftrightarrow \quad f(\boldsymbol{x}) = (1+4\pi^2)\cos(2\pi x_1)\cos(2\pi x_2)\,. \tag{2.5.15}$$

The function is to return the coefficient vector `U` of u_h, and the (approximate, due to quadrature) L^2-norm and H^1-seminorm of the discretization error $u - u_h$.

You can make use of the functions

```
Eigen::VectorXd assemLoad_LFE(
  const TriaMesh2D &Mesh,
  const LocalVectorHandle_t &getElementVector,
  const FHandle_t &FHandle);
```

and

```
Eigen::SparseMatrix<double> assembleGalMatLFE(
  const TriaMesh2D &Mesh,
  const LocalMatrixHandle_t &getElementMatrix);
```

given in the file `simplelinearfiniteelements.cc`. Those implement cell-oriented assembly according to the "distribute scheme", see [Lecture → Code 2.4.6.8] and [Lecture → Code 2.4.5.23].

The sparse linear system of equations resulting from the finite-element Galerkin discretization can be solved using EIGEN's sparse elimination solver [NumCSE course → Section 2.7.3], [NumCSE course → Code 2.7.3.1].

HIDDEN HINT 1 for (2-5.f) → 2-5-6-0:LFE41.pdf

HIDDEN HINT 2 for (2-5.f) → 2-5-6-1:LFE4hx.pdf

HIDDEN HINT 3 for (2-5.f) → 2-5-6-2:LFE41.pdf

SOLUTION for (2-5.f) → 2-5-6-3:LFE4.pdf ▲

End Problem 2-5 , 140 min.

Problem 2-6: Incidence matrices of a hybrid 2D mesh

This problem addresses an important concept related to the *topological properties* of a mesh, the **incidence matrices**. They are of fundamental importance in the mathematical theory of cellular complexes, but we will not be concerned with this here.
Rather this problem is an exercise in accessing topological information about a mesh in LEHRFEM++ and dealing with sparse matrices in EIGEN.

Implementation based on LEHRFEM++, background information is given in [Lecture → Section 2.7.2.2]. This problem is important as an introduction to the use of the LEHRFEM++ library, in particular concerning the data type **lf::mesh::Mesh** and **lf::mesh::Entity**.
▷ problem name/problem code folder: IncidenceMatrices

We are given a 2D hybrid mesh $\mathcal{M}$ that we identify with the set of its cells (entities of co-dimension 0). As in [Lecture → Section 2.5.1] we write $\mathcal{V}(\mathcal{M})$ for the set of nodes/vertices (entities of co-dimension 2), and $\mathcal{E}(\mathcal{M})$ for the set of edges (entities of co-dimension 1). Let N_k, $k = 0, 1, 2$, stand for the number of entities of co-dimension k:

$$N_0 = \sharp\mathcal{M} \quad , \quad N_1 = \sharp\mathcal{E}(\mathcal{M}) \quad , \quad N_2 = \sharp\mathcal{V}(\mathcal{M}) \, .$$

We assume the entities of co-dimension k are numbered consecutively from 1 to N_k. Look up [Lecture → § 2.5.1.2] for the meaning of "co-dimension" and that of other terms used in this paragraph.

In [Lecture → Section 2.7.2.1] we saw that in LEHRFEM++ a mesh is represented by objects of type **lf::mesh::Mesh** and that the member function `lf::mesh::Mesh::Index()` provides a consecutive numbering of all mesh entities of a specific co-dimension. We will indicate the number of a mesh entity by a subscript:

$$\begin{aligned} \mathcal{E}(\mathcal{M}) &= \{E_1, \ldots, E_{N_1}\} \, , \\ \mathcal{V}(\mathcal{M}) &= \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_{N_2}\} \, , \\ \mathcal{M} &= \{K_1, \ldots, K_{N_0}\} \, . \end{aligned}$$

The `SubEntities()` member function of **lf::mesh::Entity** returns an array of sub-entities and, thus, fixes their ordering [Lecture → § 2.7.2.14]. Therefore, with LEHRFEM++ data structures in mind, it makes sense to talk about, e.g., the first and the second endpoint/node of an edge.

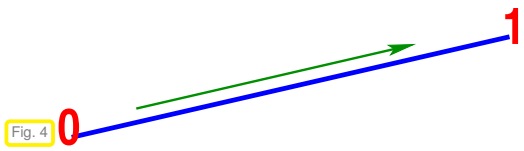
Definition 2.6.1. Edge-node incidence matrix

The entries of the **edge-node incidence matrix** $\mathbf{G} \in \{-1, 0, 1\}^{N_1, N_2}$ are defined as

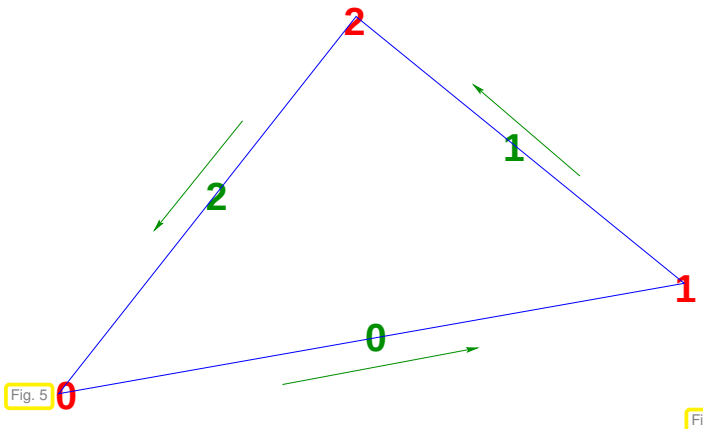
$$(\mathbf{G})_{i,j} := \begin{cases} 1 & \text{, if node } j \text{ is the first endpoint of edge } i \, , \\ -1 & \text{, if node } j \text{ is the second endpoint of edge } i \, , \\ 0 & \text{, if node } j \text{ is no endpoint of edge } i \, , \end{cases} \qquad \begin{aligned} &1 \le i \le N_1 \, , \\ &1 \le j \le N_2 \, . \end{aligned}$$

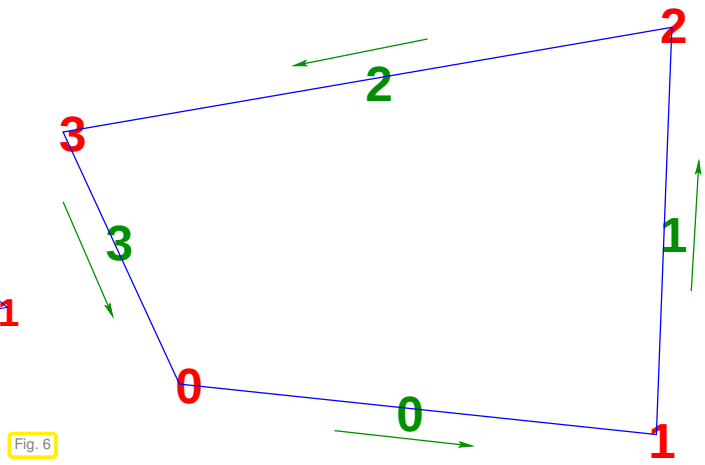
To define the second relevant incidence matrix, we first have to establish the **orientation** of edges, see also [Lecture → Rem. 2.7.2.15].

The **intrinsic orientation** of an edge $E \in \mathcal{E}(\mathcal{M})$ is given by the direction from its first endpoint to its second endpoint.
in LEHRFEM++ the ordering of endpoints is determined by the order of the elements of the array returned by a call to `lf::mesh::Entity::SubEntities(1)` for an EDGE object, see above.



Different from the intrinsic orientation is the **induced orientation** of an edge $E \in \mathcal{E}(\mathcal{M})$ relative to a cell $K \in \mathcal{M}$ to which it belongs: $E \subset \partial K$. This induced orientation is defined by the directions given by the green arrows in Fig. 5 and Fig. 6:





Keep in mind, that the induced orientation of an edge always depends on both the edge and an adjacent cell.

Definition 2.6.2. Cell-edge incidence matrix

The entries of the **cell-edge incidence matrix** $\mathbf{D} \in \{-1,0,1\}^{N_0,N_1}$ are defined as

$$(\mathbf{D})_{i,j} := \begin{cases} 1 & \text{if } E_j \subset \partial K_i \text{ and its induced orientation with respect to } K_i \text{ agrees with its intrinsic orientation,} \\ -1 & \text{if } E_j \subset \partial K_i \text{ and its induced orientation with respect to } K_i \text{ is opposite to its intrinsic orientation,} \\ 0 & \text{if edge } j \text{ is not part of the boundary of cell } i\,, \end{cases} \qquad \begin{array}{l} 1 \le i \le N_0\,, \\ 1 \le j \le N_1\,. \end{array}$$

(2-6.a) ⚀ (20 min.) What is the number of non-zero entries in each row of $\mathbf{G}$ and $\mathbf{D}$, respectively?

HIDDEN HINT 1 for (2-6.a) → `2-6-1-0:h1.pdf`

SOLUTION for (2-6.a) → `2-6-1-1:s1.pdf` ▲

(2-6.b) ⚁ (10 min.) [depends on Sub-problem (2-6.a)]

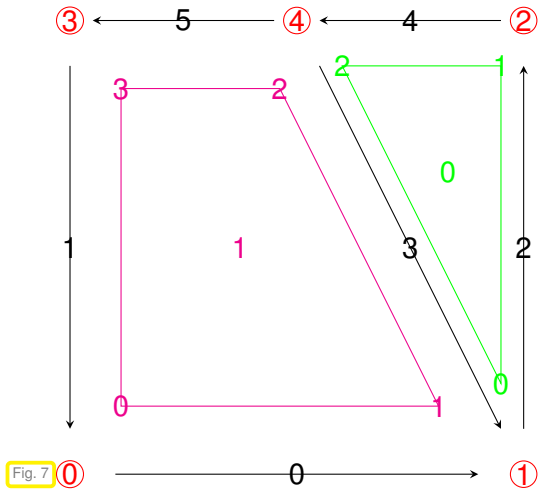
What is the topological meaning of the number of non-zero entries in the columns of $\mathbf{G}$ and $\mathbf{D}$, respectively?

SOLUTION for (2-6.b) → `2-6-2-0:s2.pdf` ▲

(2-6.c) ⚂ (15 min.)

Compute the incidence matrices $\mathbf{G}$ and $\mathbf{D}$ for the mesh depicted in Fig. 234 beside.

In red $\hat{=}$ numbers of nodes, in black $\hat{=}$ numbers of edges, orientations of edges indicated by arrows, local numbers of corners of cells in magenta. Cell indices are printed inside the cell.



SOLUTION for (2-6.c) → 2-6-3-0:s2a.pdf ▲

In the following sub-problems we rely on EIGEN's sparse matrix data type **Eigen::SparseMatrix<T>**, see EIGEN documentation and [NumCSE course → Section 2.7.2]. It will be useful to know that triplet initialization of a sparse matrix can be skipped, if one knows the (maximum) number of non-zero entries per row in advance. This information can be passed to EIGEN through the member function `Eigen::SparseMatrix<T>::reserve()` as in [NumCSE course → Code 2.7.2.1]. The following code illustrates its use.

C++11 code 2.6.3: `reserve()`-based initializatin of a sparse matrix in EIGEN ⇝ GitLab

```
2  const int n = 20, m = 10;
3  // Set up zero sparse matrix with row major storage format
4  // This format is essential for being able to set the maximal
5  // number of non-zero entries per row.
6  Eigen::SparseMatrix<int, Eigen::RowMajor> X(n, m);
7  // Reserve space for at most nnz_row non-zero entries per row
8  const std::size_t nnz_row = 3;
9  X.reserve(Eigen::VectorXi::Constant(n, nnz_row));
10 // Initialize nnz_row entries per row
11 for (int row_idx = 0; row_idx < n; ++row_idx) {
12   for (int k = 0; k < nnz_row; ++k) {
13     X.coeffRef(row_idx, (row_idx * k) % m) += 1;
14   }
15 }
```

(2-6.d) ⚄ (30 min.) In the file `incidencematrices.cc` write a C++ function

```
Eigen::SparseMatrix<int> computeEdgeVertexIncidenceMatrix(
    const lf::mesh::Mesh &mesh);
```

which computes the sparse matrix $\mathbf{G}$ for the mesh passed in `mesh` efficiently.

The `Index()` method of **lf::mesh::Mesh** provides the numbering of entities which underlies indexing of the entries of the incidence matrices.

SOLUTION for (2-6.d) → 2-6-4-0:s3.pdf ▲

(2-6.e) ⚄ (30 min.) Now implement a C++ function

```
Eigen::SparseMatrix<int> computeCellEdgeIncidenceMatrix(
    const lf::mesh::Mesh &mesh);
```

which returns the cell-edge incidence matrix $\mathbf{D}$ for the mesh encoded in `mesh`. Again rely on the

numbering of mesh entities supplied by the `Index()` method of **lf::mesh::Mesh**.

HIDDEN HINT 1 for (2-6.e) → 2-6-5-0:h14.pdf

SOLUTION for (2-6.e) → 2-6-5-1:s4.pdf ▲

(2-6.f) ⚄ (30 min.) Prove the following fundamental theorem ($\mathbf{O}$ is the zero matrix):

Theorem 2.6.6. Co-chain complex property

For any mesh holds

$$\mathbf{DG} = \mathbf{O} \,, \tag{2.6.7}$$

with $\mathbf{G}$/$\mathbf{D}$ as defined in Def. 2.6.1 and Def. 2.6.2.

HIDDEN HINT 1 for (2-6.f) → 2-6-6-0:h5.pdf

SOLUTION for (2-6.f) → 2-6-6-1:s5.pdf ▲

(2-6.g) ⚄ (20 min.) [depends on Sub-problem (2-6.e), Sub-problem (2-6.d)]

Inspired by the statement of Cor. 13.1.6 to implement in LEHRFEM++ a testing function.

```
bool testZeroIncidenceMatrixProduct(const lf::mesh::Mesh &mesh);
```

that returns **true**, whenever the two incidence matrices of the 2D hybrid mesh satisfy the relationship asserted in Cor. 13.1.6.

SOLUTION for (2-6.g) → 2-6-7-0:s5.pdf ▲

End Problem 2-6 , 155 min.

Problem 2-7: Computing the length of the boundary in LEHRFEM++

Given a **lf:mesh::Mesh** object in LEHRFEM++ describing a planar hybrid mesh with straight edges, how can one compute the length of the boundary? The challenge is to identify the edges lying on the boundary of the meshed domain. In this problem you will learn how to do this in LEHRFEM++.

This problem is meant to practice the use of tools provided by the geometry layer of meshes in LEHRFEM++, see [Lecture → Section 2.7.2.3].
▷ problem name/problem code folder: LengthOfBoundary

[Lecture → Code 2.7.1.12] demonstrated how to read mesh information from a `.msh`-file produced by GMSH. This code (↪ GITHUB) will be the starting point for code development in this problem. The task is to write a LEHRFEM++-based program that

(I) reads in a 2d hybrid mesh from a `.msh`-file using the LEHRFEM++'s **lf::io::GmshReader**,

(II) computes the area of the meshed domain Ω,

(III) and determines the length of the boundary $\partial\Omega$.

(2-7.a) ⚀ (20 min.) In the file `boundarylength.cc` write a LEHRFEM++-based C++ function

```
double volumeOfDomain(const std::shared_ptr<lf::mesh::Mesh>
  mesh_p);
```

that returns the area of the domain covered by a 2D hybrid mesh passed through the `mesh_p` pointer argument. You may use the function `lf::geometry::Volume()` ↪ GITHUB.

SOLUTION for (2-7.a) → 2-7-1-0:s1.pdf ▲

(2-7.b) ⚁ (20 min.) Study the description and the implementation of the LEHRFEM++ function (↪ GITHUB)

```
CodimMeshDataSet<bool>
lf::mesh::utils::flagEntitiesOnBoundary(
  const std::shared_ptr<const lf::mesh::Mesh> &mesh_p,
  lf::base::dim_t codim);
```

and explain how edges on the boundary are identified in this function. Explanations pertinent to the **MeshDataSet** concept of LEHRFEM++ can be found in [Lecture → Suppl. 2.7.2.8] and also in the documentation of **lf::mesh::utils::CodimMeshDataSet**.

C++ code 2.7.2: (Skeleton) Implementation of the LEHRFEM++ function flagEntitiesOnBoundary()

```
1  CodimMeshDataSet<bool> flagEntitiesOnBoundary(
2      const std::shared_ptr<const Mesh>& mesh_p, lf::base::dim_t codim) {
3    // count cells adjacent to entities of co-dimension 1
4    CodimMeshDataSet<lf::base::size_type> no_adjacent_cells{
5        CountNumSuperEntities(mesh_p, 1, 1)};
6    // flag array
7    CodimMeshDataSet<bool> bd_flags{mesh_p, codim, false};
8    // relative codimension with respect to faces (entities of
       co-dimension 1)
9    const lf::base::dim_t rel_codim = codim - 1;
10   // Run through faces and flag sub-entities
11   for (const lf::mesh::Entity *edge_p : mesh_p->Entities(1)) {
12     if (no_adjacent_cells(*edge_p) == 1) {
13       // Boundary face detected!
```

```
14        // Traverse all sub-entities of a specific relative co-dimension
15        for (const lf::mesh::Entity* subent : edge->SubEntities(rel_codim)) {
16          bd_flags(*subent) = true;
17        }
18      }
19    }
20    return bd_flags;
21  }
```

SOLUTION for (2-7.b) → 2-7-2-0:s2.pdf ▲

(2-7.c) ⚄ (20 min.) [depends on Sub-problem (2-7.b)]

Implement a function (in the file `boundarylength.cc`)

```
double lengthOfBoundary(
  const std::shared_ptr<lf::mesh::Mesh> mesh_p)
```

that calculates the length of the boundary of the domain covered by the mesh passed through the `mesh_p` pointer.

HIDDEN HINT 1 for (2-7.c) → 2-7-3-0:h2a.pdf

SOLUTION for (2-7.c) → 2-7-3-1:s2a.pdf ▲

(2-7.d) ⚄ (20 min.) [depends on Sub-problem (2-7.b)]

Based on LEHRFEM++ implement an LEHRFEM++-based C++ function

```
std::pair<double, double> measureDomain(std::string filename);
```

that reads a 2D (hybrid) mesh from the `.msh`-file [Lecture → Ex. 2.7.1.5] with the name `filename` and returns both the volume and the length of the boundary of the domain covered by the mesh.

Use this function to enhance the `main()` routine in the file `boundarylength_main.cc` so that the resulting executable solves the tasks described in the introduction to this problem: It should read the name of the `.msh`-file from the command line and print the area of Ω and the length of $\partial\Omega$.

HIDDEN HINT 1 for (2-7.d) → 2-7-4-0:h31.pdf

SOLUTION for (2-7.d) → 2-7-4-1:s4.pdf ▲

End Problem 2-7, 80 min.

Problem 2-8: Introduction to local assembly in LEHRFEM++

In this algorithmic problem we study assembly in LEHRFEM++ for (bi-)linear Lagrangian finite elements on hybrid meshes with triangular and rectangular cells. The focus is twofold, on

(i) quadrature-based local computations yielding element matrices and vectors,
(ii) the use of LEHRFEM++'s assembly facilties from the module **lf::assemble**.

This problem relies on LEHRFEM++, mainly the capabilities covered in [Lecture → Section 2.7.2] and and heavily harks back to [Lecture → Section 2.4], in particular [Lecture → Section 2.4.5] and [Lecture → Section 2.4.6].
▷ problem name/problem code folder: ElementMatrixComputation

In this problem we rely on piecewise linear Lagrangian finite element spaces $\mathcal{S}_1^0(\mathcal{M})$ on hybrid meshes throughout, see [Lecture → Section 2.4.2] for triangular meshes and [Lecture → Ex. 2.6.2.1] for tensor-product meshes. Also consult [Lecture → Rem. 2.6.2.12].

LEHRFEM++ can deal with hybrid meshes comprising general quadrilaterals. In [Lecture → Section 2.8.2] we will learn how to extend linear Lagrangian finite elements to this setting. For this problem we just pretend that all hybrid meshes used merely contain rectangles, as drawn, for instance, in [Lecture → Fig. 120].

(2-8.a) ⊡ (15 min.) LEHRFEM++ usually uses EIGEN's sparse elimination solvers (↬ documentation) to solve the linear systems of equations arising from finite element Galerkin discretizations. The standard algorithm is implemented in **Eigen::SparseLU**, see ↬ documentation.

Refresh your knowledge of how to use sparse solvers from the `SparseLU` module of EIGEN.

SOLUTION for (2-8.a) → 2-8-1-0:s0.pdf ▲

(2-8.b) ⊡ (15 min.) Study the local assembly functions `AssembleMatrixLocally()` and `AssembleVectorLocally()` of LEHRFEM++, presented in [Lecture → Ex. 2.7.4.22] and [Lecture → Ex. 2.7.4.27]:

- ```
  template <typename TMPMATRIX, class ENTITY_MATRIX_PROVIDER>
  void AssembleMatrixLocally(dim_t codim,
    const DofHandler &dof_handler_trial,
    const DofHandler &dof_handler_test,
    ENTITY_MATRIX_PROVIDER &entity_matrix_provider,
    TMPMATRIX &matrix);

  template <typename TMPMATRIX, class ENTITY_MATRIX_PROVIDER>
  TMPMATRIX AssembleMatrixLocally(
      dim_t codim, const DofHandler &dof_handler_trial,
      const DofHandler &dof_handler_test,
      ENTITY_MATRIX_PROVIDER &entity_matrix_provider);
  ```

- ```
  template <typename TMPMATRIX, class ENTITY_MATRIX_PROVIDER>
  TMPMATRIX AssembleMatrixLocally(
      dim_t codim, const DofHandler &dof_handler,
      ENTITY_MATRIX_PROVIDER &entity_matrix_provider)
  ```

and

- ```
  template <typename VECTOR, class ENTITY_VECTOR_PROVIDER>
  ```
  ```

```
void AssembleVectorLocally(dim_t codim,
  const DofHandler &dof_handler,
  ENTITY_VECTOR_PROVIDER &entity_vector_provider,
  VECTOR &resultvector)

template <typename VECTOR, class ENTITY_VECTOR_PROVIDER>
VECTOR AssembleVectorLocally(dim_t codim, const DofHandler
   &dof_handler,
                                ENTITY_VECTOR_PROVIDER
                                   &entity_vector_provider);
```

Inform yourself about the type concepts of **ENTITY_MATRIX_PROVIDER** ⇝ (documentation) and **ENTITY_VECTOR_PROVIDER** ⇝ (documentation) used in LehrFEM++. ▲

(2-8.c) ⚂ (15 min.) [depends on Sub-problem (2-8.a)]

Supplement the few missing lines in the implementation of the function

```
template <class ELMAT_PROVIDER, class ELVEC_PROVIDER>
Eigen::VectorXd solve(ELMAT_PROVIDER& elmat_builder,
                      ELVEC_PROVIDER& elvec_builder);
```

in the file `solve.h` to enable it to compute the basis expansion coefficient vector when provided with **PROVIDER** objects for element matrices and vectors. Use Eigen's sparse solver facilities to that end.

Solution for (2-8.c) → `2-8-3-0:s1.pdf` ▲

(2-8.d) ⚂ (15 min.) [depends on Sub-problem (2-8.c)]

In the file `solve.cc` complete the function `solvePoissonBVP()` that is to call `solve()` in order to compute the basis expansion coefficients of a solution of the Neumann boundary value problem

$$-\Delta u = f \quad \text{in} \quad \Omega \quad , \quad \mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega \, , \tag{2.8.2}$$

on the unit square $\Omega =]0,1[^2$.

Also make use of LehrFEM++'s local **PROVIDER** types for element matrices for the bilinear form ("weak $-\Delta$")

$$(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \, \mathrm{d}\boldsymbol{x} \, , \quad u,v \in H^1(\Omega) \, ,$$

and element vectors for the linear form

$$v \mapsto \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \, , \quad f,v \in L^2(\Omega) \, ,$$

respectively,

- **lf::uscalfe::LinearFELaplaceElementMatrix**,
- **lf::uscalfe::LinearFELocalLoadVector**.

Initialize the **lf::uscalfe::LinearFELocalLoadVector** appropriately so that the right-hand-side source function is provided by the function **double** `f(Eigen::Vector2d x)`.

Hidden hint 1 for (2-8.d) → `2-8-4-0:28dh1.pdf`

Solution for (2-8.d) → `2-8-4-1:s2.pdf` ▲

(2-8.e) ⚄ (15 min.)

The `main()` function in the file `elementmatrixcomputation_main.cc` calls `solvePoissonBVP()`. Run the executable. What do you observe and why.

HIDDEN HINT 1 for (2-8.e) → `2-8-5-0:h2a1.pdf`

SOLUTION for (2-8.e) → `2-8-5-1:s2a.pdf` ▲

(2-8.f) ⚂ (10 min.) [depends on Sub-problem (2-8.e)]

The problem haunting a finite element Galerkin discretization of (2.8.2) does not arise for the rather similar boundary value problem

$$-\Delta u + u = f \quad \text{in} \quad \Omega \quad , \quad \mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega \, , \tag{2.8.7}$$

also posed on the unit square $\Omega =]0,1[^2$. Explain why based on the weak (variational) formulation that you should derive first.

SOLUTION for (2-8.f) → `2-8-6-0:2bs.pdf` ▲

(2-8.g) ⚂ (10 min.) For a triangle K deduce the formula for the 3×3 element matrix $\mathbf{M}_K$ related to the bilinear form

$$\mathrm{m}(u,v) := \int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \, , \quad u,v \in L^2(\Omega) \, , \tag{2.8.9}$$

and the finite element space $\mathcal{S}_1^0(\mathcal{M})$. This element matrix is also called the **element mass matrix**. Only the area $|K|$ of K will enter your formula.

HIDDEN HINT 1 for (2-8.g) → `2-8-7-0:LFE11.pdf`

SOLUTION for (2-8.g) → `2-8-7-1:s2c.pdf` ▲

(2-8.h) ⚂ (15 min.) [depends on Sub-problem (2-8.g)]

For the finite element space $\mathcal{S}_1^0(\mathcal{M})$ of piecewise (bi-)linear Lagrangian finite elements, derive the 4×4 **element mass matrix** $\mathbf{M}_K$ for a *rectangular cell* K. As in Sub-problem (2-8.g) it will be a constant matrix multiplied with a multiple of the area $|K|$ of K.

SOLUTION for (2-8.h) → `2-8-8-0:s2d.pdf` ▲

(2-8.i) ⚂ (15 min.) [depends on Sub-problem (2-8.g) and Sub-problem (2-8.h)]

Based on LEHRFEM++'s **lf::uscalfe::LinearFELaplaceElementMatrix** implement a class **ElementMatrixComputation::MyLinearFEElementMatrix** compatible with the concept **ENTITY_MATRIX_PROVIDER** whose `Eval()` member function returns the element matrix for the finite element space $\mathcal{S}_1^0(\mathcal{M})$ and the bilinear form occurring in the weak formulation of (2.8.7).

HIDDEN HINT 1 for (2-8.i) → `2-8-9-0:h2f.pdf`

SOLUTION for (2-8.i) → `2-8-9-1:s2f.pdf` ▲

The source function $f \in C^0(\overline{\Omega})$ is given in procedural form only [Lecture → Rem. 2.1.2.5] which entails the use of local numerical quadrature for the computation of element vectors, *cf.* [Lecture → Section 2.4.6]. For this problem we opt for the 2D **composite edge-midpoint rule** on hybrid meshes: To define it we rely on the numbering of edges and vertices of triangular and quadrilateral cells as given in [Lecture → Fig. 131] and [Lecture → Fig. 132] and write $\boldsymbol{m}_\ell$, $\ell = 1,\ldots,\sharp\mathcal{V}(K)$ for the midpoint of the ℓ-ht edge of the cell K ($\sharp\mathcal{V}(K)$ is the number of vertices of K, mathematical indexing is used). Then, for $\varphi \in C^0(\overline{K})$,

$$\int_K \varphi\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{\sharp\mathcal{V}(K)} \sum_{\ell=1}^{\sharp\mathcal{V}(K)} \varphi(\boldsymbol{m}_\ell) \, . \tag{2.8.14}$$

(2-8.j) ⚄ (15 min.) Complete the implementation of the `Eval()` member function of the class **MyLinearLoadVector** which supplies a builder type for the element vector arising from the linear form $v \mapsto \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$. Use the composite edge midpoint rule with the local definition given in (2.8.14).

SOLUTION for (2-8.j) → `2-8-10-0:s3.pdf` ▲

End Problem 2-8 , 140 min.

Problem 2-9: Handling degrees of freedom (DOFs) in LEHRFEM++

[Lecture → § 2.7.4.13] presents the class **lf::assemble::DofHandler** and its functionality. In this problem we practice using the capabilities of this class and we will learn how to use it to index and manage the global shape functions of Lagrangian finite element spaces.

Takes for granted familiarity with [Lecture → Section 2.6.1] and LEHRFEM++ as far as covered in [Lecture → Section 2.7.2] and [Lecture → Section 2.7.4.2].
▷ problem name/problem code folder: LFPPDofHandling

Assume that $\mathcal{M}$ is a *triangular* mesh of a polygonally bounded domain $\Omega \subset \mathbb{R}^2$, see [Lecture → Section 2.4.1]. From [Lecture → Section 2.4.3] and [Lecture → Ex. 2.6.1.2] recall the definition of the "nodal" basis functions (= global shape functions) for the Lagrangian finite element spaces $\mathcal{S}_1^0(\mathcal{M})$ and $\mathcal{S}_2^0(\mathcal{M})$. Those have been introduced as cardinal basis functions with respect to suitable sets of interpolation nodes. Also remember the definition of the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$ and $\mathcal{S}_2^0(\mathcal{M})$. For the latter space they are given in [Lecture → (2.6.1.6)].

(2-9.a) ⚀ (10 min.) Explain the relationship and the difference of the concepts of

- **local shape functions**
- **global shape functions**

as they are commonly used in the context of finite-element theory and algorithms.

SOLUTION for (2-9.a) → 2-9-1-0:s0.pdf ▲

(2-9.b) ⚁ (20 min.) Express all the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$ (the barycentric coordinate functions) as linear combinations of the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$. Use the local numbering convention of LEHRFEM++ explained in [Lecture → § 2.7.4.11].

SOLUTION for (2-9.b) → 2-9-2-0:s1.pdf ▲

(2-9.c) ⚂ (15 min.)

For a single triangle $K \in \mathcal{M}$ the figure beside shows in blue the indices of global shape functions associated with nodes, in magenta the indices of those global shape functions that belong to edges.

The local numbers of the vertices of the triangles are given as ①, ②, ③.

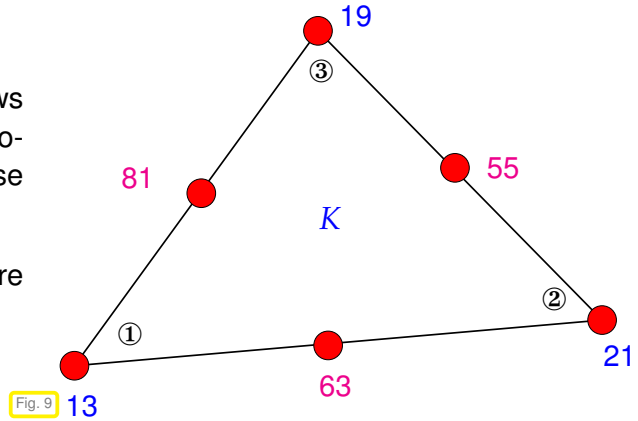


Fig. 9

Describe the output of all member functions of **lf::assemble::DofHandler** (as listed in [Lecture → § 2.7.4.13]) that take an **lf::mesh::Entity** object as an argument, when that object refers to the above triangle K.

SOLUTION for (2-9.c) → 2-9-3-0:1as.pdf ▲

The `main()`-function in `lfppdofhandling_main.cc` constructs an object of type **lf::mesh::Mesh** containing topological and geometric information on a particular triangular mesh. See [Lecture → Section 2.7.2] for more information about **lf::mesh::Mesh**. All the functions that you will be asked to implement below are called from that `main()` function.

Note that all the functions you realize should have optimal asymptotic complexity $O(\sharp\mathcal{M})$ for $\sharp\mathcal{M} \to \infty$. Throughout implement appropriate error handling in case a quadrilateral cell is encountered.

(2-9.d) ⚀ (15 min.) In the file `lfppdofhandling.cc` complete the C++ function

```
std::array<std::size_t,3> countEntityDofs(const
   lf::assemble::DofHandler &dofhandler);
```

that returns the number of global shape functions (managed by the local→global index mapping encoded in `dofhandler`) *associated with* mesh entities of co-dimension 0, 1, and 2 respectively (the co-dimension also serves as index for the returned array).

SOLUTION for (2-9.d) → `2-9-4-0:s2.pdf` ▲

(2-9.e) ⚁ (20 min.) In the file `lfppdofhandling.cc` complete the C++ function

```
std::size_t countBoundaryDofs(const lf::assemble::DofHandler
   &dofhandler);
```

that tells the number of global shape functions (as encoded in `dofhandler`) associated with mesh entities located *on the boundary* $\partial\Omega$.

Use the LEHRFEM++ function (➙ GITHUB)

```
lf::mesh::utils::AllCodimMeshDataSet<bool>
  lf::mesh::utils::flagEntitiesOnBoundary(
    const std::shared_ptr<const lf::mesh::Mesh> &mesh_p);
```

to obtain an array of flags telling whether an entity is located on the boundary.

SOLUTION for (2-9.e) → `2-9-5-0:s3.pdf` ▲

(2-9.f) ⚂ (15 min.) In the file `lfppdofhandling.cc` complete the C++ function

```
double integrateLinearFEFunction(
    const lf::assemble::DofHandler &dofhandler,
    const Eigen::VectorXd &mu);
```

that computes $\int_\Omega u_h \,\mathrm{d}\boldsymbol{x}$ for $u_h \in \mathcal{S}_1^0(\mathcal{M})$, whose nodal basis expansion coefficients are passed in `mu`. The `dofhandler` arguments provides the local→global index mapping. Check whether the **lf::assemble::DofHandler** object `dofhandler` really fits the Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$.

HIDDEN HINT 1 for (2-9.f) → `2-9-6-0:h41.pdf`

SOLUTION for (2-9.f) → `2-9-6-1:s4.pdf` ▲

(2-9.g) ⚁ (20 min.) In the file `lfppdofhandling.cc` write a C++ function

```
double integrateQuadraticFEFunction(
  const lf::assemble::DofHandler &dofhandler,
  const Eigen::VectorXd &mu);
```

that computes $\int_\Omega u_h \,\mathrm{d}\boldsymbol{x}$ for $u_h \in \mathcal{S}_2^0(\mathcal{M})$, whose nodal basis expansion coefficients are passed in `mu`. The `dofhandler` arguments provides the local→global index mapping for the quadratic Lagrangian finite element space. Check whether the **lf::assemble::DofHandler** object `dofhandler` really fits the Lagrangian finite element space $\mathcal{S}_2^0(\mathcal{M})$.

Use the formula given in

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex[a] K with barycentric coordinate functions $\lambda_1,\ldots,\lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1,\ldots,d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \,\mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1!\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1+\alpha_2+\cdots+\alpha_{d+1}+d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \,.$$

[Lecture → (2.7.5.6)]

[a] A d-simplex is the convex hull of $d+1$ points in $\mathbb{R}^d$, which do not all lie in a single plane.

to compute the integrals of the local shape functions

$$\begin{aligned} b_K^1 &= (2\lambda_1 - 1)\lambda_1 \,, & b_K^4 &= 4\lambda_1\lambda_2 \,, \\ b_K^2 &= (2\lambda_2 - 1)\lambda_2 \,, & b_K^5 &= 4\lambda_2\lambda_3 \,, \\ b_K^3 &= (2\lambda_3 - 1)\lambda_3 \,, & b_K^6 &= 4\lambda_1\lambda_3 \,. \end{aligned}$$

[Lecture → (2.6.1.6)]

over a single triangular cell.

SOLUTION for (2-9.g) → 2-9-7-0:.pdf ▲

(2-9.h) ⚄ (25 min.) [depends on Sub-problem (2-9.b)]

By definition $\mathcal{S}_1^0(\mathcal{M}) \subset \mathcal{S}_2^0(\mathcal{M})$. Hence, every $u_h \in \mathcal{S}_1^0(\mathcal{M})$, which may be given through its basis expansion coefficient vector $\vec{\mu} \in \mathbb{R}^{\sharp\mathcal{V}(\mathcal{M})}$ with respect to the tent function basis, can be written as a linear combination of nodal basis functions of $\mathcal{S}_2^0(\mathcal{M})$ with an associated coefficient vector $\vec{\zeta} \in \mathbb{R}^{\sharp\mathcal{V}(\mathcal{M})+\sharp\mathcal{E}(\mathcal{M})}$.

In the file `lfppdofhandling.cc` complete the C++ function

```
Eigen::VectorXd convertDOFsLinearQuadratic(
    const lf::assemble::DofHandler &dofh_Linear_FE,
    const lf::assemble::DofHandler &dofh_Quadratic_FE,
    const Eigen::VectorXd &mu);
```

that computes $\vec{\zeta}$ from $\vec{\mu}$ passed in `mu`. The two **lf::assemble::DofHandler** objects belong to the finite element spaces $\mathcal{S}_1^0(\mathcal{M})$ and $\mathcal{S}_2^0(\mathcal{M})$, respecively (This should be checked in the code).

HIDDEN HINT 1 for (2-9.h) → 2-9-8-0:h61.pdf

SOLUTION for (2-9.h) → 2-9-8-1:s5.pdf ▲

End Problem 2-9 , 140 min.

Problem 2-10: Projection onto Gradients

This problem discusses the computation of what is known in mathematics as the Helmholtz decomposition of a vectorfield, addressed in [PR05, Example 10.2] as "Recostructing a function from its gradient". Lofty terms aside, all this boils down to solving Poisson equations [Lecture → Rem. 1.5.3.6] numerically by means of finite elements. Thus this problem is about a non-standard application of a linear finite element Ritz-Galerkin method [Lecture → Rem. 2.2.1.3] for the solution of a particular quadratic minimization problem.

This problem asks for implementation in LEHRFEM++ related to [Lecture → Section 2.7.4]. The contents of [Lecture → Section 1.4] will also be essential.
▷ problem name/problem code folder: ProjectionOntoGradients

Let $\Omega \subset \mathbb{R}^2$ be a polygonal domain. We are given a continuous vector field $\mathbf{f} : \overline{\Omega} \to \mathbb{R}^2$, available in *procedural form* [Lecture → Rem. 2.1.2.5]. The goal is to solve the following *quadratic minimization problem* [Lecture → Def. 1.2.3.11]

$$u = \operatorname*{argmin}_{v \in H^1_0(\Omega)} J(v) \,, \quad J(v) := \int_\Omega \|\mathbf{f}(\boldsymbol{x}) - \mathbf{grad}\, v(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \,. \tag{2.10.1}$$

If u solves (2.10.1), then

$$\mathbf{f} = \mathbf{f}_0 + \mathbf{f}_\perp \quad \text{with} \quad \mathbf{f}_0 := \mathbf{grad}\, u \,, \quad \mathbf{f}_\perp := \mathbf{f} - \mathbf{grad}\, u \,, \tag{2.10.2}$$

is called a **Helmholtz decomposition** of $\mathbf{f}$. In this problem we intend to compute the terms of the Helmholtz decomposition approximately by means of a finite element method.

(2-10.a) ⚄ (5 min.) State the linear variational problem that is equivalent to (2.10.1). Do not forget to specify both the trial and the test space.

SOLUTION for (2-10.a) → 2-10-1-0:s3.pdf ▲

(2-10.b) ⚄ (10 min.) [depends on Sub-problem (2-10.a)]

Discuss existence and uniqueness of solutions of the linear variational problem found in Sub-problem (2-10.a)

HIDDEN HINT 1 for (2-10.b) → 2-10-2-0:h1a.pdf

SOLUTION for (2-10.b) → 2-10-2-1:s1a.pdf ▲

(2-10.c) ⚄ (5 min.) [depends on Sub-problem (2-10.a)]

Show that the components $\mathbf{f}_0$ and $\mathbf{f}_\perp$ of the Helmholtz decomposition (2.10.2) are **$L^2(\Omega)$-orthogonal**, that is,

$$\int_\Omega \mathbf{f}_0(\boldsymbol{x}) \cdot \mathbf{f}_\perp(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0 \,. \tag{2.10.6}$$

SOLUTION for (2-10.c) → 2-10-3-0:s3.pdf ▲

(2-10.d) ⚄ (10 min.) [depends on Sub-problem (2-10.a)]

Assume that $\mathbf{f} \in (C^1(\overline{\Omega}))^2$. Show that (2.10.1) has the trivial solution $u = 0$, *if and only if* $\operatorname{div} \mathbf{f} = 0$.

HIDDEN HINT 1 for (2-10.d) → 2-10-4-0:h3.pdf

SOLUTION for (2-10.d) → 2-10-4-1:s3.pdf ▲

Let a triangular mesh $\mathcal{M}$ of Ω be given. In order to solve the variational problem from Sub-problem (2-10.a) we rely on linear Lagrangian finite elements $\mathcal{S}^0_{1,0}(\mathcal{M}) \subset H^1_0(\Omega)$. For all occurring integrals, we use the 2D **composite midpoint rule** that employs the local approximation

$$\int_K \varphi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx |K|\varphi(\boldsymbol{c}_K)\;, \tag{2.10.7}$$

where $\boldsymbol{c}_K := \frac{1}{3}(\boldsymbol{a}^1 + \boldsymbol{a}^2 + \boldsymbol{a}^3)$ is the center of gravity of the triangle K having vertices $\boldsymbol{a}^i$, $i = 1,2,3$.

(2-10.e) ⚀ (15 min.) [depends on Sub-problem (2-10.a)]

Implement a class **ElementMatrixProvider** compatible with the concept of **ENTITY_MATRIX_PROVIDER** as discussed in connection with [Lecture → Code 2.7.4.23] that computes the element matrices for the left-hand side bilinear form of the linear variational problem found in Sub-problem (2-10.a) based on linear Lagrangian finite elements on flat triangular cells.

HIDDEN HINT 1 for (2-10.e) → 2-10-5-0:h44.pdf

HIDDEN HINT 2 for (2-10.e) → 2-10-5-1:h4II.pdf

SOLUTION for (2-10.e) → 2-10-5-2:s4.pdf ▲

(2-10.f) ⚁ (10 min.) [depends on Sub-problem (2-10.a)]

From [Lecture → Code 2.4.5.11] we learned that it is easy to obtain a matrix $\mathbf{G} \in \mathbb{R}^{2,3}$, whose columns contain the gradients of the barycentric coordinate functions λ_ℓ, $\ell = 1,2,3$, for a triangle K.

Based on $\mathbf{G}$, find a formula for the entries of the element vector [Lecture → Def. 2.7.4.5] with respect to the finite element space $\mathcal{S}^0_1(\mathcal{M})$ for a general triangular cell K with vertices $\boldsymbol{a}^i_K$, $i = 1,2,3$. Employ quadrature rule (2.10.7). The vector field $\mathbf{f}$ and the area $|K|$ of the triangle may also be used in the expressions.

SOLUTION for (2-10.f) → 2-10-6-0:s4a.pdf ▲

(2-10.g) ⚂ (20 min.) [depends on Sub-problem (2-10.f)]

Realize a class **GradProjRhsProvider** according to the concept **ENTITY_VECTOR_PROVIDER** as introduced in [Lecture → Code 2.7.4.28].

```
template <typename FUNCTOR>
class GradProjRhsProvider {
public:
// Constructor:  initialization of functor data member
  explicit GradProjRhsProvider(FUNCTOR f) : f_(f) {}

Eigen::Vector3d Eval(const lf::mesh::Entity &entity);
bool isActive(const lf::mesh::Entity & /*entity*/) const { return
   true; }

ivate:
FUNCTOR f_;
```

Here **FUNCTOR** must provide an evaluation operator that returns an object that behaves like an **Eigen::Vector2d**.

SOLUTION for (2-10.g) → 2-10-7-0:s5.pdf ▲

(2-10.h) ⚃ (30 min.) [depends on Sub-problem (2-10.e) and Sub-problem (2-10.g)]

Taking the cue from [Lecture → Code 2.7.6.20], in the file `projectionontogradients.h` implement a C++ function

```
template <typename FUNCTOR>
  Eigen::VectorXd projectOntoGradients(
    const lf::assemble::DofHandler &dofh,
    FUNCTOR f);
```

which computes the vector of nodal basis expansion coefficients of the finite element Galerkin approximation $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ of the solution of (2.10.1).

The argument `dofh` passes a LEHRFEM++ **DofHandler** object, for details refer to [Lecture → § 2.7.4.13]. Through this object the mesh can be accessed as well. The other argument `f` contains a functor object providing $\mathbf{f}$ in procedural form.

Of course you may rely on the codes of Sub-problem (2-10.e) and Sub-problem (2-10.g) for carrying out the local computations.

HIDDEN HINT 1 for (2-10.h) → 2-10-8-0:h61.pdf

SOLUTION for (2-10.h) → 2-10-8-1:s6.pdf ▲

(2-10.i) ⚂ (15 min.) [depends on Sub-problem (2-10.d)]

As we saw in Sub-problem (2-10.d), for the *divergence-free* vector field

$$\mathbf{f}(\boldsymbol{x}) = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \quad \operatorname{div} \mathbf{f} \equiv 0\,, \tag{2.10.13}$$

we can expect $u_h \equiv 0$, where $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ is the finite element Ritz-Galerkin solution of (2.10.1).

Using the facilities of GOOGLETEST implement a **unit test**

```
TEST(ProjectionOntoGradients, div_free_test) { ... }
```

that uses $\mathbf{f}$ from (2.10.13) to test the correct implementation of the function `projectOntoGradients()` from Sub-problem (2-10.h). For testing please use the triangular test mesh of $\Omega =]0,3[^2$ provided by the LEHRFEM++ function call

```
auto mesh_p = lf::mesh::utils::GenerateHybrid2DTestMesh(3);
```

SOLUTION for (2-10.i) → 2-10-9-0:s7.pdf ▲

(2-10.j) ⚂ (3 min.) For which vector fields $\mathbf{f}$ can we expect $\mathbf{f} = \mathbf{grad}\, u_h$, where $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ is the finite element Ritz-Galerkin solution of (2.10.1)?

SOLUTION for (2-10.j) → 2-10-10-0:s8.pdf ▲

(2-10.k) ⚄ (60 min.) [depends on Sub-problem (2-10.j), Sub-problem (2-10.h)]

LEHRFEM++'s testing facilities implement a class **TPTriagMeshBuilder** that can be used to generate a "triangular tensor-product mesh", as demonstrated in the following code snippet:

C++11 code 2.10.15: Generation of tensor-product triangular mesh

```
1 // Construct a structured mesh with 8 triangles
2 hybrid2d::TPTriagMeshBuilder builder(
3     std::make_shared<hybrid2d::MeshFactory>(2));
4 // Set mesh parameters following the Builder pattern
5 // Domain is the unit square Ω = ]0,1[^2
6 builder.setBottomLeftCorner(Eigen::Vector2d{0, 0})
7     .setTopRightCorner(Eigen::Vector2d{1, 1})
8     .setNoXCells(2)
9     .setNoYCells(2);
```

```
10 auto mesh_p = builder.Build();
```

The 8-cell triangular mesh generated by this code is drawn beside.

The mesh covers the unit square $\Omega =]0,1[^2$.

The green numbers give the cell indices.

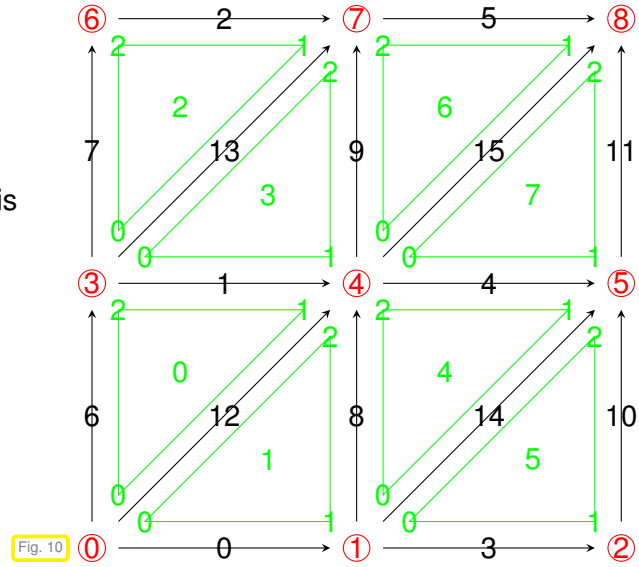


Fig. 10

Exploiting the insight from Sub-problem (2-10.i), around Code 2.10.15 build a unit test for validating the function `projectOntoGradients()` from Sub-problem (2-10.h)

```
TEST(ProjectionOntoGradients, exact_sol_test) { ... }
```

that uses $\Omega =]0,1[^2$ and the test vector field (triangle numbers as in Fig. 10)

$$\mathbf{f}(\boldsymbol{x}) = \begin{cases} \begin{bmatrix} 2 & 0 \end{bmatrix}^\top & \text{, if } \boldsymbol{x} \in K_0\,, \\ \begin{bmatrix} 0 & 2 \end{bmatrix}^\top & \text{, if } \boldsymbol{x} \in K_1\,, \\ \begin{bmatrix} -2 & 2 \end{bmatrix}^\top & \text{, if } \boldsymbol{x} \in K_4\,, \\ \begin{bmatrix} -2 & 0 \end{bmatrix}^\top & \text{, if } \boldsymbol{x} \in K_7\,, \\ \begin{bmatrix} 0 & -2 \end{bmatrix}^\top & \text{, if } \boldsymbol{x} \in K_6\,, \\ \begin{bmatrix} 2 & -2 \end{bmatrix}^\top & \text{, if } \boldsymbol{x} \in K_3\,, \\ \mathbf{0} & \text{elsewhere.} \end{cases} \tag{2.10.16}$$

Note that this vector field is just the gradient of the tent function associated with the central node of the 8-cell triangular mesh from Fig. 10. In the code you should define $\mathbf{f}$ without using a mesh object: just implement the different cases. Well, tedious, but that's an integral part of the development of simulation software!

The test should be carried out for a tensor-product triangular mesh $\mathcal{M}$ with 10 cells in each direction, for which $\mathbf{f}$ from (2.10.16) will belong to $\mathbf{grad}\, \mathcal{S}^0_{1,0}(\mathcal{M})$.

SOLUTION for (2-10.k) → `2-10-11-0:s4.pdf` ▲

References

[PR05] Yehuda Pinchover and Jacob Rubinstein. *An introduction to partial differential equations*. Cambridge University Press, Cambridge, 2005, pp. xii+371 (cit. on p. 65).

End Problem 2-10 , 183 min.

Problem 2-11: Hybrid-mesh Galerkin matrices and right-hand-side vectors

In [Lecture → Rem. 2.6.2.10] we saw that both linear and bilinear Lagrangian finite elements can be easily blended on a 2D hybrid mesh comprising both quadrilaterals and triangles. In this exercise we study the details of such a finite element method with focus on local computations and cell-oriented assembly. The computations, in particular in (2-11.e), may appear unnecessarily tedious, but it is instructive to carry out "manual assembly" at least once to get a full grasp of cell-oriented assembly as introduced in [Lecture → Section 2.4.5] and [Lecture → Section 2.7.4.2].

This is a purely theoretical problem related to [Lecture → Section 2.4.5] and [Lecture → Ex. 2.6.2.1] involving some tedious computations. Techniques from [Lecture → Section 2.8.3] are mentioned, but not essential.

Fig. 11 displays a hybrid mesh $\mathcal{M}$ consisting of 13 vertices, 8 triangular elements and 4 quadrilateral elements. The coordinates of some of the vertices are

$$\boldsymbol{a}^7 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad \boldsymbol{a}^1 := \begin{bmatrix} 0 \\ 1 \end{bmatrix},$$
$$\boldsymbol{a}^4 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} / \sqrt{2}, \quad \boldsymbol{a}^3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} / \sqrt{2}\,.$$

The coordinates of the other vertices follow from symmetry.

The figure also specifies a numbering of the vertices.

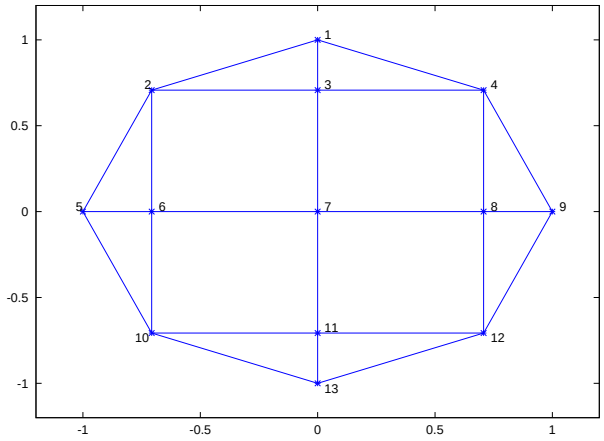


Fig. 11

In this problem we will compute the Galerkin matrix for (bi-)linear Lagrangian finite elements [Lecture → Section 2.6] on such a mesh for the bilinear form asscociated with $-\Delta$

$$\mathsf{a}(u,v) = \int_\Omega \mathbf{grad}\, u(\mathbf{x}) \cdot \mathbf{grad}\, v(\mathbf{x})\, \mathrm{d}\mathbf{x}, \quad u,v \in H^1(\Omega), \tag{2.11.1}$$

and the right-hand side vector arising from the linear form

$$\ell(v) = \int_\Omega f(\mathbf{x}) v(\mathbf{x})\, \mathrm{d}\mathbf{x}, \tag{2.11.2}$$

with $f \in C^0(\Omega)$. Note that the same model variational problem was studied in [Lecture → Section 2.4.5] and [Lecture → Section 2.4.6].

(2-11.a) ⚀ (1 min.) What is the dimension of the finite element space $\mathcal{S}_1^0(\mathcal{M})$?

HIDDEN HINT 1 for (2-11.a) → 2-11-1-0:hgm11.pdf

SOLUTION for (2-11.a) → 2-11-1-1:hgm1.pdf ▲

(2-11.b) ⚅ (15 min.) Compute the 4×4 element (Galerkin) matrix for one of the squares using the standard bilinear local shape functions from [Lecture → (2.6.2.3)]

HIDDEN HINT 1 for (2-11.b) → 2-11-2-0:hgm21.pdf

HIDDEN HINT 2 for (2-11.b) → 2-11-2-1:hgm31.pdf

SOLUTION for (2-11.b) → 2-11-2-2:hgm2.pdf ▲

(2-11.c) ⚂ (5 min.) Compute the 3×3 element (Galerkin) matrix for the triangle with vertices $1, 2, 3$ using the standard linear local shape functions (barycentric coordinate functions, see [Lecture → Section 2.4.5]).

HIDDEN HINT 1 for (2-11.c) → 2-11-3-0:hgm31.pdf

HIDDEN HINT 2 for (2-11.c) → 2-11-3-1:hgm32.pdf

SOLUTION for (2-11.c) → 2-11-3-2:hgm3.pdf ▲

(2-11.d) ⚀ (5 min.) Compute the element right-hand side vector $\vec{\varphi}_K$ for a quadrilateral cell K. For this, use the quadrature formula (quadrilateral trapezoidal rule)

$$\int_K f(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \frac{|K|}{4}\sum_{i=1}^{4} f(\mathbf{a}_K^i), \tag{2.11.4}$$

where $\mathbf{a}_K^i$ are the vertices of the square K and $|K|$ stands for its area.

HIDDEN HINT 1 for (2-11.d) → 2-11-4-0:hgm41.pdf

SOLUTION for (2-11.d) → 2-11-4-1:hgm4.pdf ▲

(2-11.e) ⚅ (24 min.) [depends on Sub-problem (2-11.b) and Sub-problem (2-11.c)]

What is the full 13×13 Galerkin matrix for the numbering of nodes given in Fig. 11?

HIDDEN HINT 1 for (2-11.e) → 2-11-5-0:hgm51.pdf

HIDDEN HINT 2 for (2-11.e) → 2-11-5-1:hgm52.pdf

HIDDEN HINT 3 for (2-11.e) → 2-11-5-2:hgm53.pdf

SOLUTION for (2-11.e) → 2-11-5-3:hgm5.pdf ▲

(2-11.f) ⚂ (10 min.) [depends on Sub-problem (2-11.d)]

Compute the full right-hand side vector $\vec{\varphi}$ using the local contributions found in (2-11.d). For the local contributions from the triangles, you can use the corresponding quadrature rule there (2D trapezoidal rule [Lecture → (2.4.6.10)]),

$$\int_K f(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \frac{|K|}{3}\sum_{i=1}^{3} f(\mathbf{a}^i),$$

with $\mathbf{a}^i$ the vertices of the triangle.

SOLUTION for (2-11.f) → 2-11-6-0:hgm6.pdf ▲

(2-11.g) ⚂ (5 min.) [Lecture → Rem. 2.6.2.12] discusses the choice of interpolation nodes and, thus, implicitly, the choice of global shape functions, for quadratic Lagrangian finite elements on hybrid meshes. What is the dimension of $\mathcal{S}_2^0(\mathcal{M})$, if $\mathcal{M}$ is the hybrid mesh displayed in Fig. 11?

SOLUTION for (2-11.g) → 2-11-7-0:hgm7.pdf ▲

(2-11.h) ⚄ (15 min.) Consider the Galerkin matrix $\mathbf{A}_Q$ for a general linear second-order elliptic Neumann boundary value problem when the space $\mathcal{S}_2^0(\mathcal{M})$ of quadratic Lagrangian finite elements on the hybrid mesh from Fig. 11 is used as a trial and test space. Give a sharp bound on the number $\mathrm{nnz}(\mathbf{A}_Q)$ of nonzero entries of $\mathbf{A}_Q$.

HIDDEN HINT 1 for (2-11.h) → 2-11-8-0:hgm81.pdf

SOLUTION for (2-11.h) → 2-11-8-1:hgm8.pdf ▲

End Problem 2-11, 80 min.

Problem 2-12: Testing built-in quadrature rules of LEHRFEM++

In [Lecture → § 2.7.5.39] we learned about the **lf::quad:QuadRule** interface to local quadrature rules in LEHRFEM++. In this problem you will practice its use and also validate the promised orders of the quadrature rules.

This problem uses LEHRFEM++ and the module **lf:quad**, see ↬ GITHUB. It requires familiarity with local numerical quadrature in the context of FEM, see [Lecture → Section 2.7.5.2].
▷ problem name/problem code folder: TestQuadratureRules

Through LEHRFEM++'s interface **lf::quad::QuadRule** to local quadrature rules one can request the nodes and weights of quadrature rules *defined on the reference entities*,

- the unit interval $[0,1]$ for type **SEGMENT**,
- the unit triangle [Lecture → (2.7.5.18b)] for type **TRIA**,
- and the unit square [Lecture → (2.8.1.13)] for type **QUAD**.

Definition [Lecture → Def. 2.7.5.9]. (Local) quadrature rule

A local quadrature rule on the element $K \in \mathcal{M}$ is an approximation

$$\int_K f(\mathbf{x})\,\mathrm{d}\boldsymbol{x} \approx \sum_{l=1}^{P_K} \omega_l^K f(\boldsymbol{\zeta}_l^K)\,, \quad \boldsymbol{\zeta}_l^K \in K\,, \omega_l^K \in \mathbb{R}\,, \quad P_K \in \mathbb{N}\,. \tag{2.12.1}$$

Terminology:

$\omega_l^K \to$ weights , $\boldsymbol{\zeta}_l^K \to$ quadrature nodes
(2.12.1) = P-point local quadrature rule

When constructing a quadrature via LEHRFEM++'s convenience function `lf::quad::make_QuadRule()` one can request a certain minimal order.

Definition [Lecture → Def. 2.7.5.29]. Order of a local quadrature rule

A local quadrature rule according to [Lecture → Def. 2.7.5.9] is said to be of order $q \in \mathbb{N}$, if
- for a simplex K (triangle tetrahedron) it is exact for all *polynomials* $f \in \mathcal{P}_{q-1}(\mathbb{R}^d)$,
- for a tensor product element K (rectangle, brick) it is exact for all *tensor product polynomials* $f \in \mathcal{Q}_{q-1}(\mathbb{R}^d)$.

Can we trust this function? In this problem we will find out and determine the order of the supplied quadrature rule by ourselves.

(2-12.a) ⊡ (5 min.) Refresh your knowledge about how the order of a quadrature rule is defined for various types of reference elements.

SOLUTION for (2-12.a) → 2-12-1-0:s1.pdf ▲

(2-12.b) ⚃ (30 min.) In the file `testquadraturerules.cc` complete the C++ function

```
bool testQuadOrderTria(const lf::quad::QuadRule &quad_rule,
  unsigned int order);
```

that returns true, if the passed quadrature rule for a *triangular* reference element has order `order`. Do not forget to check whether the quadrature rule is really meant for the reference triangle.

HIDDEN HINT 1 for (2-12.b) → 2-12-2-0:h2.pdf

SOLUTION for (2-12.b) → 2-12-2-1:s4.pdf ▲

(2-12.c) ⚄ (30 min.) In the file `testquadraturerules.cc` complete the C++ function

```
bool testQuadOrderQuad(const lf::quad::QuadRule &quad_rule,
  unsigned int order);
```

that confirms order `order` for a quadrature rule on the unit square. Make sure that the quadrature rule is really meant for the unit square.

SOLUTION for (2-12.c) → 2-12-3-0:s3.pdf ▲

(2-12.d) ⚄ (15 min.) [depends on Sub-problem (2-12.b), Sub-problem (2-12.c)]

In the file `testquadraturerules.cc` complete the C++ function

```
unsigned int calcQuadOrder(const lf::quad::QuadRule &quad_rule);
```

that computes the *maximal order* of the quadrature rule in `quad_rule`.

HIDDEN HINT 1 for (2-12.d) → 2-12-4-0:h41.pdf

SOLUTION for (2-12.d) → 2-12-4-1:s4.pdf ▲

End Problem 2-12 , 80 min.

Problem 2-13: Local computations for parametric Lagrangian finite elements

In [Lecture → Section 2.8.3] we have come across the notion of **parametric finite element methods** and learned how this construction simplifies the implementation of finite element method on very general meshes. We also studied the LEHRFEM++ tools for handling parametric finite elements, see [Lecture → § 2.8.3.6]. In this problem we practise using the tools offered by the `lf:geometry` layer to carry out local computations in the context of a Galerkin discretization by means of linear Lagrangian finite elements.

This problem requires knowledge about LEHRFEM++'s `lf:mesh` and `lf:geometry` layers and **PROVIDER** classes, see [Lecture → Code 2.7.4.23] and [Lecture → Code 2.7.4.28]. Its last part will also rely on the LEHRFEM++ built-in classes for local computations as introduced in [Lecture → Ex. 2.8.3.29].
▷ problem name/problem code folder: ParametricElementMatrices

Let $\Omega \subset \mathbb{R}^2$ be a bounded domain with piecewise smooth boundary [Lecture → § 1.2.1.14]. The weak form of the linear second-order scalar elliptic boundary value problem considered in this problem is: seek $u \in H^1(\Omega)$

$$\int_\Omega (\mathbf{I} + \boldsymbol{d}(\boldsymbol{x})\boldsymbol{d}(\boldsymbol{x})^\top)\,\mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} w^2(\boldsymbol{x}) u(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega \frac{v(\boldsymbol{x})}{1+w(\boldsymbol{x})^2}\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{2.13.1}$$

where $\boldsymbol{d} : \Omega \to \mathbb{R}^2$ is a continuous vectorfield $\boldsymbol{d} \in (C^0(\overline{\Omega}))^2$, and $w \in C^0(\overline{\Omega})$.

Remark. (2.13.1) models heat conduction in a medium with **anisotropic** and spatially varying heat conductivity $\boldsymbol{\kappa}(\boldsymbol{x}) := \mathbf{I} + \boldsymbol{d}(\boldsymbol{x})^\top \boldsymbol{d}(\boldsymbol{x})$: A temperature gradient in the direction of $\boldsymbol{d}(\boldsymbol{x})$ engenders stronger heat flow than the same temperature gradient orthogonal to it. Such an effect can be observed with materials composed of microscopic fibers.

(2-13.a) ⚀ (10 min.) Assuming sufficient smoothness of its solution u, state the strong form of the boundary value problem (2.13.1)

SOLUTION for (2-13.a) → `2-13-1-0:s4.pdf` ▲

For the remainder of this problem we assume that a general hybrid mesh $\mathcal{M}$ of Ω is given. On that mesh we use parametric linear Lagrangian finite elements, that is, the trial and test space $\mathcal{S}_1^0(\mathcal{M})$, for the Galerkin finite element discretization of (2.13.1).

Concerning the data, we make the following assumptions:

- The vectorfield $\boldsymbol{d}$ is given in procedural form.
- The function w is the result of another finite element computation and is given as a finite element function $w \in \mathcal{S}_1^0(\mathcal{M})$ described by its basis expansion coefficients with respect to the "tent-function" basis of $\mathcal{S}_1^0(\mathcal{M})$.

For all local integrations over cells of the mesh, you should use the local edge-midpoint quadrature rule

$$\int_K \varphi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{\sharp\mathcal{V}(K)} \sum_{\ell=1}^{\sharp\mathcal{V}(K)} \varphi(\boldsymbol{m}_K^\ell)\,, \quad K \in \mathcal{M}\,. \tag{2.13.2}$$

where $\boldsymbol{m}_K^\ell$ stands for the midpoints of the edges of K, and $\sharp\mathcal{E}(K)$ for the number of edges of K.

(2-13.b) ⚀ (45 min.) Implement a class **AnisotropicDiffusionElementMatrixProvider** (file `anisotropicdiffusionelementmatrixprovider.cc`)

```
class AnisotropicDiffusionElementMatrixProvider {
public:
  AnisotropicDiffusionElementMatrixProvider(
    std::function<Eigen::Vector2d(Eigen::Vector2d)>
      anisotropy_vec_field):
      anisotropy_vec_field_(anisotropy_vec_field) {};
  bool isActive(const lf::mesh::Entity &) { return true; }
  Eigen::MatrixXd Eval(const lf::mesh::Entity &cell);

private:
  std::function<Eigen::Vector2d(Eigen::Vector2d)>
    anisotropy_vec_field_;
};
```

compliant with LEHRFEM++'s concept of an **ENTITY_MATRIX_PROVIDER** is discussed in [Lecture → Ex. 2.7.4.33]. Its `Eval()` member function is to compute (based on (2.13.2)) the $\mathcal{S}_1^0(\mathcal{M})$-element matrix for the bilinear form $(u,v) \mapsto \int_\Omega \left(\mathbf{I} + \boldsymbol{d}(\boldsymbol{x})\boldsymbol{d}(\boldsymbol{x})^\top\right) \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$.

Exclusively use functions of the `lf::mesh` and `lf::geometry` layer of LEHRFEM++, no built-in quadrature rules.

HIDDEN HINT 1 for (2-13.b) → `2-13-2-0:h11.pdf`

SOLUTION for (2-13.b) → `2-13-2-1:s1.pdf` ▲

(2-13.c) ⚂ (20 min.) Local computations related to finite element Galerkin right-hand-side vectors are usually farmed out to objects matching the LEHRFEM++ concept of **ENTITY_VECTOR_PROVIDER**, see [Lecture → Ex. 2.7.4.33], [Lecture → Code 2.7.4.36]. Implement a corresponding class (file `fesourceelemvecprovider.cc`)

```
class FESourceElemVecProvider {
public:
  FESourceElemVecProvider(
    std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
      fe_space, Eigen::VectorXd coeff_expansion) :
      fe_space_(fe_space), coeff_expansion_(coeff_expansion) {}
  bool isActive(const lf::mesh::Entity &cell) { return true; }
  Eigen::VectorXd Eval(const lf::mesh::Entity &cell);

private:
  std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
    fe_space_;
  Eigen::VectorXd coeff_expansion_;
};
```

whose `Eval()` member functions returns the element load vector for the right-hand-side functional of (2.13.1) discretized on $\mathcal{S}_1^0(\mathcal{M})$ equipped with the customary tent-function global shape functions. The constructor accepts an object encoding the finite element space $\mathcal{S}_1^0(\mathcal{M})$, see [Lecture → § 2.8.3.28] and the basis expansion coefficient vector `coeff_expansion` for the function w. Again, use the local quadrature rule (2.13.2).

SOLUTION for (2-13.c) → `2-13-3-0:s2.pdf` ▲

(2-13.d) ⚂ (20 min.) Compute the $\mathcal{S}_1^0(\mathcal{M})$-element matrix spawned by the bilinear form

$$(u,v) \mapsto \int_{\partial\Omega} w^2(\boldsymbol{x})u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \quad u,v \in H^1(\Omega)\,,$$

for a *straight edge* of $\mathcal{M}$ located on $\partial\Omega$, *cf.* [Lecture → Ex. 2.5.3.7]. Use that $w \in \mathcal{S}_1^0(\mathcal{M})$ and express the entries of the element matrix in terms of the values w_1 and w_2 of w in the endpoints of the edge.

HIDDEN HINT 1 for (2-13.d) → 2-13-4-0:h03.pdf

HIDDEN HINT 2 for (2-13.d) → 2-13-4-1:h31.pdf

SOLUTION for (2-13.d) → 2-13-4-2:s4.pdf ▲

(2-13.e) ⚂ (30 min.) [depends on Sub-problem (2-13.d)]

Implement a class (file `impedanceboundaryedgematrixprovider.cc`)

```
class ImpedanceBoundaryEdgeMatrixProvider {
public:
  ImpedanceBoundaryEdgeMatrixProvider(
    std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
      fe_space,
    Eigen::VectorXd coeff_expansion);
  bool isActive(const lf::mesh::Entity &edge);
  Eigen::MatrixXd Eval(const lf::mesh::Entity &cell);
private:
  std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
    fe_space_;
  Eigen::VectorXd coeff_expansion_;
  std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<bool>>
    bd_flags_;
};
```

fitting LEHRFEM++'s concept of **ENTITY_MATRIX_PROVIDER** that computes the edge element matrix derived in Sub-problem (2-13.d) using the result of `lf::geometry::Volume` as a substitute for the length $|e|$ of the edge. The arguments for the constructor are the same as those for the constructor of the class **FESourceElemVecProvider** from Sub-problem (2-13.c).

Note that this time you also have to supply a non-trivial `isActive()` method that returns true, when given an edge on the boundary.

HIDDEN HINT 1 for (2-13.e) → 2-13-5-0:h41.pdf

HIDDEN HINT 2 for (2-13.e) → 2-13-5-1:h42.pdf

SOLUTION for (2-13.e) → 2-13-5-2:s4.pdf ▲

(2-13.f) ⚂ (30 min.) [depends on Sub-problem (2-13.b)]

In the Googletest framework develop a unit test for the `Eval()` member function of **AnisotropicDiffusionElementMatrixProvider** (file `test/parametricelementmatrices_test.cc`). The policy is to compare the results with the element matrices returned by the `Eval()` method of **lf::uscalfe::ReactionDiffusionElementMatrixProvider**, where objects of this type have to be initilized properly with edge-midpoint quadrature rule. These are available through

```
lf::quad::QuadRule lf::quad::make_TriaQR_EdgeMidpointRule();
lf::quad::QuadRule lf::quad::make_Quad Qr_EdgeMidpointRule();
```

Then,

- rely on the vectorfield

$$\boldsymbol{d}(\boldsymbol{x}) := \sin(\|\boldsymbol{x}\|)\boldsymbol{x}\,, \quad \boldsymbol{x} \in \mathbb{R}^2\,,$$

- use the hybrid mesh generated by the lfpp function call `lf::mesh::test_utils::GenerateHybrid2DTestMesh(5, 1)`, and
- for each of its cells compare the element matrices obtained by the two `Eval()` functions. As usual, when comparing results of floating point computations, you should use the `EXPECT_NEAR` macro.

HIDDEN HINT 1 for (2-13.f) → `2-13-6-0:h51.pdf`

HIDDEN HINT 2 for (2-13.f) → `2-13-6-1:h25.pdf`

SOLUTION for (2-13.f) → `2-13-6-2:s5.pdf` ▲

(2-13.g) ⚄ (15 min.) Parallel to Sub-problem (2-13.f) implement a unit test for the `Eval()` member function of **FESourceElemVecProvider** comparing it to the corresponding member function of **lf::fe::ScalarLoadElementVectorProvider** (file `test/parametricelementmatrices_test.cc`).

As w use the function $w_h \in \mathcal{S}_1^0(\mathcal{M})$ obtained by interpolating the affine linear function $\boldsymbol{x} \to 3x_1 - x_2$ in the nodes of the mesh. To that end you may use the function `lf::uscalfe::NodalProjection<double>()`, see ↪ (documentation).

HIDDEN HINT 1 for (2-13.g) → `2-13-7-0:h21.pdf`

SOLUTION for (2-13.g) → `2-13-7-1:s6.pdf` ▲

End Problem 2-13, 170 min.

Problem 2-14: Non-conforming Crouzeix-Raviart finite element method

From [Lecture → Thm. 1.3.4.23] we have learned that global continuity of an $\mathcal{M}$-piecewise smooth ($\mathcal{M}$ a finite element mesh of a bounded domain Ω) function is sufficient and necessary for it to belong to $H^1(\Omega)$. This is why we took pains to ensure that finite element spaces designed for the Galerkin discretization of 2nd-order elliptic boundary value problems were contained in $C^0(\overline{\Omega})$. However, there are a few exotic finite element schemes that relax strict C^0-continuity and rely on trial and test spaces that fail to be subspaces of $H^1(\Omega)$. They are known as *non-conforming* finite element schemes [Bra07, Sect. III.1]. In this problem we will encounter such alien species, which take us beyond Lagrangian FEM. An application of this "exotic" method is presented in [Lecture → Section 12.3.5].

This problem offers an A-Z repetition of important concepts in FEM from, among others, [Lecture → Section 2.5], [Lecture → Section 2.7.4]. You will also be asked to implement a several functions in C++ within the LEHRFEM++ framework. Knowing the contents of [Lecture → Section 2.7], [Lecture → Section 2.8], in particular [Lecture → Section 2.8.3], and also [Lecture → Section 2.5.3] will be important.
▷ problem name/problem code folder: NonConformingCrouzeixRaviartFiniteElements

Remark. Do not be discouraged by the sheer length of this problem. It has been designed to guide you through many steps, covering both theoretical aspects and the complete implementation.

Remark. For customary 2nd-order elliptic BVPs, the use of non-conforming finite element methods is largely of theoretical interest. Yet for the finite element discretization of coupled or constrained problems, for e.g. in fluid mechanics and electromagnetics, non-conforming finite element methods may sometimes offer specific advantages.

In this problem a non-conforming finite element scheme will offer an opportunity to practice notions and techniques for finite elements concerning the handling of degrees of freedom (global and local shape functions), as they have been discussed in [Lecture → Section 2.7.4]. You will also learn how to define a "scalar reference element", that is an interface of type **lf::fe::ScalarReferenceFiniteElement**, in LEHRFEM++.

Let a *triangular* mesh $\mathcal{M}$ of a 2D bounded polygonal domain $\Omega \subset \mathbb{R}^2$ be given and write $\mathcal{N} = \{\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N\}$, $N \in \mathbb{N}$, for the set of midpoints of its edges. A numbering of these points is assumed and in LEHRFEM++ will be provided by an object of type **lf::assemble::DofHandler**, see [Lecture → § 2.7.4.13].

The so-called Crouzeix-Raviart finite element space $\mathcal{CR}(\mathcal{M}) \subset L^2(\Omega)$ on the mesh $\mathcal{M}$ is defined as the span of the functions b_h^j, $j = 1, \ldots, N$, which satisfy

$$b^i_{h\,|K} \in \mathcal{P}_1(K) \quad \forall K \in \mathcal{M} \quad , \quad b^i_h(\boldsymbol{m}_j) = \begin{cases} 1 & \text{, if } i = j \text{ ,} \\ 0 & \text{else,} \end{cases} \quad i,j \in \{1,\ldots,N\} \text{ ,} \tag{2.14.1}$$

see Fig. 12.

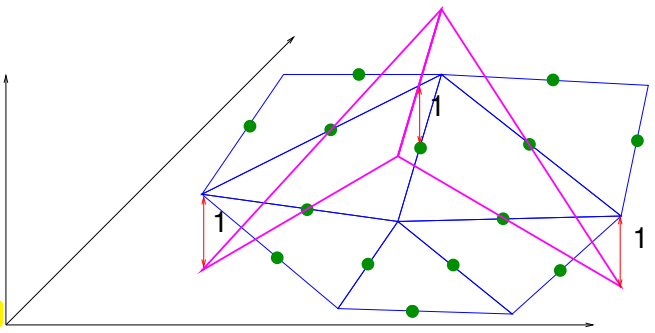


Fig. 12

◁ A *piecewise linear* global shape function of the finite element space $\mathcal{CR}(\mathcal{M})$ associated with the midpoint of an edge of a mesh.

The subspace that takes into account homogeneous Dirichlet boundary conditions is

$$\mathcal{CR}_0(\mathcal{M}) := \{v_h \in \mathcal{CR}(\mathcal{M}) : \ v_h(\boldsymbol{m}) = 0 \ \forall \boldsymbol{m} \in \mathcal{N} \cap \partial\Omega\} \ . \tag{2.14.2}$$

(2-14.a) ⚂ (10 min.) Show that (2.14.1) provides a valid definition of the functions b_h^j.

HIDDEN HINT 1 for (2-14.a) → 2-14-1-0:cppcrfe11.pdf

SOLUTION for (2-14.a) → 2-14-1-1:cppcrfe1.pdf ▲

(2-14.b) ⚂ (5 min.) Show that the set of functions $\{b_h^j : j = 1, \ldots, N\}$ is linearly independent.

SOLUTION for (2-14.b) → 2-14-2-0:cppcrfe2.pdf ▲

(2-14.c) ⚀ (3 min.) Describe the set $\mathcal{N}$ of **interpolation nodes** (→ [Lecture → Ex. 2.6.1.2]) for which $\{b_h^j, j = 1, \ldots, N\}$ is a **cardinal basis** for $\mathcal{CR}(\mathcal{M})$, *cf.* [Lecture → (2.6.1.4)].

SOLUTION for (2-14.c) → 2-14-3-0:2bs.pdf ▲

(2-14.d) ⚂ (10 min.) Assume that the triangular mesh $\mathcal{M}'$ is generated by regular refinement of another coarser triangular mesh $\mathcal{M}$. This means that every triangle is split into four smaller congruent triangles by connecting the midpoints of the edges.

In contrast to what is true in the case of Lagrangian finite element spaces, it may happen that $\mathcal{CR}(\mathcal{M}) \not\subset \mathcal{CR}(\mathcal{M}')$! Explain why.

HIDDEN HINT 1 for (2-14.d) → 2-14-4-0:cppcrfe2a1.pdf

SOLUTION for (2-14.d) → 2-14-4-1:cppcrfe2a.pdf ▲

(2-14.e) ⚂ (5 min.) Show that $\mathcal{CR}(\mathcal{M}) \not\subset H^1(\Omega)$.

SOLUTION for (2-14.e) → 2-14-5-0:cppcrfe3.pdf ▲

(2-14.f) ⚁ (2 min.) Describe the support of a basis function b_h^i.

SOLUTION for (2-14.f) → 2-14-6-0:cppcrfe4.pdf ▲

(2-14.g) ⚂ (5 min.) Is the set of functions defined in (2.14.1) a valid set of finite element basis functions in the sense of [Lecture → Section 2.5.3]?

SOLUTION for (2-14.g) → 2-14-7-0:cppcrfe1a.pdf ▲

(2-14.h) ⚂ (5 min.)

In the file `nonconformingcrouzeixraviartfiniteelements_main.cc` ⇢ GITHUB add those few lines of C++ code that effect the initialization of a LEHRFEM++ **lf::assemble::DofHandler** object suitable for the finite element space $\mathcal{CR}(\mathcal{M})$. A pointer to a **lf::mesh::Mesh** object describing the triangular mesh $\mathcal{M}$ is available in the variable `mesh_p`.

HIDDEN HINT 1 for (2-14.h) → 2-14-8-0:h4x.pdf

SOLUTION for (2-14.h) → 2-14-8-1:s4x.pdf ▲

(2-14.i) ⚃ (15 min.) We use the b_h^i from (2.14.1) as global shape functions and, thus, the corresponding local shape functions of $\mathcal{CR}(\mathcal{M})$ are fixed by the formula from [Lecture → Def. 2.5.3.4]. Write these local shape functions on a triangle K as linear combinations of the barycentric coordinate functions λ_1, λ_2, and λ_3 of K. To do so, use LEHRFEM++'s local numbering convention from [Lecture → § 2.7.2.14].

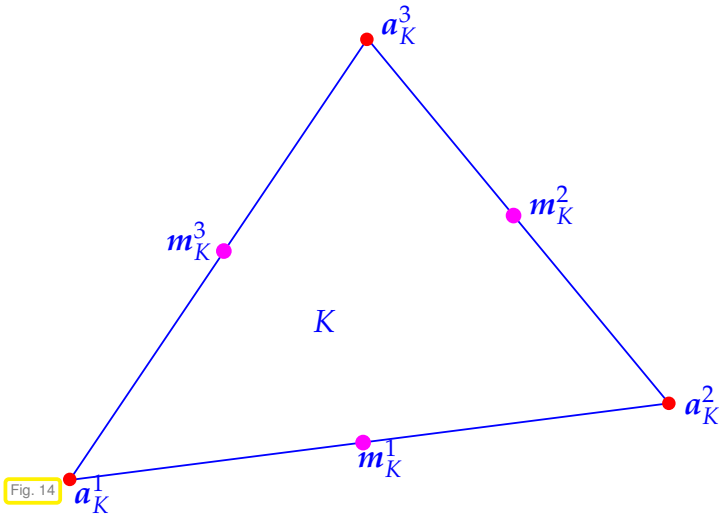


Fig. 14

◁ local numbering convention for vertices and edges of a triangle K in LEHRFEM++, see also [Lecture → Fig. 131]:

local edge j		opposite vertex $\mathrm{opp}(j)$
1	$\mapsto$	3
2	$\mapsto$	1
3	$\mapsto$	2

For ease of notation use the mapping

$$\mathrm{opp}(j) := (j+1) \mod 3 + 1 \,.$$

This induces a numbering of local shape functions associated with the various geometric (sub-)entities of K.

SOLUTION for (2-14.i) → 2-14-9-0:cppcrfe5.pdf ▲

(2-14.j) ⚀ (5 min.) [depends on Sub-problem (2-14.i)]

Show that the finite element space $\mathcal{CR}(\mathcal{M})$ on a triangular mesh $\mathcal{M}$ is *parametric* in the sense of [Lecture → Def. 2.8.3.1].

Definition [Lecture → Def. 2.8.3.1]. Parametric finite elements

A finite element space on a mesh $\mathcal{M}$ is called parametric, if there exist a few reference elements $\widehat{K}_1, \ldots, \widehat{K}_R$, $R \in \mathbb{N}$, numbers $Q_R \in \mathbb{N}$, and functions $\widehat{b}_r^i \in C^0(\widehat{K})$, $i = 1, \ldots, Q_R$, $r = 1, \ldots, R$, such that

$$\forall K \in \mathcal{M}: \quad \exists r \in \{1, \ldots, R\}, \text{ bijection } \boldsymbol{\Phi}_K : \widehat{K} \mapsto K: \quad \widehat{b}_r^i = \boldsymbol{\Phi}_K^* b_K^i, \quad i = 1, \ldots, Q_R \,,$$

where $\{b_K^1, \ldots, b_K^Q\}$ is the set of local shape functions on K.

SOLUTION for (2-14.j) → 2-14-10-0:s8x.pdf ▲

(2-14.k) ⚀ (10 min.) [depends on Sub-problem (2-14.i)]

Compute explicit formulas for the local shape functions of the finite element space $\mathcal{CR}(\mathcal{M})$ and for their gradients on the "unit reference triangle" $\widehat{K}$ (→ [Lecture → § 2.7.5.17]) with vertices $\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}$.

SOLUTION for (2-14.k) → 2-14-11-0:cppcrfe5a.pdf ▲

(2-14.l) ⚂ (10 min.) [depends on Sub-problem (2-14.i)]

Compute the element (Galerkin) matrix for a triangle K and the finite element space $\mathcal{CR}(\mathcal{M})$ and the bilinear form

$$\mathsf{a}(u,v) = \int_\Omega uv \,\mathrm{d}\boldsymbol{x} \,, \quad u, v \in L^2(\Omega) \,.$$

This element matrix is also called the **element mass matrix** $\mathbf{M}_K$.

HIDDEN HINT 1 for (2-14.l) → 2-14-12-0:cppcrfe61.pdf

SOLUTION for (2-14.l) → 2-14-12-1:cppcrfe6.pdf ▲

Next, we want to use $\mathcal{CR}(\mathcal{M})/\mathcal{CR}_0(\mathcal{M})$ for the non-conforming finite element Galerkin discretization of

the linear 2nd-order elliptic Neumann boundary value problem

$$-\Delta u + \gamma(\boldsymbol{x})u = f(\boldsymbol{x}) \quad \text{in } \Omega \qquad \mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \text{on } \partial\Omega \ , \tag{2.14.6}$$

and for the 2nd-order elliptic homogeneous Dirichlet boundary value problem

$$-\Delta u + \gamma(\boldsymbol{x})u = f(\boldsymbol{x}) \quad \text{in } \Omega \qquad u = 0 \quad \text{on } \partial\Omega \ , \tag{2.14.7}$$

with spatially varying reaction coefficient $\gamma \in C^0(\overline{\Omega})$, $\gamma \geq 0$ in Ω, $f \in L^2(\Omega)$, $\Omega \subset \mathbb{R}^2$ a bounded polygon.

Of course, by [Lecture → Thm. 1.3.4.23] the gradient of a function $\in \mathcal{CR}(\mathcal{M})$ is not square integrable and therefore we cannot rely on the standard variational formulation for (2.14.7) and (2.14.6) from, e.g., [Lecture → Ex. 1.8.0.2]. Instead we resort to a variational formulation with "broken-up bilinear form" in order to state the discrete variational problem correspoding to (2.14.7): seek $u_h \in \mathcal{CR}_0(\mathcal{M})$

$$\mathsf{a}_{\mathcal{M}}(u_h, v_h) + \int_\Omega \gamma(\boldsymbol{x}) u_h(\boldsymbol{x}) v_h(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v_h(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v_h \in \mathcal{CR}_0(\mathcal{M}) \ , \tag{2.14.8}$$

$$\text{with} \qquad \mathsf{a}_{\mathcal{M}}(u_h, v_h) := \sum_{K\in\mathcal{M}} \int_K \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \ . \tag{2.14.9}$$

This is well defined, because on each cell the gradient of a function in $\mathcal{CR}(\mathcal{M})$ is just a constant vector field.

(2-14.m) ⚄ (10 min.) Show that

$$\mathsf{a}_{\mathcal{M}}(v_h, v_h) > 0 \quad \forall v_h \in \mathcal{CR}_0(\mathcal{M}) \setminus \{\mathbf{0}\} \ . \tag{2.14.10}$$

HIDDEN HINT 1 for (2-14.m) → 2-14-13-0:cppcrfe61.pdf

SOLUTION for (2-14.m) → 2-14-13-1:cppcrfe6.pdf ▲

(2-14.n) ⚄ (15 min.) [depends on Sub-problem (2-14.i)]

Let $\mathbf{A}_K^L$, $K \in \mathcal{M}$ a triangle, be the element matrix for the bilinear form $(u, v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$ and the Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$ equipped with the standard "tent-function basis", see [Lecture → (2.4.5.8)].

Write $\mathbf{A}_K^{\mathcal{CR}}$ for the element matrix spawned by the bilinear form $\mathsf{a}_{\mathcal{M}}$, the finite element space $\mathcal{CR}(\mathcal{M})$ and the basis introduced in (2.14.1). Which *congruence transformation* converts $\mathbf{A}_K^L$ into $\mathbf{A}_K^{\mathcal{CR}}$?

Definition [Lecture → Def. 2.2.3.5]. Congruent matrices

Two matrices $\mathbf{A} \in \mathbb{R}^{N,N}$, $\mathbf{B} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$, are called congruent, if there is a regular matrix $\mathbf{S} \in \mathbb{R}^{N,N}$ such that $\mathbf{B} = \mathbf{S}\mathbf{A}\mathbf{S}^T$.

SOLUTION for (2-14.n) → 2-14-14-0:s6a.pdf ▲

We now focus on developing an efficient C++ implementation using LEHRFEM++ to solve the discrete variational problem (2.14.8) for a triangular mesh $\mathcal{M}$. All occuring integrals should be tackled by means of local edge-midpoint quadrature:

$$\int_K \psi(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3} \sum_{\ell=1}^{3} \psi(\boldsymbol{m}_K^\ell) \ , \quad K \in \mathcal{M} \ , \tag{2.14.15}$$

where $\boldsymbol{m}_K^\ell$ stands for the midpoint of the ℓ-th edge of K.

(2-14.o) ⚄ (10 min.) [depends on Sub-problem (2-14.n) and Sub-problem (2-14.m)]

Give the formula for the element matrix for the finite element space $\mathcal{CR}(\mathcal{M})$ equipped with the basis introduced in (2.14.1), a triangular cell $K \in \mathcal{M}$, and for the bilinear form

$$(u_h, v_h) \mapsto \mathsf{a}_{\mathcal{M}}(u_h, v_h) + \int_\Omega \gamma(\boldsymbol{x}) u_h(\boldsymbol{x}) v_h(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\,, \quad u_h, v_h \in \mathcal{CR}_0(\mathcal{M})\,,$$

on the left-hand side of (2.14.8), which is evaluated approximately by means of the local quadrature formula (2.14.15).

You can use the symbol $\mathbf{A}_K^{\mathcal{CR}}$ for the element matrix computed in Sub-problem (2-14.n).

HIDDEN HINT 1 for (2-14.o) → 2-14-15-0:h7a.pdf

SOLUTION for (2-14.o) → 2-14-15-1:s7.pdf ▲

(2-14.p) ⚂ Give the formula for the element right-hand-side vector (element load vector) $\vec{\boldsymbol{\varphi}}_K$, $K \in \mathcal{M}$, for the variational problem (2.14.8) discretized by means of $\mathcal{CR}(\mathcal{M})$. Since the source function f is available only in procedural form, use the quadrature formula (2.14.15) locally.

HIDDEN HINT 1 for (2-14.p) → 2-14-16-0:h7a.pdf

SOLUTION for (2-14.p) → 2-14-16-1:s7a.pdf ▲

(2-14.q) ⚄ [depends on Sub-problem (2-14.k) and Sub-problem (2-14.j)]

[Lecture → § 2.8.3.25] introduced the concept of **lf::fe::ScalarReferenceFiniteElement** as LEHRFEM++'s interface for scalar-valued parametric finite element spaces. Now the task is to implement such an interface for the finite element space $\mathcal{CR}(\mathcal{M})$ on a triangular mesh, which, by Sub-problem (2-14.j) is parametric in the sense of [Lecture → Def. 2.8.3.1], even **affine equivalent** [Lecture → Section 2.8.1]. The interface should be realized as a class **CRReferenceFiniteElement** derived from **lf::fe::ScalarReferenceFiniteElement**.

Implement the following member functions of **CRReferenceFiniteElement**:

- ```
  lf::base::RefEl CRReferenceFiniteElement::RefEl() const;
  unsigned int CRReferenceFiniteElement::Degree() const;

  lf::assemble::size_type
     CRReferenceFiniteElement::NumRefShapeFunctions() const;
  lf::assemble::size_type
     CRReferenceFiniteElement::NumRefShapeFunctions(
    lf::base::dim_t codim) const;
  lf::assemble::size_type
     CRReferenceFiniteElement::NumRefShapeFunctions(
    lf::base::dim_t codim, lf::base::sub_idx_t subidx) const;
  ```

- ```
  Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
     CRReferenceFiniteElement::EvalReferenceShapeFunctions(
       const Eigen::MatrixXd& refcoords) const;
  ```

HIDDEN HINT 1 for (2-14.q) → 2-14-17-0:cppcrfe9a1.pdf

SOLUTION for (2-14.q) → 2-14-17-1:cppcrfe9a.pdf ▲

(2-14.r) ⚂ (15 min.) [depends on Sub-problem (2-14.k)]

Implement the member function

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
  CRReferenceFiniteElement::GradientsReferenceShapeFunctions(
    const Eigen::MatrixXd& refcoords) const;
```

Look up the web page ↪ (documentation) to find the specification for this method.

SOLUTION for (2-14.r) → 2-14-18-0:s9b.pdf ▲

(2-14.s) ⚄ (10 min.) Implement the following member functions of **CRReferenceFiniteElement**:

-
```
Eigen::MatrixXd CRReferenceFiniteElement::EvaluationNodes()
   const;
```

-
```
Eigen::Matrix<double, 1, Eigen::Dynamic>
   CRReferenceFiniteElement::NodalValuesToDofs(
    const Eigen::Matrix<SCALAR, 1, Eigen::Dynamic>& nodvals)
      const;
```

For explanations see [Lecture → § 2.8.3.25] and ↪ (documentation). Note that in LEHRFEM++ an "evaluation node" corresponds to a local interpolation node. The local interpolation nodes for $\mathcal{RM}(\mathcal{M})$ are just the midpoints of the edges of a triangle.

HIDDEN HINT 1 for (2-14.s) → 2-14-19-0:h19c.pdf

SOLUTION for (2-14.s) → 2-14-19-1:s9c.pdf ▲

(2-14.t) ⚄ (5 min.) [depends on Sub-problem (2-14.q)]

Implement a class **CRFeSpace** compliant with LEHRFEM++'s interface **lf::uscalfe::UniformScalarFESpace** (and derived from it) for the finite element space $\mathcal{CR}(\mathcal{M})$. This interface is presented in [Lecture → § 2.8.3.28].

Take the cue from the implementation of **lf::uscalfe::FeSpaceLagrangeO1**, but note that you need only supply the specification for local shape functions on triangles, none for QUADs and EDGEs (pass **nullptr** for those).

HIDDEN HINT 1 for (2-14.t) → 2-14-20-0:h1x.pdf

SOLUTION for (2-14.t) → 2-14-20-1:s9x.pdf ▲

(2-14.u) ⚄ (30 min.) [depends on Sub-problem (2-14.q) and Sub-problem (2-14.t)]

Implement a C++ function

```
template<typename GAMMA_COEFF, typename F_FUNCTOR>
  Eigen::VectorXd solveCRNeumannBVP(
      std::shared_ptr<CRFeSpace> fe_space,
      GAMMA_COEFF &&gamma
      F_FUNCTOR &&f);
```

that solves the discrete variational problem (2.14.8) with $\mathcal{CR}_0(\mathcal{M})$ replaced with $\mathcal{CR}(\mathcal{M})$ (no boundary conditions enforced on trial and test space any more) and returns the basis expansion coefficient vector of the solution $u_h \in \mathcal{CR}(\mathcal{M})$. The argument `gamma` passes a functor (with evaluations operator

```
double operator () (Eigen::Vector2d x) const;
```

for the coefficient function $\gamma \in C^0(\overline{\Omega})$ that is supposed to be uniformly positive.

lf::fe::ScalarReferenceFiniteElement as "plugin"

Of course, your implementation should rely on the assembly functions in the LEHRFEM++ module `lf::assemble`. Moreover, you can make use of the LEHRFEM++ classes **lf::uscalfe::ReactionDiffusionElementMatrixProvider** ([Lecture → Ex. 2.8.3.29]) and **lf::fe::ScalarLoadElementVectorProvider**, which can serve as builder helper classes for element matrices and element vectors. Note that functions are passed as **MeshFunction** objects to the constructors of these classes!
Examining [Lecture → Code 2.8.3.31] convince yourself that the `Eval()` member functions of theses classes will smoothly work in the case of $\mathcal{CR}(\mathcal{M})$ provided that the finite element space contains the right **lf::fe::ScalarReferenceFiniteElement** object.

Thus, you can simply copy large parts of the code from ➙ GITHUB.

SOLUTION for (2-14.u) → `2-14-21-0:s12a.pdf` ▲

(2-14.v) ⚄ (20 min.) [depends on Sub-problem (2-14.u)]

Based on LEHRFEM++'s assembly facilities and the classes **lf::uscalfe::ReactionDiffusionElementMatrixProv** and **lf::fe::ScalarLoadElementVectorProvider** create a C++ function

```
template <typename GAMMA_COEFF, typename F_FUNCTOR>
Eigen::VectorXd solveCRDirichletBVP(
  std::shared_ptr<CRFeSpace> fe_space,
  GAMMA_COEFF &&gamma,
  F_FUNCTOR &&f);
```

that solves (2.14.8) with trial and test space $\mathcal{CR}_0(\mathcal{M})$, which is supposed to provide a non-conforming finite element discretization for the a Dirichlet boundary value problem with homogeneous Dirichlet boundary conditions. The arguments and return value are the same as in Sub-problem (2-14.u).

To solve this problem remember how to enforce Dirichlet boundary conditions in LEHRFEM++: [Lecture → § 2.7.6.13]. Also look at the demo code ➙ GITHUB.

SOLUTION for (2-14.v) → `2-14-22-0:s13.pdf` ▲

(2-14.w) ⚄ (30 min.) We now want to compute the $L^2(\Omega)$-norm of the finite element discretization error $u_h - u$ of the $\mathcal{CR}_0(\mathcal{M})$ non-conforming finite element solution of the homogeneous Dirichlet problem (2.14.7), where $u \in H^1(\Omega)$ stands for its exact solution.

For this purpose, write the code for a lfpp-based C++ function

```
template <typename FUNCTION>
double computeCRL2Error(
  std::shared_ptr<CRFeSpace> fe_space,
  const Eigen::VectorXd &mu,
  FUNCTION && u);
```

which takes as input arguments

- a **CRFeSpace** object for the finite element space $\mathcal{CR}(\mathcal{M})$,
- a column vector `mu` supplying the basis expansion coefficients a finite element function $u_h(\boldsymbol{x}) = \sum_{j=0}^{N-1} \mu_j b_h^j(\boldsymbol{x}) \in \mathcal{CR}(\mathcal{M})$, and
- a functor object `u`, which allows access to the exact solution u through a point-evaluation operator

with signature **double operator** (Eigen::Vector2d x) **const**.

and returns the $L^2(\Omega)$-norm of the $\mathcal{CR}_0$ finite element discretization error: $\|u_h - u\|_{L^2(\Omega)}$.

Compute the norm directly in a cell oriented fashion using a loop over the cells and the local quadrature formula (2.14.15).

HIDDEN HINT 1 for (2-14.w) → 2-14-23-0:h1sp15.pdf

SOLUTION for (2-14.w) → 2-14-23-1:cppcrfe15.pdf ▲

(2-14.x) ⚄ (20 min.) [depends on Sub-problem (2-14.v)]

Empirically we study the behavior of the $L^2(\Omega)$-norm of the $\mathcal{CR}_0(\mathcal{M})$-discretization error for the 2nd-order elliptic boundary value problem with a known solution u, with the data

$$\begin{aligned} f(\boldsymbol{x}) &= (2\pi^2 + x_1x_2)\sin(\pi x_1)\sin(\pi x_2) \\ c(\boldsymbol{x}) &= x_1x_2\,, \\ u(\boldsymbol{x}) &= \sin(\pi x_1)\sin(\pi x_2)\,, \end{aligned} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{2.14.27}$$

Code a C++ function

```
double L2errorCRDiscretizationDirichletBVP(std::string filename);
```

that reads a 2D triangular mesh $\mathcal{M}$ from a .msh-file with name filename, computes the $\mathcal{CR}_0(\mathcal{M})$-finite element solution u_h of the homogeneous Dirichlet boundary value problem (2.14.7) on it using the data from (2.14.27), and returns the $\|u - u_h\|_{L^2(\Omega)}$.

HIDDEN HINT 1 for (2-14.x) → 2-14-24-0:cppcrfe141.pdf

SOLUTION for (2-14.x) → 2-14-24-1:cppcrfe14.pdf ▲

(2-14.y) ⚄ (20 min.) [depends on Sub-problem (2-14.x)]

In the repository ➙ GitLab you find a number of .msh-files refined_square*.msh, *=1,2,3,4, providing a sequence of refined meshes of the unit square.

Extend the main()-function in templates such that it writes a table listing

(i) the dimensions of $\mathcal{CR}_0(\mathcal{M})$ for each mesh,

(ii) the $L^2(\Omega)$-norms for the $\mathcal{CR}_0$-discretization errors for the testing problem with data (2.14.27) on that mesh.

SOLUTION for (2-14.y) → 2-14-25-0:cppcrfe16.pdf ▲

(2-14.z) ⚄ (15 min.) [depends on Sub-problem (2-14.y)]

Describe qualitatively and quantitatively the convergence of the $L^2(\Omega)$-norm of the $\mathcal{CR}_0$-solution of the testing boundary value problem from Sub-problem (2-14.x).

See [Lecture → § 3.2.2.5] for details on what you are expected to do.

SOLUTION for (2-14.z) → 2-14-26-0:sz.pdf ▲

References

[Bra07] Dietrich Braess. *Finite elements. Theory, Fast Solvers, and Applications in Elasticity Theory*. Third. Cambridge: Cambridge University Press, 2007, p. 365. DOI: 10.1017/CBO9780511618635 (cit. on p. 77).

End Problem 2-14 , 285 min.

Problem 2-15: Regularized Neumann Problem

Pure 2nd-order elliptic Neumann problems may not have a solution as was discussed in [Lecture → Rem. 1.8.0.14]. In this problem we will how this can be cured in the context of finite element Galerkin discretization by imposing a vanishing-average constraint.

Problem based on the `lf::assembly` module of LEHRFEM++, see [Lecture → Section 2.7.4.3].
▷ problem name/problem code folder: RegularizedNeumannProblem

Let $\Omega \subset \mathbb{R}^2$ be a bounded and *connected* polygonal domain and $f \in L^2(\Omega)$, $h \in L^2(\partial\Omega)$. The variational Neumann problem is a linear variational problem and reads

$$u \in H^1(\Omega) \quad : \underbrace{\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = \underbrace{\int_\Omega f v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h v \,\mathrm{d}S}_{=:\ell(v)} \quad \forall v \in H^1(\Omega)\,. \tag{2.15.1}$$

Obviously, it fails to have a unique solution, even if the compatibility condition [Lecture → (1.8.0.13)]

$$-\int_{\partial\Omega} h \,\mathrm{d}S = \int_\Omega f \,\mathrm{d}\boldsymbol{x} \tag{2.15.2}$$

is satisfied (see [Lecture → Rem. 1.8.0.14]).

A remedy is to replace the trial/test space with [Lecture → (1.8.0.15)]

$$H^1_*(\Omega) := \{v \in H^1(\Omega) : \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0\}. \tag{2.15.3}$$

(2-15.a) ⚀ (5 min.) Argue why the linear variational problem (2.15.1) will always posses a *unique* solution when posed on $H^1_*(\Omega)$, even if (2.15.2) is violated.

SOLUTION for (2-15.a) → `2-15-1-0:regNeu1.pdf` ▲

(2-15.b) ⚅ (20 min.) Let $u \in H^1_*(\Omega)$ stand for the solution of (2.15.1) on $H^1_*(\Omega)$. We do not assume (2.15.2).

Propose a modification of the data f and h such that u also solves (2.15.1) on $H^1(\Omega)$.

SOLUTION for (2-15.b) → `2-15-2-0:regNeu2.pdf` ▲

In the file `getgalerkinlse.h` you can find the LEHRFEM++-based function

```
template <typename FUNCT_F, typename FUNCT_H>
std::pair<Eigen::SparseMatrix<double>, Eigen::VectorXd>
   getGalerkinLSE(
   const std::shared_ptr<lf::uscalfe::ScalarUniformFESpace<double>>
      fe_space,
   const FUNCT_F &f, const FUNCT_H &h);
```

modeled after `SecOrdEllBVPLagrFELinSys()` in ↬ GITHUB that returns the Galerkin matrix and right hand side vector arising from the Galerkin discretization of (2.15.1) based on Lagrangian finite elements as specified in the `fe_space` argument. The types `FUNCT_F` and `FUNCT_H` must match the functor concept and `f`, `h` provide function objects for f and h.

C++11 code 2.15.5: Implementation of `getGalerkinLSE`

```
2 template <typename FUNCT_F, typename FUNCT_H>
3 std::pair<Eigen::SparseMatrix<double>, Eigen::VectorXd> getGalerkinLSE(
```

```
    const std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space,
    const FUNCT_F &f, const FUNCT_H &h) {
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};

  const std::size_t N_dofs = dofh.NumDofs();
  // Right-hand-side vector; don't forget to set to zero initially!
  Eigen::VectorXd rhs_vec(N_dofs);
  rhs_vec.setZero();

  // ASSEMBLE MATRIX
  lf::assemble::COOMatrix<double> A_aux(N_dofs, N_dofs);
  lf::uscalfe::LinearFELaplaceElementMatrix my_mat_provider{};
  // co-dimension 0 because we locally assemble on cells
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, my_mat_provider, A_aux);

  // ASSEMBLE GLOBAL RHS VECOR
  // Volume part v ↦ ∫_Ω f v dx of the rhs linear
  // functional
  lf::uscalfe::ScalarLoadElementVectorProvider my_vec_provider(fe_space, f);
  // co-dimension 0 because we locally assemble on cells
  lf::assemble::AssembleVectorLocally(0, dofh, my_vec_provider, rhs_vec);

  // Boundary part v ↦ ∫_∂Ω h v dS of the rhs
  // linear functional.  Only edges on the boundary should be included
     in
  // assembly, which is achieved by passing a suitable selector
     predicate to the
  // builder object through a lambda function
  auto bd_edges{lf::mesh::utils::flagEntitiesOnBoundary(dofh.Mesh(), 1)};
  lf::uscalfe::ScalarLoadEdgeVectorProvider my_vec_provider_edge(
      fe_space, h,
      [&bd_edges](const lf::mesh::Entity &edge) { return bd_edges(edge); });
  // co-dimension 1 because we locally assemble on edges !
  lf::assemble::AssembleVectorLocally(1, dofh, my_vec_provider_edge, rhs_vec);

  return std::make_pair(A_aux.makeSparse(), rhs_vec);
}
```

(2-15.c) ⚀ (15 min.) By slightly modifying the `getGalerkinLSE()` create another function

```
template <typename FUNCT_F, typename FUNCT_H>
std::pair<Eigen::SparseMatrix<double>, Eigen::VectorXd>
   getGalerkinLSE_dropDof(
     const
        std::shared_ptr<lf::uscalfe::ScalarUniformFESpace<double>>
        fe_space,
     const FUNCT_F &f, const FUNCT_H &h);
```

(in the file `regularizedneumannproblem.h`) that returns the Galerkin matrix and right hand side vector of (2.15.1) discretized by means of the Finite element space

$$\mathcal{S}^0_{1,*}(\mathcal{M}) := \{\mathbf{v}_h \in \mathcal{S}^0_1(\mathcal{M}) : \mathbf{v}_h(\boldsymbol{p}) = 0\}. \tag{2.15.6}$$

Here $\boldsymbol{p} \in \mathbb{R}^2$ stands for the node of the mesh (`NODE` entity) whose associated nodal basis function of $\mathcal{S}^0_1(\mathcal{M})$ has global index number 0 (as fixed through the **lf::assemble::DofHandler** contained in `fe_space`).

Remark. Of course, any node of the mesh could be chosen as $\boldsymbol{p}$ in the definition of $\mathcal{S}^0_{1,*}$. That we use the one with index 0 is purely arbitrary.

HIDDEN HINT 1 for (2-15.c) → 2-15-3-0:rnh1.pdf

HIDDEN HINT 2 for (2-15.c) → 2-15-3-1:regNeu31.pdf

SOLUTION for (2-15.c) → 2-15-3-2:regNeu3.pdf ▲

(2-15.d) ⚂ (5 min.) Argue why the Galerkin matrix returned by `getGalerkinLSE_dropDof()` is invertible.

SOLUTION for (2-15.d) → 2-15-4-0:s3x.pdf ▲

It is not possible to find a global finite element basis for $\mathcal{S}_1^0(\mathcal{M}) \cap H^1_*(\Omega)$. Therefore the constraint in the definition of $H^1_*(\Omega)$ is attached to the variational formulation by means of a **Lagrange multiplier**, which leads to the augmented linear variational **saddle point problem** posed on $H^1(\Omega) \times \mathbb{R}$: Seek $u \in H^1(\Omega)$, $\lambda \in \mathbb{R}$ (Lagrange multiplier) such that

$$\begin{cases} \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \lambda \int_\Omega v \,\mathrm{d}\boldsymbol{x} &= \int_\Omega f v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h v \,\mathrm{d}S \quad \forall v \in H^1(\Omega)\,, \\ \int_\Omega u \,\mathrm{d}\boldsymbol{x} &= 0\,. \end{cases} \tag{2.15.8}$$

(2-15.e) ⚄ (15 min.) [depends on Sub-problem (2-15.b)]

Show that the solution of (2.15.1) on $H^1_*(\Omega)$ also solves (2.15.8).

SOLUTION for (2-15.e) → 2-15-5-0:regNeu4.pdf ▲

The Galerkin discretization of (2.15.8) by means of linear Lagrangian Finite elements relies on $\mathcal{S}_1^0(\mathcal{M}) \times \mathbb{R}$ as trial and test space, uses the basis $\{b_h^1, \ldots, b_h^N, 1\}$, $b_h^j \hat{=}$ tent functions, and leads to an $(N+1) \times (N+1)$ linear system of equations whose unknowns are the basis expansions coefficients of u_h and the Lagrange multiplier λ. ($N := \dim \mathcal{S}_1^0(\mathcal{M})$).

You may assume that the Lagrange multiplier λ is stored in the $N+1$-th component of the solution vector.

Then the FE Galerkin linear system for (2.15.8) has the structure

$$\begin{bmatrix} \mathbf{A} & \mathbf{c} \\ \mathbf{c}^T & 0 \end{bmatrix} \vec{\boldsymbol{\mu}} = \vec{\tilde{\boldsymbol{\varphi}}}, \tag{2.15.9}$$

where $\mathbf{A} \in \mathbb{R}^{N,N}$ is the Finite Element Galerkin matrix for (2.15.1) and $\mathbf{c} \in \mathbb{R}^N$ a suitable column vector.

(2-15.f) ⚄ (15 min.) Let $\mathbf{A} \in \mathbb{R}^{N,N}$ and $\vec{\boldsymbol{\varphi}} \in \mathbb{R}^N$ be the matrix and vector returned by Code 2.15.5. How do $\mathbf{A}$ and $\vec{\boldsymbol{\varphi}}$ have to be augmented to get the Galerkin matrix and right hand side vector for the linear variational problem (2.15.8) discretized by means of linear Lagrangian Finite elements?

HIDDEN HINT 1 for (2-15.f) → 2-15-6-0:regNeu51.pdf

SOLUTION for (2-15.f) → 2-15-6-1:regNeu5.pdf ▲

(2-15.g) ⚂ (20 min.) [depends on Sub-problem (2-15.f)]

Based on LEHRFEM++'s function `lf::assemble:AssembleVectorLocally()` implement a routine (in the file `regularizedneumannproblem.h`)

```
Eigen::VectorXd assembleVector_c(const lf::assemble::DofHandler
  &dofh);
```

that computes the vector $\mathbf{c} \in \mathbb{R}^N$ from (2.15.9). The `dofh` argument passes the local-to-global index mapping and mesh information for the finite element space $\mathcal{S}_1^0(\mathcal{M})$.

As an auxliary class that provides the **ENTITY_VECTOR_PROVIDER**, see [Lecture → Ex. 2.7.4.33], implement

```
class VecHelper {
 public:
  explicit VecHelper() {}
  bool isActive(const lf::mesh::Entity &entity) const;
  Eigen::Vector3d Eval(const lf::mesh::Entity &entity);
};
```

SOLUTION for (2-15.g) → 2-15-7-0:s5x.pdf ▲

(2-15.h) ⚀ (10 min.) [depends on Sub-problem (2-15.g)]

Modify the function `getGalerkinLSE()` from Code 2.15.5 into

```
template <typename FUNCT_F, typename FUNCT_H>
std::pair<Eigen::SparseMatrix<double>, Eigen::VectorXd>
  getGalerkinLSE_augment(
    const
       std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
       fe_space,
    const FUNCT_F &f, const FUNCT_H &h);
```

such that it returns the Galerkin matrix and right hand side vector of (2.15.8) discretized by means of piecewise linear Lagrangian finite elements.

HIDDEN HINT 1 for (2-15.h) → 2-15-8-0:regNeu61.pdf

SOLUTION for (2-15.h) → 2-15-8-1:regNeu6.pdf ▲

End Problem 2-15 , 105 min.

Problem 2-16: Rigidity of Piecewise Polynomial Continuous Functions

[Lecture → Section 2.4] and, particular, [Lecture → Section 2.6] probably created the impression that the construction of a viable finite element space is straightforward: one starts from a mesh, fixes a piecewise polynomial space and, finally, finds suitable locally supported basis functions. However, at each stage this procedure can fail, which is strikingly demonstrated in this problem.

This is a purely theoretical problem
▷ problem name/problem code folder: DegenerateFiniteElementSpace

Let $\mathcal{M} = \{K\}$ be a tensor product mesh of the rectangle domain $\Omega :=]a,b[\times]c,d[$ with N_x, N_y grid lines in x- and y-direction, respectively. See [Lecture → Fig. 93] in [Lecture → § 2.5.1.3] for an illustration. All cells (elements) are rectangles, there are $(N_x-1)(N_y-1)$ of them, and there are $N = N_x N_y$ vertices in the mesh.

(2-16.a) ⚄ (10 min.) Define the function space

$$W_N = \left\{ v \in H^1_0(\Omega) : \ v|_K \in \mathcal{P}_1(\mathbb{R}^2), \forall K \in \mathcal{M} \right\},$$

of piecewise linear functions (see [Lecture → Def. 2.5.2.2] for $p = 1$) on each element of $\mathcal{M}$ that are zero at the boundary. What is the dimension of W_N?

HIDDEN HINT 1 for (2-16.a) → 2-16-1-0:deg11.pdf

HIDDEN HINT 2 for (2-16.a) → 2-16-1-1:deg12.pdf

SOLUTION for (2-16.a) → 2-16-1-2:deg1.pdf ▲

(2-16.b) ⚄ (10 min.) Define the function space

$$V_N = \left\{ v \in H^1(\Omega) : \ v|_K \in \mathcal{P}_1(\mathbb{R}^2) \, \forall K \in \mathcal{M} \right\},$$

of piecewise linear functions on each element of $\mathcal{M}$. What is the dimension of V_N?

HIDDEN HINT 1 for (2-16.b) → 2-16-2-0:dfes2h1.pdf

SOLUTION for (2-16.b) → 2-16-2-1:deg2.pdf ▲

(2-16.c) ⚁ (2 min.) Define the function space

$$B_N = \left\{ v \in H^1(\Omega) : \ v|_K \in \mathcal{Q}_1(\mathbb{R}^2) \, \forall K \in \mathcal{M} \right\},$$

of piecewise bi-linear functions [Lecture → Def. 2.5.2.7] for $p = 1$) on each element of $\mathcal{M}$, see [Lecture → Def. 2.5.2.7]. What is the dimension of this B_N?

SOLUTION for (2-16.c) → 2-16-3-0:deg3.pdf ▲

(2-16.d) ⚄ (10 min.)

If we abandon nice "confoming" finite element meshes and even admit "hanging nodes", additional difficulties rear their heads. To appreciate this, now consider the non-conforming triangular mesh $\mathcal{M}$ of $\Omega =]0,1[^2$ displayed in Fig. 18. There, the hanging nodes are located on the midpoints of the edges of the other triangle.

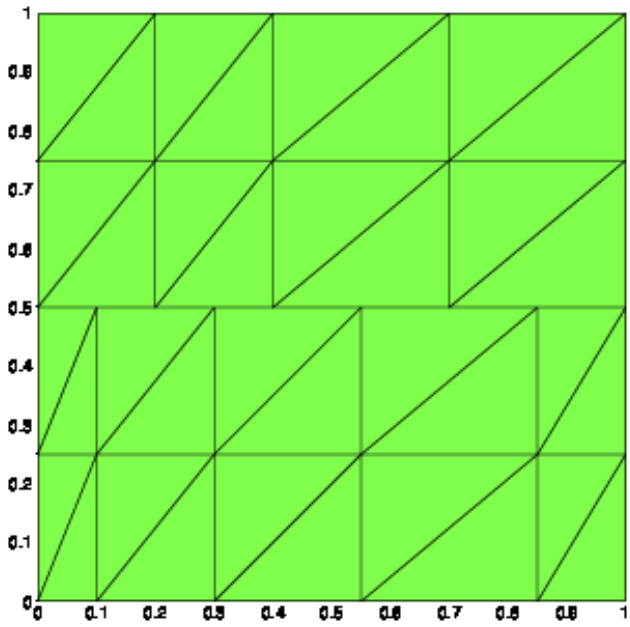


Fig. 18

Determine the dimension of the space

$$U_N = \left\{ v \in C^0(\overline{\Omega}) : \ v|_K \in \mathcal{P}_1(\mathbb{R}^2) \, \forall K \in \mathcal{M}, v|_{\partial\Omega} = 0 \right\} ,$$

and describe a basis of locally supported functions.

SOLUTION for (2-16.d) → 2-16-4-0:deg4.pdf ▲

(2-16.e) ⚄ (10 min.) What is the dimension of the space obtained from U_N by dropping the boundary condition $v|_{\partial\Omega} = 0$. Also in this case describe a basis and specify the supports of the basis functions.

SOLUTION for (2-16.e) → 2-16-5-0:deg5.pdf ▲

End Problem 2-16 , 42 min.

Problem 2-17: Local Computations for Convection Bilinear Form

This problem is concerned with a non-standard bilinear form arising from the weak formulation of so-called linear convection terms. In particular it studies the local computations needed for standard Galerkin finite-element discretization.

Requires familiarity with simple finite-element spaces, the concept of local shape functions and [Lecture → Ex. 2.7.4.14].
▷ problem name/problem code folder: ConvBLFMatrixProvider

Let $\Omega \subset \mathbb{R}^2$ be a bounded domain. For a given uniformly bounded vector field $\boldsymbol{a} : \Omega \to \mathbb{R}^2$ we consider the **convection bilinear form**

$$(u,v) \mapsto \mathsf{b}(u,v) := \int_\Omega \boldsymbol{a}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \tag{2.17.1}$$

for sufficiently smooth functions $u, v : \Omega \to \mathbb{R}$.

(2-17.a) ⚂ (10 min.) What are the largest possible Sobolev spaces V_0 and W_0 among $L^2(\Omega)$, $H^1(\Omega)$, and $H^2(\Omega)$ such that $\mathsf{b} : V_0 \times W_0 \to \mathbb{R}$ is *continuous*/bounded.

$V_0 =$ ☐ (space for u) ,

$W_0 =$ ☐ (space for v) .

(In each box, write one of $L^2(\Omega)$, $H^1(\Omega)$, or $H^2(\Omega)$.)

SOLUTION for (2-17.a) → 2-17-1-0:s1s.pdf

▲

Based on a 2D hybrid mesh $\mathcal{M}$ of Ω we perform the Galerkin discretization of b from (2.17.1) by choosing

$$u \in V_{0,h} := \mathcal{S}_1^0(\mathcal{M}) ,$$
$$v \in W_{0,h} := \mathcal{S}_0^{-1}(\mathcal{M}) ,$$

where $\mathcal{S}_1^0(\mathcal{M})$ is the space of $\mathcal{M}$-piecewise linear Lagrangian finite-element functions, and

$$\mathcal{S}_0^{-1}(\mathcal{M}) := \{v : \Omega \to \mathbb{R} \colon v|_K \in \mathcal{P}_0(\mathbb{R}^2)\ \forall K \in \mathcal{M}\}$$

stands for the space of $\mathcal{M}$-*piecewise constant* functions ($\mathcal{P}_0(\mathbb{R}^2)$ is the space of constant functions on $\mathbb{R}^2$).

(2-17.b) ⚂ (20 min.) What are the dimensions of $V_{0,h}$ and $W_{0,h}$ in terms of the numbers

- N_V of nodes of $\mathcal{M}$,
- N_e of edges of $\mathcal{M}$,
- N_T of triangles of $\mathcal{M}$,
- N_Q of quadrilaterals of $\mathcal{M}$?

$$\dim V_{0,h} = \square \cdot N_V + \square \cdot N_e + \square \cdot N_T + \square \cdot N_Q ,$$
$$\dim W_{0,h} = \square \cdot N_V + \square \cdot N_e + \square \cdot N_T + \square \cdot N_Q .$$

SOLUTION for (2-17.b) → 2-17-2-0:s2s.pdf ▲

As global shape functions we choose

- the "tent functions" for $V_{0,h} = \mathcal{S}_1^0(\mathcal{M})$, that is, the standard nodal basis,
- the characteristic functions of the cells of $\mathcal{M}$ for $W_{0,h} = \mathcal{S}_0^{-1}(\mathcal{M})$, that is, functions equal to 1 on a single cell and vanishing on all the others.

(2-17.c) ⚂ (20 min.) In a LEHRFEM++-based code local-to-global index mappings for $V_{0,h}$ and $W_{0,h}$ are encoded by means of **lf::assemble::UniformFEDofHandler** objects. Complete the following lines of C++ code meant to initialize such objects (`mesh_p` is a pointer to a **lf::mesh::Mesh** object for $\mathcal{M}$):

- For $V_{0,h}$

```
lf::assemble::UniformFEDofHandler dh_V(
mesh_p, {{lf::base::RefEl::kPoint(),  },
  {lf::base::RefEl::kSegment(),  },
  {lf::base::RefEl::kTria(),  },
  {lf::base::RefEl::kQuad(),  }});
```

- For $W_{0,h}$

```
lf::assemble::UniformFEDofHandler dh_W(
mesh_p, {{lf::base::RefEl::kPoint(),  },
  {lf::base::RefEl::kSegment(),  },
  {lf::base::RefEl::kTria(),  },
  {lf::base::RefEl::kQuad(),  }});
```

SOLUTION for (2-17.c) → 2-17-3-0:s2ax.pdf ▲

(2-17.d) ⚂ (20 min.)

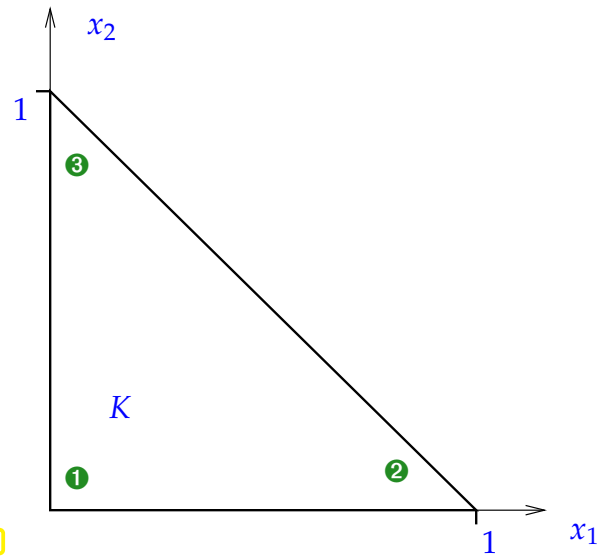


Fig. 20

Compute the element matrix $\mathbf{A}_K \in \mathbb{R}^{1,3}$ for the bilinear form b, with the constant vector field

$$\boldsymbol{a}(\boldsymbol{x}) \equiv \begin{bmatrix} 1 \\ 2 \end{bmatrix},$$

and the finite-element Galerkin discretization introduced above on the triangle K with vertices $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$, numbered as in the figure beside.

$$\mathbf{A}_K = \begin{bmatrix} \square & \square & \square \end{bmatrix}.$$

SOLUTION for (2-17.d) → 2-17-4-0:.pdf

▲

(2-17.e) ⚄ (20 min.) We implement an **ENTITY_MATRIX_PROVIDER** class for the finite-element Galerkin discretization of b as specified above:

C++ code 2.17.2: Definition of CDBLFElemMatProvider

```
template <typename MeshFunction>
class CDBLFElemMatProvider {
  public:
  // No default constructor
  CDBLFElemMatProvider() = delete;
  // Constructor takes a vectorfield argument
  explicit CDBLFElemMatProvider(MeshFunction av) : av_(std::move(av)) {}
  // The crucial interface methods for ENTITY_MATRIX_PROVIDER
  virtual bool isActive(const lf::mesh::Entity& /*cell*/) { return true; }
  Eigen::Matrix<double, 1, 3> Eval(const lf::mesh::Entity& tria);

  private:
  MeshFunction av_;
};
```

The data member `av_` will contain the coefficient $\boldsymbol{x} \mapsto \boldsymbol{a}(\boldsymbol{x})$ as a **MeshFunction** object, which is supposed to have an evaluation operator

```
std::vector<double>
  operator () (const lf::mesh::Entity &K,
               const Eigen::MatrixXd &refcoords);
```

Supply the missing parts of the following implementation of the crucial `Eval()` member function. It relies on the auxiliary function

```
std::pair<Eigen::Matrix<double, 2, 3>, double>
getGradBaryCoords(const lf::mesh::Entity &tria);
```

which returns, for a triangular cell passed in `tria`,

(i) a 2×3-matrix, whose columns contain the constant gradients of the barycentric coordinate functions,

(ii) the area of the triangle.

C++ code 2.17.3: Implementation of `Eval()` member function for CDBLFElemMatProvider

```
template <typename MeshFunction>
Eigen::Matrix<double, 1, 3> CDBLFElemMatProvider<MeshFunction>::Eval(
const lf::mesh::Entity& tria) {
  // Throw error in case no triangular cell
  LF_VERIFY_MSG(tria.RefEl() == lf::base::RefEl::kTria(),
  "Unsupported cell type " << tria.RefEl());
  // Fetch constant gradients of barycentric coordinate functions
  // and the area of the triangle
  auto [grad_bary_coords, area] = getGradBaryCoords(tria);
  // Reference coordinates of quadrature nodes
  const Eigen::MatrixXd refqrnodes{
    (Eigen::MatrixXd(2, 3) << 0.5, 0.5, 0.0, 0.0, 0.5, 0.5).finished()};
  std::vector<Eigen::Vector2d> av_values{av_(                ,                )};
```

```
14  Eigen::Matrix<double, 1, 3> elmat{Eigen::Matrix<double, 1, 3>::Zero()};
15  for (int i = 0; i < 3; ++i) {
16    for (int j =  0; j <     ; ++j) {
17      elmat(0, i)     (grad_bary_coords.col(     )).transpose()*av_values[     ];
18    }
19  }
20  return (                ) * elmat;
```

SOLUTION for (2-17.e) → 2-17-5-0:s4s.pdf

▲

End Problem 2-17, 90 min.

Problem 2-18: Nitsche's Method for Elliptic BVPs

This problem studies a non-standard variational formulation of a scalar linear 2nd-order elliptic boundary value problem. This formulation has practical relevance for cases when curved boundaries of the computational domain cannot be resolved by the finite-element mesh. However, in this problem this application is not discussed.

This problem does not involve any implementation, but expects familiarity with LEHRFEM++ nevertheless.

For the bounded computational domain $\Omega \subset \mathbb{R}^2$ and sufficiently smooth functions $u, v : \Omega \to \mathbb{R}$ we consider the bilinear form

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} (\mathbf{grad}\, u \cdot \boldsymbol{n}) v \,\mathrm{d}S - \int_{\partial\Omega} (\mathbf{grad}\, v \cdot \boldsymbol{n}) u \,\mathrm{d}S + c \int_{\partial\Omega} uv \,\mathrm{d}S\,, \tag{2.18.1}$$

and the linear form

$$\ell(v) := c \int_{\partial\Omega} gv \,\mathrm{d}S - \int_{\partial\Omega} (\mathbf{grad}\, v \cdot \boldsymbol{n}) g \,\mathrm{d}S\,, \tag{2.18.2}$$

where $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ is the exterior unit normal vector field on $\partial\Omega$, $c \in \mathbb{R}$, and $g \in C^0(\partial\Omega)$.

(2-18.a) ⚀ (20 min.) In the case that Ω is the unit disk $\Omega = \{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\| < 1\}$, show that a is *not continuous* on $H^1(\Omega)$.

HIDDEN HINT 1 for (2-18.a) → 2-18-1-0:nmh1.pdf

SOLUTION for (2-18.a) → 2-18-1-1:nms1.pdf ▲

(2-18.b) ⚁ (25 min.) State a second-order elliptic boundary value problem whose solution u, if it belongs to $H^2(\Omega)$, satisfies

$$\mathsf{a}(u,v) = \ell(v) \quad \forall v \in C^\infty(\overline{\Omega})\,. \tag{2.18.4}$$

PDE: [ ] in Ω,

Boundary conditions: [ ] on $\partial\Omega$.

HIDDEN HINT 1 for (2-18.b) → 2-18-2-0:nmsp2h1.pdf

SOLUTION for (2-18.b) → 2-18-2-1:nms2.pdf ▲

Now we restrict ourselves to bounded polygons Ω and assume that Ω is equipped with a *triangular* finite-element mesh $\mathcal{M}$. We consider the discrete variational problem:

$$u_h \in \mathcal{S}_1^0(\mathcal{M}):\quad \mathsf{a}(u_h, v_h) = \ell(v_h) \quad \forall v_h \in \mathcal{S}_1^0(\mathcal{M})\,.$$

(2-18.c) ⚂ (30 min.)

Compute the $\mathcal{S}_1^0(\mathcal{M})$-element matrices $\in \mathbb{R}^{3,3}$ for three different parts of the bilinear form a defined in (2.18.1), the full bilinear form, and for the triangle K with vertices $\begin{bmatrix}0\\0\end{bmatrix}$, $\begin{bmatrix}1\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\end{bmatrix}$, numbered as in the figure beside.

The edge e connecting vertices ❷ and ❸ lies on $\partial\Omega$: $e \subset \partial\Omega$. All other edges are located in the interior of Ω.

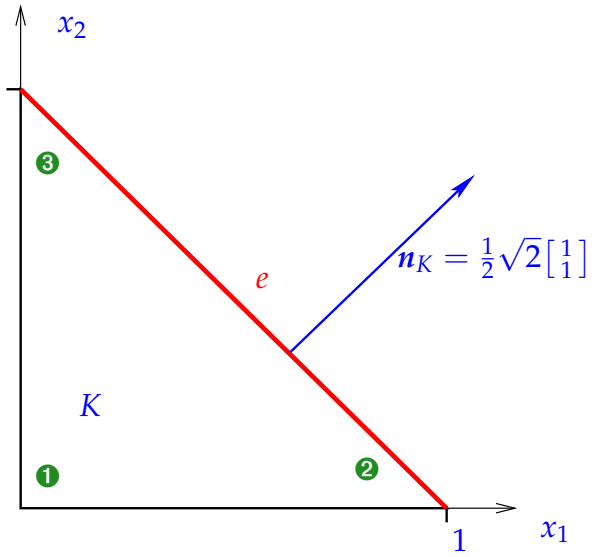


Fig. 21

Element matrix

for $(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}x$:

$$\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix},$$

for $(u,v) \mapsto \int_{\partial\Omega} (\mathbf{grad}\, u \cdot \boldsymbol{n}) v \,\mathrm{d}S$:

$$\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix},$$

for $(u,v) \mapsto \int_{\partial\Omega} uv \,\mathrm{d}S$:

$$\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix},$$

for $(u,v) \mapsto \mathsf{a}(u,v)$:

$$\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}.$$

SOLUTION for (2-18.c) → 2-18-3-0:nms3.pdf ▲

(2-18.d) ⚄ (30 min.) The C++ function `assembleNitscheGalerkinMatrix()` listed as Code 2.18.10 is supposed to compute the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix for the bilinear form a from (2.18.1) in EIGEN's CRS format. To that end it relies on the **ENTITY_MATRIX_PROVIDER** class **NitscheBoundaryMatProvider** shown as Code 2.18.9.

Explain, why the function `assembleNitscheGalerkinMatrix()` *can never work* no matter what is implemented in parts of the codes not shown here.

C++ code 2.18.9: Definition/implementation of NitscheBoundaryMatProvider ↪ GitLab

```
class NitscheBoundaryMatProvider {
 public:
  NitscheBoundaryMatProvider(lf::mesh::utils::CodimMeshDataSet<bool> &bd_flags,
                             double c)
      : bd_flags_(bd_flags), c_(c) {}
  virtual ~NitscheBoundaryMatProvider() = default;
  [[nodiscard]] virtual bool isActive(const lf::mesh::Entity &edge) const {
    return bd_flags_(edge);
  }
  [[nodiscard]] Eigen::Matrix2d Eval(const lf::mesh::Entity &edge) const;

 private:
  lf::mesh::utils::CodimMeshDataSet<bool> &bd_flags_;
  double c_;
};
```

C++ code 2.18.10: Implementation of `assembleNitscheGalerkinMatrix()` ↪ GitLab

```
Eigen::SparseMatrix<double> assembleNitscheGalerkinMatrix(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> lin_fes_p,
    double c) {
  // (Pointer to) underlying mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p{lin_fes_p->Mesh()};
  // Obtain local-to-global index mapper
  const lf::assemble::DofHandler &dofh{lin_fes_p->LocGlobMap()};
  // Provider object for element matrices for negative Laplacian and linear
  // Lagrangian finite elements
  lf::uscalfe::LinearFELaplaceElementMatrix lapl_elmat_builder{};
  // Object for sparse matrix to be filled by cell-oriented assembly
  const int N_dofs = dofh.NumDofs();
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Invoke cell-oriented assembly for the volume part of the finite-element
  // Galerkin matrix
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, lapl_elmat_builder, A);
  // Flag all edge (co-dimension-1 entities) on the boundary
  lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
  NitscheBoundaryMatProvider nitsche_bd_builder(bd_flags, c);
  // Add boundary contributions to Galerkin matrix stored in A
  // Assembly covers edges (co-dimensions-1 entities)!
  lf::assemble::AssembleMatrixLocally(1, dofh, dofh, nitsche_bd_builder, A);
  // Convert the matrix from triplet format to CRS format
  return A.makeSparse();
}
```

SOLUTION for (2-18.d) → `2-18-4-0:nm4s.pdf` ▲

(2-18.e) ⚀ (90 min.) A correct implementation of a **ENTITY_MATRIX_PROVIDER** class for $\mathcal{S}_1^0(\mathcal{M})$ and the bilinear form a from (2.18.1) is provided by the class **LinearFENitscheElementMatrix**, whose definition is given in Code 2.18.11. The function `computeNitscheGalerkinMatrix()` from Code 2.18.12, implemented in `nitschemethod.cc` demonstrates its use.

C++ code 2.18.11: Definition/implementation of LinearFENitscheElementMatrix ↪ GitLab

```
class LinearFENitscheElementMatrix {
 public:
```

```
  LinearFENitscheElementMatrix(
      lf::mesh::utils::CodimMeshDataSet<bool> &bd_flags, double c)
      : bd_flags_(bd_flags), c_(c) {}
  virtual ~LinearFENitscheElementMatrix() = default;
  [[nodiscard]] virtual bool isActive(const lf::mesh::Entity & /*cell*/) const {
    return true;
  }
  [[nodiscard]] Eigen::Matrix3d Eval(const lf::mesh::Entity &cell) const;

 private:
  lf::mesh::utils::CodimMeshDataSet<bool> &bd_flags_;
  double c_;
  // Constant matrices for use in Eval()
  const Eigen::MatrixXd c_hat_ = Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
  const Eigen::MatrixXd G_hat_ =
      (Eigen::Matrix<double, 2, 3>() << -1.0, 1.0, 0.0, -1.0, 0.0, 1.0)
          .finished();
  const Eigen::Matrix3d L_ =
      (Eigen::Matrix3d() << 0.5, 0.0, 0.5, 0.5, 0.5, 0.0, 0.0, 0.5, 0.5)
          .finished();
};
```

C++ code 2.18.12: Implementation of `computeNitscheGalerkinMatrix()` ⇢ GitLab

```
Eigen::SparseMatrix<double> computeNitscheGalerkinMatrix(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> lin_fes_p,
    double c) {
  // Pointer to underlying mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p{lin_fes_p->Mesh()};
  // Obtain local-to-global index mapper "D.o.f. handler"
  const lf::assemble::DofHandler &dofh{lin_fes_p->LocGlobMap()};
  // Flag all edge (co-dimension-1 entities) on the boundary
  lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
  // Provider object for element matrices
  LinearFENitscheElementMatrix nitsche_mat_builder(bd_flags, c);
  // Object for sparse matrix to be filled by cell-oriented assembly
  const int N_dofs = dofh.NumDofs();
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Cell-oriented assembly
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, nitsche_mat_builder, A);
  return A.makeSparse();
}
```

Devise an implementation of the C++Eval() member function of **LinearFENitscheElementMatrix**

```
Eigen::Matrix3d LinearFENitscheElementMatrix::Eval(
const lf::mesh::Entity &cell) const;
```

that complies with the LEHRFEM++ concept of an **ENTITY_MATRIX_PROVIDER**.

HIDDEN HINT 1 for (2-18.e) → 2-18-5-0:nelmh1.pdf

HIDDEN HINT 2 for (2-18.e) → 2-18-5-1:nelh2.pdf

SOLUTION for (2-18.e) → 2-18-5-2:.pdf ▲

(2-18.f) ⚀ (30 min.) [depends on Sub-problem (2-18.e)]

The following class fits the concept of **ENTITY_VECTOR_PROVIDER** ⇢ (documentation) and is sup-

posed to supply the $\mathcal{S}_1^0(\mathcal{M})$ element vectors spawned by the right-hand-side linear functional $v \mapsto \ell(v)$ as defined in (2.18.2) through its `Eval()` member function.

C++ code 2.18.15: Definition of class NitscheElemVecProvider. ➙ GitLab

```
template <typename FUNCTOR>
class NitscheElemVecProvider {
 public:
  NitscheElemVecProvider(FUNCTOR g,
                         lf::mesh::utils::CodimMeshDataSet<bool> &bd_flags,
                         double c)
      : g_(g), bd_flags_(bd_flags), c_(c) {}
  virtual ~NitscheElemVecProvider() = default;
  [[nodiscard]] virtual bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  [[nodiscard]] Eigen::Vector3d Eval(const lf::mesh::Entity &cell) const;

 private:
  FUNCTOR g_;  // x ↦ g(x)
  // Flags marking edges on the boundary
  lf::mesh::utils::CodimMeshDataSet<bool> &bd_flags_;
  double c_;  // Penalty parameter c
  // Reference coordinate of barycentre of a triangle
  const Eigen::MatrixXd c_hat_ = Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
  // Gradients of reference shape functions
  const Eigen::MatrixXd G_hat_ =
      (Eigen::Matrix<double, 2, 3>() << -1.0, 1.0, 0.0, -1.0, 0.0, 1.0)
          .finished();
};
```

The following implementation of the `Eval()` member function of **NitscheElemVecProvider** is *flawed*!

C++ code 2.18.16: `Eval()` member function of class NitscheElemVecProvider: Implementation with errors! ➙ GitLab

```
template <typename FUNCTOR>
Eigen::Vector3d NitscheElemVecProvider<FUNCTOR>::Eval(
    const lf::mesh::Entity &cell) const {
  // Throw error in case no triangular cell
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type " << cell.RefEl());
  // Fetch geometry object for current cell
  const lf::geometry::Geometry &K_geo{*(cell.Geometry())};
  const Eigen::MatrixXd cell_pts{lf::geometry::Corners(K_geo)};
  LF_ASSERT_MSG(K_geo.DimGlobal() == 2, "Mesh must be planar");
  // Obtain physical coordinates of barycenter of triangle
  const Eigen::Vector2d center{K_geo.Global(c_hat_).col(0)};
  // Compute gradients of barycentric coordinate functions
  // Transformation matrix for gradients on reference triangle
  const Eigen::Matrix2d JinvT(K_geo.JacobianInverseGramian(c_hat_));
  // Transform gradients
  const Eigen::Matrix<double, 2, 3> G = JinvT * G_hat_;
  // Element vector
  Eigen::Vector3d el_vec = Eigen::Vector3d::Zero();
  // Loop over edges and check whether they are
  // located on the bondary
  std::span<const lf::mesh::Entity *const> edges{cell.SubEntities(1)};
  for (int k = 0; k < 3; ++k) {
```

```
    if (bd_flags_(*edges[k])) {
      // Edge with local index k is an edge on the boundary
      // Fetch the coordinates of its endpoints
      const lf::geometry::Geometry &ed_geo{*(edges[k]->Geometry())};
      const Eigen::MatrixXd ed_pts{lf::geometry::Corners(ed_geo)};
      // Evaluate Dirichlet data function in the endpoints
      const double g0 = g_(ed_pts.col(0));
      const double g1 = g_(ed_pts.col(1));

      // I: Contribution from consistency correction
      // Direction vector of the edge
      const Eigen::Vector2d dir = ed_pts.col(1) - ed_pts.col(0);
      // Rotate counterclockwise by 90 degrees
      const Eigen::Vector2d ed_normal = Eigen::Vector2d(dir(1), -dir(0));
      // For adjusting direction of normal
      const int ori = (ed_normal.dot(center - ed_pts.col(0)) > 0) ? -1 : 1;
      // Dirichlet data assumed to be piecewise linear
      el_vec -= 0.5 * (g0 + g1) * (G.transpose() * (ori * ed_normal));

      // II: Contribution from penalty correction
      const double fac = c_ * dir.norm();
      const int l = (k + 1) % 3;
      el_vec[k] += fac * (g0 / 3.0 + g1 / 6.0);
      el_vec[l] += fac * (g1 / 3.0 + g0 / 6.0);
    }
  }
  return el_vec;
}
```

Fix the error!

SOLUTION for (2-18.f) → 2-18-6-0:.pdf ▲

Running the main executable ➙ GitLab uses the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin discretization of (2.18.1), (2.18.2) to solve Dirichlet boundary for $-\Delta u = 0$ on the unit square with Dirichlet data chosen such that the exact solution will be the harmonic polynomial $u(\boldsymbol{x}) = x_1^2 - x_2^2$. The Galerkin solutions $u_\ell \in \mathcal{S}_1^0(\mathcal{M}_\ell)$ are computed for a sequence of triangular meshes $\mathcal{M}_0 \prec \mathcal{M}_1 \prec \cdots \prec \mathcal{M}_6$ generated by uniform regular refinement.

$\sharp\mathcal{M}$	$\|u-u_\ell\|_{L^2(\Omega)}$	$\lvert u-u_\ell\rvert_{H^1(\Omega)}$
15	0.0233035	0.371194
60	0.00585961	0.18121
240	0.0013711	0.0898262
960	0.000322264	0.0447222
3840	7.72974e-05	0.022314
15360	1.88665e-05	0.0111452
61440	4.65618e-06	0.00556968

The displays the $L^2(\Omega)$- and $H^1(\Omega)$-seminorms of the Galerkin discretization errors. From row to row the meshwidth h is halved. The penalty parameter c is set to $\sim h^{-1}$.

We observe that the $L^2(\Omega)$-norm of the FE discretization error is reduced by a factor of about 4 and that the $H^1(\Omega)$-seminorm drops by a factor of about 2 when the mesh is refined.

As explained in [Lecture → Section 3.2.2] this hints at **algebraic convergence**

$$\|u-u_\ell\|_{L^2(\Omega)} = O(h^2) \quad \text{as} \quad h \to 0 \quad \text{(rate 2)}\,,$$
$$\lvert u-u_\ell\rvert_{H^1(\Omega)} = O(h) \quad \text{as} \quad h \to 0 \quad \text{(rate 1)}\,.$$

End Problem 2-18 , 225 min.

Problem 2-19: Lagrangian Finite Elements on Criss-Cross Meshes

In this problem we study Lagrangian finite element spaces [Lecture → Section 2.6] and related stiffness matrices on a special family of meshes.

Th problem assumes knowledge about the contents of [Lecture → Section 2.6].
▷ problem name/problem code folder: FESpacesCrissCross

The figure beside shows a so-called **criss-cross mesh** $\mathcal{M}_M$ of the unit square $\Omega :=]0,1[^2$, which arises from splitting every square of an $M \times M$, $M \in \mathbb{N}$, $M \geq 3$, tensor-product mesh into four congruent isosceles triangles.

criss-cross mesh $\mathcal{M}_M$ ▷

Marked in magenta in the figure is the part of the boundary

$$\Gamma_D := (\{0\} \times [0,1]) \cup (\{1\} \times [0,1]) .$$

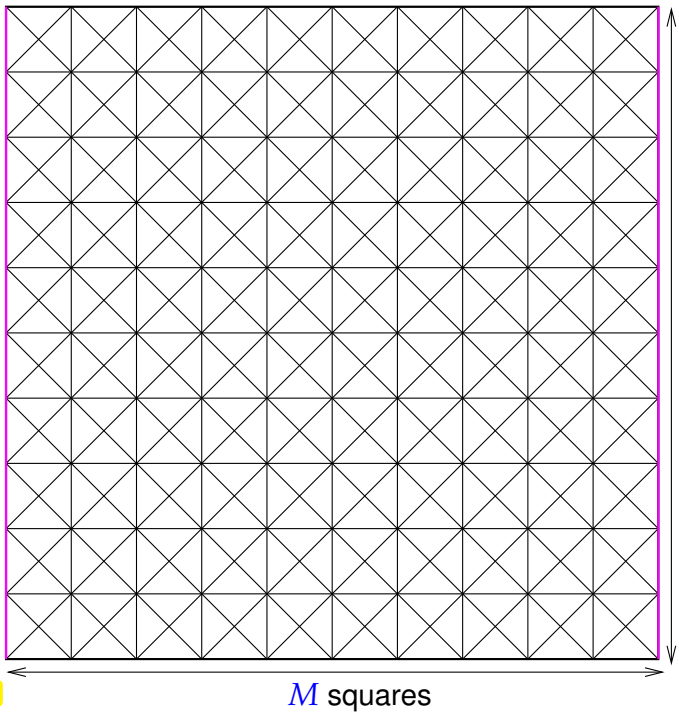


Fig. 22

(2-19.a) ⚀ (20 min.) What are the dimensions of the following Lagrange finite-element spaces on the criss-cross mesh $\mathcal{M}_M$ of Fig. 22, $M \in \mathbb{N}$, $M \geq 3$?

$$\dim \mathcal{S}_1^0(\mathcal{M}_M) = \quad , \tag{2.19.1}$$

$$\dim \mathcal{S}_2^0(\mathcal{M}_M) = \quad , \tag{2.19.2}$$

$$\dim V = \quad , \quad V := \{v_h \in \mathcal{S}_1^0(\mathcal{M}_M) : \ v_h|_{\Gamma_D} = 0\} , \tag{2.19.3}$$

$$\dim W = \quad , \quad W := \{v_h \in \mathcal{S}_2^0(\mathcal{M}_M) : \ v_h|_{\Gamma_D} = 0\} . \tag{2.19.4}$$

SOLUTION for (2-19.a) → 2-19-1-0:ccs1.pdf ▲

From now consider the discrete variational problem

$$u \in V: \quad \int_\Omega u(\boldsymbol{x})v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \quad \forall v \in V , \tag{2.19.5}$$

where V is defined in (2.19.3).

(2-19.b) ⚄ (30 min.) Compute the following (maximal/minimal) numbers of non-zero elements of (rows of) the Galerkin matrix $\mathbf{A}$ for (2.19.5) in terms of M, assuming that the customary minimally

locally supported "tent" basis functions [Lecture → Section 2.4.3] are used.

$$\max\{\sharp\{j \in \{1,\ldots,\dim V\} : (\mathbf{A})_{i,j} \neq 0\}, i \in \{1,\ldots,\dim V\}\} = \boxed{},$$

$$\min\{\sharp\{j \in \{1,\ldots,\dim V\} : (\mathbf{A})_{i,j} \neq 0\}, i \in \{1,\ldots,\dim V\}\} = \boxed{},$$

$$\operatorname{nnz}(\mathbf{A}) := \sharp\{(i,j) \in \{1,\ldots,\dim V\}^2 : (\mathbf{A})_{i,j} \neq 0\}$$
$$= \boxed{}.$$

SOLUTION for (2-19.b) → 2-19-2-0:ccs2.pdf ▲

(2-19.c) ⚄ (15 min.) What is the number of non-zero entries of the right-hand-side vector $\vec{\varphi}$ of the linear system of equations arising from (2.19.5) when using the customary minimally locally supported "tent" basis functions of V [Lecture → Section 2.4.3]?

$$\operatorname{nnz}(\vec{\varphi}) = \sharp\{i \in \{1,\ldots,\dim V\} : (\vec{\varphi})_i \neq 0\} = \boxed{}.$$

SOLUTION for (2-19.c) → 2-19-3-0:cc3s.pdf ▲

End Problem 2-19, 65 min.

Problem 2-20: `DofHandler` and Assembly

A key component in the cell-oriented assembly of finite-element Galerkin matrices is a local→global index mapping/d.o.f. handler, in LehrFEM++ encoded in **lf::assemble::DofHandler** objects. In this problem we consider such an object for a concrete small mesh and try to figure out what the member functions will return.

This problem is linked to [Lecture → Section 2.7.4.2], in particular [Lecture → § 2.7.4.13]. It requires knowledge about LehrFEM++.
▷ problem name/problem code folder: DOFHandlerAssembly

Assume that in a LehrFEM++-based C++ code the variable `mesh_p` holds a pointer to an object of type **lf::mesh::hybrid2d::Mesh**, which describes the following *2D triangular mesh* with 15 cells, 27 edges, and 13 nodes for the square domain $\Omega :=]0,3[^2$.

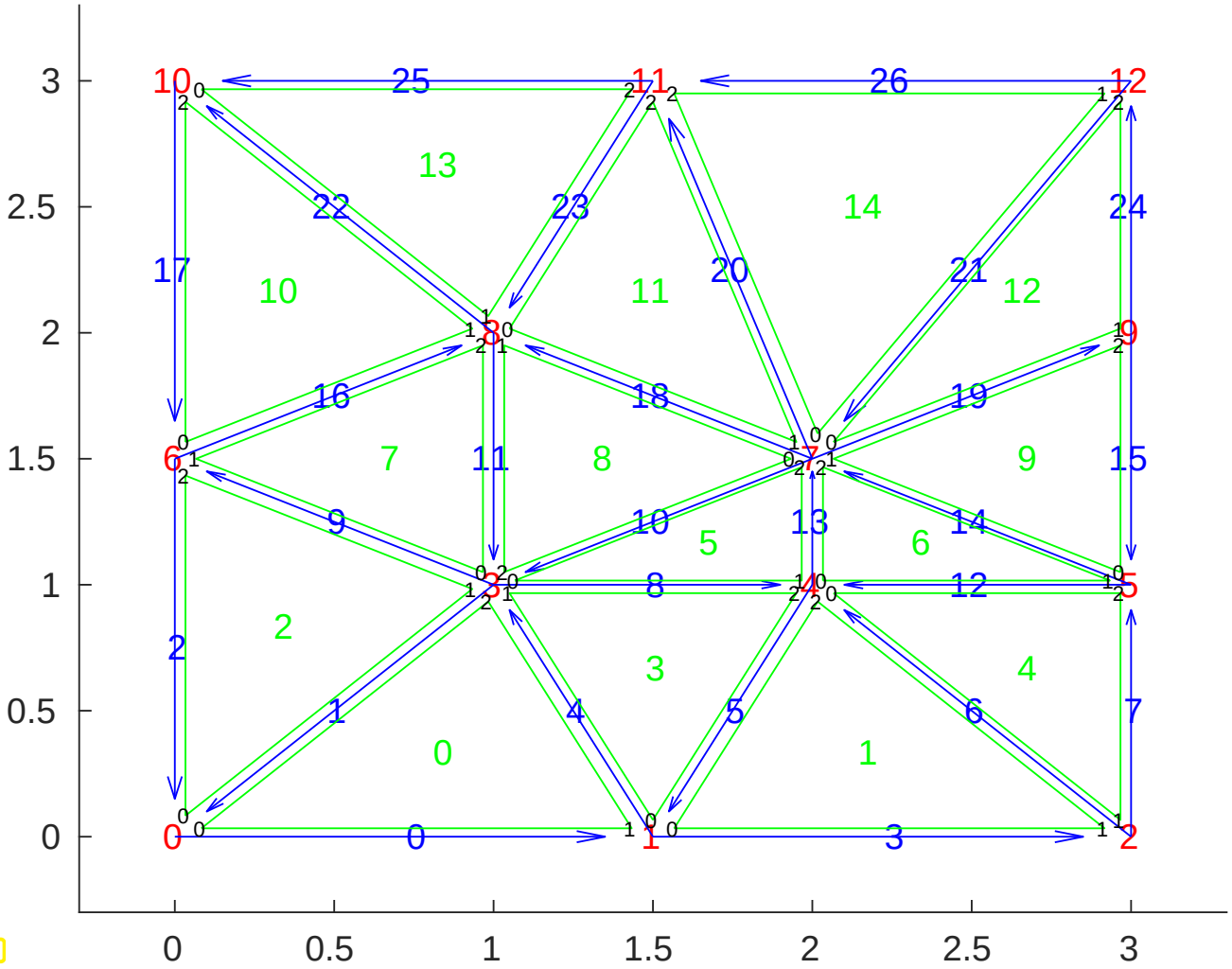


Fig. 25

The big red numbers $\in \{0,\ldots,12\}$ give the global index numbers of the mesh nodes as returned by the `Index()` member function of **lf::mesh::Mesh**. The blue numbers $\in \{0,\ldots,26\}$ on top of mesh edges give their respective global index numbers, while the green numbers in the center of triangles tell the global index numbers of the mesh cells. The small black numbers at the vertices of every triangle provide their local indices within that triangle.

In the code we initialize the d.o.f. handler object `dofh` as follows:

```
lf::assemble::UniformFEDofHandler dofh(
mesh_p, {{lf::base::RefEl::kPoint(), 0},
         {lf::base::RefEl::kSegment(), 1},
         {lf::base::RefEl::kTria(), 2},
         {lf::base::RefEl::kQuad(), 2}});
```

Moreover, below the variable `cell_p`, whenever it occurs, holds a pointer to an object of type **lf::mesh::hybrid2d::Triangle**, which means that

```
cell_p.RefEl() == lf::base::RefEl::kTria()
```

evaluates to **true**.

(2-20.a) ⚄ (15 min.) What integer numbers will the following function calls return?

- `dofh.NumDofs()` = ☐,
- `dofh.NumLocalDofs(*cell_p)` = ☐,
- `dofh.NumInteriorDofs(*cell_p)` = ☐.

SOLUTION for (2-20.a) → 2-20-1-0:dfog.pdf ▲

(2-20.b) ⚅ (15 min.) Now assume that `cell_p` points to the triangle object for *cell no. 8* of the given mesh of Fig. 25, that is `mesh_p->Index(*cell_p)` returns `8`. What sequences of integers $\in \mathbb{N}_0$ are returned by the following function calls? Write them as comma-separated numbers.

- `dofh.GlobalDofIndices(*cell_p)` = (☐)
- `dofh.InteriorGlobalDofIndices(*cell_p)` = (☐).

HIDDEN HINT 1 for (2-20.b) → 2-20-2-0:nce.pdf

HIDDEN HINT 2 for (2-20.b) → 2-20-2-1:dofh31.pdf

SOLUTION for (2-20.b) → 2-20-2-2:.pdf ▲

End Problem 2-20 , 30 min.

Problem 2-21: Blended Parametric Representation of Curvi-Linear triangles

This problem explores a new way for the construction of parametric finite elements with "arbitrary" element shapes.

This problem relates to [Lecture → Section 2.8] and relies on EIGEN for an implementation in C++.
▷ problem name/problem code folder: BlendedParameterization

The figure beside displays a so-called curvi-linear triangle that is a bounded domain $K \subset \mathbb{R}^2$ with three corners $\boldsymbol{a}^0$, $\boldsymbol{a}^1$, and $\boldsymbol{a}^2$, joined by non-intersecting curves γ_{01}, γ_{12}, and γ_{20}.

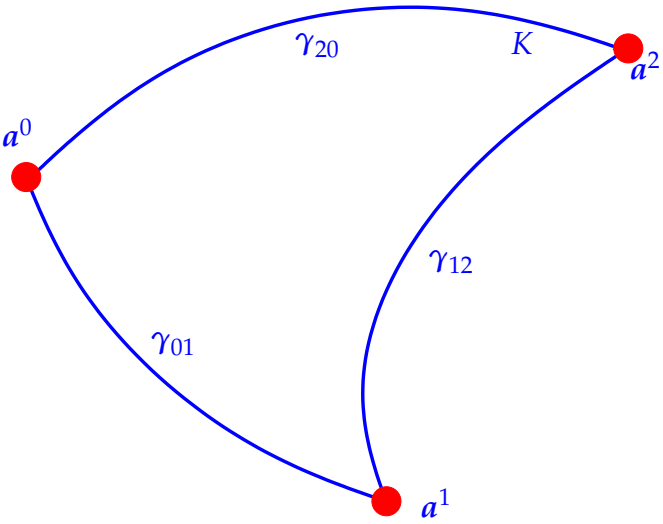


Fig. 27

These three curves are given by smooth parameterizations:

$$\gamma_{01}, \gamma_{12}, \gamma_{20} : [0,1] \to \mathbb{R}^2$$

such that

$$\begin{aligned} \gamma_{01}(0) &= \boldsymbol{a}^0 , \quad & \gamma_{01}(1) &= \boldsymbol{a}^1 , \\ \gamma_{12}(0) &= \boldsymbol{a}^1 , \quad & \gamma_{12}(1) &= \boldsymbol{a}^2 , \\ \gamma_{20}(0) &= \boldsymbol{a}^2 , \quad & \gamma_{20}(1) &= \boldsymbol{a}^0 . \end{aligned}$$

Based on this description of the curvi-linear triangle, we introduce the mapping $\boldsymbol{\Phi} : [0,1]^2 \to \mathbb{R}^2$,

$$\begin{aligned} \boldsymbol{\Phi}(\widehat{\boldsymbol{x}}) := \gamma_{01}(\widehat{x}_1) + \gamma_{20}(1-\widehat{x}_2) - \gamma_{01}(0) + \widehat{x}_1\gamma_{12}(\widehat{x}_2) + \widehat{x}_2\gamma_{12}(1-\widehat{x}_1) - \\ \widehat{x}_1(\gamma_{01}(1-\widehat{x}_2) + \gamma_{20}(1-\widehat{x}_2) - \gamma_{01}(0)) - \widehat{x}_2(\gamma_{01}(\widehat{x}_1) + \gamma_{20}(\widehat{x}_1) - \gamma_{01}(0)) , \end{aligned} \tag{2.21.1}$$

for $\widehat{\boldsymbol{x}} = [\widehat{x}_1, \widehat{x}_2]^\top \in [0,1]^2$.

(2-21.a) ⚀ (20 min.) Show that $\boldsymbol{\Phi}$ from (2.21.1) satisfies

$$\boldsymbol{\Phi}\left(\begin{bmatrix} \xi \\ 0 \end{bmatrix}\right) = \gamma_{01}(\xi) , \quad 0 \le \xi \le 1 . \tag{2.21.2}$$

SOLUTION for (2-21.a) → 2-21-1-0:blsp1s.pdf ▲

(2-21.b) ⚁ (10 min.) Give a geometric interpretation of (2.21.2).

SOLUTION for (2-21.b) → 2-21-2-0:blsp2.pdf ▲

(2-21.c) ⚂ (10 min.) We write $\widehat{K}$ for the (reference) triangle with vertices $\begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$, and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$.

State a formula analogous to (2.21.2) that expresses the fact that $\boldsymbol{\Phi}$ from (2.21.1) maps the edge of $\widehat{K}$ connecting $\begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$ to the edge of K connecting $\boldsymbol{a}^1$ and $\boldsymbol{a}^2$.

SOLUTION for (2-21.c) → 2-21-3-0:bl4s.pdf ▲

(2-21.d) ⚃ (15 min.) We write $\dot{\gamma}_*$ for the derivative of γ_* with respect to its argument, given $* \in \{01, 12, 20\}$.

Using this notation give a formula for the Jacobian $\mathrm{D}\boldsymbol{\Phi} : [0,1]^2 \to \mathbb{R}^{2,2}$ of $\boldsymbol{\Phi}$.

HIDDEN HINT 1 for (2-21.d) → 2-21-4-0:bl5h1.pdf

SOLUTION for (2-21.d) → 2-21-4-1:bl5s.pdf ▲

In some EIGEN-based C++ code curves $[0,1] \mapsto \mathbb{R}^2$ are represented by child classes of the following virtual base class:

```
class Curve {
  public:
  virtual Eigen::Vector2d operator()(double parameter) const = 0;
  virtual Eigen::Vector2d derivative(double parameter) const = 0;
};
```

The **operator()** returns a point on the curve when given a parameter $\in [0,1]$, whereas the method `derivative()` returns the derivative vector.

(2-21.e) ⚄ (30 min.) [depends on Sub-problem (2-21.d)]

In the file `blendedparameterization.cc` implement a C++ function

```
Eigen::MatrixXd JacobianPhi(const Eigen::Vector2d& point,
const Curve& gamma01, const Curve& gamma12, const Curve& gamma20)
```

that provides the Jacobian of $\boldsymbol{\Phi}$ from (2.21.1) when one supplies the coordinates of a point $\in [0,1]^2$, given γ_{01}, γ_{12}, and γ_{20}.

SOLUTION for (2-21.e) → 2-21-5-0:blsp6.pdf ▲

(2-21.f) ⚄ (60 min.) [depends on Sub-problem (2-21.e)]

The observations made in Sub-problem (2-21.b) and Sub-problem (2-21.c) hint that $\boldsymbol{\Phi}(\widehat{K}) = K$, which we are going to assume.

In the file `blendedparameterization.cc` complete the implementation of a C++ function

```
Eigen::MatrixXd evalBlendLocMat(const Curve& gamma01,
                                const Curve& gamma12,
                                const Curve& gamma20);
```

that *approximately* $(*)$ computes the **element matrix** on K for the bilinear form

$$(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$$

for the Laplacian $-\Delta$ on K using parametric linear Lagrangian finite elements based on the parametrization of K through $\boldsymbol{\Phi}$ from (2.21.1). Function arguments `gamma01`, `gamma12`, and `gamma20` of type `const Curve&` specify the curves γ_{01}, γ_{12}, and γ_{20}, respectively. As explained above these curves bound K.

$(*)$ Use the following quadrature rule to approximately evaluate any integral over a triangle: for a generic triangle T with vertices $\boldsymbol{x},\boldsymbol{y},\boldsymbol{z} \in \mathbb{R}^2$, use

$$\int_T f \,\mathrm{d}\boldsymbol{x} \approx \frac{|T|}{3}\left(f\left(\tfrac{1}{2}(\boldsymbol{x}+\boldsymbol{y})\right) + f\left(\tfrac{1}{2}(\boldsymbol{y}+\boldsymbol{z})\right) + f\left(\tfrac{1}{2}(\boldsymbol{z}+\boldsymbol{x})\right)\right).$$

HIDDEN HINT 1 for (2-21.f) → 2-21-6-0:bl7h1.pdf

SOLUTION for (2-21.f) → 2-21-6-1:bl7.pdf ▲

End Problem 2-21, 145 min.

Problem 2-22: Constrained Neumann Problem

This problem deals with a scalar second-order elliptic boundary with non-standard "Neumann-type" boundary conditions.

This problem draws on techniques from [Lecture → Chapter 1], [Lecture → Chapter 2], and [Lecture → Chapter 3]. It involves implementation in LEHRFEM++.
▷ problem name/problem code folder: SolAvgBoundary

For a connected bounded Lipschitz polygon $\Omega \subset \mathbb{R}^2$ define the Sobolev space

$$H^1_\partial(\Omega) := \{v \in H^1(\Omega) : \int_{\partial\Omega} v \,\mathrm{d}S = 0\} \,. \tag{2.22.1}$$

On this Sobolev space we consider the variational problem

$$u \in H^1_\partial(\Omega): \quad \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_\partial(\Omega) \,, \tag{2.22.2}$$

with source function $f \in L^2(\Omega)$.

(2-22.a) ⚀ (15 min.) Discuss the uniqueness of solutions of (2.22.2).

HIDDEN HINT 1 for (2-22.a) → 2-22-1-0:sabh1.pdf

SOLUTION for (2-22.a) → 2-22-1-1:sab0.pdf ▲

(2-22.b) ⚅ (15 min.) State the second-order elliptic boundary value problem for which (2.22.2) is the weak formulation.

SOLUTION for (2-22.b) → 2-22-2-0:asb1.pdf ▲

Since there is no simple finite element subspace of $H^1_\partial(\Omega)$, the following **augmented variational formulation** is often used:

$$\begin{array}{c} u \in H^1(\Omega) \\ \eta \in \mathbb{R} \end{array} : \quad \begin{array}{lll} \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \eta\, v \,\mathrm{d}\boldsymbol{x} &= \int_\Omega f v \,\mathrm{d}\boldsymbol{x} & \forall v \in H^1(\Omega) \,, \\ \int_{\partial\Omega} \mu\, u \,\mathrm{d}\boldsymbol{x} &= 0 & \forall \mu \in \mathbb{R} \,. \end{array} \tag{2.22.5}$$

(2-22.c) ⚁ (15 min.) Show that the solution u of the variational problem (2.22.5) solves the variational problem (2.22.2).

SOLUTION for (2-22.c) → 2-22-3-0:sab2.pdf ▲

(2-22.d) ⚂ (15 min.) [depends on Sub-problem (2-22.b)]

How is the value for the scalar solution component $\eta \in \mathbb{R}$ in (2.22.5) related to the u-component of the solution of (2.22.5)?

SOLUTION for (2-22.d) → 2-22-4-0:sab3.pdf ▲

We equip Ω with a triangular mesh and consider the finite element Galerkin discretization of (2.22.5) based on the piecewise linear Lagrangian finite element space $\mathcal{S}^0_1(\mathcal{M})$ and its nodal ordered basis $\{b^\ell_h\}_{\ell=1}^N$, $N := \dim \mathcal{S}^0_1(\mathcal{M})$, of "tent functions". This converts (2.22.5) into a linear system of equations of the following block form

$$\begin{bmatrix} \mathbf{A} & \mathbf{c} \\ \mathbf{c}^\top & 0 \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\mu}} \\ \eta \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\varphi}} \\ 0 \end{bmatrix} \,, \tag{2.22.7}$$

where $\vec{\boldsymbol{\mu}}$ is the vector of expansion coefficients of the Galerkin solution $u_N \in H^1(\Omega)$ w.r.t. the nodal basis.

(2-22.e) ⚀ (15 min.) Give formulas for the entries of the matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ and of the column vectors $\mathbf{c} \in \mathbb{R}^N$ and $\vec{\boldsymbol{\varphi}} \in \mathbb{R}^N$.

SOLUTION for (2-22.e) → 2-22-5-0:sab4.pdf ▲

We have at our disposal a LEHRFEM++-based C++ function (in the file `solavgboundary.h`)

```
template <typename FUNC_ALPHA, typename FUNC_GAMMA, typename
    FUNC_BETA>
lf::assemble::COOmatrix<double>
compGalerkinMatrix(
const lf::assemble::DofHandler &dofh,
FUNC_ALPHA &&alpha, FUNC_GAMMA &&gamma, FUNC_BETA &&beta);
```

that, given a mesh $\mathcal{M}$, (approximately, using "sufficiently high-order" local numerical quadrature) computes the $\mathcal{S}_1^0(\mathcal{M})$-based finite element Galerkin matrix (in triplet format, see the documentation of **lf::assemble::COOMatrix**) for the left-hand side of the variational boundary value problem

$$u \in H^1(\Omega): \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\beta(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{2.22.8}$$

where $\alpha, \gamma : \Omega \to \mathbb{R}$, $\beta : \partial\Omega \to \mathbb{R}$ are bounded coefficient functions, and $f \in L^2(\Omega)$. The use of the standard nodal basis consisting of "tent functions" is assumed.

The argument `dofh` passes the local-to-global index mapping for $\mathcal{S}_1^0(\mathcal{M})$ and the mesh $\mathcal{M}$, while `alpha`, `gamma`, and `beta` are functors matching **std::function<double(Vector2D)** realizing the coefficient functions $\boldsymbol{x} \mapsto \alpha(\boldsymbol{x})$, $\boldsymbol{x} \mapsto \gamma(\boldsymbol{x})$, and $\boldsymbol{x} \mapsto \beta(\boldsymbol{x})$.

(2-22.f) ⚄ (30 min.) [depends on Sub-problem (2-22.f)]

In the file `solavgboundary.cc`, based on the function `compGalerkinMatrix()`, write a LEHRFEM++-based C++ function

```
Eigen::SparseMatrix<double> augmentMatrix(
std::shared_ptr<const lf::uscalefe::FeSpaceLagrangeO1<double>>
    fe_spc,
const Eigen::VectorXd& c)
```

that augments the appropriate sparse coefficient matrix assembled by `compGalerkinMatrix()` in triplet format to obtain the full Galerkin matrix for the linear system of equations (2.22.7). Argument `c` passes vector $\mathbf{c}$ from (2.22.7), while argument `fe_test` passes an object describing the piecewise linear Lagrangian finite element space.

HIDDEN HINT 1 for (2-22.f) → 2-22-6-0:sab4h1.pdf

SOLUTION for (2-22.f) → 2-22-6-1:.pdf ▲

(2-22.g) ⚅ (20 min.) [depends on Sub-problem (2-22.f)]

Based on the function `compGalerkinMatrix()`, write another C++ function

```
template <typename FESPACE>
Eigen::VectorXd computeCVector(const FESPACE& fe_test)
```

that returns the column vector $\mathbf{c} \in \mathbb{R}^N$ as defined in (2.22.7).

For your implementation you should use *only* the function `compGalerkinMatrix()` and tools offered by EIGEN.

HIDDEN HINT 1 for (2-22.g) → 2-22-7-0:sabxh1.pdf

SOLUTION for (2-22.g) → 2-22-7-1:sabxsl.pdf ▲

End Problem 2-22 , 125 min.

Problem 2-23: Galerkin Matrices

Abstract Galerkin discretization of a linear variational problem leads a linear system of equations, whose system matrix is called a **Galerkin matrix**. This problem examines Galerkin matrices for typical bilinear forms connected with second-order elliptic BVPs from in an abstract setting and for a concrete finite-element trial and test space.

You need to be familiar with [Lecture → Section 2.2.3] and [Lecture → Section 2.4.5.2].
▷ problem name/problem code folder: GalerkinMatrices

Let $\Omega \subset \mathbb{R}^2$ be a regular hexagon in the plane. Throughout this problem we deal with the two bilinear forms

$$\mathsf{m}(u,v) := \int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \qquad u,v \in L^2(\Omega)\,, \tag{2.23.1}$$

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \qquad u,v \in H^1(\Omega)\,. \tag{2.23.2}$$

(2-23.a) ⚂ (10 min.) Let $V_h \subset H^1(\Omega)$ be some *finite-dimensional* space equipped with some basis $\mathfrak{B} := \{b_h^1,\ldots,b_h^N\}$, $N := \dim V_h$. We write $\mathbf{M} \in \mathbb{R}^{N,N}$ and $\mathbf{A} \in \mathbb{R}^{N,N}$ for the Galerkin matrices induced by m and a using $\mathfrak{B}$, that is,

$$\mathbf{M} = \left[\mathsf{m}(b_h^j, b_h^i)\right]_{i,j=1}^N \quad , \quad \mathbf{A} = \left[\mathsf{a}(b_h^j, b_h^i)\right]_{i,j=1}^N . \tag{2.23.3}$$

Decide, which properties of $\mathbf{A}$, $\mathbf{M}$, and $\mathbf{A}+\mathbf{M}$ hold for *any* choice of V_h and $\mathfrak{B}$.

matrix	symmetric	positive definite[a]	regular[b]	sparse[c]
$\mathbf{M}$	yes ◯	yes ◯	yes ◯	yes ◯
	no ◯	no ◯	no ◯	no ◯
$\mathbf{A}$	yes ◯	yes ◯	yes ◯	yes ◯
	no ◯	no ◯	no ◯	no ◯
$\mathbf{A}+\mathbf{M}$	yes ◯	yes ◯	yes ◯	yes ◯
	no ◯	no ◯	no ◯	no ◯

[a] A matrix $\mathbf{X} \in \mathbb{R}^{N,N}$ is called **positive definite**, if $\vec{\xi}^\top \mathbf{X}\vec{\xi} > 0$ for all $\vec{\xi} \in \mathbb{R}^N \setminus \{0\}$.
[b] A square matrix is called **regular**, when it is invertible.
[c] Here we call a matrix **sparse**, when it has a *fixed* small N-independent number of non-zero entries per row.

SOLUTION for (2-23.a) → 2-23-1-0:gms1.pdf ▲

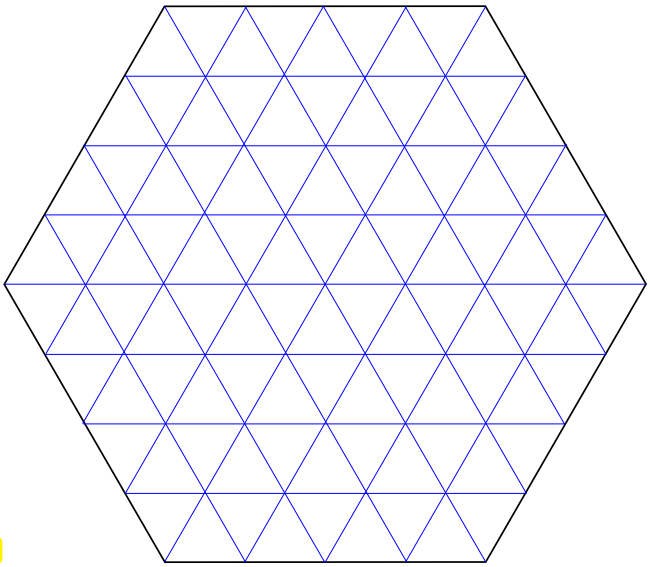


Fig. 29

In the sequel we consider finite-element Galerkin discretization based on the trial and test space $\mathcal{S}_1^0(\mathcal{M})$ of lowest-order Lagrangian finite element functions based on a mesh whose cells are all congruent equilateral triangles, see Fig. 29.

Throughout, we use the nodal basis of $\mathcal{S}_1^0(\mathcal{M})$ comprised of *tent functions*.

(2-23.b) ⚄ (15 min.) What are the dimensions of the following spaces

$$\dim \mathcal{S}_1^0(\mathcal{M}) = \boxed{}\,,$$
$$\dim \mathcal{N}(\mathbf{M}) = \boxed{}\,,$$
$$\dim \mathcal{N}(\mathbf{A}) = \boxed{}\,?$$

Here,

- $\mathbf{M}$ and $\mathbf{A}$ are the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrices for the bilinear forms m and a, respectively.
- $\mathcal{N}(\mathbf{X})$ stands for the kernel/nullspace of a matrix $\mathbf{X}$.

Note. You may write your answer in terms of the number $\sharp\mathcal{V}(\mathcal{M})$ of vertices, the number $\sharp\mathcal{E}(\mathcal{M})$ of edges, and the number $\sharp\mathcal{M}$ of triangles of $\mathcal{M}$.

SOLUTION for (2-23.b) → 2-23-2-0:gms2.pdf ▲

(2-23.c) ⚅ (10 min.) It is known that for our $\mathcal{S}_1^0(\mathcal{M})$-based finite element Galerkin discretization using the mesh sketched in Fig. 29 all **element matrices** for the bilinear form a are

$$\begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix} \in \mathbb{R}^{3,3}\,.$$

Explicitly write down the set of diagonal entries of the Galerkin matrix $\mathbf{A}$:

$$\left\{ (\mathbf{A})_{i,i}\,,\ i = 1,\ldots,\dim \mathcal{S}_1^0(\mathcal{M}) \right\} = \left\{ \boxed{} \right\}.$$

SOLUTION for (2-23.c) → 2-23-3-0:gms3.pdf ▲

End Problem 2-23, 35 min.

Problem 2-24: Parametric FEM: Local Computations

In the case of parametric finite element methods the computation of element matrices can be organized efficiently by precomputing many expressions on the reference element(s), see [Lecture → § 2.8.3.6]. In this problem we recall the relevant formulas.

Assumes familiarity with [Lecture → § 2.8.3.6], [Lecture → § 2.8.3.14], [Lecture → § 2.8.3.15], and [Lecture → (2.8.3.13)].
▷ problem name/problem code folder: LocCompParamFE

As in [Lecture → Ex. 2.8.3.29], in this problem we consider the bilinear form

$$\mathsf{a}(u,v) := \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u,v\in H^1(\Omega)\,,$$

on a polygonally bounded computational domain $\Omega\subset\mathbb{R}^2$. Here, $\boldsymbol{\alpha}:\overline{\Omega}\to\mathbb{R}^{2,2}$ and $\gamma:\overline{\Omega}\to\mathbb{R}$ are uniformly positive (definite) coefficient functions.

(2-24.a) ⚄ (10 min.) In the context of *parametric* Lagrangian finite elements the following formula provides the entries of the **element matrix** [Lecture → Def. 2.7.4.5] for a mesh cell $K\in\mathcal{M}$:

$$(\mathbf{A}_K)_{ij} = \sum_{\ell=1}^{P} \boxed{\mathbf{A}} \Big(\boldsymbol{\alpha}(\boxed{\mathbf{B}})(\mathrm{D}\boldsymbol{\Phi}(\boxed{\mathbf{C}}))^{-\top}\mathbf{grad}\,\widehat{b}^i(\boxed{\mathbf{D}}))\cdot((\mathrm{D}\boldsymbol{\Phi}(\boxed{\mathbf{E}}))^{-\top}\mathbf{grad}\,\boxed{\mathbf{F}})+ \gamma(\boxed{\mathbf{G}})\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell)\Big)\,|\boxed{\mathbf{H}}|\,, \quad i,j\in\{1,\ldots,Q\}\,.$$

Put the appropriate expression into the placeholders by ticking the corresponding column:

expression	A	B	C	D	E	F	G	H
$\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)$	◯	◯	◯	◯	◯	◯	◯	◯
$\widehat{\omega}_\ell$	◯	◯	◯	◯	◯	◯	◯	◯
$\widehat{\boldsymbol{\zeta}}_\ell$	◯	◯	◯	◯	◯	◯	◯	◯
$\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)$	◯	◯	◯	◯	◯	◯	◯	◯
$\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)$	◯	◯	◯	◯	◯	◯	◯	◯
$\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)$	◯	◯	◯	◯	◯	◯	◯	◯
$(\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))^\top$	◯	◯	◯	◯	◯	◯	◯	◯

Symbols:
- $\widehat{\omega}_\ell\in\mathbb{R}$, $\ell\in\{1,\ldots,P\}$, $P\in\mathbb{N}$ ≙ quadrature weights of local quadrature rule on $\widehat{K}$,
- $\widehat{\boldsymbol{\zeta}}_\ell$, $\ell\in\{1,\ldots,P\}$ ≙ quadrature nodes in reference element $\widehat{K}$,
- $\boldsymbol{\Phi}:\widehat{K}\to K$ bijective mapping from reference element $\widehat{K}$ to K,
- $\widehat{b}^j$, $j\in\{1,\ldots,Q\}$, $Q\in\mathbb{N}$, local shape functions on reference element $\widehat{K}$.

SOLUTION for (2-24.a) → 2-24-1-0:pfls1.pdf ▲

(2-24.b) ⚀ (10 min.) The **lf::geometry::Geometry** and **lf::quad::QuadRule** classes of LEHRFEM++ provide a number of member functions to support the local computations for parametric Lagrangian finite elements.

In the table below match the member functions and expressions that play a role in local computations in the context of parametric Lagrangian finite elements: the member function is to offer direct access to the expression (We use the symbols introduced in Sub-problem (2-24.a)).

Member function	$\widehat{\zeta}_\ell$	$\boldsymbol{\Phi}(\widehat{\zeta})$	$\mathrm{D}\boldsymbol{\Phi}(\widehat{\zeta})^{-\top}$	$\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\zeta})$
`Geometry::Global()`	◯	◯	◯	◯
`lf::quad::QuadRule::Weights()`	◯	◯	◯	◯
`Geometry::IntegrationElement()`	◯	◯	◯	◯
`lf::quad::QuadRule::Points()`	◯	◯	◯	◯
`Geometry::JacobianInverseGramian()`	◯	◯	◯	◯

Here `Geometry` stands for `lf::geometry::Geometry`.

SOLUTION for (2-24.b) → 2-24-2-0:lcps2.pdf

▲

End Problem 2-24, 20 min.

Chapter 3

FEM: Convergence and Accuracy

Problem 3-1: Computing Averages over the Boundary

This problem studies a linear output functional depending on the solution of a second-order elliptic boundary value problem. We take a look at issues of implementation and examine asymptotic convergence of finite-element Galerkin approximation.

Relies on tools and results from [Lecture → Chapter 2] and [Lecture → Chapter 3], in particular [Lecture → Section 3.6] and [Lecture → Section 3.4].

Given a bounded polygonal domain $\Omega \subset \mathbb{R}^2$ and a *fixed* weight function $w : \partial\Omega \to \mathbb{R}$, $w \in L^2(\partial\Omega)$, we consider the functional

$$F(v) = \int_{\partial\Omega} w(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})\,, \tag{3.1.1}$$

for sufficiently smooth functions $v : \partial\Omega \to \mathbb{R}$.

(3-1.a) ⚂ (10 min.) Argue why the functional F from (3.1.1) is continuous on $H^1(\Omega)$.

HIDDEN HINT 1 for (3-1.a) → `3-1-1-0:h11.pdf`

SOLUTION for (3-1.a) → `3-1-1-1:avb1.pdf` ▲

In the file `avgvalboundary.h` we have at our disposal a LEHRFEM++-based C++ function

```
template <typename FUNC_ALPHA,
          typename FUNC_GAMMA,
          typename FUNC_BETA>
  Eigen::SparseMatrix<double> compGalerkinMatrix(
      const lf::assemble::DofHandler &dofh,
      FUNC_ALPHA &&alpha, FUNC_GAMMA &&gamma, FUNC_BETA &&beta);
```

that, given a mesh $\mathcal{M}$, (approximately, using "sufficiently high-order" local numerical quadrature) computes the $\mathcal{S}_1^0(\mathcal{M})$-based finite element Galerkin matrix for the left-hand side of the variational boundary value problem

$$u \in H^1(\Omega)\colon \int_\Omega \alpha(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \gamma(\boldsymbol{x})\, u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \beta(\boldsymbol{x})\, u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{3.1.2}$$

where $\alpha, \gamma : \Omega \to \mathbb{R}$, $\beta : \partial\Omega \to \mathbb{R}$ are bounded coefficient functions, and $f \in L^2(\Omega)$. The use of the standard nodal basis consisting of "tent functions" is assumed.

The argument `dofh` passes the local-to-global index mapping for $\mathcal{S}_1^0(\mathcal{M})$ and the mesh $\mathcal{M}$, while `alpha`, `gamma`, and `beta` are functors matching **std::function<double(Vector2D)** realizing the coefficient functions $\boldsymbol{x} \mapsto \alpha(\boldsymbol{x})$, $\boldsymbol{x} \mapsto \gamma(\boldsymbol{x})$, and $\boldsymbol{x} \mapsto \beta(\boldsymbol{x})$.

The following code gives an excellent demonstrations of how to assemble Galerkin matrices for general 2nd-order elliptic scalar boundary value problems in LehrFEM++.

C++11 code 3.1.3: Sub-problem (3-1.a): Implementation of `compGalerkinMatrix()`
➪ GitLab

```
template <typename FUNC_ALPHA, typename FUNC_GAMMA, typename FUNC_BETA>
Eigen::SparseMatrix<double> compGalerkinMatrix(
    const lf::assemble::DofHandler &dofh, FUNC_ALPHA &&alpha,
    FUNC_GAMMA &&gamma, FUNC_BETA &&beta) {
  // obtain mesh and set up fe_space (p.w. linear Lagrangian FEM)
  auto mesh = dofh.Mesh();
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh);

  // get the number of degrees of freedom = dimension of FE space
  const lf::base::size_type N_dofs(dofh.NumDofs());
  // Set up an empty sparse matrix to hold the Galerkin matrix
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Initialize ELEMENT_MATRIX_PROVIDER object
  lf::mesh::utils::MeshFunctionGlobal mf_alpha{alpha};
  lf::mesh::utils::MeshFunctionGlobal mf_gamma{gamma};
  lf::uscalfe::ReactionDiffusionElementMatrixProvider elmat_builder(
      fe_space, std::move(mf_alpha), std::move(mf_gamma));
  // Cell-oriented assembly over the whole computational domain
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);

  // Add contributions of boundary term in the bilinear form using
  // a LehrFEM++ built-in high-level ENTITY_MATRIX_PROVIDER class
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh, 1)};
  lf::mesh::utils::MeshFunctionGlobal mf_beta{beta};
  lf::uscalfe::MassEdgeMatrixProvider edgemat_builder(
      fe_space, std::move(mf_beta), bd_flags);
  lf::assemble::AssembleMatrixLocally(1, dofh, dofh, edgemat_builder, A);
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
  return A_crs;
}
```

(3-1.b) ⚀ (20 min.) Based on the function `compGalerkinMatrix()`, complete the LehrFEM++-based C++ function (file `avgvalboundary.cc`)

```
double compH1seminorm(lf::assemble::DofHandler &dofh,
                      const Eigen::VectorXd &u);
```

that computes the norm $|v_h|_{H^1(\Omega)}$ for a function $v_h \in \mathcal{S}_1^0(\mathcal{M})$ passed through its vector `u` of expansion coefficients with respect to the customary "tent functions" nodal basis of $\mathcal{S}_1^0(\mathcal{M})$. The argument `dofh` passes an **lf::assemble::DofHandler** object providing the mesh $\mathcal{M}$ and the local-to-global index mapping for $\mathcal{S}_1^0(\mathcal{M})$.

SOLUTION for (3-1.b) → `3-1-2-0:avb2.pdf` ▲

(3-1.c) ⚂ (25 min.) Again based on the function `compGalerkinMatrix()`, write a LehrFEM++-based C++ function (file `avgvalboundary.h`)

```
template <typename FUNCTION>
```

```
double compBoundaryFunctional(
       const lf::assemble::DofHandler &dofh,
       const Eigen::VectorXd &u, FUNCTION &&w);
```

that evaluates the functional F from (3.1.1) for a function $v_h \in \mathcal{S}_1^0(\mathcal{M})$ passed through its expansion coefficient vector `u` with respect to the customary "tent function" nodal basis of $\mathcal{S}_1^0(\mathcal{M})$. The weight function w is made available through the functor `w`, while `dofh` provides the mesh $\mathcal{M}$ and the local-to-global index mapping for $\mathcal{S}_1^0(\mathcal{M})$.

HIDDEN HINT 1 for (3-1.c) → 3-1-3-0:avb2h.pdf

SOLUTION for (3-1.c) → 3-1-3-1:avb4.pdf ▲

Now we assume that $\Omega \subset \mathbb{R}^2$ is a *convex* and *polygonally* bounded domain, and that the coefficients in (3.1.2) are $\alpha = \gamma = 1$ and and $\beta = 0$. This means that in the sequel we consider the variational problem

$$u \in H^1(\Omega): \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,. \tag{3.1.6}$$

We adopt the generic notation $u \in H^1(\Omega)$ and $u_h \in \mathcal{S}_1^0(\mathcal{M})$ for the exact solution and the piecewise linear finite element solution, respectively, of (3.1.6) on a triangular mesh $\mathcal{M}$ of Ω. Whenever asymptotic convergence of discretization errors is mentioned, we study it on families of triangular meshes created by uniform regular refinement.

In your answers you may appeal to the following result from elliptic regularity theory, which can be applied to (3.1.6):

Theorem 3.1.7. Regularity of solutions of Neumann problems

Assume that Ω is a convex bounded Lipschitz domain. If $f \in L^2(\Omega)$ and $g \in H^1(\partial\Omega)$ such that $\int_\Omega f\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} g\,\mathrm{d}S = 0$, then every solution $u \in H^1(\Omega)$, $\int_\Omega u\,\mathrm{d}\boldsymbol{x} = 0$, of

$$-\Delta u + u = f \quad \text{in} \quad \Omega\,, \quad \mathbf{grad}\, u \cdot \boldsymbol{n} = g \quad \text{on} \quad \partial\Omega\,,$$

satisfies

$$u \in H^2(\Omega) \quad \text{and} \quad \|u\|_{H^2(\Omega)} \le C(\|f\|_{L^2(\Omega)} + \|g\|_{H^1(\partial\Omega)})\,,$$

with a constant $C > 0$ depending only on Ω.

(3-1.d) ⚄ (15 min.) Based on the multiplicative trace estimate from [Lecture → Thm. 1.9.0.10] and assuming $f \in L^2(\Omega)$, describe qualitatively and quantitatively the expected asymptotic convergence of

$$\|u - u_h\|_{L^2(\partial\Omega)} := \left(\int_{\partial\Omega} |u(\boldsymbol{x}) - u_h(\boldsymbol{x})|^2\,\mathrm{d}S(\boldsymbol{x})\right)^{\frac{1}{2}} \tag{3.1.8}$$

in terms of the meshwidth $h := h_{\mathcal{M}} \to 0$.

SOLUTION for (3-1.d) → 3-1-4-0:avb4.pdf ▲

(3-1.e) ⚄ (10 min.) State (in PDE form, "strong form") the boundary value problem satisfied by the dual solution that is crucial for duality estimates targetting the error $|F(u) - F(u_h)|$.

HIDDEN HINT 1 for (3-1.e) → 3-1-5-0:avb5h1.pdf

SOLUTION for (3-1.e) → 3-1-5-1:avb5.pdf ▲

(3-1.f) ⚄ (15 min.) [depends on Sub-problem (3-1.e), Sub-problem (3-1.d)]

Which asymptotic convergence of $|F(u) - F(u_h)|$ in terms of $h := h_{\mathcal{M}} \to 0$ is predicted by duality estimates for a piecewise continuously differentiable weight function $w \in C^1_{\mathrm{pw}}(\partial\Omega)$? Justify your answer.

SOLUTION for (3-1.f) → 3-1-6-0:avb6.pdf ▲

(3-1.g) ⚄ (30 min.) [depends on Sub-problem (3-1.c)]

Finally, we want to test the predictions of our theoretical estimates in a numerical experiment. To that end the LEHRFEM++ function

```
std::shared_ptr<MeshHierarchy>
  generateTestMeshSequence(unsigned int L);
```

in the file `avgvalboundary.cc` provides a (pointer to a) **lf::refinement::MeshHierarchy** object that contains a sequence $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_L$ of $L+1 \in \mathbb{N}$ meshes created by successive regular refinement of a triangular mesh of the unit square $\Omega =]0,1[^2$. In this domain we consider (3.1.6) with $f(\boldsymbol{x}) = \|\boldsymbol{x}\|$ and its Galerkin finite element discretization based on $\mathcal{S}^0_1(\mathcal{M})$.

We write $u_\ell \in \mathcal{S}^0_1(\mathcal{M}_\ell)$ for the finite element solution on $\mathcal{M}_\ell$. Based on `compGalerkinMatrix()` (also for the computation of the right-hand-side vector of the Galerkin linear system of equations) and `compBoundaryFunctional()` implement a C++ function (also in the file `avgvalboundary.cc`)

```
std::vector<std::pair<unsigned int, double>>
  approxBoundaryFunctionalValues(unsigned int L);
```

that computes the values $F(u_\ell)$ for $\ell = 0, \ldots, L$ and returns them in a vector together with the dimension of the finite element space $\mathcal{S}^0_1(\mathcal{M}_\ell)$.

SOLUTION for (3-1.g) → 3-1-7-0:sx.pdf ▲

(3-1.h) ⚂ (20 min.) [depends on Sub-problem (3-1.g)]

In the setting of the Sub-problem (3-1.g) we choose $L = 7$ and use $F(u_7)$ as a substitute for the unknown value $F(u)$.

- Extend the `main()` function in `avgvalboundary_main.cc` so that it tabulates $F(u_\ell) - F(u_7)$ as a function of $N_\ell := \dim \mathcal{S}^0_1(\mathcal{M}_\ell)$.
- Describe qualitatively and quantitatively the "convergence" of $F(u_\ell) - F(u_7)$ as a function of $N_\ell := \dim \mathcal{S}^0_1(\mathcal{M}_\ell)$ for $\ell = 0, \ldots, 6$ as ℓ increases.

SOLUTION for (3-1.h) → 3-1-8-0:sy.pdf ▲

End Problem 3-1 , 145 min.

Problem 3-2: Debugging Finite Element Codes

[Lecture → Chapter 3] confronted you with theoretical results on the asymptotic convergence of finite element Galerkin solutions for 2nd-order elliptic boundary value problems. On the one hand, these estimates can be used to gauge the relative efficiency of different finite element approximations, as was discussed in [Lecture → Section 3.3.5]. On the other hand, expected rates of convergence are a fine probe for detecting errors in a finite element code. For instance, the observed convergence to a known analytic solution should match the theoretical predictions, unless the code is flawed. Another way of using the approximation results of [Lecture → Section 3.3.5] to identify a faulty finite element implementation is demonstrated in this problem.

The considerations underlying this problem are presented in detail in [Lecture → Section 3.8]. This problem complements this section of the lecture material, in particular [Lecture → § 3.8.0.10]. Study this paragraph before you continue. This problem relies on LEHRFEM++. Thus, familiarity with the contents of [Lecture → Section 2.7] is expected.
▷ problem name/problem code folder: DebuggingFEM

Three different local **ENTITY_MATRIX_PROVIDER** classes

$$\textbf{LocalLaplaceQFEX}, \quad \mathrm{X} = 1,2,3 \ ,$$

implemented in `locallaplaceqfe.cc` , purport to provide element matrices matrices and element load vectors for the finite element discretization of the variational problem

$$u \in H^1(\Omega): \quad \mathsf{a}(u,v) := \int_\Omega \mathbf{grad} u \cdot \mathbf{grad} v \, \mathrm{d}\boldsymbol{x} = \ell(v) := \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \tag{3.2.1}$$

using *quadratic* Lagrangian finite elements (space $\mathcal{S}_2^0(\mathcal{M})$) on a triangular mesh $\mathcal{M}$ of some polygonal domain $\Omega \subset \mathbb{R}^2$.

For a successful attack on the problem, a solid understanding of the LEHRFEM++ framework for assembly as discussed in the lecture in [Lecture → Section 2.7.4] is crucial. In this problem we want to find out which **ENTITY_MATRIX_PROVIDER** is implemented correctly, **without peeking at the code**!.

(3-2.a) ⚃ (30 min.) Implement a C++ function

```
template <typename FUNCTOR>
Eigen::VectorXd interpolateOntoQuadFE(const
   lf::assemble::DofHandler &dofh, FUNCTOR &&f);
```

in `qfeinterpolator.h` that accepts a **lf::assemble::DofHandler** argument `dofh` for the finite element space $\mathcal{S}_2^0(\mathcal{M})$ and a functor object for a continuous function $f \in C^0(\overline{\Omega})$. It should return a vector with the basis coefficients of the nodal interpolant $\mathsf{I}_2 f \in \mathcal{S}_2^0(\mathcal{M})$. This particular piecewise quadratic interpolation is presented in [Lecture → Ex. 2.6.1.2]. Do not forget to equip your function with checks for consistency using the `LF_ASSERT_MSG()` macro.

HIDDEN HINT 1 for (3-2.a) → 3-2-1-0:loc1h1.pdf

HIDDEN HINT 2 for (3-2.a) → 3-2-1-1:loc1h2.pdf

SOLUTION for (3-2.a) → 3-2-1-2:loc1.pdf

▲

(3-2.b) ⚀ (15 min.) Please refresh your knowledge of the "direct testing of bilinear form" debugging strategy explained in [Lecture → § 3.8.0.10]. Mull over the estimates presented in that §.
▲

(3-2.c) ⚄ (15 min.) Determine a sharp bound $T(h_{\mathcal{M}})$ in the estimate

$$|\mathsf{a}(u,u) - \mathsf{a}(\mathsf{I}_2 u, \mathsf{I}_2 u)| \leq CT(h_{\mathcal{M}}), \tag{3.2.3}$$

where $u : \overline{\Omega} \mapsto \mathbb{R}$ is supposed to be smooth and the unknown constant $C > 0$ may depend only on u, Ω and the shape regularity measure of $\mathcal{M}$.

Use the following result, which is a generalization of [Lecture → Cor. 3.3.3.4] to quadratic Lagrangian finite element spaces.

Theorem 3.2.4. Interpolation error estimate for quadratic Lagrangian FEM

Let $\Omega \subset \mathbb{R}^d$, $d = 1,2,3$, be a bounded polygonal/polyhedral domain equipped with a simplicial mesh $\mathcal{M}$. Then the following interpolation error estimate holds for the nodal interpolation operator I_2 onto $\mathcal{S}_2^0(\mathcal{M})$

$$\|u - \mathsf{I}_2 u\|_{H^1(\Omega)} \leq C h_{\mathcal{M}}^{\min\{3,k\}-1} |u|_{H^k(\Omega)} \quad \forall u \in H^k(\Omega), \quad k = 2,3\,,$$

with a constant $C > 0$ depending only on k and the shape regularity measure $\rho_{\mathcal{M}}$.

HIDDEN HINT 1 for (3-2.c) → 3-2-3-0:loc2h1.pdf

SOLUTION for (3-2.c) → 3-2-3-1:loc2.pdf ▲

(3-2.d) ⚅ (30 min.) [depends on Sub-problem (3-2.a)]

In the file `qfeprovidertester.h` implement the constructor and the member function `energyOfInterpolant` of the templated class

```
template <typename ENTITY_MATRIX_PROVIDER>
class QFEProviderTester {
  public:
  QFEProviderTester(const lf::assemble::DofHandler &dofh,
  ENTITY_MATRIX_PROVIDER &element_matrix_provider);

  template <typename FUNCTOR>
  double energyOfInterpolant(FUNCTOR &&u) const;

  private:
  const lf::assemble::DofHandler &dofh_;
  ENTITY_MATRIX_PROVIDER &element_matrix_provider_;
  Eigen::SparseMatrix<double> A_;
};
```

The template parameter **ENTITY_MATRIX_PROVIDER** must be a type that fits the local Laplace element matrix provider classes **LocalLaplaceQFEX**, X= 1,2,3.

The method `energyOfInterpolant()` should accept a functor object `u` encoding a continuous function u defined on the domain covered by the mesh associated with the `dofh` object with which the current instance of **QFEProviderTester** was initialized. The method should return $\mathsf{a}(\mathsf{I}_2 u, \mathsf{I}_2 u)$ based on the element matrix provider object with which the current object of type **QFEProviderTester** was set up.

HIDDEN HINT 1 for (3-2.d) → 3-2-4-0:h3a.pdf

HIDDEN HINT 2 for (3-2.d) → 3-2-4-1:loc3h1.pdf

SOLUTION for (3-2.d) → 3-2-4-2:loc3.pdf ▲

(3-2.e) ⚀ (15 min.) Test your implementation from Sub-problem (3-2.a) and Sub-problem (3-2.d) by running the executable created by compiling `debuggingfem_main.cc`. It uses your implementation of **QFEProvideTester** to compute

$$\mathrm{err}_i = |\mathsf{a}(u,u) - \mathsf{a}(\mathsf{I}_2 u, \mathsf{I}_2 u)| \quad , \quad \mathsf{I}_2 : C^0(\overline{\Omega}) \to \mathcal{S}_2^0(\mathcal{M}_i) \quad \hat{=} \text{ nodal interpolation ,}$$

for the function $u(\boldsymbol{x}) = \exp(\|\boldsymbol{x}\|^2)$ from (3-2.d), and for a sequence $\mathcal{M}_0 \preceq \mathcal{M}_1 \preceq \mathcal{M}_2 \preceq \mathcal{M}_3 \preceq \mathcal{M}_4$ of nested meshes created by uniform regular refinement. Tabular and graphical output (`error.eps` -file) is generated.

SOLUTION for (3-2.e) → `3-2-5-0:loc4.pdf` ▲

(3-2.f) ⚁ (15 min.) [depends on Sub-problem (3-2.b)]

Which implementations of the assembly routine are wrong and which are correct? Can you decide? Explain your answer.

SOLUTION for (3-2.f) → `3-2-6-0:loc5.pdf` ▲

End Problem 3-2 , 120 min.

Problem 3-3: Dirichlet BVP with point-evaluation right-hand-side functional

One of the messages of [Lecture → Section 1.2.3.4] was that not all linear variational problems on function spaces are well-posed. In this problem we will en counter a specimen of an innocent looking ill-posed linear variational problem that was discussed in detail in [Lecture → Ex. 1.2.3.45]. We are going to study how the ill-posed nature of the continuous variational problem will be reflected by the behavior of Galerkin solutions. This will be done empirically based on a C++ implementation using the LEHRFEM++ library.

Knowledge about handling of meshes and assmbly in LEHRFEM++ is taken for granted and essential for this problem.
▷ problem name/problem code folder: PointEvaluationRhs

We are given a polygon Ω and a point $\boldsymbol{p} \in \Omega$. We consider the linear variational problem

$$u \in H^1_0(\Omega): \quad \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = v(\boldsymbol{p}) \quad \forall v \in H^1_0(\Omega)\,, \tag{3.3.1}$$

and attempt its numerical solution by means of a Galerkin discretization based on lowest-order (piecewise linear) Lagrangian finite elements on planar hybrid meshes.

(3-3.a) (10 min.) Describe a physical system for which (3.3.1) provides a model of certain aspects. What is the meaning of u in this model.

HIDDEN HINT 1 for (3-3.a) → 3-3-1-0:BETL-d-r11.pdf

SOLUTION for (3-3.a) → 3-3-1-1:BETL-d-r1.pdf ▲

(3-3.b) (15 min.) Which problem haunts the linear variational problem (3.3.1)? What can you say about the existence of a minimizer of the associated quadratic functional?

HIDDEN HINT 1 for (3-3.b) → 3-3-2-0:BEM-d-r21.pdf

SOLUTION for (3-3.b) → 3-3-2-1:BEM-d-r2.pdf ▲

(3-3.c) (15 min.) Let $\mathcal{M}$ be a triangular mesh of Ω. The Galerkin discretization of (3.3.1) based on $\mathcal{S}^0_{1,0}(\mathcal{M})$ leads to a discrete variational problem, *cf.* [Lecture → Section 2.2]. Explain why the associated discrete quadratic minimization problem (see [Lecture → Rem. 2.2.1.3]) always has a unique solution.

SOLUTION for (3-3.c) → 3-3-3-0:BEM-d-r3.pdf ▲

(3-3.d) ⚄ (15 min.) In the file `pointevaluationrhs.cc`, implement a function

```
Eigen::Vector2d GlobalInverseTria(
  Eigen::Matrix<double,2,3> vertices,
  Eigen::Vector2d x);
```

accepting the arguments

- `vertices`, a matrix whose columns store the coordinates of the vertices of a planar triangle K,
- `x`, the coordinates of a point $\boldsymbol{x}$ in the plane.

The function should return the coordinates of the pre-image $\widehat{\boldsymbol{x}}$ of $\boldsymbol{x}$ under the affine mapping $\boldsymbol{\Phi}_K : \widehat{K} \to K$, $\widehat{K}$ the usual "unit triangle".

The returned coordinates need not be located inside $\widehat{K}$, which indicates that $\boldsymbol{x} \notin K$.

For the sake of stable implementation use EIGEN's direct elimination solver also for small dense matrices.

HIDDEN HINT 1 for (3-3.d) → 3-3-4-0:h3a1.pdf

SOLUTION for (3-3.d) → 3-3-4-1:s3a.pdf ▲

(3-3.e) ⚄ (20 min.) In the file `pointevaluationrhs.cc`, implement a C++ function

```
std::pair<double, double> solveQuadraticEquation(
  double a, double b, double c);
```

that computes the real roots of the quadratic equation

$$a\xi^2 + b\xi + c = 0\,,$$

and returns `NAN` ("Not a number"', see **std::nan()**), in case none exist.

Remember that numerical stability is an issue when applying the standard formula for the roots of a parabola. Distinguish cases to achieve a cancellation-free implementation, see [NumCSE course → Ex. 1.5.4.1].

SOLUTION for (3-3.e) → 3-3-5-0:s3x.pdf ▲

(3-3.f) ⚄ (45 min.) [depends on Sub-problem (3-3.e)]

In the file `pointevaluationrhs.cc`, implement a function

```
Eigen::Vector2d GlobalInverseQuad(
  Eigen::Matrix<double,2,4> vertices,
  Eigen::Vector2d x);
```

accepting the arguments

- `vertices`, a matrix passing the coordinates of the vertices of a planar quadrilateral K in its columns,
- `x`, the coordinates of a point $\boldsymbol{x}$ in the plane.

The function should return the coordinates of the pre-image $\widehat{\boldsymbol{x}}$ of $\boldsymbol{x}$ under the bilinear mapping $\boldsymbol{\Phi}_K : \widehat{K} \to K$, see [Lecture → § 2.8.2.2], where $\widehat{K}$ is the unit square.

The returned coordinates need not be located inside $\widehat{K}$, which indicates that $\boldsymbol{x} \notin K$. However, there can also be cases, where $\boldsymbol{x} \notin K$ does not have a pre-image. Then the function should terminate with an error.

HIDDEN HINT 1 for (3-3.f) → 3-3-6-0:h3b1.pdf

HIDDEN HINT 2 for (3-3.f) → 3-3-6-1:h3b2.pdf

SOLUTION for (3-3.f) → 3-3-6-2:s3b.pdf ▲

(3-3.g) ⚂ (20 min.) Based on the Googletest framework implement in `pointevaluationrhs_test.cc` a C++ function

```
void testGlobalInverseQuad(
   const lf::mesh::Entity &quad,
   Eigen::Vector2d xh);
```

which can be used to create a **unit test** for the function `GlobalInverseQuad()`. The argument `quad` should pass an LEHRFEM++ mesh entity of topological type QUAD, the vector `xh` gives the coordinates of $\widehat{\boldsymbol{x}} \in \widehat{K}$ to be used for testing.

Rely on the `EXPECT_NEAR()` macro to test the correctness of the result of `GlobalInverseQuad()`.

SOLUTION for (3-3.g) → 3-3-7-0:3cs.pdf ▲

(3-3.h) ⚂ (10 min.) Let $\mathcal{M}_k$ be a sequence of hybrid meshes, where $\mathcal{M}_{k+1}$ is generated from $\mathcal{M}_k$ by regular refinement of all cells. Guess how the minimal value of the quadratic functional associated with the variational problem (3.3.1) will behave as $k \to \infty$.

SOLUTION for (3-3.h) → 3-3-8-0:BEM-d-r4.pdf ▲

(3-3.i) ⚀ (15 min.) Study [Lecture → Ex. 2.7.4.27] and remember how right-hand-side vectors of linear systems of equations arising from finite element Galerkin discretization are built in LEHRFEM++ using `lf::assemble::AssembleVectorLocally()`.

▲

(3-3.j) ⚄ (30 min.) [depends on Sub-problem (3-3.d), Sub-problem (3-3.f)]

In the file `pointevaluationrhs.cc` implement a class **DeltaLocalVectorAssembler** fitting LEHRFEM++'s concept of an **ENTITY_VECTOR_PROVIDER** for the the computation of the element vector associated to the specific right-hand side of (3.3.1).

```
class DeltaLocalVectorAssembler{
private:
  Eigen::Vector2d x_0;
  bool already_found;

public:
  explicit DeltaLocalVectorAssembler(Eigen::Vector2d x)
      : x_0(x), already_found(false) {}
  bool isActive(const lf::mesh::Entity &entity) const{
    return (!already_found);
  }
  Eigen::VectorXd Eval(const lf::mesh::Entity &entity);
};
```

The core task is to complete the code for the `Eval()` method of the **DeltaLocalVectorAssembler** class. The method should check whether the point $\boldsymbol{x}$ (with coordinates `x` passed to the constructor) belongs to the the cell referenced by `entity` and then compute the corresponding values using linear Lagrangian finite elements.

HIDDEN HINT 1 for (3-3.j) → 3-3-10-0:BETL-d-r51.pdf

HIDDEN HINT 2 for (3-3.j) → 3-3-10-1:h52.pdf

SOLUTION for (3-3.j) → 3-3-10-2:BETL-d-r5.pdf ▲

(3-3.k) ⚂ (30 min.) Write a C++ function (in the file `pointevaluationrhs_norms.cc`)

```
double computeL2normLinearFE(
    const lf::assemble::DofHandler &dofh,
    const Eigen::VectorXd &mu);
```

that approximately computes $\|u_h\|_{L^2(\Omega)}$ for a finite element function $u_h \in \mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a hybrid mesh with straight edges.

HIDDEN HINT 1 for (3-3.k) → 3-3-11-0:sp6h1.pdf

SOLUTION for (3-3.k) → 3-3-11-1:BETL-d-r6.pdf ▲

(3-3.l) ⚂ (30 min.) Code a C++ function (in the file `pointevaluationrhs_norms.cc`)

```
double computeH1seminormLinearFE(
 const lf::assemble::DofHandler &dofh,
 const Eigen::VectorXd &mu);
```

that approximately computes $|u_h|_{H^1(\Omega)}$ for a finite element function $u_h \in \mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a hybrid mesh with straight edges. The argument `dofh` gives the local→global index mapping for $\mathcal{S}_1^0(\mathcal{M})$, whereas `mu` passes the basis expansion coefficients of u_h.

HIDDEN HINT 1 for (3-3.l) → 3-3-12-0:h16a.pdf

SOLUTION for (3-3.l) → 3-3-12-1:BETL-d-r6a.pdf ▲

(3-3.m) ⚄ [depends on Sub-problem (3-3.k) and Sub-problem (3-3.l)]

Copying the relevant parts of the function `lecturedemoDirichlet()` from `examples/lecturedemos/lecturedemoassemble.cc` ➙ GITHUB create a function (in the file `pointevaluationrhs.cc`)

```
std::pair<double, double> normsSolutionPointLoadDirichletBVP(
      const lf::assemble::DofHandler &dofh,
      Eigen::Vector2d source_point,
      Eigen::VectorXd &sol_vec)
```

that solves (3.3.1) on the domain covered by the mesh pointed to by `mesh_p`, for $\boldsymbol{p}$ passed through `source_point`, and by means of lowest-order Lagrangian finite elements. It should return the $L^2(\Omega)$-norm and $H^1(\Omega)$-seminorm of the finite element solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$. The basis expansion coefficient vector of the solution is returned via the argument `sol_vec`.

SOLUTION for (3-3.m) → 3-3-13-0:s6x.pdf ▲

(3-3.n) ⚄ (25 min.) Now we consider (3.3.1) on the square $\Omega :=]0,4[^2$ and for $\boldsymbol{p} := \begin{bmatrix} 1.3 \\ 1.7 \end{bmatrix}$. Based on the mesh returned by `lf::mesh::test_utils::GenerateHybrid2DTestMesh(1, 1.0)` we created a sequence of nested meshes by seven steps of regular refinement, see [Lecture → Ex. 3.1.4.4] for explanations.

Tabulate the $L^2(\Omega)$-norms and the $H^1(\Omega)$-seminorms of the $\mathcal{S}_1^0(\mathcal{M})$-finite element solutions of (3.3.1) obained on these meshes versus the dimensions of the finite element spaces.

SOLUTION for (3-3.n) → 3-3-14-0:BETL-d-r7.pdf ▲

(3-3.o) ⚄ (15 min.) Consider an abstract linear variational problem

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0, \tag{3.3.16}$$

with s.p.d. (→ Def. [Lecture → Def. 1.2.3.27]) bilinear form a and a linear form ℓ that is continuous in the energy norm in the sense of [Lecture → (1.2.3.40)].

Elaborate the relationship between the energy norm of the solution of (3.3.16) and the minimal value of the associated quadratic functional according to [Lecture → Thm. 1.4.1.8].

SOLUTION for (3-3.o) → 3-3-15-0:BETL-d-r8.pdf ▲

(3-3.p) ⚄ (10 min.) Explain your observations on the behavior of the H^1-seminorm in Subproblem (3-3.n) in light of the theoretical conclusions obtained in subproblems (3-3.h).

SOLUTION for (3-3.p) → 3-3-16-0:BETL-d-r9.pdf ▲

End Problem 3-3, 305 min.

Problem 3-4: A boundary value problem modelling stationary heat conduction

We "reverse engineer" a boundary value problem from a code and then discuss empiric convergence in different situations. This problem is related to [Lecture → Section 2.7.4] and, loosely, to [Lecture → Chapter 3]. It will also encounter a particular instability, whose explanation draws on [Lecture → § 1.9.0.6].

Just involves reading of LehrFEM++ code; otherwise a purely theoretical problem.
▷ problem name/problem code folder: UnstableBVP

The temperature distribution of a body Ω which is partly or entirely submerged in water can be obtained as the solution $u = u(\mathbf{x})$ of some second-order elliptic boundary value problem (BVP), see [Lecture → Section 1.6].

A LehrFEM++ implementation of a Galerkin finite element solver for that BVP using linear Lagrangian basis functions is provided to you in the file `unstablebvp.cc` ➙ GitLab.

(3-4.a) ⊡ (10 min.)

The LehrFEM++ built-in type **lf::uscalfe::LinearFELaplaceElementMatrix** implements an **ENTITY_MATRIX_PROVIDER**-compatible type. What are objects of this type meant for?

Solution for (3-4.a) → 3-4-1-0:s0a.pdf ▲

(3-4.b) ⊡ (10 min.)

Describe the purpose of the LehrFEM++ class **lf::uscalfe::FeSpaceLagrangeO1** and relate it to concepts introduced in the course.

Solution for (3-4.b) → 3-4-2-0:0bs.pdf ▲

(3-4.c) ⊡ (10 min.)

Study the documentation of the LehrFEM++ function `lf::fe::InitEssentialConditionFromFunct` and explain the purpose of its template parameters and arguments.

```
template <typename SCALAR,
          typename EDGESELECTOR,
          typename FUNCTION>
std::vector<std::pair<bool, SCALAR>>
   InitEssentialConditionFromFunction(
     const lf::fe::ScalarFESpace<SCALAR> &fes,
     EDGESELECTOR &&esscondflag, FUNCTION &&g);
```

Solution for (3-4.c) → 3-4-3-0:0cs.pdf ▲

(3-4.d) ⚀ (20 min.) [depends on Sub-problem (3-4.a), Sub-problem (3-4.b), Sub-problem (3-4.c)]

The LehrFEM++-based C++ function

```
double solveTemperatureDistribution(
  std::shared_ptr<const lf::mesh::Mesh> mesh_p);
```

is provided in the (copiously commented) file `unstablebvp.cc` ➙ GitLab. The function is meant to solve a second-order scalar elliptic boundary value problem and supposed to return the $H^1(\Omega)$-seminorm of the finite element solution u_h computed when the function is called.

Give the strong form of the boundary value problems tackled by `solveTemperatureDistribution()`. Which finite element method is used for its discretization?

HIDDEN HINT 1 for (3-4.d) → 3-4-4-0:SHC11.pdf

SOLUTION for (3-4.d) → 3-4-4-1:SHC1.pdf ▲

(3-4.e) ⚀ (15 min.) [depends on Sub-problem (3-4.d)]

The function `solveTemperatureDistribution()` is now used to solve concrete BVPs on particular domains (passed through the `mesh_p` argument in the form of covering meshes).

For which of the three triangular domains Ω drawn in Fig. 37, Fig. 38, and Fig. 39 will the solution of the BVP be a linear function on Ω?

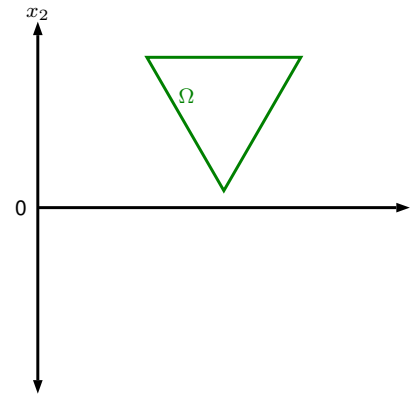


Fig. 37

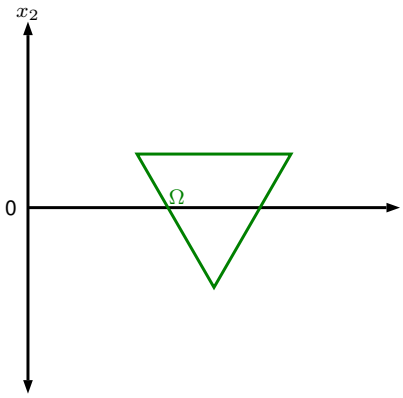


Fig. 38

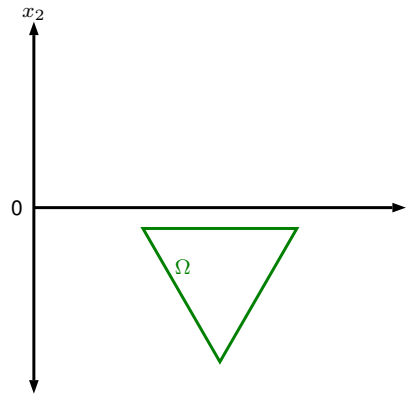


Fig. 39

SOLUTION for (3-4.e) → 3-4-5-0:SHC2.pdf ▲

(3-4.f) ⚀ (15 min.) [depends on Sub-problem (3-4.d) and Sub-problem (3-4.e)]

For the three different domains shown introduced in Sub-problem (3-4.e), we compute the finite element solutions u_k on sequences of regularly refined triangular meshes $\mathcal{T}_0, \mathcal{T}_1, \ldots, \mathcal{T}_L$ and plot $k \mapsto |u_k|_{H^1(\Omega)}$ in Fig. 40 and $k \mapsto |u_k|_{H^1(\Omega)} - |u_L|_{H^1(\Omega)}$ in Fig. 41, $k = 1, \ldots, L-1$.

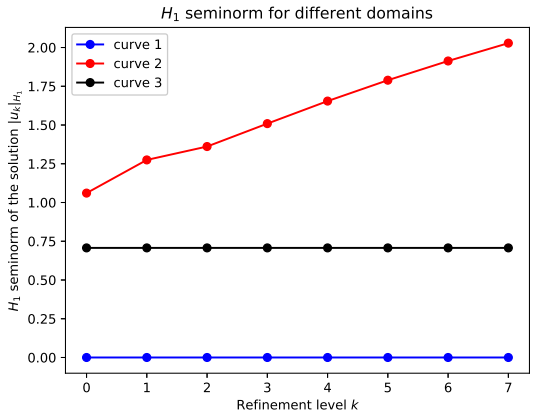


Fig. 40

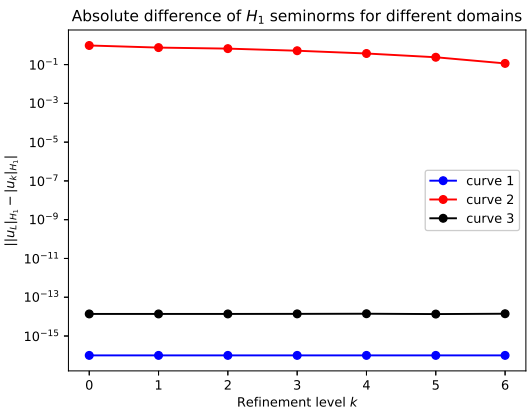


Fig. 41

Which curve in Fig. 41/Fig. 40 belongs to which domain? Explain your answer.

HIDDEN HINT 1 for (3-4.f) → 3-4-6-0:SHC31.pdf

SOLUTION for (3-4.f) → 3-4-6-1:SHC3.pdf ▲

(3-4.g) ⚁ (15 min.) Explain, why curve #2 (red) in Fig. 41 is increasing.

HIDDEN HINT 1 for (3-4.g) → 3-4-7-0:shc4h1.pdf

SOLUTION for (3-4.g) → 3-4-7-1:SHC4.pdf ▲

End Problem 3-4 , 95 min.

Problem 3-5: Error estimates for traces

This problem is connected with [Lecture → Chapter 3] and addresses special a priori finite element convergence estimates. It should also give you an idea how to use the a-priori convergence theory for finite element methods treated in class [Lecture → Section 3.3] for the prediction of the rate of algebraic convergence of derived quantities.

This problem involves moderate implementation in LEHRFEM++.
▷ problem name/problem code folder: ErrorEstimatesForTraces

On a convex polygon $\Omega \subset \mathbb{R}^2$ with exterior unit normal vector field $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ we consider the boundary value problem with impedance boundary conditions, *cf.* [Lecture → Ex. 1.7.0.5],

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega\,, \\ \boldsymbol{n} \cdot \mathbf{grad}\, u + u &= 0 \quad \text{on } \partial\Omega\,, \end{aligned} \tag{3.5.1}$$

with source function $f \in L^2(\Omega)$.

(3-5.a) State the complete variational formulation of (3.5.1).

HIDDEN HINT 1 for (3-5.a) → 3-5-1-0:trace11.pdf

SOLUTION for (3-5.a) → 3-5-1-1:trace1.pdf ▲

(3-5.b) ⚀ (5 min.) Discuss the uniqueness of solutions of the variational formulation obtained in Sub-problem (3-5.a).

HIDDEN HINT 1 for (3-5.b) → 3-5-2-0:tee2h1.pdf

SOLUTION for (3-5.b) → 3-5-2-1:trace2.pdf ▲

(3-5.c) ⚄ (20 min.) [depends on Sub-problem (3-5.a)]

Show the *continuity* (→ [Lecture → Def. 1.2.3.42]) of the right-hand-side linear form of the variational problem obtained in Sub-problem (3-5.a) with respect to the energy norm induced by its bilinear form.

HIDDEN HINT 1 for (3-5.c) → 3-5-3-0:tee2h2.pdf

SOLUTION for (3-5.c) → 3-5-3-1:1xs.pdf ▲

The following counterpart of [Lecture → Thm. 3.4.0.10] holds, another example of an insight from elliptic regularity theory.

Theorem 3.5.8. Elliptic lifting theorem for convex domains

If $\Omega \subset \mathbb{R}^d$ is convex, $u \in H^1(\Omega)$, $\Delta u \in L^2(\Omega)$, $\mathbf{grad}\, u \cdot \boldsymbol{n} + u = 0$ on $\partial\Omega$, then

$$u \in H^2(\Omega) \quad \textit{and} \quad \exists C = C(\Omega) > 0\colon \quad \|u\|_{H^2(\Omega)} \le C\|\Delta u\|_{L^2(\Omega)}\,.$$

(3-5.d) ⚀ (5 min.) What properties of the solution of (3.5.1) can be concluded from Thm. 3.5.8 and the information that Ω is convex?

SOLUTION for (3-5.d) → 3-5-4-0:tee2a.pdf ▲

The boundary value problem (3.5.1) is discretized by means of linear Lagrangian finite elements on a sequence $(\mathcal{M}_i)$ of triangular meshes of Ω created by uniform regular refinement of a coarse mesh $\mathcal{M}_0$. Let $u_i \in \mathcal{S}_1^0(\mathcal{M}_i)$ denote the finite element solution on mesh $\mathcal{M}_i$.

(3-5.e) ⚄ (20 min.) [depends on Sub-problem (3-5.d)]

Based on [Lecture → Thm. 3.3.2.21], qualitatively and quantitatively predict the asymptotic behavior of the error norm $|u-u_i|_{H^1(\Omega)}$ in terms of $N_i := \dim \mathcal{S}_1^0(\mathcal{M}_i)$ for $N_i \to \infty$.

HIDDEN HINT 1 for (3-5.e) → 3-5-5-0:tee3hi.pdf

SOLUTION for (3-5.e) → 3-5-5-1:trace3.pdf ▲

(3-5.f) ⚃ (20 min.) [depends on Sub-problem (3-5.e)]

Employ the duality techniques introduced in [Lecture → § 3.6.3.3] to derive a sharp asymptotic estimate for the error norm $\|u-u_i\|_{L^2(\Omega)}$ as $N_i := \dim \mathcal{S}_1^0(\mathcal{M}_i) \to \infty$.

HIDDEN HINT 1 for (3-5.f) → 3-5-6-0:tee3ahi.pdf

SOLUTION for (3-5.f) → 3-5-6-1:s3a.pdf ▲

(3-5.g) ⚂ (15 min.) [depends on Sub-problem (3-5.e)]

In the setting of the previous sub-problem, what asymptotic behavior can be deduced from the multiplicative trace inequality [Lecture → Thm. 1.9.0.10] for the error norm $\|u-u_i\|_{L^2(\partial\Omega)}$ on the boundary in terms of $N_i := \dim \mathcal{S}_1^0(\mathcal{M}_i) \to \infty$?

SOLUTION for (3-5.g) → 3-5-7-0:trace4.pdf ▲

Now we consider the linear output functional

$$B(v) := \int_{\partial\Omega} v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \quad v \in H^1(\Omega)\,. \tag{3.5.14}$$

(3-5.h) ⚀ (10 min.)

Read [Lecture → § 2.8.3.28] and study the documentation of **lf::uscalfe::UniformScalarFESpace** and **lf::uscalfe::FeSpaceLagrangeO1**. ▲

(3-5.i) ⚃ (30 min.) In the file `teelaplrobinassembly.cc` (name due to historical reasons) implement an LEHRFEM++-based function

```
double bdFunctionalEval(
    std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>&
      fe_space,
    Eigen::VectorXd& coeff_vec);
```

that computes $B(v_h) := \int_{\partial\Omega} v_h\,\mathrm{d}S(\boldsymbol{x})$, where $v_h \in \mathcal{S}_1^0(\mathcal{M})$ and its basis expansion coefficients are passed through `coeff_vec`. The argument `fe_space` also passes information about the (planar triangular) mesh $\mathcal{M}$ and the "d.o.f. handler" for the finite element space.

SOLUTION for (3-5.i) → 3-5-9-0:c1s.pdf ▲

(3-5.j) (30 min.) [depends on Sub-problem (3-5.i)]

The files `errorestimatesfortraces_main.cc`, `teelaplrobinassembly.h`, and `teelaplrobinassembly.cc` contain a LEHRFEM++-based code for solving (3.5.1) with $f \equiv \cos(\|\boldsymbol{x}\|)$ on a polygon using linear Lagrangian finite elements on triangular meshes.

For the output functional $B(v) := \int_{\partial\Omega} v\,\mathrm{d}S$, $v \in H^1(\Omega)$, determine qualitatively and quantitatively the rate of convergence of $B(u_h) \to B(u)$ by computing the numerical values on meshes obtained by successive refinement of the triangular mesh (of a regular hexagon) stored in `hexX.msh`, $X = 1,2,3,4$. The "exact" value is $B(u) = 2.08154105279$.

To do this, extend the `main()` function in `errorestimatesfortraces_main.cc` so that it writes a table of values $B(u_h)$ and errors $|B(u_h)-B(u)|$, with $u_h \in \mathcal{S}_1^0(\mathcal{M})$ the finite element solution of (3.5.1) on the four meshes.

SOLUTION for (3-5.j) → 3-5-10-0:trace6.pdf ▲

(3-5.k) (20 min.) [depends on Sub-problem (3-5.a)]

Prove that for the linear functional $v \mapsto B(v)$ as defined in (3.5.14) holds

$$B(u) = \int_\Omega f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \text{and} \quad B(u_h) = \mathbf{1}^\top \vec{\boldsymbol{\varphi}}\,, \tag{3.5.17}$$

where $u \in H^1(\Omega)$ is the exact solution of (3.5.1), $u_h \in \mathcal{S}_1^0(\mathcal{M})$ the finite element solution, and $\vec{\boldsymbol{\varphi}} \in \mathbb{R}^N$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$, is the right-hand-side vector of the Galerkin linear system. The symbol $\mathbf{1}$ stands for the vector $\in \mathbb{R}^N$ with entries all $= 1$.

HIDDEN HINT 1 for (3-5.k) → 3-5-11-0:trace71.pdf

SOLUTION for (3-5.k) → 3-5-11-1:trace7.pdf ▲

(3-5.l) ⚀ (10 min.) [depends on Sub-problem (3-5.k)]

Show that, with the notations used in Sub-problem (3-5.k), $B(u) = B(u_h)$ on *any* mesh $\mathcal{M}$, provided that all integrals defining the entries of the Galerkin matrix and of the right-hand-side vector $\vec{\boldsymbol{\varphi}}$ are computed exactly.

Remark. This reveals that the error observed in Sub-problem (3-5.j) is actually the error due to the use of a local quadrature rule for the approximation of the right-hand-side functional.

SOLUTION for (3-5.l) → 3-5-12-0:sx.pdf ▲

End Problem 3-5 , 185 min.

Problem 3-6: Projection Onto Constants

In [Lecture → Section 3.3.1] we derived L^2- and H^1-estimates for the error of piecewise linear interpolation on a grid, see [Lecture → (3.3.1.13)] and [Lecture → (3.3.1.17)]. The key tool was the integral representation formula [Lecture → (3.3.1.8)]. In this problem we practice these techniques for an even simpler projection operator.

This is a purely theoretical problem closely connected to [Lecture → Section 3.3.1], whose study is strongly recommended.
▷ problem name/problem code folder: ProjectionOntoConstants

Given a one-dimensional mesh $\mathcal{M} := \{]x_{j-1}, x_j[\colon 1 \le j \le M\}$, $M \in \mathbb{N}$, of the interval $[a,b] \subset \mathbb{R}$ with $x_{j-1} < x_j$ we define a projection onto the space $\mathcal{S}_0^{-1}(\mathcal{M})$ of piecewise constant discontinuous "staircase" functions on $\mathcal{M}$

$$\mathcal{S}_0^{-1}(\mathcal{M}) := \{v : [a,b] \to \mathbb{R} : \ v|_{]x_{j-1},x_j[} \in \mathcal{P}_0(\mathbb{R})\} , \tag{3.6.1}$$

according to

$$\mathsf{I}_0 : \left\{ \begin{array}{ccc} L^2(]a,b[) & \mapsto & \mathcal{S}_0^{-1}(\mathcal{M}) \\ u & \mapsto & \sum_{j=1}^{M} \frac{1}{x_j - x_{j-1}} \int_{x_{j-1}}^{x_j} u(\xi)\, \mathrm{d}\xi \cdot \chi_{]x_{j-1},x_j[} \end{array} \right\}. \tag{3.6.2}$$

where $\chi_{]x_{j-1},x_j[}$ stands for the characteristic function of the interval $]x_{j-1}, x_j[$, that is

$$\chi_{]x_{j-1},x_j[}(x) = \begin{cases} 1 & \text{, if } x \in]x_{j-1}, x_j[, \\ 0 & \text{else.} \end{cases} \tag{3.6.3}$$

Remark. The linear projection I_0 is an instance of an **L^2-projection**. Generically, given a (closed) subspace $V \subset L^2(\Omega)$, the associated L^2-projection operator $\mathsf{Q}_V : L^2(\Omega) \mapsto V$ is defined through as solution operator of the variational problem

$$\mathsf{Q}_V u \in V : \quad \langle \mathsf{Q}_V u, v \rangle_{L^2(\Omega)} = \langle u, v \rangle_{L^2(\Omega)} \quad \forall v \in V ,$$

where $\langle \cdot, \cdot \rangle_{L^2(\Omega)}$ stands for the inner product in $L^2(\Omega)$:

$$\langle u, v \rangle_{L^2(\Omega)} := \int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} , \quad u, v \in L^2(\Omega) .$$

Again, the operator Q_V is a specimen of an **orthogonal projection** onto a closed subspace of a Hilbert space.

(3-6.a) ⚀ (15 min.) Compute $\mathsf{I}_0 u$ on $[0,1]$ for $u(x) = x$ and an equidistant mesh with meshwidth $h := M^{-1}$. Sketch the function $\mathsf{I}_0 u$.

SOLUTION for (3-6.a) → 3-6-1-0:poc1.pdf ▲

(3-6.b) ⚅ (15 min.) Derive the following local **integral representation formula** for the projection error

$$(u - \mathsf{I}_0 u)(x) = \frac{1}{x_j - x_{j-1}} \int_{x_{j-1}}^{x_j} \int_y^x u'(\xi)\, \mathrm{d}\xi\, \mathrm{d}y, \quad x_{j-1} < x < x_j. \tag{3.6.4}$$

HIDDEN HINT 1 for (3-6.b) → 3-6-2-0:poc21.pdf

SOLUTION for (3-6.b) → 3-6-2-1:poc2.pdf ▲

(3-6.c) ⚀ (30 min.) Starting from (3.6.4) deduce the estimate

$$\|u - \mathrm{I}_0 u\|^2_{L^2(]x_{j-1},x_j[)} \leq (x_j - x_{j-1})^2 |u|^2_{H^1(]x_{j-1},x_j[)}. \tag{3.6.5}$$

HIDDEN HINT 1 for (3-6.c) → 3-6-3-0:h2aI.pdf

HIDDEN HINT 2 for (3-6.c) → 3-6-3-1:poc2a1.pdf

SOLUTION for (3-6.c) → 3-6-3-2:poc2a.pdf ▲

(3-6.d) ⚁ (10 min.) [depends on Sub-problem (3-6.c)]

Based on (3.6.5) derive the global projection error estimate

$$\|u - \mathrm{I}_0 u\|_{L^2(]a,b[)} \leq h_{\mathcal{M}} |u|_{H^1(]a,b[)} \,, \tag{3.6.6}$$

where $h_{\mathcal{M}}$ is the meshwidth of $\mathcal{M}$.

SOLUTION for (3-6.d) → 3-6-4-0:poc3.pdf ▲

End Problem 3-6 , 70 min.

Problem 3-7: Maximum principle for reaction-diffusion boundary value problems

An important structural property of solutions of scalar second-order elliptic problems is expressed through the **maximum principle** of [Lecture → Thm. 3.7.1.2].
In this problem we deal with continuous and discrete maximum principles for second-order elliptic PDEs with an additional zero-order term. In particular we study linear finite element Galerkin discretization on structured meshes, which are closely related to so-called finite-difference schemes.

Of course, the contents of [Lecture → Section 3.7] are prerequisite for this problem. It involves a little C++ coding based on EIGEN only.
▷ problem name/problem code folder: MaximumPrinciple

On a bounded domain $\Omega \subset \mathbb{R}^2$ and for a constant $0 \leq c < 1$ we consider the 2nd-order reaction-diffusion elliptic Dirichlet boundary value problem

$$\begin{aligned} -(1-c)\Delta u + cu &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned} \tag{3.7.1}$$

where the source function $f \in L^2(\Omega)$ is also continuous on $\overline{\Omega}$ and satisfies $f(\boldsymbol{x}) \leq 0$ for all $\boldsymbol{x} \in \Omega$.

(3-7.a) ⚀ (10 min.) State the variational formulation of (3.7.1) on a suitable Sobolev space.

SOLUTION for (3-7.a) → 3-7-1-0:s0.pdf ▲

(3-7.b) ⚁ (10 min.) [depends on Sub-problem (3-7.a)]

State a quadratic minimization problem on a suitable function space, whose solution agrees with the solution of (3.7.1).

SOLUTION for (3-7.b) → 3-7-2-0:max-principle0.pdf ▲

(3-7.c) ⚂ (20 min.) Following the reasoning of [Lecture → Section 3.7.1] argue why

$$\max_{\boldsymbol{x}\in\Omega} u(\boldsymbol{x}) = 0 \tag{3.7.3}$$

holds. Why is $f \leq 0$ important for your argument?

HIDDEN HINT 1 for (3-7.c) → 3-7-3-0:max-principle11.pdf

SOLUTION for (3-7.c) → 3-7-3-1:max-principle1.pdf ▲

For the finite element Galerkin discretization of (3.7.1) on $\Omega =]0,1[^2$ we employ the linear Lagrangian finite element space $\mathcal{S}^0_{1,0}(\mathcal{M})$ on a "equidistant tensor-product triangular mesh" $\mathcal{M}$ with meshwidth $h := 1/M+1$, $M \in \mathbb{N}$, as depicted in Fig. 46.

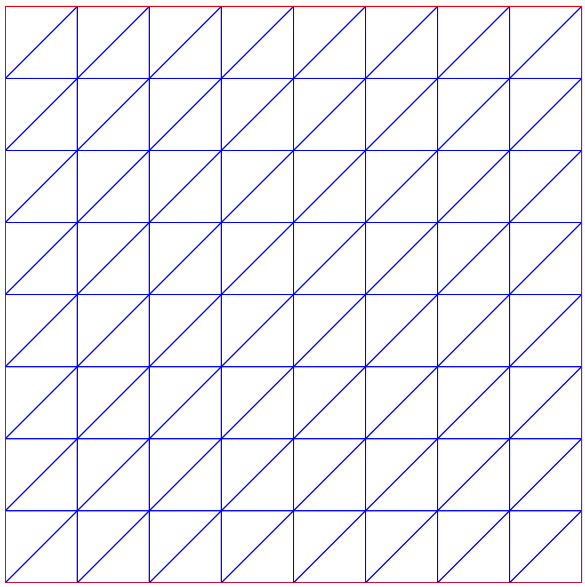


Fig. 46

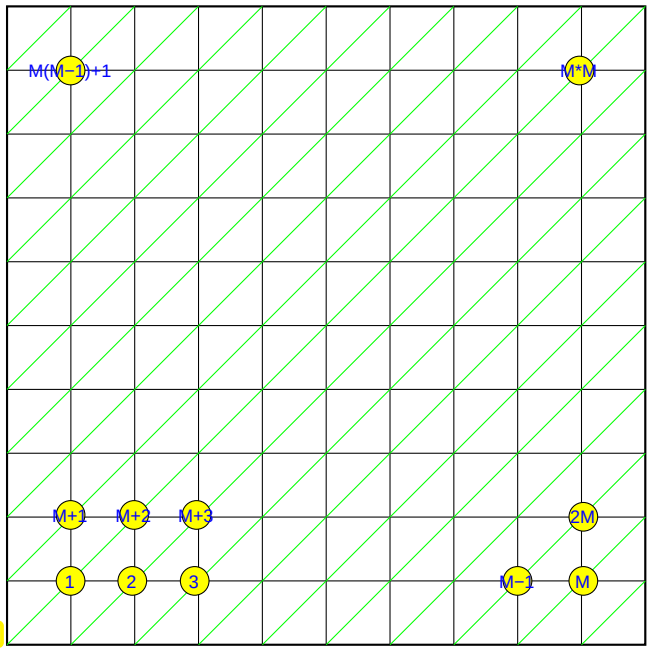


Fig. 47

Structured finite element mesh and lexikographic node numbering.

We number the nodes of the mesh **lexikographically** from left to right and bottom to top, see Fig. 47. Since, the "tent-function" global shape functions of $\mathcal{S}^0_{1,0}(\mathcal{M})$ are associated with the nodes of the mesh this induces a lexikographic ordering of the basis functions of $\mathcal{S}^0_{1,0}(\mathcal{M})$. This is used throughout this problem.

Remark. We point out that the discretization of $-\Delta$ by means of lowest-order piecewise linear Lagrangian finite elements on is presented in [Lecture → Section 4.1] in the context of finite difference schemes.

(3-7.d) ⚀ (10 min.) All triagles of $\mathcal{M}$ have the same shape and, therefore, will give rise to the same element matrices for the bilinear form $(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$ and the lowest-order Lagrangian finite element space $\mathcal{S}^0_1(\mathcal{M})$. Compute this element matrix.

SOLUTION for (3-7.d) → 3-7-4-0:s1b.pdf ▲

(3-7.e) ⚁ (15 min.) Compute the $\mathcal{S}^0_1(\mathcal{M})$-element matrix for the triangles of $\mathcal{M}$ and the bilinear form $(u,v) \mapsto \int_\Omega u\, v \,\mathrm{d}\boldsymbol{x}$. This element matrix is also known as **element mass matrix**.

SOLUTION for (3-7.e) → 3-7-5-0:1c.pdf ▲

(3-7.f) ⚀ (10 min.) What is $N := \dim \mathcal{S}^0_{1,0}(\mathcal{M})$? What is the maximal number of non-zero entries per row/column of the finite element Galerkin matrix for the $\mathcal{S}^0_{1,0}(\mathcal{M})$-Galerkin finite element discretization of (3.7.1)?

SOLUTION for (3-7.f) → 3-7-6-0:s1d.pdf ▲

(3-7.g) ⚄ (45 min.) [depends on Sub-problem (3-7.d), Sub-problem (3-7.e), Sub-problem (3-7.f)]

Write a C++ function

```
Eigen::SparseMatrix<double> computeGalerkinMatrix(int M, double c);
```

that initializes the finite element Galerkin matrix for (3.7.1), the mesh from Fig. 46 and the finite element space $\mathcal{S}^0_{1,0}(\mathcal{M})$, assuming lexikographic numbering of global shape functions. The argument `M` passes the number of nodes in each direction, see Fig. 47, and `c` the constant $c \in]0,1[$.

HIDDEN HINT 1 for (3-7.g) → 3-7-7-0:2h1.pdf

SOLUTION for (3-7.g) → 3-7-7-1:s2.pdf ▲

(3-7.h) ⚄ (20 min.) Implement a C++ function

```
template <typename FUNCTOR>
  Eigen::VectorXd computeLoadVector(int M, FUNCTOR &&f);
```

that computes the right-hand-side vector $\vec{\varphi} \in \mathbb{R}^N$ for the finite element Galerkin discretization of (3.7.1), using the mesh from Fig. 46 and the finite element space $\mathcal{S}^0_{1,0}(\mathcal{M})$, and assuming lexicographic numbering of global shape functions. The argument `f` passes a functor matching `std::function<double(double,double)>` for the source function f.

Evaluate the local integrals approximately with the 2D local trapezoidal rule [Lecture → (2.4.6.10)].

SOLUTION for (3-7.h) → 3-7-8-0:s2a.pdf ▲

(3-7.i) ⚄ (30 min.) Show by means of a counterexample that for certain values of the constant $c \in]0,1[$ the finite element discretization of (3.7.1) using the mesh from Fig. 46 with $M = 4$ and the finite element space $\mathcal{S}^0_{1,0}(\mathcal{M})$ violates the **discrete maximum principle** established in Sub-problem (3-7.c) in the sense that the finite element Galerkin solution $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ can attain values > 0 though $f \leq 0$.

HIDDEN HINT 1 for (3-7.i) → 3-7-9-0:max-principle21.pdf

SOLUTION for (3-7.i) → 3-7-9-1:max-principle2.pdf ▲

(3-7.j) ⚄ (15 min.) We consider the same finite element Galerkin discretization of (3.7.1) on $\mathcal{M}$ as above. However, now we evaluate all occurring local integrals by means of the local 2D trapezoidal rule [Lecture → (2.4.6.10)]

$$\int_K \phi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{1}{3}|K| \sum_{i=1}^{3} \phi(\boldsymbol{a}^i)\,, \quad K \in \mathcal{M}\,.$$

Write a C++ function

```
Eigen::SparseMatrix<double>
  computeGalerkinMatrixTR(int M, double c);
```

with the same purpose and arguments as `computeGalerkinMatrix()` from Sub-problem (3-7.g), but now employing the local 2D trapezoidal rule.

SOLUTION for (3-7.j) → 3-7-10-0:s4.pdf ▲

(3-7.k) ⚄ (20 min.) Show that for the Galerkin finite element discretization of (3.7.1) on $\Omega =]0,1[^2$ introduced in Sub-problem (3-7.k) the property $u_h(\boldsymbol{x}) \leq 0$ is satisfied for the finite element solution, if $f(\boldsymbol{x}) \leq 0$ for all $\boldsymbol{x} \in \Omega$.

HIDDEN HINT 1 for (3-7.k) → 3-7-11-0:max-principle41.pdf

SOLUTION for (3-7.k) → 3-7-11-1:max-principle4.pdf ▲

End Problem 3-7, 205 min.

Problem 3-8: Output Functionals for a 2nd-Order Elliptic Boundary Value Problem with Impedance Boundary Conditions

This problem studies linear output functional depending on the solution of a second-order elliptic boundary value problem. We have seen in class that sometimes these functionals can be recast in several equivalent forms, some of which offer superior accuracy in the context of finite element Galerkin discretization.

This problem is connected with [Lecture → Section 1.8] and [Lecture → Section 3.6.1] and it involves implementation in LEHRFEM++
▷ problem name/problem code folder: OutputImpedanceBVP

We write $\Omega_0 \subset [0.1, 0.9]^2$ for a bounded domain with piecewise smooth boundary strictly contained in the unit square $\square :=]0,1[^2$, see Fig. 50. On the domain $\Omega := \square \setminus \overline{\Omega_0}$ we consider the linear second-order elliptic boundary value problem with mixed Dirichlet/impedance boundary conditions:

$$-\Delta u = 0 \quad \text{in } \Omega\,, \tag{3.8.1a}$$
$$\mathbf{grad}\, u \cdot \boldsymbol{n}_0 - u = 0 \quad \text{on } \Gamma_0 := \partial\Omega_0\,, \tag{3.8.1b}$$
$$u = g \quad \text{on } \Gamma_g := \partial\square\,. \tag{3.8.1c}$$

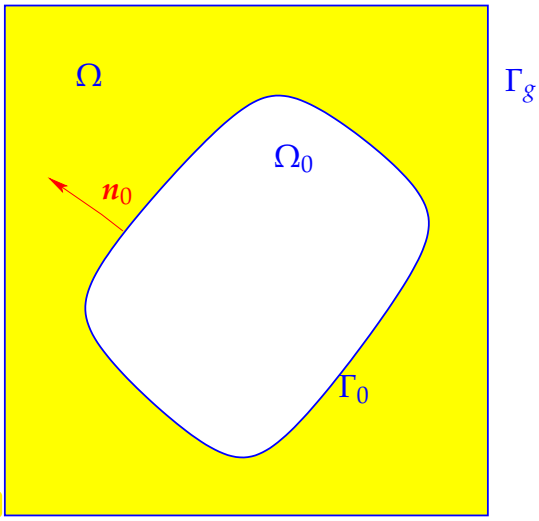


Fig. 50

Here, $\boldsymbol{n}_0$ is the *exterior* unit normal vectorfield on $\partial\Omega_0$, see Fig. 50, and the Dirichlet data g are given by the concrete expression

$$g(\boldsymbol{x}) := \mathbf{g} \cdot \boldsymbol{x}\,, \tag{3.8.2}$$

for a vector $\mathbf{g} \in \mathbb{R}^2$.

(3-8.a) ⚀ (15 min.)

State the complete variational formulation of (3.8.1a)–(3.8.1c).

SOLUTION for (3-8.a) → 3-8-1-0:s1.pdf ▲

(3-8.b) ⚁ (10 min.) [depends on Sub-problem (3-8.a)]

Argue why weak solutions of the boundary value problem (3.8.1) will always be unique.

SOLUTION for (3-8.b) → 3-8-2-0:s2.pdf ▲

(3-8.c) ⚂ (30 min.) [depends on Sub-problem (3-8.a)]

In the file `outputimpedancebvp.cc` complete the implementation of the function

```
Eigen::VectorXd
solveImpedanceBVP(
  const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
    &fe_space_p,
    Eigen::Vector2d g);
```

that discretizes (3.8.1) on a 2D hybrid mesh by means of lowest-order Lagrangian finite elements, that is, it realizes a finite element Galerkin discretization based on a suitable subspace of $\mathcal{S}_1^0(\mathcal{M})$, where $\mathcal{M}$ is a triangulation of Ω (possibly using a polygonal approximation of $\partial\Omega$). The argument `fe_space_p` passes a pointer to an object of type **lf::uscalfe::FeSpaceLagrangeO1** that contains both the underlying mesh object and the local-to-global index mapping. The function is supposed to

return the basis expansion coefficient vector of the finite element solution with respect to the nodal basis of the *full* finite element space $\mathcal{S}_1^0(\mathcal{M})$.

HIDDEN HINT 1 for (3-8.c) → 3-8-3-0:h31.pdf

SOLUTION for (3-8.c) → 3-8-3-1:s4.pdf ▲

(3-8.d) ⚂ (15 min.) [depends on Sub-problem (3-8.a)]

We consider a *uniformly shape-regular* sequence of meshes $(\mathcal{M}_\ell)_\ell$, $\ell \in \mathbb{N}_0$, of Ω with meshwidth h_ℓ of $\mathcal{M}_\ell$ tending to zero as $\ell \to \infty$.

Denote by $u \in H^2(\Omega)$ the exact solution of (3.8.1) and by $u_\ell \in \mathcal{S}_1^0(\mathcal{M})$ the finite element Galerkin solution of (3.8.1) & (3.8.2) for some given $\mathbf{g}$. Qualitatively and quantitatively describe the asymptotic convergence of the energy norm

$$\|u - u_\ell\|_{\mathsf{a}} := \left(\|\mathbf{grad}(u - u_\ell)\|^2_{L^2(\Omega)} + \|u - u_\ell\|^2_{L^2(\Gamma_0)} \right)^{\frac{1}{2}}$$

of the finite element discretization errors for $\ell \to \infty$, if Ω_0 is a disk.

SOLUTION for (3-8.d) → 3-8-4-0:s4.pdf ▲

In the remainder of this problem we examine the linear output functional

$$F(\boldsymbol{d}; v) := \int_{\Gamma_0} (\boldsymbol{d} \cdot \boldsymbol{x})\, \mathbf{grad}\, v(\boldsymbol{x}) \cdot \boldsymbol{n}_0(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \,. \tag{3.8.12}$$

for functions $v \in C^2(\overline{\Omega})$ and a vector $\boldsymbol{d} \in \mathbb{R}^2$, where the unit normal vectorfield $\boldsymbol{n}_0$ on Γ_0 is oriented as indicated in Fig. 50 (opposite to the exterior unit normal of Ω).

(3-8.e) ⚂ (10 min.)

Let u stand for the solution of (3.8.1) and assume that u is "sufficiently smooth". Then we have

$$F(\boldsymbol{d}; u) = F_\partial(\boldsymbol{d}; u) \quad \text{with} \quad v \mapsto F_\partial(\boldsymbol{d}; v) = \int_{\Gamma_0} \ldots \mathrm{d}S \quad \textit{continuous} \text{ on } H^1(\Omega) \,.$$

Give a concrete expression for $F_\partial(\boldsymbol{d}; v)$ in the form of an integral over Γ_0.

HIDDEN HINT 1 for (3-8.e) → 3-8-5-0:h51.pdf

SOLUTION for (3-8.e) → 3-8-5-1:s5.pdf ▲

(3-8.f) ⚄ (10 min.)

Again, let u stand for the solution of (3.8.1) and assume that u is "sufficiently smooth". Relying on a suitable cut-off function $\psi : \Omega \to \mathbb{R}$, of which you need to specify only general properties, give the formula for a *continuous* linear functional $F_\Omega(\boldsymbol{d}; \cdot) : H^1(\Omega) \to \mathbb{R}$, which has the form of an *integral over* Ω

$$F_\Omega(\boldsymbol{d}; v) = \int_\Omega \ldots \mathrm{d}\boldsymbol{x}$$

and satisfies

$$F_\Omega(\boldsymbol{d}; u) = F(\boldsymbol{d}; u) \,, \quad \boldsymbol{d} \in \mathbb{R}^2 \,.$$

SOLUTION for (3-8.f) → 3-8-6-0:s6.pdf ▲

(3-8.g) ⚄ (10 min.) [depends on Sub-problem (3-8.a) and Sub-problem (3-8.e)]

Give the strong (PDE) form of the boundary value problem corresponding to the dual variational problem for the output functional $v \mapsto F_\partial(\boldsymbol{d}, v)$ from Sub-problem (3-8.e) applied to the solution of (3.8.1).

SOLUTION for (3-8.g) → 3-8-7-0:s8.pdf ▲

(3-8.h) ⚄ (20 min.) [depends on Sub-problem (3-8.e)]

In the file `outputimpedancebvp.cc` implement an LEHRFEM++-based C++ function

```
double
computeBoundaryOutputFunctional(
  const Eigen::VectorXd eta,
  const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
     &fe_space_p,
  Eigen::Vector2d d);
```

which evaluates the boundary-integral-based version $F_\partial(\boldsymbol{d}; v)$ for $v \in \mathcal{S}_1^0(\mathcal{M})$. The vector `eta` passes the basis expansion coefficients of v with respect to the standard nodal basis of $\mathcal{S}_1^0(\mathcal{M})$, `fe_space_p` provides information about the mesh and the degrees of freedom, and `d` supplies the vector $\boldsymbol{d}$.

Polygonal approximation of Γ_0 is acceptable, but for polygonal Γ_0 the evaluation should be exact.

SOLUTION for (3-8.h) → 3-8-8-0:s7.pdf ▲

(3-8.i) ⚄ (25 min.) [depends on Sub-problem (3-8.c) and Sub-problem (3-8.h)]

In the files `evalclass.h` and `evalclass.cc` complete the implementation of the following class

```
class EvalResponse {
public:
  explicit EvalResponse(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
          &fe_space_p);
  double operator ()(Eigen::Vector2d g, Eigen::Vector2d d) const;
private:
  // suitable data members
}
```

This class should allow the *efficient* approximate evaluation of $F(\boldsymbol{d}, u(\mathbf{g}))$, for many different vectors $\mathbf{g}, \boldsymbol{d} \in \mathbb{R}^2$ through its evaluation operator **operator** `()`. Here $u(\mathbf{g})$ stands for the solution of (3.8.1) regarded as dependent on the vector $\mathbf{g}$ defining the Dirichlet data $g(\boldsymbol{x})$ according to (3.8.2).

In the implementation you can rely on the functions `solveImpedanceBVP()` from Sub-problem (3-8.c) `computeBoundaryOutputFunctional()` from Sub-problem (3-8.h).

HIDDEN HINT 1 for (3-8.i) → 3-8-9-0:h91.pdf

SOLUTION for (3-8.i) → 3-8-9-1:s9.pdf ▲

End Problem 3-8 , 145 min.

Problem 3-9: Zienkiewicz-Zhu A-Posteriori Error Estimator

A-posteriori error estimators provide information about a relevant norm of the finite element discretization error based on the computed finite element solution. In this problem we study the implementation and the behavior of a simple a-posteriori error estimator for the energy norm of the discretization error for linear Lagrangian finite elements applied to scalar second-order elliptic problems [Rod94].

Involves substantial implementation based on LEHRFEM++.
▷ problem name/problem code folder: ZienkiewiczZhuEstimator

On a polygonal domain $\Omega \subset \mathbb{R}^2$ we consider a Dirichlet boundary value problem for $-\Delta$:

$$-\Delta u = f \quad \text{in } \Omega \quad , \quad u = 0 \quad \text{on } \partial\Omega \, , \tag{3.9.1}$$

where $f \in C^0(\overline{\Omega})$ is given in procedural form [Lecture → Rem. 2.1.2.5].

Given a 2D triangular mesh $\mathcal{M}$ of Ω, we denote by $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ the finite element Galerkin solution of (3.9.1), obtained by employing some local quadrature formula for the right-hand-side vector, of course.

The starting point for the construction of the Zienkiewicz-Zhu a-posteriori error estimator is the approximation of $\mathbf{grad}\, u_h$ by means of a piecewise linear *continuous* vectorfield. Setting $\mathbf{V}_h := \mathcal{S}^0_1(\mathcal{M}) \times \mathcal{S}^0_1(\mathcal{M})$ for the space of C^0-vectorfields with piecewise linear components, we define

$$\mathbf{g}_h \in \mathbf{V}_h : \quad \int_\Omega \mathbf{g}_h \cdot \mathbf{v}_h \,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{grad}\, u_h \cdot \mathbf{v}_h \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{v}_h \in \mathbf{V}_h \, . \tag{3.9.2}$$

(3-9.a) ⚀ (10 min.) Show that $\mathbf{g}_h$ is the $L^2(\Omega)$-best approximation of $\mathbf{grad}\, u_h$ in $\mathbf{V}_h$

SOLUTION for (3-9.a) → 3-9-1-0:s1.pdf ▲

Denoting by $\{b^1_h, \ldots, b^N_h\}$, $N := \sharp\mathcal{V}(\mathcal{M})$, the standard tent function basis of $\mathcal{S}^0_1(\mathcal{M})$, we use

$$\left\{ \begin{bmatrix} b^1_h \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ b^1_h \end{bmatrix}, \begin{bmatrix} b^2_h \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ b^2_h \end{bmatrix}, \ldots, \begin{bmatrix} b^N_h \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ b^N_h \end{bmatrix} \right\} \tag{3.9.5}$$

as a basis for the $2N$-dimensional space $\mathbf{V}_h$. In our LEHRFEM++-based code the corresponding local-to-global index mapping represented by a **lf::assemble::DofHandler** object will be initialized as follows:

```
lf::assemble::UniformFEDofHandler dof_handler(
  mesh_p, {{lf::base::RefEl::kPoint(),   2},
           {lf::base::RefEl::kSegment(), 0},
           {lf::base::RefEl::kTria(),    0},
           {lf::base::RefEl::kQuad(),    0}});
```

where `mesh_p` is a (shared) pointer to an object of type **lf::mesh::Mesh** describing the underlying finite element mesh $\mathcal{M}$. Writing $b^p_h \in \mathcal{S}^0_1(\mathcal{M})$ for the tent function associated with the node $\boldsymbol{p} \in \mathcal{V}(\mathcal{M})$, we order the two basis functions of $\mathbf{V}_h$ associated with $\boldsymbol{p}$ as follows, *cf.* (3.9.5)

$$\text{first:} \quad \begin{bmatrix} b^p_h \\ 0 \end{bmatrix} \quad , \quad \text{second:} \quad \begin{bmatrix} 0 \\ b^p_h \end{bmatrix} . \tag{3.9.6}$$

(3-9.b) ⚀ (5 min.) Specify the set of local shape functions of the finite element space $\mathbf{V}_h$ for a triangle $K \in \mathcal{M}$ in terms of the barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ of that triangle.

SOLUTION for (3-9.b) → 3-9-2-0:s1a.pdfAny other ordering is also acceptable, but once LEHRFEM++ implementation is requested the above ordering is the only one compatible with LEHRFEM++'s internal conventions, see [Lecture → § 2.7.4.11]. ▲

(3-9.c) ⚁ (5 min.) [depends on Sub-problem (3-9.b)]

Denote by $\mathbf{M}_K^{\mathrm{vec}}$ the element matrix belonging to the finite element space $\mathbf{V}_h$, the bilinear form on the left-hand-side of (3.9.2), and a triangle $K \in \mathcal{M}$. How many entries of $\mathbf{M}_K^{\mathrm{vec}}$ will vanish for any K?

SOLUTION for (3-9.c) → 3-9-3-0:s1b.pdf ▲

(3-9.d) ⚄ (20 min.) [depends on Sub-problem (3-9.c)]

In the files `zienkiewiczhuEstimator.h` and `zienkiewiczhuEstimator.cc` you find the definition and a partial implementation of the class **VectorProjectionMatrixProvider** compatible with the concept of **ENTITY_MATRIX_PROVIDER** as required by the LEHRFEM++ function `AssembleMatrixLocally()`.

This class is supposed to supply the element matrices for the discrete variational problem (3.9.2) using the (ordered) global shape functions of $\mathbf{V}_h$ as introduced above.

Complete the implementation of the `Eval()` method of **VectorProjectionMatrixProvider** that provides the element matrices for *triangular cells*.

HIDDEN HINT 1 for (3-9.d) → 3-9-4-0:h22.pdf

SOLUTION for (3-9.d) → 3-9-4-1:s2.pdf

An alternative is to use numerical quadrature based on a quadrature rule that is exact for polynomials of degree ≤ 2. ▲

(3-9.e) ⚄ (20 min.) As in the previous sub-problem you are asked to supplement missing code for the `Eval()` method of the class **VectorProjectionMatrixProvider** in the file `zienkiewiczhuestimator.cc`, this time the code after the comment `//For QUADRILATERAL CELLS`, which is supposed to compute the $\mathbf{V}_h$ element matrix for the bilinear form on the left-hand-side of (3.9.2) and for *quadrilateral cells*.

HIDDEN HINT 1 for (3-9.e) → 3-9-5-0:h31.pdf

SOLUTION for (3-9.e) → 3-9-5-1:s3.pdf ▲

(3-9.f) ⚄ (15 min.) In the file `zienkiewiczhuestimator.cc` complete the implementation of the `Eval()` method for *triangular cells* (only!) of the following class

```
class GradientProjectionVectorProvider {
public:
  explicit GradientProjectionVectorProvider(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
      &fe_space_p,
    const Eigen::VectorXd &mu);
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true; }
  Eigen::VectorXd Eval(const lf::mesh::Entity &entity);
private:
  std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
    _fe_space_p;
  Eigen::VectorXd _mu;
};
```

This class is meant to comply with the concept of an **ENTITY_VECTOR_PROVIDER** and supposed to compute the element vectors for the right-hand-side linear form of the discrete variational problem (3.9.2). Through the argument `mu` of its constructor we communicate the $\mathcal{S}_1^0(\mathcal{M})$ basis expansion coefficient vector of u_h, while `fe_space_p` passes the mesh $\mathcal{M}$ and the local-to-global index mapping.

HIDDEN HINT 1 for (3-9.f) → 3-9-6-0:hgpc.pdf

SOLUTION for (3-9.f) → 3-9-6-1:s4.pdf ▲

The solution (3.9.2) involves solving a linear system of equations for a large sparse matrix. This can be avoided when using the local trapezoidal rule

$$\int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{n_K}\sum_{\ell=1}^{n_K}\psi(\boldsymbol{a}_K^\ell);, \quad K\in\mathcal{M}\,, \tag{3.9.15}$$

where $\boldsymbol{a}_K^1,\ldots,\boldsymbol{a}_K^{n_K}$ are the vertices of the cell K, e.g. $n_K=3$ for triangles. If (3.9.15) is used for assembly, the Galerkin matrix for (3.9.2) becomes a diagonal matrix.

(3-9.g) ⚀ (12 min.) Assume that (3.9.2) is solved approximately on a triangular mesh using the local quadrature formula (3.9.15) and write $\widetilde{\mathbf{g}}_h\in\mathbf{V}_h$ for its solution. Writing $\boldsymbol{d}_K$ for the gradient of u_h on a triangle K, give an explicit formula for the value $\widetilde{\mathbf{g}}_h(\mathbf{p})$, where $\mathbf{p}$ is a node of the mesh.

SOLUTION for (3-9.g) → 3-9-7-0:s5.pdf ▲

(3-9.h) ⚄ (20 min.) [depends on Sub-problem (3-9.g)]

For a *triangular mesh* implement a C++ function

```
Eigen::VectorXd computeLumpedProjection(
  const lf::assemble::DofHandler &scal_dofh,
  const Eigen::VectorXd &mu,
  const lf::assemble::DofHandler &vec_dofh);
```

that computes the $\mathbf{V}_h$-basis-expansion coefficient vector for $\widetilde{\boldsymbol{g}}_h\in\mathbf{V}_h$ without using LEHRFEM++'s functions `AssembleMatrixLocally` and `AssembleVectorLocally`, but by directly evaluating the formula obtained in Sub-problem (3-9.g).

SOLUTION for (3-9.h) → 3-9-8-0:s6.pdf ▲

(3-9.i) ⚄ (30 min.) For a *triangular mesh* implement a C++ function

```
double computeL2Deviation(
  const lf::assemble::DofHandler &scal_dofh,
  const Eigen::VectorXd &eta,
  const lf::assemble::DofHandler &vec_dofh
  const Eigen::VectorXd &gamma);
```

that evaluates the deviation norm

$$\|\mathbf{r}_h-\mathbf{grad}\,v_h\|_{L^2(\Omega)}\,,\quad \mathbf{r}_h\in\mathbf{V}_h\,,\quad v_h\in\mathcal{S}_1^0(\mathcal{M})\,,$$

where the arguments have the following meaning

- `scal_dofh` is a **lf::assemble::DofHandler** object encoding the local-to-global index mapping for the finite element space $\mathcal{S}_1^0(\mathcal{M})$,
- `eta` is the basis-expansion coefficient vector for v_h,
- `vec_dofh` gives a **lf::assemble::DofHandler** object for the vector-valued finite element space $\mathbf{V}_h$. It follows the convention introduced above.
- `gamma` provides the coefficient vector defining $\mathbf{r}_h$.

HIDDEN HINT 1 for (3-9.i) → 3-9-9-0:s6h1.pdf

HIDDEN HINT 2 for (3-9.i) → 3-9-9-1:s6h2.pdf

SOLUTION for (3-9.i) → 3-9-9-2:.pdf ▲

Now we consider the particular boundary value problem with $\Omega =]0,1[^2$, $f(\boldsymbol{x}) := \sin(\pi x_1)\sin(2\pi x_2)$, which yields the exact solution $u(\boldsymbol{x}) = \frac{1}{5\pi^2}\sin(\pi x_1)\sin(2\pi x_2)$ for (3.9.1). Starting from a coarse unstructured triangular mesh $\mathcal{M}_0$ we obtain meshes $\mathcal{M}_0 \prec \mathcal{M}_1 \prec \mathcal{M}_2 \prec \ldots$ by successive global regular refinement.

(3-9.j) ⚄ (15 min.) We write $u_{h,\ell} \in \mathcal{S}_1^0(\mathcal{M}_\ell)$ for the finite element Galerkin solution of (3.9.1) on $\mathcal{M}_\ell$ and $\widetilde{\mathbf{g}}_{h,\ell} \in \mathbf{V}_\ell$, $\mathbf{V}_\ell := \mathcal{S}_1^0(\mathcal{M}_\ell) \times \mathcal{S}_1^0(\mathcal{M}_\ell)$ for the approximately projected gradient of $u_{h,\ell}$.

The following table lists the quantities

- $\delta_\ell := \|\mathbf{g}_{h,\ell} - \mathbf{grad}\, u_{h,\ell}\|_{L^2(\Omega)}$ and
- $\epsilon_\ell := \left| |u_{h,\ell} - u|_{H^1(\Omega)} - \|\mathbf{g}_{h,\ell} - \mathbf{grad}\, u_{h,\ell}\|_{L^2(\Omega)} \right|$

for $\ell = 1,2,3,4,5$.

ℓ	1	2	3	4	5
δ_ℓ	0.0039	0.0019	0.00099	0.00049	0.00025
ϵ_ℓ	7.18e-05	1.90e-05	4.79e-06	1.18e-06	2.90e-07

Describe qualitatively and quantitatively the asymptotic convergence of δ_ℓ and ϵ_ℓ as functions of the meshwidth $h_\ell = h(\mathcal{M}_\ell)$ as $\ell \to \infty$.

SOLUTION for (3-9.j) → 3-9-10-0:s8.pdf ▲

References

[Rod94] Rodolfo Rodríguez. "Some remarks on Zienkiewicz-Zhu estimator". In: *Numer. Methods Partial Differential Equations* 10.5 (1994), pp. 625–635. DOI: 10.1002/num.1690100509 (cit. on p. 139).

End Problem 3-9 , 152 min.

Problem 3-10: Parametric Finite Elements

The idea underlying parametric finite elements [Lecture → Section 2.8] can also be used to map boundary value problems to simpler domains. This possibility will be demonstrated in this problem and then exploited for finite element discretization on a structured mesh.

This problem requires skills in transforming variational problems as you should have learned them in [Lecture → Section 2.8.3]. Substantial C++ implementation based on EIGEN will be part of this problem.
▷ problem name/problem code folder: ParametricFiniteElements

To model the temperature distribution inside the earth's crust with topography we rely on the following boundary value problem

$$\begin{aligned} -\operatorname{div}(\alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) &= 0 \text{ on } \Omega\,, \\ \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x}) + u(\boldsymbol{x}) &= 0 \text{ on } \Gamma_S\,, \\ \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x}) &= 0 \text{ on } \Gamma_N\,, \\ u(\boldsymbol{x}) &= 1 \text{ on } \Gamma_D\,, \end{aligned} \tag{3.10.1}$$

where $u = u(\boldsymbol{x})$ stands for the unknown temperature distribution, $\alpha = \alpha(\boldsymbol{x})$ denotes a uniformly positive heat conductivity, and the domain is

$$\Omega := \left\{ [x_1, x_2]^T \in \mathbb{R}^2 : 0 \le x_1 \le 1 \text{ and } 0 \le x_2 \le \Psi(x_1) \right\} \tag{3.10.2}$$

for the topography function $\Psi \in C^1([0,1])$, $\Psi > 0$, and the domain boundaries are

$$\begin{aligned} \Gamma_S &:= \left\{ [x_1, x_2]^T \in \mathbb{R}^2 : 0 \le x_1 \le 1 \text{and } x_2 = \Psi(x_1) \right\}, \\ \Gamma_N &:= \left\{ [x_1, x_2]^T \in \mathbb{R}^2 : x_1 = 0 \text{ and } x_1 = 1 \right\}, \\ \Gamma_D &:= \left\{ [x_1, x_2]^T \in \mathbb{R}^2 : x_2 = 0 \right\}. \end{aligned} \tag{3.10.3}$$

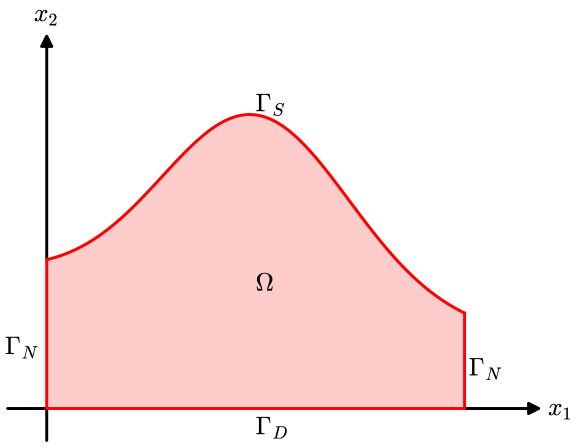


Fig. 51

◁ Computational domain Ω. The top boundary part Γ_S is defined as the graph of the function $\Psi : [0,1] \to \mathbb{R}^+$.

The finite element discretization of (3.10.1) is done by means of bilinear parametric Lagrangian finite elements [Lecture → Ex. 2.6.2.1] based on the mesh

$$\mathcal{M} := \left\{ K_{j,l} : 0 \le j, l < n \right\}, \quad n \in \mathbb{N}\,, \tag{3.10.4}$$

where the quadrilaterals $K_{j,l}$ (with straight edges) are defined through their vertices

$$\begin{aligned} \mathbf{a}^1 &= \begin{bmatrix} jh \\ lh\Psi(jh) \end{bmatrix}, \ \mathbf{a}^2 = \begin{bmatrix} (j+1)h \\ lh\Psi((j+1)h) \end{bmatrix}, \\ \mathbf{a}^3 &= \begin{bmatrix} (j+1)h \\ (l+1)h\Psi((j+1)h) \end{bmatrix}, \ \mathbf{a}^4 = \begin{bmatrix} jh \\ (l+1)h\Psi(jh) \end{bmatrix}. \end{aligned} \tag{3.10.5}$$

Moreover, $h = 1/n$ denotes the mesh width and $n \in \mathbb{N}$ is the number of mesh cells in one direction.

We make use of the following local parametric quadrature rule on the reference quadrilateral (unit square $\hat{K} =]0,1[^2$), the local trapezoidal rule

$$\int_{\hat{K}} \varphi(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \approx \frac{1}{4}\Big(\varphi([0,0]^T) + \varphi([1,0]^T) + \varphi([0,1]^T) + \varphi([1,1]^T)\Big) \, . \tag{3.10.6}$$

(3-10.a) ⚂ (15 min.) State the variational formulation of (3.10.1).

HIDDEN HINT 1 for (3-10.a) → 3-10-1-0:param-FE11.pdf

SOLUTION for (3-10.a) → 3-10-1-1:param-FE1.pdf ▲

(3-10.b) ⚂ (10 min.) Sketch the mesh $\mathcal{M}$ for $n = 3$ and $\Psi(x_1) = x_1^2 + 1$.

SOLUTION for (3-10.b) → 3-10-2-0:param-FE1p.pdf ▲

(3-10.c) ⚂ (15 min.)

Give the formula for the unique *bilinear transformation* $\boldsymbol{\Phi}_{j,l} : \mathbb{R}^2 \to \mathbb{R}^2$ taking the reference quadrilateral with vertices

$$\hat{\mathbf{a}}^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \ \hat{\mathbf{a}}^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \hat{\mathbf{a}}^3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \ \hat{\mathbf{a}}^4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \tag{3.10.8}$$

to $K_{j,l}$ for $0 \le j,l < n$, i.e. find $\boldsymbol{\Phi}_{j,l}$ such that $\boldsymbol{\Phi}_{j,l}(\hat{\mathbf{a}}_i) = \mathbf{a}_i$, $i = 1,2,3,4$, where the $\mathbf{a}_i$ are given by (3.10.5).

HIDDEN HINT 1 for (3-10.c) → 3-10-3-0:sp2h1.pdf

SOLUTION for (3-10.c) → 3-10-3-1:param-FE2.pdf ▲

(3-10.d) ⚂ (15 min.) Compute the Jacobian and the transposed of the inverse Jacobian of your mapping found in Sub-problem (3-10.c)

$$\begin{aligned} \mathbf{J}_{j,l}(\hat{\boldsymbol{x}}) &:= D\boldsymbol{\Phi}_{j,l}(\hat{\boldsymbol{x}}) \in \mathbb{R}^{2,2} \, , \\ \mathbf{J}_{j,l}^{-T}(\hat{\boldsymbol{x}}) &:= D\boldsymbol{\Phi}_{j,l}^{-T}(\hat{\boldsymbol{x}}) \in \mathbb{R}^{2,2} \, , \end{aligned} \tag{3.10.10}$$

as an expression depending on j, l, Ψ and h.

HIDDEN HINT 1 for (3-10.d) → 3-10-4-0:param-FE31.pdf

SOLUTION for (3-10.d) → 3-10-4-1:param-FE3.pdf ▲

(3-10.e) ⚂ (20 min.) [depends on Sub-problem (3-10.d)]

Write two C++ functions

```
template <typename FUNCTOR>
double integrationElement(unsigned int n,
 unsigned int j, unsigned int l,
 FUNCTOR &&Psi, Eigen::Vector2d xhat);
```

and

```
template <typename FUNCTOR>
Eigen::Matrix2d jacobianInverseTransposed(unsigned int n,
  unsigned int j, unsigned int l,
  FUNCTOR &&Psi, Eigen::Vector2d xhat);
```

that compute the absolute value of the determinant of the Jacobian $|\det(\mathbf{J}_{j,l}(\hat{\boldsymbol{x}}))|$ and the inverse Jacobian transposed $\mathbf{J}_{j,l}^{-\top}(\hat{\boldsymbol{x}})$ based on your results from Sub-problem (3-10.d). These functions take as input the mesh discretization parameter n, the indices of the considered element j and l, the topography function Ψ and the point in the reference element $\hat{\boldsymbol{x}}$ where their evaluation is desired.

SOLUTION for (3-10.e) → 3-10-5-0:param-FE3p.pdf ▲

(3-10.f) ⚂ (20 min.) Write a C++ function

```
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::MatrixXd geoThermElemMat(unsigned int n,
  unsigned int j, unsigned int l,
  FUNCTOR1 &&alpha, FUNCTOR2 &&Psi)
```

that computes the *volume contributions* to the element matrix for $K_{j,l}$ with the quadrature rule (3.10.6) on the reference quadrilateral element.

The function's input parameters are the mesh discretization parameter n, the indices of the considered element j and l, the topography function Ψ and the scalar-valued material function $\alpha = \alpha(\boldsymbol{x})$.

For your convenience the values of the four reference shape/basis functions and their gradients on the reference element are provided by the functions

```
Eigen::Vector4d bhats(Eigen::Vector2d xhat);
```

and

```
Eigen::MatrixXd bhats_grad(Eigen::Vector2d xhat);
```

The return vector of `bhats()` contains the values of the reference shape functions in its components. The numbering is the same as that of the vertices in (3.10.8). The columns of the matrix furnished by `bhats_grad()` give the gradients at the passed point.

HIDDEN HINT 1 for (3-10.f) → 3-10-6-0:param-FE42.pdf

SOLUTION for (3-10.f) → 3-10-6-1:param-FE4.pdf ▲

(3-10.g) ⚄ (15 min.) Assume the following ordering of the vertices $P_{j,l} = [jh, lh\Psi(jh)]^T$, $0 \le j,l \le n$, of the mesh $\mathcal{M}$:

$$P_{0,0}, P_{1,0}, P_{2,0}..., P_{n,0}, P_{0,1}, P_{1,1}, ..., P_{0,n}, P_{1,n}, ...P_{n,n}\ , \tag{3.10.18}$$

which is known as **lexicographical ordering**. It induces an ordering of the nodal basis functions b_1, b_2, ..., b_N, $N := (n+1)^2$, spanning the space $\mathcal{S}_1^0(\mathcal{M})$ of parametric bilinear Lagrangian finite elements.

Numbering of mesh nodes $P_{i,j}$ for $n = 3$, global node numbers in purple:

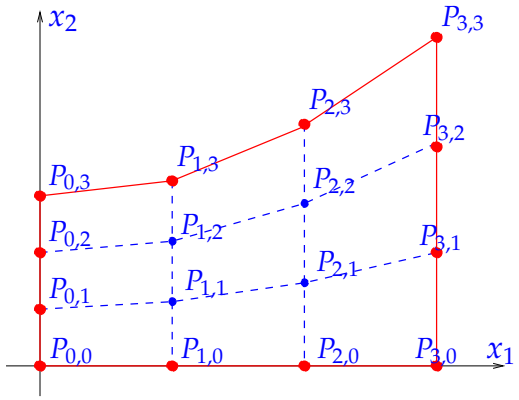


Fig. 54

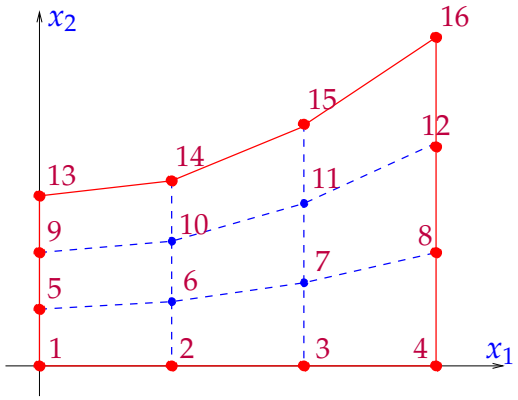


Fig. 55

Which entries of the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ for the bilinear form associated with (3.10.1) will invariably be zero?

SOLUTION for (3-10.g) → 3-10-7-0:param-FE5.pdf ▲

(3-10.h) ⚀ (15 min.) Assuming the ordering of basis functions induced by the lexicographical ordering (3.10.18), what are the global index numbers of the four local shape functions associated with the corners $\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3, \mathbf{a}^4$ (in that order, see (3.10.5)) of $K_{j,l}$?

Write a C++ function

```
int geoThermLocalToGlobal(unsigned int n,
  unsigned int j, unsigned int l,
  unsigned int local_dof);
```

that returns the global index of the local shape function `local_dof` on element $K_{j,l}$.

SOLUTION for (3-10.h) → 3-10-8-0:param-FE6.pdf ▲

(3-10.i) ⚁ (30 min.) [depends on Sub-problem (3-10.g) and Sub-problem (3-10.h)]

Write a C++ function

```
template <typename FUNCTOR1, typename FUNCTOR2>
  std::vector<Eigen::Triplet<double>>
    assembleGeoTherm(unsigned int n,
      FUNCTOR1 &&alpha, FUNCTOR2 &&Psi);
```

that computes the Galerkin matrix in triplet/COO format based on the element matrices from Sub-problem (3-10.g) and the local to global index map from Sub-problem (3-10.h).

SOLUTION for (3-10.i) → 3-10-9-0:param-FE7.pdf ▲

(3-10.j) ⚂ (30 min.) [depends on Sub-problem (3-10.i)]

We write $\mathbf{A}$ for the Galerkin matrix obtained in Sub-problem (3-10.i). Implement a C++ function

```
void geoThermBdElim(unsigned int n,
   std::vector<Eigen::Triplet<double>> &A)
```

that accomplishes the following transformation of $\mathbf{A}$ given in triplet format:

> If the m-th row of $\mathbf{A}$ belongs to a node on Γ_D, then replace it with $\mathbf{e}_m^\top$ ($\mathbf{e}_m$ is the m-th unit vector).

SOLUTION for (3-10.j) → 3-10-10-0:param-FE8.pdf ▲

(3-10.k) ⚄ (30 min.) Implement the C++ function

```
template <typename FUNCTOR1, typename FUNCTOR2>
  Eigen::VectorXd geoThermSolve(
    unsigned int n,
    FUNCTOR1 &&alpha,
    FUNCTOR2 &&Psi);
```

that computes the basis expansion coefficient vector $\boldsymbol{\mu}$ of the finite element solution u_h of (3.10.1) in $\mathcal{S}_1^0(\mathcal{M})$. You should rely on the `assembleGeoTherm()` implemented in a previous exercise. Do not forget to account for the Dirichlet boundary condition using the function `geoThermBdElim()` and to implement the correct right-hand side vector. You may use the `Eigen::SparseLU` solver to solve the linear system. Explain the details of the implementation by commenting your code.

SOLUTION for (3-10.k) → 3-10-11-0:param-FE9.pdf ▲

(3-10.l) Again assume the lexicographic numbering of basis functions from (3.10.5). Implement a C++ function

```
template <typename FUNCTOR>
double geoThermSurfInt(unsigned int n, FUNCTOR &&Psi, const
   Eigen::VectorXd &mu)
```

that takes a vector of basis expansion coefficients $\boldsymbol{\mu}$ and computes an approximation of the surface integral

$$\int_{\Gamma_S} u_h \, \mathrm{d}S \,. \tag{3.10.23}$$

Here `Psi` is again a factor that passes the topography function Ψ.

SOLUTION for (3-10.l) → 3-10-12-0:param-FE10.pdf ▲

(3-10.m) ⚂ (10 min.) What variational crimes are committed in the Galerkin finite element discretization discussed above?

SOLUTION for (3-10.m) → 3-10-13-0:param-FE11.pdf ▲

(3-10.n) ⚂ (10 min.) Writing u for the exact solution of (3.10.1) and assuming $u \in H^2(\Omega)$, $1 \leq \alpha(x) \leq 2$, what asymptotic behavior of $|u - u_h|_{H^1(\Omega)}$ in terms of $N \to \infty$ will be observed?

HIDDEN HINT 1 for (3-10.n) → 3-10-14-0:pfe12h.pdf

SOLUTION for (3-10.n) → 3-10-14-1:param-FE12.pdf ▲

End Problem 3-10 , 235 min.

Problem 3-11: Stable Evaluation at a Point

In [Lecture → Section 3.6.2] we saw an amazing case, where the conversion of a unbounded (w.r.t. the natural energy norm) output functional into an equivalent bounded functional worked miracles in terms of enhancing accuracy. In this problem we will apply a similar manipulation to the point evaluation output functional.

In this problem you will be asked to implement a few simple functions in C++ in the LEHRFEM++ framework.
▷ problem name/problem code folder: StableEvaluationAtAPoint

On a bounded domain Ω with polygonal connected boundary $\partial\Omega$ we consider the Dirichlet problem for the Laplace equation

$$\Delta u = 0 \quad \text{in } \Omega, \quad u = g \quad \text{on } \partial\Omega, \tag{3.11.1}$$

for a sufficiently smooth continuous function $g : \partial\Omega \mapsto \mathbb{R}$.

From now on let $\boldsymbol{x} \in \Omega$ be fixed. We are interested in an approximation of $u(\boldsymbol{x})$, that is, we aim for an approximate evaluation of the point value output functional

$$J(u) := u(\boldsymbol{x}) \,. \tag{3.11.2}$$

However, from [Lecture → Ex. 1.2.3.45] we know that J is not continuous with respect to the energy norm associated with (3.11.1), which denies us the fast convergence endowed by the duality argument of [Lecture → Thm. 3.6.1.7].

As a starting point for a suitable modification of J we consider the function

$$G_{\boldsymbol{x}}(\boldsymbol{y}) = -\frac{1}{2\pi}\log(\|\boldsymbol{x}-\boldsymbol{y}\|)\,, \quad \boldsymbol{y} \in \Omega\,, \quad \boldsymbol{y} \neq \boldsymbol{x}\,, \tag{3.11.3}$$

and the two functionals on $C^0(\overline{\Omega})$

$$P_{\mathrm{SL}}(v)(\boldsymbol{x}) := \int_{\partial\Omega} v(\boldsymbol{y}) G_{\boldsymbol{x}}(\boldsymbol{y}) \,\mathrm{d}S(\boldsymbol{y})\,, \qquad v \in C^0(\overline{\Omega})\,, \tag{3.11.4}$$

$$P_{\mathrm{DL}}(v)(\boldsymbol{x}) := \int_{\partial\Omega} v(\boldsymbol{y}) (\mathbf{grad}\, G_{\boldsymbol{x}})(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y}) \,\mathrm{d}S(\boldsymbol{y})\,, \qquad v \in C^0(\overline{\Omega})\,, \tag{3.11.5}$$

where $\boldsymbol{n}$ is the exterior unit normal vector field on $\partial\Omega$.

(3-11.a) ⚄ (40 min.) Implement the LEHRFEM++-based C++ functions

```
template <typename FUNCTOR>
double PSL(std::shared_ptr<const lf::mesh::Mesh> mesh, FUNCTOR &&v,
           const Eigen::Vector2d x);
```

and

```
template <typename FUNCTOR>
double PDL(std::shared_ptr<const lf::mesh::Mesh> mesh, FUNCTOR &&v,
           const Eigen::Vector2d x);
```

that evaluate P_{SL} and P_{DL}, respectively, for a triangulated polygonal domain. The parameter `mesh` passes a **lf::mesh::Mesh** object, which also describes the domain Ω. `FUNCTOR v` with signature

`std::function<double(Vector2d)>` gives the argument function $v : \overline{\Omega} \to \mathbb{R}$, and `x` is a 2-vector with the coordinates of $\boldsymbol{x}$.

The approximate evaluation of the integrals should be done by means of the local midpoint rule on the partitioning of $\partial\Omega$ induced by the mesh. For the implementation you can use the two already supplied functions

```
double G(Eigen::Vector2d x, Eigen::Vector2d y);
```

and

```
Eigen::Vector2d gradG(Eigen::Vector2d x, Eigen::Vector2d y);
```

that return $G_{\boldsymbol{x}}(\boldsymbol{y})$ and $\mathbf{grad}\, G_{\boldsymbol{x}}(\boldsymbol{y})$; the inputs `x` and `y` are vectors of coordinates of a point.

For the computation of the outward-pointing unit normal $\boldsymbol{n}$, you may use a shortcut rely on the fact that all supplied meshes for this problem provide triangulations of the unit square: $\Omega =]0,1[^2$. Thus $\boldsymbol{n}(\boldsymbol{x})$ for any $\boldsymbol{x} \in \partial\Omega$ can easily be determined from the coordinates of $\boldsymbol{x}$.

Remark. Of course, there is a more general way to compute $\boldsymbol{n}(\boldsymbol{x})$, but this entails visiting the cells adjacent to boundary edges, an access operation not really supported by LEHRFEM++'s data structures.

HIDDEN HINT 1 for (3-11.a) → `3-11-1-0:spe11.pdf`

SOLUTION for (3-11.a) → `3-11-1-1:spe1.pdf` ▲

(3-11.b) ⚄ (30 min.) [depends on Sub-problem (3-11.a)]

Somebody contends that for a harmonic function u on Ω, that is, $\Delta u = 0$, holds

$$u(\boldsymbol{x}) = P_{\mathrm{SL}}(\mathbf{grad}\, u \cdot \boldsymbol{n}) - P_{\mathrm{DL}}(u) \ . \tag{3.11.8}$$

Complete the code of the C++ function

```
double pointEval(std::shared_ptr<const lf::mesh::Mesh> mesh);
```

that is supposed to provide evidence for this claim for

- $\Omega =]0,1[^2$ (unit square),
- $\boldsymbol{x} = \begin{bmatrix} 0.3 \\ 0.4 \end{bmatrix}$,
- and $u(\boldsymbol{x}) = \log\left(\left\|\boldsymbol{x} + \binom{1}{0}\right\|\right)$, which satisfies $\Delta u = 0$ in Ω.

The function should return the deviation $|u(\boldsymbol{x}) - P_{\mathrm{SL}}(\mathbf{grad}\, u \cdot \boldsymbol{n}) + P_{\mathrm{DL}}(u)|$.

Apply the functions `PSL()` and `PDL()` to this u to evaluate the right hand side of (3.11.8) approximately for a sequence of meshes obtained by global regular refinement. Then you should plot the difference of both sides versus the mesh-width in a doubly logarithmic ('loglog') plot.

HIDDEN HINT 1 for (3-11.b) → `3-11-2-0:spe21.pdf`

SOLUTION for (3-11.b) → `3-11-2-1:spe2.pdf` ▲

(3-11.c) Assume that a function $\Psi_x \in C^2(\overline{\Omega})$ is given with the following properties:

(P1) $0 \le \Psi_x \le 1$;
(P2) $\Psi_x \equiv 1$ close to $\partial\Omega$;
(P3) $\Psi_x \equiv 0$ in a ball around x.

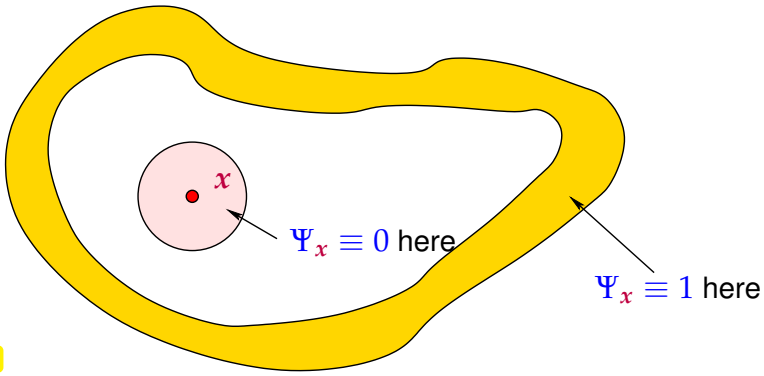


Fig. 59

By applying Green's formula [Lecture → Thm. 1.5.2.7], show that, if $\Delta u = 0$ in Ω,

$$P_{\mathrm{SL}}(\mathbf{grad}\, u \cdot \boldsymbol{n}) = \int_\Omega \mathbf{grad}\, u(\boldsymbol{y}) \cdot \mathbf{grad}(G_x \Psi_x)(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y}\,. \tag{3.11.10}$$

Note that $G_x\Psi_x$ is a smooth function.

SOLUTION for (3-11.c) → 3-11-3-0:spe3.pdf ▲

(3-11.d) [depends on Sub-problem (3-11.c)]

Appealing to the result (3.11.10) of sub-problem (3-11.c), use Green's formula [Lecture → Thm. 1.5.2.7] again to establish

$$\begin{aligned} P_{\mathrm{SL}}(\mathbf{grad}\, u \cdot \boldsymbol{n}) &= \int_{\partial\Omega} (\mathbf{grad}\, u(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y}))\, G_x\, \mathrm{d}S(\boldsymbol{y}) \\ &= -\int_\Omega u(\boldsymbol{y})\, (2\,\mathbf{grad}\, G_x(\boldsymbol{y}) \cdot \mathbf{grad}\, \Psi_x(\boldsymbol{y}) + G_x(\boldsymbol{y})\Delta\Psi_x(\boldsymbol{y}))\, \mathrm{d}\boldsymbol{y} \\ &\quad + P_{\mathrm{DL}}(u)\,. \end{aligned} \tag{3.11.12}$$

HIDDEN HINT 1 for (3-11.d) → 3-11-4-0:spe41.pdf

SOLUTION for (3-11.d) → 3-11-4-1:spe4.pdf ▲

(3-11.e) From (3.11.12) and (3.11.8) we conclude that for the solution u of (3.11.1) we have the equivalent form of the point evaluation functional

$$J(u) = J^*(u) := -\int_\Omega u(\boldsymbol{y})\, (2\,\mathbf{grad}\, G_x(\boldsymbol{y}) \cdot \mathbf{grad}\, \Psi_x(\boldsymbol{y}) + G_x(\boldsymbol{y})\Delta\Psi_x(\boldsymbol{y}))\, \mathrm{d}\boldsymbol{y}\,. \tag{3.11.14}$$

Show that J^* is continuous on $H^1_0(\Omega)$ with respect to the energy norm associated with (3.11.1).

SOLUTION for (3-11.e) → 3-11-5-0:spe5.pdf ▲

(3-11.f) We consider $\Omega =]0,1[^2$ and know a priori that $\left\| \boldsymbol{x} - \binom{0.5}{0.5} \right\| \le \frac{1}{4}$. Hence, independently of $\boldsymbol{x}$ we can choose

$$\Psi(\boldsymbol{y}) := \Psi_x(\boldsymbol{y}) = \begin{cases} 0 & \text{, if } \left\| \boldsymbol{y} - \binom{0.5}{0.5} \right\| \le \frac{1}{4}\sqrt{2}\,, \\ \cos^2\left(\frac{\pi}{\frac{1}{2}\sqrt{2}-1} \left(\left\| \boldsymbol{y} - \binom{0.5}{0.5} \right\| - \frac{1}{2} \right) \right)\,, & \\ 1 & \text{, if } \left\| \boldsymbol{y} - \binom{0.5}{0.5} \right\| \ge \frac{1}{2}\,. \end{cases} \tag{3.11.17}$$

which is in the space $C^2_{\mathrm{pw}}(\overline{\Omega}) \subset C^1(\overline{\Omega})$. This function is already implemented in the C++ functions

```
double Psi(const Eigen::Vector2d y);
Eigen::Vector2d gradPsi(const Eigen::Vector2d y);
double laplPsi(const Eigen::Vector2d y);
```

which returns $\Psi(\boldsymbol{y})$, $\mathbf{grad}\,\Psi(\boldsymbol{y})$, and $\Delta\Psi(\boldsymbol{y})$.

Write a C++ function

```
double Jstar(
  std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
  const Eigen::VectorXd &mu,
  const Eigen::Vector2d x);
```

that takes a finite element space defined on a triangular mesh of the square domain $\Omega =]0,1[^2$, the coefficient vector `mu` for a finite element function $u_N \in \mathcal{S}_1^0(\mathcal{M})$ (w.r.t. to the standard tent function, of course), and the point coordinates `x` of $\boldsymbol{x} \in \Omega$ as arguments and returns an approximation of $J^*(u_N)$ from (3.11.14). Use the usual midpoint quadrature rule for the approximate evaluation of the integral in (3.11.14).

HIDDEN HINT 1 for (3-11.f) → 3-11-6-0:spe61.pdf

SOLUTION for (3-11.f) → 3-11-6-1:spe6.pdf ▲

(3-11.g) The supplied LEHRFEM++ function

```
template <typename FUNCTOR>
Eigen::VectorXd
Solve_LaplDir_LFE(std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
   fe_space,
                  FUNCTOR &&g);
```

solves (3.11.1) by means of piecewise linear Lagrangian finite elements on a *triangular mesh* passed through the **lf::uscalfe::FeSpaceLagrangeO1** object `fe_space`. `FUNCTOR g` gives the Dirichlet data g. The finite element solution is returned in the form of its basis expansion coefficient vector based on the numering of global shape functions encoded in `fe_space`.

Use `Solve_LaplDir_LFE()` together with `Jstar()` implemented in sub-problem (3-11.f) to realize a C++ function

```
template <typename FUNCTOR>
std::pair<double, double>
comparePointEval(
  std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
  FUNCTOR &&g, Eigen::Vector2d x);
```

that computes approximations of $u(\boldsymbol{x})$ for the solution u of (3.11.1) by first computing a finite element solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh passed through the **lf::uscalfe::FeSpaceLagrangeO1** object `fe_space`, and then

1. directly evaluating $u_h(\boldsymbol{x})$ (→ first return value)
2. or using the (3.11.12) with $u := u_h$ (→ second return value).

We take for granted that the meshed domain is the unit square $\Omega =]0,1[^2$. The point $\boldsymbol{x}$ must satisfy the assumptions of sub-problem (3-11.f) and its coordinates are passed in `x`.

SOLUTION for (3-11.g) → 3-11-7-0:spe7.pdf ▲

(3-11.h) [depends on Sub-problem (3-11.g)]

For $\Omega =]0,1[^2$, $\boldsymbol{x} = \binom{0.3}{0.4}$, and $u(\boldsymbol{y}) = \log(\left\|\boldsymbol{y} + \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]\right\|)$ examine numerically the convergence of the values returned by `comparePointEval` to $u(\boldsymbol{x})$. To that end perform the evaluation on a sequence of four meshes created by uniform regular refinement and plot the error on a doubly logarithmic plot ('loglog'). Describe the convergence in a qualitatively and quantitatively way.

Achieve this goal by completing the supplied C++ `main` function in `stableevaluationatapoint_main.cc`.

For the handle function for the Dirichlet boundary condition on you, remember that $g = u|_{\partial\Omega}$ on $\partial\Omega$.

SOLUTION for (3-11.h) → `3-11-8-0:spe8.pdf` ▲

End Problem 3-11, 70 min.

Problem 3-12: Computation of Electrostatic Forces

In [Lecture → Section 3.6.1] we arrived at the key insight that continuous (w.r.t. the energy norm) linear output functionals enjoy enhanced convergence rates in the context of Galerkin discretization. However, sometimes it takes smart tweaking of the functional in order to achieve this, as was demonstrated for the heat flux functional in [Lecture → Section 3.6.2]. We return to this topic in the current problem, extending the discussion to non-linear output functional.

Implementation relies on LEHRFEM++.
▷ problem name/problem code folder: ElectrostaticForce

A straight cylindrical wire is enclosed in a straight cylindrical conducting pipe as depicted in Fig. 61. There is a voltage drop between both. If the axial extension of both wire and pipe is very long, translational symmetry along the axis can be assumed, which allows two-dimensional modelling. As explained in [Lecture → Section 1.2.2], the (non-dimensional) electrostatic potential u in the homogeneous space Ω between the surface Γ_1 of the wire and the inner wall Γ_0 of the pipe, is given as the solution of the 2nd-order elliptic boundary value problem

$$-\Delta u = 0 \quad \text{in } \Omega\,, \tag{3.12.1}$$

with Dirichlet boundary conditions

$$u = 0 \quad \text{on } \Gamma_0, \quad u = 1 \quad \text{on } \Gamma_1\,. \tag{3.12.2}$$

Of course, a suitable scaling is assumed that yielded the non-dimensional equation (3.12.1). In this problem the use of a finite element method to compute the total attracting *force* between wire and pipe approximately is explored. Then, we will demonstrate another case for which a smart modification of an output functions can render it continuous and, thus, enable faster convergence, as we have seen for the heat flux functional in [Lecture → Section 3.6.2].

Caution: In contrast to the cases discussed in class, the output functionals studied in this problem are *non-linear*!

In the above model the electrostatic force on the wire can be computed from the potential u according to

$$\boldsymbol{F}(u) = \frac{1}{2}\int_{\Gamma_1} (\mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}))\, \mathbf{grad}\, u(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})\,. \tag{3.12.3}$$

To understand this formula recall $\mathbf{E} = -\mathbf{grad}\, u$, see [Lecture → (1.2.2.2)], and that $\mathbf{E}\cdot\boldsymbol{n}$ gives the surface charge density on a conductor. Thus the integrand in (3.12.3) can be read as the force density effected by the electric field "pulling at the surface charges".

(3-12.a) ⚄ (10 min.) Show that an equivalent expression for the force is

$$\boldsymbol{F}(u) = \frac{1}{2}\int_{\Gamma_1} \|\mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\|^2\, \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S\,. \tag{3.12.4}$$

HIDDEN HINT 1 for (3-12.a) → 3-12-1-0:s0h0.pdf

SOLUTION for (3-12.a) → 3-12-1-1:s0s.pdf ▲

In order to obtain an analytical solution for u the following special geometric situation will be considered. The pipe wall Γ_0 is a circle centered in $(4/15, 0)$ with radius $r_0 := 2/3$, and the wire boundary Γ_1 is a circle centered in the origin, with radius $r_1 := 4/15$.

The domain used in Problem 3-12. ▷

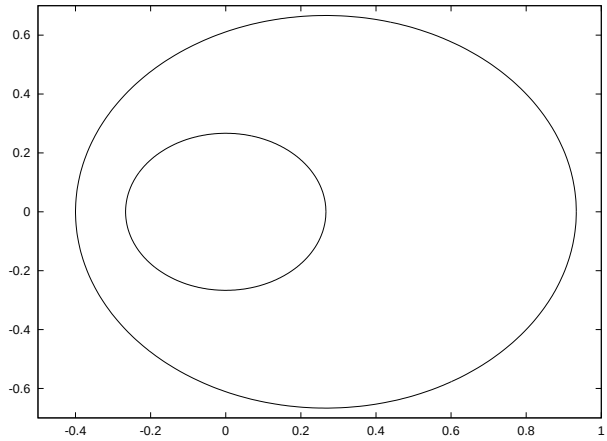


Fig. 61

(3-12.b) ⚁ (15 min.) Show that the solution to (3.12.1)-(3.12.2) can be implemented as a C++ lambda function as follows

C++ code 3.12.5: Sub-problem (3-12.b): Implementation of exact solution ↬ GitLab

```
2   Eigen::Vector2d a(-16.0 / 15.0, 0.0);
3   Eigen::Vector2d b(-1.0 / 15.0, 0.0);
4   // Lambda function providing exact solution
5   auto uExact = [&a, &b](Eigen::VectorXd x) -> double {
6     return (log((x - a).norm()) - log((x - b).norm())) / log(2) - 1;
7   };
```

SOLUTION for (3-12.b) → 3-12-2-0:emforce1.pdf ▲

(3-12.c) ⚂ (25 min.) Compute the force between wire and pipe for the exact solution found in Sub-problem (3-12.b).

HIDDEN HINT 1 for (3-12.c) → 3-12-3-0:emforce21.pdf

HIDDEN HINT 2 for (3-12.c) → 3-12-3-1:emforce22.pdf

SOLUTION for (3-12.c) → 3-12-3-2:emforce2.pdf ▲

(3-12.d) ⚀ (10 min.) Inside the function `solvePoissonBVP()` in the file `electrostaticforce.cc` implement the C++ lambda function `mf_bd_values` that creates a LEHRFEM++ **MeshFunction** object (→ [Lecture → Suppl. 2.8.3.32]) for the following function: it takes as input the coordinates of a point on the boundary and assigns to it the value 1 or 0 depending on whether the point belongs to the inner boundary or the outer boundary, respectively.

HIDDEN HINT 1 for (3-12.d) → 3-12-4-0:emforce31.pdf

SOLUTION for (3-12.d) → 3-12-4-1:emforce3.pdf ▲

(3-12.e) ⚂ (30 min.) [depends on Sub-problem (3-12.d)]

Complete that part of the function `solvePoissonBVP()` in the file `electrostaticforce.cc` that is meant to assemble the left- and right-hand sides that should solve (3.12.1)-(3.12.2) using a linear Lagrangian finite element Galerkin method.

HIDDEN HINT 1 for (3-12.e) → 3-12-5-0:emforce41.pdf

SOLUTION for (3-12.e) → 3-12-5-1:emforce4.pdf ▲

(3-12.f) ⚂ (15 min.) Implement a C++ function

```
double getMeshSize(const std::shared_ptr<const lf::mesh::Mesh>
  &mesh_p);
```

that computes the mesh width $h_{\mathcal{M}}$ of the LEHRFEM++ mesh object passed through `mesh_p` according to

Definition [Lecture → Def. 3.2.1.4]. Mesh width

Given a mesh $\mathcal{M} = \{K\}$, its mesh width $h_{\mathcal{M}}$ is defined as

$$h_{\mathcal{M}} := \max\{\operatorname{diam} K \colon K \in \mathcal{M}\} \quad , \quad \operatorname{diam} K := \max\{|\mathbf{p}-\mathbf{q}| \colon \mathbf{p},\mathbf{q} \in K\} \, .$$

SOLUTION for (3-12.f) → 3-12-6-0:s4a.pdf ▲

(3-12.g) ⚄ (30 min.) [depends on Sub-problem (3-12.e)]

Supplied are five meshes of increasingly fine resolution of Ω in GMSH files `emforce1.msh` through `emforce5.msh` in ➙ GitLab. The code that reads these meshes is already implemented in the `main()` function in the file `electrostaticforce_main.cc`. In addition, `solvePoissonBVP()` is called from `main()`.

Extend the `main()` function so that it computes and lists approximate L^2-norms of the errors, $\|u-u_h\|_{L^2(\Omega)}$, for the given sequence of meshes and tabulate them together with the meshwidth. Which type of convergence do you (and should you in light of [Lecture → Section 3.6.3]) observe?

HIDDEN HINT 1 for (3-12.g) → 3-12-7-0:l2c.pdf

HIDDEN HINT 2 for (3-12.g) → 3-12-7-1:emforce51.pdf

SOLUTION for (3-12.g) → 3-12-7-2:emforce5.pdf ▲

(3-12.h) ⚄ (5 min.) Show that the functional $u \mapsto \boldsymbol{F}(u)$ from (3.12.3) is *not* linear.

SOLUTION for (3-12.h) → 3-12-8-0:emforce6.pdf ▲

(3-12.i) In the file `electrostaticforce.cc` complete the implementation of the C++ function

```
Eigen::Vector2d computeForceBoundaryFunctional(
  const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
    &fe_space_p,
  Eigen::VectorXd approx_sol);
```

that evaluates the the force functional $\boldsymbol{F}(u)$ from (3.12.3) for a finite element solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ of (3.12.1) & (3.12.2). The argument `fe_space_p` provides all information about $\mathcal{S}_1^0(\mathcal{M})$ and `approx_sol` passes the basis expansion coefficient vector of u_h with respect to the customary tent-function basis of $\mathcal{S}_1^0(\mathcal{M})$.

Specifically, you are supposed to write the body of the for-loop that iterates over the cells of the mesh adjacent to the inner boundary Γ_1.

HIDDEN HINT 1 for (3-12.i) → 3-12-9-0:emforce71.pdf

SOLUTION for (3-12.i) → 3-12-9-1:emforce7.pdf ▲

(3-12.j) ⚅ (15 min.) Show that there are functions $u \in H^1(\Omega)$ for which $\boldsymbol{F}(u)$ from (3.12.3) is not defined (i.e. "$\boldsymbol{F}(u) = \infty$").

HIDDEN HINT 1 for (3-12.j) → 3-12-10-0:emforce81.pdf

SOLUTION for (3-12.j) → 3-12-10-1:emforce8.pdf ▲

(3-12.k) ⚄ (10 min.) [depends on Sub-problem (3-12.i)]

For the given meshes the `main()` function tabulates the Euclidean norm of the error of $\boldsymbol{F}(u_h)$,

$u_h \in \mathcal{S}_1^0(\mathcal{M})$ the finite-element solution, with the exact value of the force from Sub-problem (3-12.c). What (kind and rate) of convergence $\boldsymbol{F}(u_h) \to \boldsymbol{F}(u)$ in terms of the meshwidth do you observe?

SOLUTION for (3-12.k) → 3-12-11-0:emforce9.pdf ▲

(3-12.l) ⚄ (10 min.) According to Sub-problem (3-12.k) it might be possible to encounter infinitely large electrostatic forces attracting the wire towards the pipe. Why will this never be observed?

HIDDEN HINT 1 for (3-12.l) → 3-12-12-0:h10h1.pdf

SOLUTION for (3-12.l) → 3-12-12-1:emforce10.pdf ▲

[Lecture → Section 3.6.2] strikingly demonstrated the benefit of replacing an unbounded linear output functional with an equivalent bounded one. By sub-problem Sub-problem (3-12.j) the output functional $\boldsymbol{F}$ from (3.12.3) may be haunted by the same problems as [Lecture → (3.6.2.2)], though it is non-linear. Thus, it may pay off to reformulate (3.12.3) in a form $\tilde{\boldsymbol{F}}$ that is continuous on $H^1(\Omega)$, and which agrees with $\boldsymbol{F}$ for all solutions of (3.12.1).

(3-12.m) ⚃ (10 min.) [depends on Sub-problem (3-12.a)]

Show that, for all $\boldsymbol{x} \in \Gamma_1$ we have

$$\frac{1}{2}(\mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}))\, \mathbf{grad}\, u(\boldsymbol{x}) = \boldsymbol{T}(u)(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \,,$$

where $\boldsymbol{T}$ is the so-called Maxwell stress tensor

$$\boldsymbol{T}(u)(\boldsymbol{x}) = \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x})^T - \frac{1}{2} \| \mathbf{grad}\, u(\boldsymbol{x}) \|^2 \cdot \boldsymbol{I} \,, \tag{3.12.15}$$

where $\boldsymbol{I}$ is the 2×2 identity matrix.

HIDDEN HINT 1 for (3-12.m) → 3-12-13-0:emforce111.pdf

SOLUTION for (3-12.m) → 3-12-13-1:emforce11.pdf ▲

(3-12.n) ⚃ (10 min.) Show that if u solves (3.12.1), then

$$\operatorname{div} \boldsymbol{T}(u)(\boldsymbol{x}) = \mathbf{0} \,, \quad \boldsymbol{x} \in \Omega \,. \tag{3.12.16}$$

Here, div is row-wise divergence of a matrix-valued function, i.e. it takes the divergence of each *row* $\boldsymbol{T}_i$ of the matrix $\boldsymbol{T}$ (regarded as a vector) and returns a vector:

$$\operatorname{div} \begin{pmatrix} t_{11}(\boldsymbol{x}) & t_{12}(\boldsymbol{x}) \\ t_{21}(\boldsymbol{x}) & t_{22}(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} \frac{\partial t_{11}}{\partial x_1}(\boldsymbol{x}) + \frac{\partial t_{12}}{\partial x_2}(\boldsymbol{x}) \\ \frac{\partial t_{21}}{\partial x_1}(\boldsymbol{x}) + \frac{\partial t_{22}}{\partial x_2}(\boldsymbol{x}) \end{pmatrix} . \tag{3.12.17}$$

SOLUTION for (3-12.n) → 3-12-14-0:emforce12.pdf ▲

(3-12.o) ⚃ (15 min.) [depends on Sub-problem (3-12.m) and Sub-problem (3-12.n)]

Show that if u solves (3.12.1), then

$$\boldsymbol{F}(u) = \tilde{\boldsymbol{F}}(u) := \int_\Omega \boldsymbol{T}(u)(\boldsymbol{x})\, \mathbf{grad}\, \Psi(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,, \tag{3.12.18}$$

where the cut-off function $\Psi \in C^\infty(\overline{\Omega})$ satisfies $\Psi(\boldsymbol{x}) = 1$ on Γ_1 and $\Psi(\boldsymbol{x}) = 0$ on Γ_0.

HIDDEN HINT 1 for (3-12.o) → 3-12-15-0:emforce131.pdf

SOLUTION for (3-12.o) → 3-12-15-1:emforce13.pdf ▲

(3-12.p) ⚁ (10 min.) Show that $\tilde{\boldsymbol{F}}(v)$, defined in (3.12.18), is is bounded for all $v \in H^1(\Omega)$, that is

$$\exists C > 0: \quad |\tilde{\boldsymbol{F}}(v)| \leq C\|v\|_{H^1(\Omega)} \quad \forall v \in H^1(\Omega) \,.$$

SOLUTION for (3-12.p) → 3-12-16-0:emforce14.pdf ▲

(3-12.q) ⚃ (45 min.) In the file `electrostaticforce.cc` finish the implementation of the C++ function

```
Eigen::Vector2d computeForceDomainFunctional(
  const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
    &fe_space_p,
  Eigen::VectorXd approx_sol)
```

that evaluates the force functional $\tilde{\boldsymbol{F}}(u_h)$ from (3.12.18) for a finite-element function $u_h \in \mathcal{S}_1^0(\mathcal{M})$. The arguments are the same as for `computeForceBoundaryFunctional` from Sub-problem (3-12.i).

HIDDEN HINT 1 for (3-12.q) → 3-12-17-0:emforce151.pdf

SOLUTION for (3-12.q) → 3-12-17-1:emforce15.pdf ▲

(3-12.r) ⚁ (10 min.) [depends on Sub-problem (3-12.q)]

As in Sub-problem (3-12.k) for the given meshes the `main()` function tabulates the Euclidean norm of the error of $\tilde{\boldsymbol{F}}(u_h)$, $u_h \in \mathcal{S}_1^0(\mathcal{M})$ the finite-element solution, with the exact value of the force from Sub-problem (3-12.c). What (kind and rate) of convergence $\boldsymbol{F}(u_h) \to \boldsymbol{F}(u)$ in terms of the meshwidth do you observe now?

Compare your observations with those made in sub-problem Sub-problem (3-12.k).

SOLUTION for (3-12.r) → 3-12-18-0:emforce16.pdf ▲

End Problem 3-12, 275 min.

Problem 3-13: Computation of Stationary Currents

This problem arises in computational electromagnetics and deals with the computation of the currents in a conductor when a voltage is applied at contacts. This boils down to the solution of a second-order scalar elliptic boundary value problem with mixed Dirichlet- and Neumann boundary conditions. A relevant output quantity is the total current and its computation will also be treated in this problem.

Heavily draws on the contents of [Lecture → Section 1.8], [Lecture → Section 2.7.6], and [Lecture → Section 3.6.2]. Involves substantial implementation in LEHRFEM++.
▷ problem name/problem code folder: StationaryCurrents

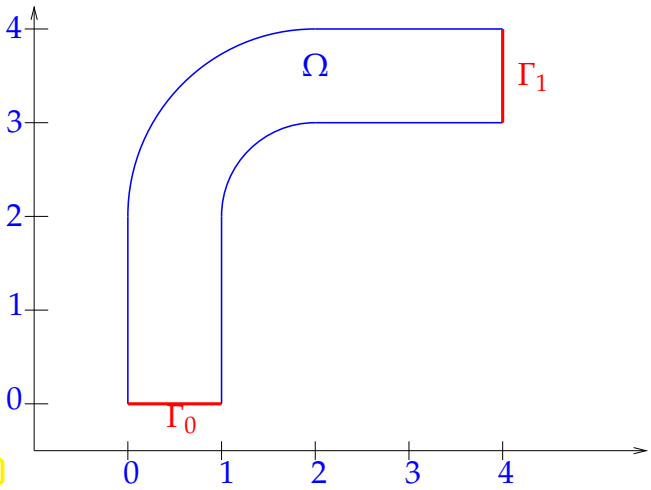


Fig. 65

◁ The figure shows a cross-section $\Omega \subset \mathbb{R}^2$ of a wide piece of metal. The boundary $\partial\Omega$ consists of straight and circular parts.

The two red boundary parts Γ_0 and Γ_1 designate contacts. The remainder $\Gamma_{-1} := \partial\Omega \setminus (\Gamma_0 \cup \Gamma_1)$ is the insulated boundary, drawn in blue.

A uniformly positive definite [Lecture → Def. 1.2.2.9] matrix-valued function $\sigma \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^{2,2}$ describes the spatially varying **electric conductivity** of the material.

A voltage drop of 1 is imposed between the contacts.

The solution of the following linear variational problem yields the **electric potential** prevailing in the conducting material.

$$\begin{array}{c} u \in H^1(\Omega) \\ u|_{\Gamma_0} = 0,\ u|_{\Gamma_1} = 1 \end{array} : \quad \underbrace{\int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = 0 \quad \forall v \in V_0\,, \tag{3.13.1}$$

where the test space is

$$V_0 := \{v \in H^1(\Omega) :\ v|_{\Gamma_0} = v|_{\Gamma_1} = 0\}\,. \tag{3.13.2}$$

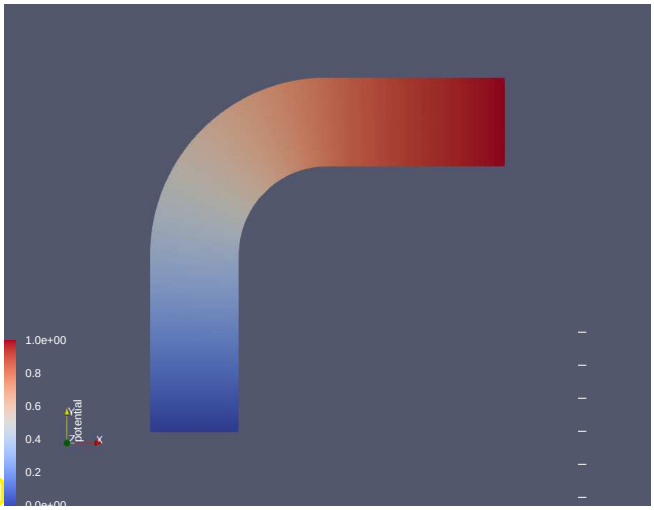


Fig. 66

◁ Finite-element solution for potential u computed with the methods investigated in this problem.

Here we used a scalar-valued conductivity coefficient

$$\sigma(\boldsymbol{x}) = \max\{1, x_1 - 1\}\,,\ \boldsymbol{x} \in \Omega\,.$$

(3-13.a) ⚀ (10 min.) What is the strong (PDE) form of the boundary value problem, whose weak form is given by (3.13.1)?

SOLUTION for (3-13.a) → 3-13-1-0:sx1.pdf ▲

(3-13.b) ⚂ (15 min.) Specify in full details a continuous linear form $\ell : H^1(\Omega) \to \mathbb{R}$ and a function $u_0 \in H^1(\Omega)$ such that the solution u of (3.13.1) can be recovered as $u = u_0 + w$, where $w \in V_0$ solves

$$\mathsf{a}(w, v) = \ell(v) \quad \forall v \in V_0 \, .$$

The bilinear form $\mathsf{a} : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ is defined in (3.13.1).

SOLUTION for (3-13.b) → 3-13-2-0:s3x.pdf ▲

We perform a finite-element Galerkin discretization of (3.13.1) relying on lowest-degree Lagrangian finite element spaces $\mathcal{S}_1^0(\mathcal{M})$, where $\mathcal{M}$ is a standard triangular mesh like the ones drawn below. In the LEHRFEM++ implementation the mesh is represented by an object of type **lf::mesh::hybrid2d::Mesh**.

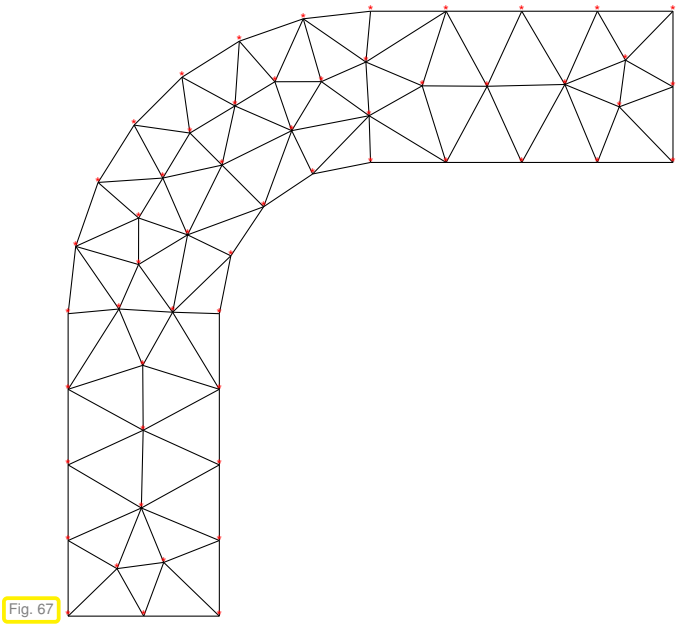


Fig. 67

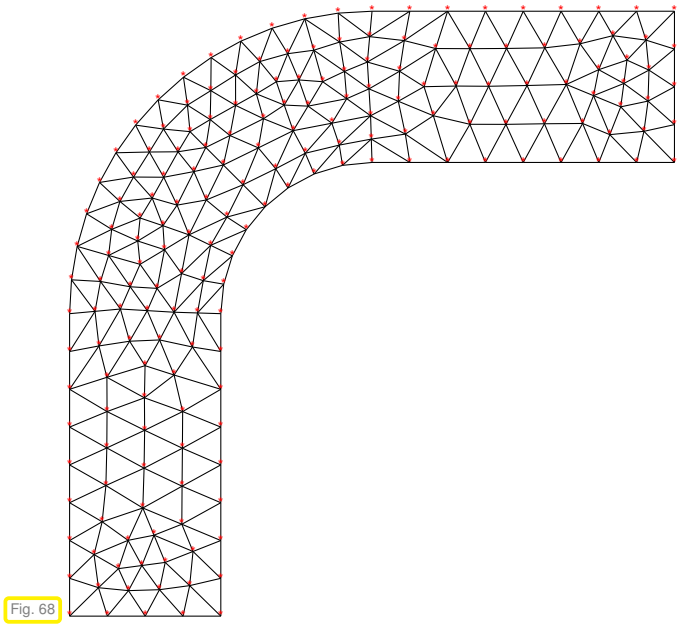


Fig. 68

(3-13.c) ⚂ (10 min.) What kind of variational crime is inevitably committed in the finite-element Galerkin discretization based on $\mathcal{S}_1^0(\mathcal{M})$?

SOLUTION for (3-13.c) → 3-13-3-0:cs3.pdf ▲

(3-13.d) ⚂ (10 min.)

The mesh $\mathcal{M}$ displayed in Fig. 67 above is a triangulation of Ω and comprises

- 55 nodes,
- 131 edges,
- and 77 cells.

Beside it is shown again with entity index numbers.

What is the dimension of the finite element space

$$\mathcal{S}_{1,C}^0(\mathcal{M}) := \mathcal{S}_1^0(\mathcal{M}) \cap V_0 \, ? \qquad (3.13.7)$$

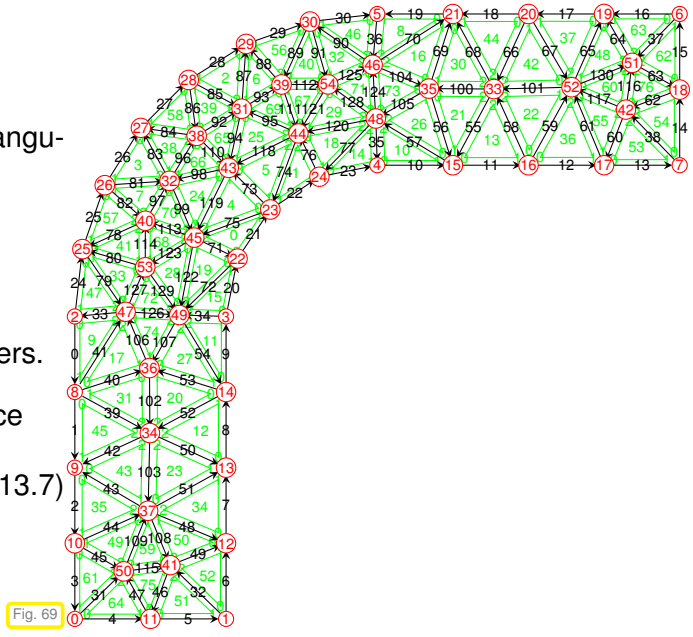


Fig. 69

SOLUTION for (3-13.d) → 3-13-4-0:sl4.pdf ▲

(3-13.e) ⚄ (20 min.) In the file `stationarycurrents.h` complete the code of the C++ function

```
template <typename SIGMAFUNCTOR>
  Eigen::VectorXd solveMixedBVP(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space,
    lf::mesh::utils::CodimMeshDataSet<int> &nodeflags,
    std::vector<double> &&voltvals, SIGMAFUNCTOR &&sigma);
```

This function is meant to compute finite-element Galerkin solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ of the mixed second-order boundary value problem (3.13.1).

- The argument `fe_space` passes an object holding information about the mesh and the numbering of global shape functions.
- The argument `nodeflags` is a *node-indexed* array based on the finite element mesh stored in `fe_space`:

$$\texttt{nodeflags}(p) = \begin{cases} 1 & \text{, if node } p \in \overline{\Gamma}_1, \\ 0 & \text{, if node } p \in \overline{\Gamma}_0, \\ -1 & \text{else,} \end{cases} \qquad p \in \mathcal{N}(\mathcal{M}) \,.$$

- The vector `voltvals` gives the Dirichlet boundary values that the solution u_h is to attain on Γ_0 and Γ_1, respectively, in its first and second entry.
- `sigma` passes a functor realizing the coefficient function $\sigma(\boldsymbol{x})$.

The function should return the basis expansion coefficients of the finite-element solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ with respect to the tent-function basis of $\mathcal{S}_1^0(\mathcal{M})$.

HIDDEN HINT 1 for (3-13.e) → 3-13-5-0:h14a.pdf

SOLUTION for (3-13.e) → 3-13-5-1:s4a.pdf ▲

A key output quantity is the total current

$$I = I(u) := \int_{\Gamma_1} \sigma(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \,, \tag{3.13.9}$$

where u is the solution of (3.13.1).

(3-13.f) ⚄ (15 min.) [depends on Sub-problem (3-13.a)]

Explain, why there holds the alternative formula

$$I(u) = -\int_{\Gamma_0} \sigma(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \,,$$

if u solves (3.13.1).

SOLUTION for (3-13.f) → 3-13-6-0:lx5.pdf ▲

(3-13.g) ⚄ (45 min.) In the file `stationarycurrents.h` supplement the missing parts of the function

```
template <typename SIGMAFUNCTION>
  double contactFlux(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space,
```

```
const Eigen::VectorXd& sol_vec, SIGMAFUNCTION&& sigma,
const lf::mesh::utils::CodimMeshDataSet<int>& edgeids,
int contact_id = 0);
```

that directly evaluates $I(u_h)$ for a finite-element function $u_h \in \mathcal{S}_1^0(\mathcal{M})$, whose basis expansion coefficient vector is available as argument `sol_vec`. The local integrals over boundary edges should be approximated by a midpoint quadrature rule, the lowest-order Gauss quadrature formula.

The other arguments have the following meanings

- `fe_space` contains information about the mesh and the numbering of the global shape functions.
- `sigma` is a functor object providing the coefficient function σ.
- `edgeids` is an *edge-indexed* array based on the mesh stored in `fe_space` with the following information

$$\texttt{edgeids}(e) = \begin{cases} 1 & \text{, if edge } e \subset \Gamma_1, \\ 0 & \text{, if edge } e \subset \Gamma_0, \\ -1 & \text{else,} \end{cases} \qquad e \in \mathcal{E}(\mathcal{M}) \,.$$

Your implementation can rely on the function

```
std::tuple<Eigen::Matrix<double, 2, 3>,
           Eigen::Matrix<double, 2, 3>,
           double>
getTriangleGradLambdaNormals(Eigen::Matrix<double, 2, 3> vertices);
```

which returns **two** 2×3-matrices, whose columns contain

1. the gradients of the barycentric coordinate functions,
2. the three exterior normal vectors, whose lengths agree with that of the associated edge ("edge-length-weighted exterior normals")

and a single number providing the area, for a triangle K, whose vertex coordinates are passed as the columns of the matrix argument `vertices`. This function can be invoked as follows

```
auto [grad_bary_coords, normals, area] =
  getTriangleGradLambdaNormals(
    lf::geometry::Corners(*(cell->Geometry())));
```

where `cell` is a pointer to a mesh entity of type `TRIA`.

SOLUTION for (3-13.g) → 3-13-7-0:s6.pdf ▲

(3-13.h) ⚀ (15 min.) Which properties does the function $\Psi \in H^2(\Omega)$ have to satisfy such that

$$I^*(u) := \int_\Omega \sigma(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, \Psi(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = I(u) \quad \text{, if } u \text{ solves (3.13.1) ?} \tag{3.13.11}$$

SOLUTION for (3-13.h) → 3-13-8-0:s7s.pdf ▲

(3-13.i) ⚁ (10 min.) Prove that the functional $u \to I^*(u)$ as defined in (3.13.11) is continuous on the Sobolev space $H^1(\Omega)$.

SOLUTION for (3-13.i) → 3-13-9-0:s81.pdf ▲

(3-13.j) ⚄ (30 min.) In the file `stationarycurrents.h` finish the implementation of the C++ function

```
template <typename SIGMAFUNCTION, typename PSIGRAD>
  double stabFlux(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space,
    const Eigen::VectorXd &sol_vec, SIGMAFUNCTION &&sigma,
    PSIGRAD &&gradpsi);
```

that approximately evaluates the functional $I^*(u_h)$ for a finite-element function $u_h \in \mathcal{S}_1^0(\mathcal{M})$. For the evaluation of the integral use the local midpoint quadrature formula for triangles

$$\int_K \varphi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx |K|\varphi(\boldsymbol{c})\,, \quad \boldsymbol{c} := \tfrac{1}{3}(\boldsymbol{a}_K^1 + \boldsymbol{a}_K^2 + \boldsymbol{a}_K^3)\,,$$

where $\boldsymbol{a}_K^\ell$, $\ell = 1,2,3$, are the vertices of the triangle K.

The arguments of the function have the following meaning:

- `fe_space` passes information about the mesh and the indexing of global shape functions.
- `sol_vec` contains the tent-function basis expansion coefficients of u_h.
- `sigma` provides a functor object describing the coefficient function σ.
- `gradpsi` is a functor object that realizes the function $\mathbf{grad}\,\Psi : \Omega \to \mathbb{R}^2$ and complies with `std::function<Eigen::Vector2d(Eigen::Vector2d)>`.

HIDDEN HINT 1 for (3-13.j) → 3-13-10-0:gtgn.pdf

SOLUTION for (3-13.j) → 3-13-10-1:sol9.pdf ▲

(3-13.k) ⚄ (15 min.) The following table provides the values

- $I(u_\ell)$,
- $I^*(u_\ell)$,
- $I(u_\ell) - I(u)$,
- $I^*(u_\ell) - I(u)$,

in its rows, but not necessarily in this order. Here u is the solution of (3.13.1) and $u_\ell \in \mathcal{S}_1^0(\mathcal{M}_\ell)$ is the finite-element Galerkin solution on a mesh from a sequence of meshes $\mathcal{M}_0, \ldots, \mathcal{M}_5$ created by uniform regular refinement in GMSH. The leftmost column provides the meshwidths h_ℓ of $\mathcal{M}_\ell$. The reference value $I(u) = 0.1859836202175363$ was used and the computations relied on the following weighting function

```
static auto psi = [](Eigen::Vector2d x) -> double {
  if (x[0] > 3)
    return 1.0;
  else if (x[0] < 2)
    return 0.0;
  const double p = std::cos((3.0 - x[0]) * M_PI / 2);
  return (p * p);
};
```

h_ℓ	(I)	(II)	(III)	(IV)
0.628289	0.199849	0.013865	0.190757	0.004774
0.346305	0.192438	0.006454	0.187415	0.001432
0.187303	0.189040	0.003056	0.186367	0.000384
0.094212	0.187457	0.001473	0.186055	0.000071
0.050521	0.186705	0.000722	0.186001	0.000018
0.024491	0.186341	0.000357	0.185986	0.000002

Column Convergence

$I(u_\ell)$:

$I^*(u_\ell)$:

$I(u_\ell) - I(u)$:

$I^*(u_\ell) - I(u)$:

Which column lists which quantity? Described qualitatively and quantitatively the empiric aysmptotic convergence of $I(u_\ell) \to I(u)$ and $I^*(u_\ell) \to I(u)$ as $h \to 0$.

SOLUTION for (3-13.k) → 3-13-11-0:sola.pdf ▲

(3-13.l) ⚀ (20 min.) Which boundary value problem in strong (PDE) form is solved by the dual solution g_F belonging to the linear output functional $F := I^*$ with I^* from (3.13.11), Ψ as defined by the C++ lambda function `psi()` in Sub-problem (3-13.k), and $\sigma(\boldsymbol{x})$ continuously differentiable and a scalar multiple of the identity matrix for all $\boldsymbol{x} \in \Omega$, $\Omega \subset \mathbb{R}^2$ sketched in Fig. 65?

Remember the definition of the dual solution from [Lecture → Thm. 3.6.1.7]:

Theorem [Lecture → Thm. 3.6.1.7]. Duality estimate for linear functional output

Define the ***dual solution*** $g_F \in V_0$ *to* F *as solution of the* ***dual variational problem***

$$g_F \in V_0: \quad \mathsf{a}(v, g_F) = F(v) \quad \forall v \in V_0 \,.$$

Then

$$|F(u) - F(u_h)| \leq \|u - u_h\|_{\mathsf{a}} \inf_{v_h \in V_{0,h}} \|g_F - v_h\|_{\mathsf{a}} \,. \tag{3.13.15}$$

SOLUTION for (3-13.l) → 3-13-12-0:solb.pdf ▲

End Problem 3-13, 215 min.

Problem 3-14: A Local Quasi-Interpolation Operator

This problems introduces an important concept in the numerical analysis of finite element methods: **quasi-interpolation operators**, which are linear projections onto finite element spaces, which enjoy some locality and continuity properties superior to those of the usual nodal interpolation operators.

This problem is based on [Lecture → Section 2.6] and involves substantial coding in LEHRFEM++.
▷ problem name/problem code folder: QuasiInterpolation

Throughout $\mathcal{M}$ denotes a *triangular* finite-element mesh of a polygonally bounded domain $\Omega \subset \mathbb{R}^2$, which comprises only triangles with straight edges. For every $K \in \mathcal{M}$ there exist three unique affine linear functions (polynomials of degree 1) $\psi_K^\ell \in \mathcal{P}_1(\mathbb{R}^2)$, $\ell = 1,2,3$, such that

$$\int_K \psi_K^\ell(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \delta_{\ell,j} := \begin{cases} 1 & \text{, if } \ell = j\,, \\ 0 & \text{, if } \ell \neq j\,, \end{cases} \qquad \ell,j \in \{1,2,3\}\,, \tag{3.14.1}$$

where λ_j, $j = 1,2,3$, are the barycentric coordinate functions on K.

(3-14.a) ⚂ (15 min.) Compute $\widehat{\psi}^\ell := \psi_{\widehat{K}}^\ell$, $\ell = 1,2,3$, for the reference triangle $\widehat{K}$ with vertices $\widehat{\boldsymbol{a}}^1 := \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$, $\widehat{\boldsymbol{a}}^2 := \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, and $\widehat{\boldsymbol{a}}^3 := \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$.

HIDDEN HINT 1 for (3-14.a) → 3-14-1-0:policy.pdf

SOLUTION for (3-14.a) → 3-14-1-1:sol1.pdf ▲

(3-14.b) ⚂ (10 min.) [depends on Sub-problem (3-14.a)]

For a general triangle $K \in \mathcal{M}$ (with numbered vertices) denote by $\boldsymbol{\Phi}_K : \widehat{K} \to K$ the unique *affine* mapping from the reference triangle $\widehat{K}$ (with vertices $\widehat{\boldsymbol{a}}^1 := \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$, $\widehat{\boldsymbol{a}}^2 := \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, and $\widehat{\boldsymbol{a}}^3 := \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$) to K, which preserves the order of the vertices. Show that ($|K| \triangleq$ area of K)

$$\psi_K^\ell = \frac{|\widehat{K}|}{|K|}\left(\boldsymbol{\Phi}_K^{-1}\right)^* \widehat{\psi}^\ell \quad \Leftrightarrow \quad \boldsymbol{\Phi}_K^* \psi_K^\ell = \frac{|\widehat{K}|}{|K|}\widehat{\psi}^\ell\,. \tag{3.14.5}$$

Here, $*$ tags the pullback of functions induced by a mapping, as introduced in [Lecture → Def. 2.8.1.2].

SOLUTION for (3-14.b) → 3-14-2-0:s2.pdf ▲

Now we are in a position to define a specimen of a **quasi-interpolation operator**. To that end, for every node $\boldsymbol{p} \in \mathcal{N}(\mathcal{M})$ of the mesh $\mathcal{M}$ denote by $K_{\boldsymbol{p}}$ the largest triangle having $\boldsymbol{p}$ as a vertex

$$K_{\boldsymbol{p}} \in \mathcal{M}_{\boldsymbol{p}} := \{K \in \mathcal{M} : \ \boldsymbol{p} \in \overline{K}\} \quad , \quad h_{K_{\boldsymbol{p}}} = \max\{h_K : \ K \in \mathcal{M}_{\boldsymbol{p}}\}\,, \tag{3.14.6}$$

where $h_K > 0$ is the size of $K \in \mathcal{M}$, that is, the length of its longest edge.

Write $\mathfrak{B}_h := \{b_h^{\boldsymbol{p}},\ \boldsymbol{p} \in \mathcal{N}(\mathcal{M})\}$ for the standard "tent-function" basis of $\mathcal{S}_1^0(\mathcal{M})$, see [Lecture → Section 2.4.3]. Here we index each basis function with that node of the mesh to which it is associated. Next, based on the mapping $\mathcal{N}(\mathcal{M}) \to \mathcal{M}$, $\boldsymbol{p} \mapsto K_{\boldsymbol{p}}$, we define

$$\mathrm{Q}_h : \begin{cases} L^2(\Omega) & \to & \mathcal{S}_1^0(\mathcal{M}) \\ v & \mapsto & \mathrm{Q}_h(v) := \displaystyle\sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})} \left(\int_{K_{\boldsymbol{p}}} v(\boldsymbol{x})\,\psi_{K_{\boldsymbol{p}}}^{\boldsymbol{p}}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right) b_h^{\boldsymbol{p}}\,. \end{cases} \tag{3.14.7}$$

By the notation $\psi_{K_{\boldsymbol{p}}}^{\boldsymbol{p}}$ we refer to that function $\in \{\psi_{K_{\boldsymbol{p}}}^1, \psi_{K_{\boldsymbol{p}}}^2, \psi_{K_{\boldsymbol{p}}}^3\}$ whose (superscript) index agrees with the local vertex number of $\boldsymbol{p}$ in $K_{\boldsymbol{p}}$.

(3-14.c) ⚄ (15 min.) Show that Q_h is a **projection** onto $\mathcal{S}_1^0(\mathcal{M})$, that is,

$$Q_h(v_h) = v_h \quad \forall v_h \in \mathcal{S}_1^0(\mathcal{M}) \,. \tag{3.14.8}$$

HIDDEN HINT 1 for (3-14.c) → 3-14-3-0:2bh.pdf

SOLUTION for (3-14.c) → 3-14-3-1:s2b.pdf ▲

(3-14.d) ⚂ (10 min.) Of course, in an implementation for a generic continuous argument function v the integral in the definition (3.14.7) of $Q_h(v)$ can be computed only approximately using local quadrature rules.

Assume that we use a P-point quadrature rule obtained from a "master rule" on the reference triangle $\widehat{K}$ with weights $\widehat{\omega}_\ell \in \mathbb{R}$ and nodes $\widehat{\zeta}_\ell \in \widehat{K}$, $\ell = 1, \ldots, P$. Write the down the definition of the resulting approximate quasi-interpolation operator $\widetilde{Q}_h : C^0(\overline{\Omega}) \to \mathcal{S}_1^0(\mathcal{M})$, taking into account (3.14.5).

HIDDEN HINT 1 for (3-14.d) → 3-14-4-0:h3ax.pdf

HIDDEN HINT 2 for (3-14.d) → 3-14-4-1:h3a.pdf

SOLUTION for (3-14.d) → 3-14-4-2:s3a.pdf ▲

(3-14.e) ⚂ (10 min.) [depends on Sub-problem (3-14.d)]

We use the notations introduced in Sub-problem (3-14.d). Which arguments have to be given to the LEHRFEM++ function

```
QuadRule lf::quad::make_QuadRule(
  lf::base::RefEl ref_el, unsigned degree);
```

such that it returns an *efficient* quadrature rule that, when used to define $\widetilde{Q}_h$ as in Sub-problem (3-14.d), ensures

$$\widetilde{Q}_h(v_h) = Q_h(v_h) \quad \forall v_h \in \mathcal{S}_3^0(\mathcal{M}) \,,$$

where $\mathcal{S}_3^0(\mathcal{M})$ is the space of piecewise cubic Lagrangian finite-element functions on $\mathcal{M}$.

SOLUTION for (3-14.e) → 3-14-5-0:sl3bb.pdf ▲

(3-14.f) ⚂ (30 min.) In the file `quasiinterpolation.cc` complete the implementation of the auxiliary LEHRFEM++ function

```
lf::mesh::utils::CodimMeshDataSet<
  std::pair<const lf::mesh::Entity *, unsigned int>>
    findKp(std::shared_ptr<const lf::mesh::Mesh mesh_p);
```

that takes a pointer to a **lf::mesh::Mesh** object (describing $\mathcal{M}$) and returns a node-indexed array of 2-tuples of

❶ pointers to mesh cells such that (& ≙ C++ address operator)

$$\texttt{findKp(mesh_p)}(p)\texttt{.first} = \&K_p \quad \forall p \in \mathcal{N}(\mathcal{M} = \texttt{*mesh_p}) \,,$$

❷ and integers $\in \{0,1,2\}$ that give the local vertex number of p in K_p.

HIDDEN HINT 1 for (3-14.f) → 3-14-6-0:h3c1.pdf

SOLUTION for (3-14.f) → 3-14-6-1:sl3.pdf ▲

(3-14.g) ⚄ (30 min.) [depends on Sub-problem (3-14.a) and Sub-problem (3-14.d)]

In the file `quasiinterpolation.h` finish the code for the LEHRFEM++-based function

```
template <typename MESHFUNCTION>
  Eigen::VectorXd quasiInterpolate(
    lf::uscalfe::FeSpaceLagrangeO1<double> &fe_space,
    MESHFUNCTION &&v_mf);
```

The argument `fe_space` passes information about the triangular mesh $\mathcal{M}$ and the local-to-global index map for the shape functions of $\mathcal{S}_1^0(\mathcal{M})$. The object in `v_mf` must comply with the concept of a real-valued LEHRFEM++ **MeshFunction**, that is, it must provide an evaluation operator with signature

```
std::vector<double> operator()(
  const lf::mesh::Entity& e, const Eigen::MatrixXd& local) const;
```

where `e` must be a cell K of the mesh, whereas the columns of the matrix argument pass the *reference or local coordinates* of points in K: if a point has coordinates $\boldsymbol{x} \in \mathbb{R}^2$, its reference/local coordinates are the components of $\boldsymbol{\Phi}_K^{-1}(\boldsymbol{x})$.

The function is to return the basis expansion coefficients of $Q_h(v) \in \mathcal{S}_1^0(\mathcal{M})$, with $v \in C^0(\overline{\Omega})$ encoded in `v_mf`:

$$\texttt{v_mf}(K, \widehat{\boldsymbol{x}}) = v(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})) \quad \forall K \in \mathcal{M}\,, \quad \widehat{\boldsymbol{x}} \in \widehat{K}\,. \tag{3.14.13}$$

Of course, as discussed in Sub-problem (3-14.d) the integral in the the definition (3.14.7) of Q_h can only be evaluated approximately using local quadrature rules. Employ a quadrature rule that is exact for polynomials of degree 2.

HIDDEN HINT 1 for (3-14.g) → `3-14-7-0:kph.pdf`

SOLUTION for (3-14.g) → `3-14-7-1:sol4.pdf` ▲

(3-14.h) ⚄ (10 min.) We can restrict the quasi-interpolation operator Q_h to the quadratic Lagrangian finite-element space $\mathcal{S}_2^0(\mathcal{M})$. This yields a linear mapping $Q_h : \mathcal{S}_2^0(\mathcal{M}) \to \mathcal{S}_1^0(\mathcal{M})$, which can be represented by a matrix $\mathbf{R} \in \mathbb{R}^{\sharp\mathcal{N}(\mathcal{M}),\sharp\mathcal{N}(\mathcal{M})+\sharp\mathcal{E}(\mathcal{M})}$ ($\mathcal{E}(\mathcal{M}) \hat{=}$ set of edges of $\mathcal{M}$) with respect to the standard nodal bases of $\mathcal{S}_2^0(\mathcal{M})$ and $\mathcal{S}_1^0(\mathcal{M})$.

Give a sharp bound for the number of non-zero entries in a row of $\mathbf{R}$.

HIDDEN HINT 1 for (3-14.h) → `3-14-8-0:h51.pdf`

SOLUTION for (3-14.h) → `3-14-8-1:sn5.pdf` ▲

(3-14.i) ⚄ (15 min.) Show that Q_h as defined in (3.14.7) is continuous in $L^2(\Omega)$, more precisely, that there exists a constant $C > 0$ depending only on the shape regularity measure $\rho_{\mathcal{M}}$ of $\mathcal{M}$ such that

$$\|Q_h(v)\|_{L^2(\Omega)} \leq C\|v\|_{L^2(\Omega)} \quad \forall v \in L^2(\Omega)\,. \tag{3.14.15}$$

HIDDEN HINT 1 for (3-14.i) → `3-14-9-0:h61.pdf`

HIDDEN HINT 2 for (3-14.i) → `3-14-9-1:h62.pdf`

SOLUTION for (3-14.i) → `3-14-9-2:sol6.pdf` ▲

(3-14.j) ⚄ (10 min.) Now $\Omega \subset \mathbb{R}^2$ is an "L-shaped" domain:

$$\Omega :=]-1,1[^2 \setminus [0,1]^2\,.$$

On Ω we consider a sequence $\mathcal{M}_0 \prec \mathcal{M}_1 \prec \cdots \prec \mathcal{M}_L$, $L = 6$, of triangular meshes generated by *uniform regular refinement*. We write h_k for the mesh width of $\mathcal{M}_k$.

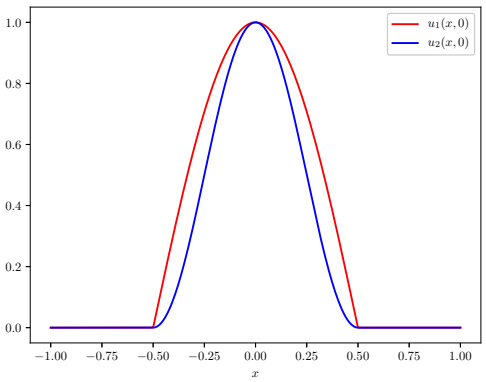


Fig. 71

We investigate the quasi-interpolation error for the two functions

$$u_1(\boldsymbol{x}) := \begin{cases} \cos(\pi\|\boldsymbol{x}\|) & \text{, if } \|\boldsymbol{x}\| < \frac{1}{2}\,, \\ 0 & \text{else,} \end{cases}$$

$$u_2(\boldsymbol{x}) := u_1^2(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Omega\,.$$

◁ cross-sections of graphs of u_1, u_2

We write Q_k for the quasi-interpolation onto $\mathcal{S}_1^0(\mathcal{M}_k)$ as introduced above. We obtain the following table of quasi-interpolation errors

	Function A (f_A)		Function B (f_B)	
h_k	$EL2_A(k)$	$EH1_A(k)$	$EL2_B(k)$	$EH1_B(k)$
1	0.244547	1.67563	0.2352667	1.36644
0.5	0.143055	1.65355	0.0772272	1.06657
0.25	0.045093	1.15569	0.0344346	0.78000
0.125	0.017011	0.81551	0.0076522	0.42977
0.0625	0.005156	0.53399	0.0016812	0.21514
0.03125	0.001986	0.38990	0.0003923	0.10608
0.015625	0.000631	0.27153	0.0000941	0.05309

($EL2_A(k) := \|f_A - Q_k(f_A)\|_{L^2(\Omega)}$, $EL2_B(k) := \|f_B - Q_k(f_B)\|_{L^2(\Omega)}$,
$EH1_A(k) := \|f_A - Q_k(f_A)\|_{H^1(\Omega)}$, $EH1_B(k) := \|f_B - Q_k(f_B)\|_{H^1(\Omega)}$)

Describe quantitatively and qualitatively the empiric convergence of the different norms of the quasi-interpolation errors in terms of the mesh width $h_k \to 0$.

We have $f_A, f_B \in \{u_1, u_2\}$. Conjecture which function f_*, $* = A, B$ corresponds to which u_*, $* = 1, 2$, and give arguments for your decision.

HIDDEN HINT 1 for (3-14.j) → `3-14-10-0:h71.pdf`

SOLUTION for (3-14.j) → `3-14-10-1:s7.pdf` ▲

End Problem 3-14 , 155 min.

Problem 3-15: Asymptotic Convergence of Finite-Element Discretization Errors and Interpolation Errors

Most theoretical results about the behavior of finite-element discretization errors are of an asymptotic nature, predicting its dependence on discretization parameters in some limit, often in the sense of h-convergence on sequences of ever finer meshes. In this short problem we review some result from the course.

This problem is based on [Lecture → Section 3.3.2] and [Lecture → Section 3.3.5].

Throughout this problem let $\mathcal{M}_0 \prec \mathcal{M}_1 \prec \cdots \prec \mathcal{M}_\ell \prec \ldots$ stand for a sequence of triangular meshes of a polygonal domain $\Omega \subset \mathbb{R}^2$ created by *successive regular refinement*. Write h_ℓ for the mesh width of $\mathcal{M}_\ell$ and assume $h_0 \leq 1$.

(3-15.a) ⚄ (15 min.) Denote by $\mathsf{I}_\ell : C^0(\overline{\Omega}) \to \mathcal{S}_1^0(\mathcal{M}_\ell)$ the nodal interpolation operator onto the space of $\mathcal{M}_\ell$-piecewise linear Lagrangian finite-element functions. Give the maximal sets of non-negative integers from which the parameters p and q in the following asymptotic interpolation estimate can be chosen: For any

- $q \in \{\quad\} \subset \mathbb{N}_0$,
- $p \in \{\quad\} \subset \mathbb{N}_0$,

there holds true

$$\exists C > 0: \quad \|u - \mathsf{I}_\ell u\|_{L^2(\Omega)} \leq C h_\ell^q \|u\|_{H^p(\Omega)} \quad \forall u \in H^p(\Omega),\ \forall \ell \in \mathbb{N}_0\ . \tag{3.15.1}$$

SOLUTION for (3-15.a) → 3-15-1-0:s1s.pdf

▲

(3-15.b) ⚄ (15 min.) In a numerical test for a Neumann problem

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^2 \quad , \quad \mathbf{grad}\, u \cdot \boldsymbol{n} = h \quad \text{on } \partial\Omega\ ,$$

the data were chosen to yield the exact solution $u(\boldsymbol{x}) = \exp(-\|\boldsymbol{x}\|^2)$.

We write $u_\ell \in \mathcal{S}_2^0(\mathcal{M}_\ell)$ for the finite-element Galerkin solution obtained on the mesh $\mathcal{M}_\ell$ by means of *quadratic* Lagrangian finite elements. Predict, how many steps of uniform regular refinement of $\mathcal{M}_\ell$ have to be conducted to reduce $|u - u_\ell|_{H^1(\Omega)}$ by a factor of 1000 in order to gain three decimal digits:

Required are [ ] refinement steps.

SOLUTION for (3-15.b) → 3-15-2-0:s2s.pdf ▲

End Problem 3-15, 30 min.

Problem 3-16: Non-conforming Crouzeix-Raviart FEM: Theoretical aspects

The Crouzeix-Raviart finite-element method relies on piecewise linear *possibly discontinuous* trial and test functions for the Galerkin discretization of of variational problems originally set in $H^1(\Omega)$. This use of trial and test spaces that are not subspaces of the function spaces on which a variational problem is posed distinguishes **non-conforming** finite-element methods.

In this problem we examine various aspect of the Crouzeix-Raviart FEM. No implementation is requested. The related Problem 2-14 covers implementation based on LEHRFEM++.
▷ problem name/problem code folder: CrouzeixRaviartFEMTheory

Let a *triangular* mesh $\mathcal{M}$ of a 2D bounded polygonal domain $\Omega \subset \mathbb{R}^2$ be given and write $\mathcal{N} = \{\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N\}$, $N \in \mathbb{N}$, for the ordered set of the midpoints of its edges.

The so-called Crouzeix-Raviart finite element space $\mathcal{CR}(\mathcal{M}) \subset L^2(\Omega)$ on the mesh $\mathcal{M}$ is defined as the span of the basis functions b_h^j, $j = 1, \ldots, N$, which satisfy

$$b_{h\,|K}^i \in \mathcal{P}_1(K) \quad \forall K \in \mathcal{M} \quad , \quad b_h^i(\boldsymbol{m}_j) = \begin{cases} 1 & \text{, if } i = j \text{ ,} \\ 0 & \text{else,} \end{cases} \quad i,j \in \{1, \ldots, N\} \text{ .} \tag{3.16.1}$$

(3-16.a) ⚀ (15 min.)

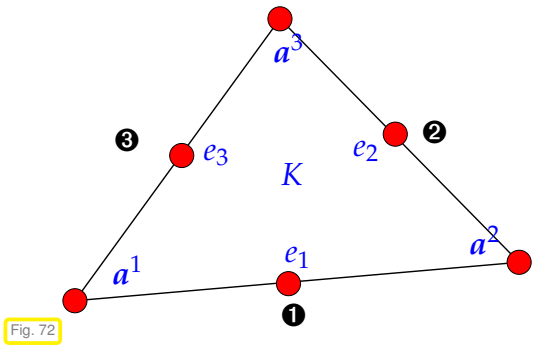


Fig. 72

The Crouzeix-Raviart finite element space $\mathcal{CR}(\mathcal{M}) \subset L^2(\Omega)$ is **affine equivalent** in the sense of [Lecture → Section 2.8.1]; all local shape functions are affine pullbacks of a few "reference local shape functions" $\widehat{b}^1, \ldots, \widehat{b}^Q$, $Q \in \mathbb{N}$, on the reference triangle $\widehat{K} := \left\langle \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\rangle$. Give explicit formulas for all of the $\widehat{b}^i$, $i = 1, 2, 3$, using the numbering of edges as in LEHRFEM++, which induces a numbering of the midpoints, see figure.

$$\widehat{b}^1(\widehat{\boldsymbol{x}}) = \boxed{} \text{ ,}$$

$$\widehat{b}^1(\widehat{\boldsymbol{x}}) = \boxed{} \text{ ,} \qquad \widehat{\boldsymbol{x}} = \begin{bmatrix} \widehat{x}_1 \\ \widehat{x}_2 \end{bmatrix} \text{ .}$$

$$\widehat{b}^2(\widehat{\boldsymbol{x}}) = \boxed{} \text{ ,}$$

SOLUTION for (3-16.a) → 3-16-1-0:s1cr.pdf ▲

(3-16.b) ⚀ (10 min.) In LEHRFEM++, with `mesh_p` providing a shared pointer to an **lf::mesh::Mesh** object describing a 2D triangular mesh $\mathcal{M}$, complete the initialization of a **lf::assemble::DofHandler** object suitable for the finite-element space $\mathcal{CR}(\mathcal{M})$:

```
lf::assemble::UniformFEDofHandler dh_CR(
mesh_p, {{lf::base::RefEl::kPoint(), [    ] },
  {lf::base::RefEl::kSegment(), [    ] },
  {lf::base::RefEl::kTria(), [    ] },
  {lf::base::RefEl::kQuad(), [    ] }});
```

SOLUTION for (3-16.b) → 3-16-2-0:s2cr.pdf ▲

Given uniformly positive continuous functions $\alpha, \gamma \in C^0(\overline{\Omega})$ and $f \in L^2(\Omega)$ we consider the variational problem

$$u \in H^1(\Omega): \quad \int_\Omega \alpha(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{3.16.2}$$

and its discrete "nonconforming" counterpart

$$u_h \in \mathcal{CR}(\mathcal{M}): \quad \sum_{K\in\mathcal{M}} \int_K \alpha(\boldsymbol{x})\, \mathbf{grad}\, u_h(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v_h \in \mathcal{CR}(\mathcal{M})\,. \tag{3.16.3}$$

(3-16.c) ⚀ (15 min.) Give a sharp upper bound for the number of non-zero entries of the matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ of the linear system arising from (3.16.3) when the basis $\{b_h^1, \ldots, b_h^N\}$ (defined in (3.16.1)) of $\mathcal{CR}(\mathcal{M})$ is used. Express the bound in terms of the numbers $\sharp\mathcal{M}$, $\sharp\mathcal{E}$, and $\sharp\mathcal{V}$ of cells, edges, and nodes, respectively, of the mesh $\mathcal{M}$:

$$\mathrm{nnz}(\mathbf{A}) \le \square \cdot \sharp\mathcal{M} + \square \cdot \sharp\mathcal{E} + \square \cdot \sharp\mathcal{V}\,.$$

SOLUTION for (3-16.c) → 3-16-3-0:crsas.pdf ▲

(3-16.d) ⚁ (30 min.) Given a triangle $K \in \mathcal{M}$ let $\mathbf{A}_K^{\mathcal{S}} \in \mathbb{R}^{3,3}$ stand for the element matrix arising in the $\mathcal{S}_1^0(\mathcal{M})$-based finite-element Galerkin discretization of (3.16.2) (assuming the use of the standard nodal tent-function basis). Conversely, we write $\mathbf{A}_K^{\mathcal{CR}}$ for the element matrix instrumental in the cell-oriented assembly of the linear system arising from (3.16.3) (using the basis of $\mathcal{CR}(\mathcal{M})$ introduced above).

Fully specify two matrices $\mathbf{S}_K$ and $\mathbf{T}_K$ such that

$$\mathbf{A}_K^{\mathcal{CR}} = \mathbf{S}_K \mathbf{A}_K^{\mathcal{S}} \mathbf{T}_K\,.$$

The local numbering conventions of LEHRFEM++ apply.

$$\mathbf{S}_K = \begin{bmatrix} & \\ & \\ & \end{bmatrix}, \quad \mathbf{T}_K = \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}.$$

SOLUTION for (3-16.d) → 3-16-4-0:s3cr.pdf ▲

(3-16.e) ⚄ (30 min.) [depends on Sub-problem (3-16.b)]

The inclusion $\mathcal{S}_1^0(\mathcal{M}) \subset \mathcal{CR}(\mathcal{M})$ holds for every triangular finite-element mesh. Hence every $v_h \in \mathcal{S}_1^0(\mathcal{M})$ can also be regarded as a function in $\mathcal{CR}(\mathcal{M})$.

Fill in the gaps in the Listing 3.16.7 of the LEHRFEM++-based C++ function

```
Eigen::VectorXd LinLagrDOFsToCRDOFs(
  const lf::assemble::DofHandler &dh_linfe,
  const lf::assemble::DofHandler &dh_cr,
  const Eigen::VectorXd &mu);
```

which expects in `dh_linfe` a **lf::assemble::DofHandler** object for $\mathcal{S}_1^0(\mathcal{M})$ (**lf::uscalfe::FeSpaceLagrangeO1**), in `dh_cr` a **lf::assemble::DofHandler** object for $\mathcal{CR}(\mathcal{M})$ as found in Sub-problem (3-16.b), and in `mu` a coefficient vector $\vec{\mu} = [\mu_1, \ldots, \mu_M]$, $M := \dim \mathcal{S}_1^0(\mathcal{M})$.

It is supposed to return a coefficient vector $\vec{\nu} = [\nu_1, \ldots, \nu_M] \in \mathbb{R}^N$, $N := \dim \mathcal{CR}(\mathcal{M})$, such that

$$\sum_{j=1}^{N} \nu_j b_h^j = \sum_{k=1}^{M} \nu_k q_h^k ,$$

where we wrote $\{q_h^1, \ldots, q_h^M\}$ for the standard nodal "tent-function" basis of $\mathcal{S}_1^0(\mathcal{M})$.

C++ code 3.16.7: Implementation of `LinLagrDOFsToCRDOFs()`

```
1  Eigen::VectorXd LinLagrDOFsToCRDOFs(const lf::assemble::DofHandler &dh_linfe,
2                                      const lf::assemble::DofHandler &dh_cr,
3                                      const Eigen::VectorXd &mu) {
4    using gdof_idx_t = lf::assemble::gdof_idx_t;
5    LF_ASSERT_MSG(dh_linfe.Mesh() == dh_cr.Mesh(),
6                  "DofHandlers must be based on the same mesh");
7    LF_ASSERT_MSG(                          == mu.size(), "Vector length mismatch");
8    std::shared_ptr<const lf::mesh::Mesh> mesh_p{dh_linfe.Mesh()};
9    const lf::mesh::Mesh &mesh{*mesh_p};
10   LF_ASSERT_MSG((                         == mesh.NumEntities(1)) &&
11                        (                  == mesh.NumEntities(2)),
12                 "#DOFs do not match #entities");
13   Eigen::VectorXd nu(                     );
14   for (const lf::mesh::Entity *e :                                ) {
15     nonstd::span<const gdof_idx_t> lf_dof_idx{dh_linfe.GlobalDofIndices(*e)};
16     nonstd::span<const gdof_idx_t> cr_dof_idx{dh_cr.GlobalDofIndices(*e)};
17     LF_ASSERT_MSG((lf_dof_idx.size() ==     ) && (cr_dof_idx.size() ==     ),
18                   "Wrong numbers of local DOFs");
19     nu[                                   ] =                                   ;
20   }
21   return nu;
22 }
```

HIDDEN HINT 1 for (3-16.e) → 3-16-5-0:crs4h1.pdf

SOLUTION for (3-16.e) → 3-16-5-1:s4cr.pdf ▲

(3-16.f) ⚀ (20 min.) The $L^2(\Omega)$**-orthogonal projection** $\mathsf{P} : \mathcal{CR}(\mathcal{M}) \to \mathcal{S}_1^0(\mathcal{M})$ is defined through

$$\mathsf{P}w_h \in \mathcal{S}_1^0(\mathcal{M}): \quad \int_\Omega (\mathsf{P}w_h(\boldsymbol{x}) - w_h(\boldsymbol{x}))\, v_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v_h \in \mathcal{S}_1^0(\mathcal{M}) . \tag{3.16.9}$$

Specify size and entries of the matrices $\mathbf{X}$ and $\mathbf{Y}$ in the linear system

$$\mathbf{X}\vec{\mu} = \mathbf{Y}\vec{\nu}$$

relating the basis expansion coefficient vector $\vec{\mu}$ of $\mathrm{P}w_h$ and basis expansion coefficient vector $\vec{\nu}$ of $w_h \in \mathcal{CR}(\mathcal{M})$. Rely on the basis $\{b_h^1, \ldots, b_h^N\}$ from (3.16.1) for $\mathcal{CR}(\mathcal{M})$ and the standard nodal basis $\{q_h^1, \ldots, q_h^M\}$ of $\mathcal{S}_1^0(\mathcal{M})$.

$\mathbf{X} =$, $\mathbf{Y} =$.

SOLUTION for (3-16.f) → 3-16-6-0:s5cr.pdf ▲

(3-16.g) ⚄ (15 min.) Let $(\mathcal{M}_h)_h$ be a sequence of triangular meshes of a bounded polygonal domain Ω. The index $h > 0$ also indicates the meshwidth and we have $h \to 0$. The shape regularity measures of all meshes $\mathcal{M}_h$ are assumed to be uniformly bounded. For $u \in H^3(\Omega)$ determine the asymptotic behavior of the following quantities:

$$\inf_{v_h \in \mathcal{CR}(\mathcal{M}_h)} \|u - v_h\|_{L^2(\Omega)} = O\Big(\quad\Big) \quad \text{for} \quad h \to 0\,,$$

$$\inf_{v_h \in \mathcal{CR}(\mathcal{M}_h)} \sum_{K \in \mathcal{M}} |u - v_h|^2_{H^1(K)} = O\Big(\quad\Big) \quad \text{for} \quad h \to 0\,.$$

SOLUTION for (3-16.g) → 3-16-7-0:s6cr.pdf ▲

End Problem 3-16, 135 min.

Problem 3-17: Residual-Based A-Posteriori Error Estimator

In the context of finite-element methods, a-posteriori error estimators are formulas that, based on the known data for a boundary value problem and the finite-element Galerkin solution, try to predict a norm of the discretization error and its distribution in the computational domain. A very important class of a-posteriori error estimators employ local cell and face residuals.

In this problem we practice the implementation of an important residual-based a-posteriori error estimator for linear Lagrangian finite elements in LEHRFEM++.
▷ problem name/problem code folder: ResidualErrorEstimator

On a bounded connected polygonal domain $\Omega \subset \mathbb{R}^2$ we consider the homogeneous Dirichlet boundary value problem

$$-\operatorname{div}(\alpha\, \mathbf{grad}\, u) = f \quad \text{in} \quad \Omega \quad , \quad u = 0 \quad \text{on} \quad \partial\Omega \, , \tag{3.17.1}$$

where $\alpha \in C^0_{\mathrm{pw}}(\overline{\Omega})$ is uniformly positive almost everywhere in Ω, and $f \in C^0(\overline{\Omega})$.

(3-17.a) ⚀ (10 min.) State the variational formulation of the boundary value problem (3.17.1) in a suitable Sobolev space.

$u \in$ [ ] : [ ] = [ ] $\forall$ [ ] .

SOLUTION for (3-17.a) → 3-17-1-0:s1s.pdf ▲

In [Ver13, Chapter 4] we find the derivation of the residual-based local a-posteriori error estimator, when (3.17.1) is discretized by means of a Lagrangian finite-element method on a triangular mesh $\mathcal{M}$ of Ω. We state the formulas under the following assumption:

Assumption 3.17.2.

We assume that the coefficient function $\alpha \in C^0_{\mathrm{pw}}(\overline{\Omega})$ is constant inside every cell of $\mathcal{M}$.

We write $\alpha_K := \alpha|_K \in \mathbb{R}$ for all $K \in \mathcal{M}$ and $u_h \in C^0(\overline{\Omega})$ for the finite-element solution. Further let $\mathcal{E}_0$ be the set of *interior* edges of $\mathcal{M}$ and $\boldsymbol{n}_e$ a unit vector normal to e for every $e \in \mathcal{E}$. The direction of $\boldsymbol{n}_e$ is arbitrary, but supposed to be fixed once and for all. Given an $\mathcal{M}$-piecewise smooth function $\varphi : \Omega \to \mathbb{R}$, we define its *jump* across the edge $e \in \mathcal{E}_0$ by

$$[\![\varphi]\!]_e(\boldsymbol{x}) := \lim_{t\to 0} \{\varphi(\boldsymbol{x} + t\boldsymbol{n}_e) - \varphi(\boldsymbol{x} - t\boldsymbol{n}_e)\} \, , \quad \boldsymbol{x} \in e \, . \tag{3.17.3}$$

Then the key formula from [Ver13, Thoerem 4.41] is

$$\mathrm{ree}(u_h) := \sum_{K\in\mathcal{M}} \mu_K^2 \|f + \alpha_K \Delta u_h|_K\|^2_{L^2(K)} + \sum_{e\in\mathcal{E}} \mu_e \|[\![\boldsymbol{n}_e \cdot (\alpha\, \mathbf{grad}\, u_h)]\!]_e\|^2_{L^2(e)} \, , \tag{3.17.4}$$

$$\mu_K := \frac{h_K}{\sqrt{\alpha_K}} \quad , \quad \mu_e := \frac{h_e}{\max\{\alpha_e^+, \alpha_e^-\}} \, , \tag{3.17.5}$$

where α_e^+, α_e^- are the two values of α on the two triangles adjacent to the edge e, and h_K and h_e denote the diameter of a triangle and the length of an edge, respectively.

From now we restrict ourselves to piecewise linear Lagrangian finite elements, that is, to the finite-element spaces $\mathcal{S}^0_1(\mathcal{M})$ and $\mathcal{S}^0_{1,0}(\mathcal{M})$.

(3-17.b) ⚂ (15 min.) For the special case $f \equiv 0$ and $\alpha \equiv 1$ characterize the function space

$$Z_h := \{v_h \in \mathcal{S}_1^0(\mathcal{M}) : \ \mathrm{ree}(v_h) = 0\} \,. \tag{3.17.6}$$

Tick the right answer ($\mathcal{P}_k(\mathbb{R}^2)$ is the space of bi-variate polynomials of degree $\leq k$):

$Z_h = \mathcal{P}_0(\mathbb{R}^2)$ ○
$Z_h = \{0\}$ ○
$Z_h = \mathcal{P}_1(\mathbb{R}^2)$ ○
$Z_h = \mathcal{S}_{1,0}^0(\mathcal{M})$ ○
$Z_h = \mathcal{S}_1^0(\mathcal{M})$ ○

SOLUTION for (3-17.b) → 3-17-2-0:s2l.pdf ▲

The implementation of $\mathrm{ree}(u_h)$ relies on two auxiliary data types, whose definitions can be found in `residualerrorestimator.h`:

- **MeshFunctionPWConst** is a LEHRFEM++ **MeshFunction**-compatible type that stores an $\mathcal{M}$-piecewise constant **double**-valued function. In particular it provides an evaluation operator

```
std::vector<double> operator()(
const lf::mesh::Entity &e,
const Eigen::MatrixXd &refc) const;
```

  that returns the constant value for the cell `e` `refc.cols()` times. The entries of `refc` are ignored.

- **dataDiscreteBVP** collects the information describing the boundary value problem and its finite-element discretization, in particular the coefficient α, the source function f, and the numbering of basis functions:

```
struct dataDiscreteBVP {
  // Constructor, which essentially copies the passed arguments
  dataDiscreteBVP(std::shared_ptr<const lf::mesh::Mesh> mesh_p,
  std::function<double(Eigen::Vector2d)> alpha,
  std::function<double(Eigen::Vector2d)> f);
  std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
     pwlinfespace_p_;
  lf::mesh::utils::MeshFunctionGlobal<
  std::function<double(Eigen::Vector2d)>> mf_f_;
  MeshFunctionPWConst mf_alpha_;
};
```

(3-17.c) ⚄ (45 min.) In the file `residualerrorestimator.cc` complete the implementation of the C++ function

```
lf::mesh::utils::CodimMeshDataSet<double> volumeResiduals(
const dataDiscreteBVP &disc_bvp, Eigen::VectorXd u_vec);
```

that returns a **lf::mesh::utils::CodimMeshDataSet** object that, for every cell $K \in \mathcal{M}$, provides the value

$$\eta_K := \mu_K^2 \| f + \alpha_K \Delta u_h|_K \|_{L^2(K)}^2 \in \mathbb{R} \,,$$

when `u_vec` is the basis expansion coefficient vector for $u_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$, and `disc_bvp` passes the information about α, f, and the finite-element space.

For $f \in \mathcal{S}_1^0(\mathcal{M})$ the exact value of η_K should be computed (assuming exact arithmetic).

HIDDEN HINT 1 for (3-17.c) → 3-17-3-0:3h1.pdf

SOLUTION for (3-17.c) → 3-17-3-1:s3lfpp.pdf ▲

(3-17.d) ⚄ (90 min.) In the file `residualerrorestimator.cc` finish the implementation of the C++ function

```
lf::mesh::utils::CodimMeshDataSet<double> edgeResiduals(
const dataDiscreteBVP &disc_bvp, Eigen::VectorXd u_vec);
```

This function is to compute for every edge $e \in \mathcal{E}_0$ the values

$$\rho_e := \mu_e \| [\![\boldsymbol{n}_e \cdot (\alpha \, \mathbf{grad}\, u_h)]\!]_e \|^2_{L^2(e)} \in \mathbb{R}$$

and stores them in a **lf::mesh::utils::CodimMeshDataSet** object, which it returns. The values for edges on $\partial\Omega$ can be left undefined. The meaning of the arguments is

- `disc_bvp` provides complete information about the boundary-value problem and the finite element space $\mathcal{S}_1^0(\mathcal{M})$,
- `u_vec` gives the basis expansion coefficient vector of $u_h \in \mathcal{S}_1^0(\mathcal{M})$ with respect to the standard nodal basis.

HIDDEN HINT 1 for (3-17.d) → 3-17-4-0:h41.pdf

HIDDEN HINT 2 for (3-17.d) → 3-17-4-1:h42.pdf

SOLUTION for (3-17.d) → 3-17-4-2:s4max.pdf ▲

References

[Ver13] Rüdiger Verfürth. *A posteriori error estimation techniques for finite element methods*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, 2013, pp. xx+393. DOI: 10.1093/acprof:oso/9780199679423.001.0001 (cit. on p. 173).

End Problem 3-17 , 160 min.

Problem 3-18: Hierarchical Local A-Posteriori Error Estimator

A local finite-element a-posteriori error estimator is an algoritm that based on a computed finite-element Galerkin solution provides an estimate for some norm of the discretization on every cell of the finite-element mesh. Hierarchical error estimators are an important class and realize the simple idea that the difference of Galerkin solutions obtained by two finite-element spaces of different polynomial degree is probably a good approximation of the discretization error.

This problem centers around implementation in LEHRFEM++ with a focus on the Lagrangian finite element spaces of polynomial degree 1 and 2. Implementation based on LEHRFEM++ is requested.
▷ problem name/problem code folder: HierarchicalErrorEstimator

For a bounded polygon $\Omega \subset \mathbb{R}^2$ we consider the linear variational problem

$$u \in H^1_0(\Omega): \quad \underbrace{\int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = \underbrace{\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:\ell(v)} \quad \forall v \in H^1_0(\Omega)\,, \tag{3.18.1}$$

for a bounded, uniformly positive definite, continuous, matrix-valued function $\boldsymbol{\alpha} : \overline{\Omega} \to \mathbb{R}^{2,2}$, and $f \in C^0(\overline{\Omega})$.

We take for granted a sequence $(\mathcal{M}_\ell)_{\ell\in\mathbb{N}_0}$ of meshes of Ω created by successive global regular refinement of an initial *triangular* mesh $\mathcal{M}_0$. We adopt the notations

- N_V^ℓ, N_E^ℓ, and N_C^ℓ for the numbers of nodes, edges, and cells of $\mathcal{M}_\ell$,
- $N_{\partial,V}^\ell$ and $N_{\partial,E}^\ell$ for the numbers of nodes and edges located on $\partial\Omega$,

and let h_ℓ stand for the meshwidth of $\mathcal{M}_\ell$.

Further, we write u for the exact solution of (3.18.1),

- $u_\ell \in \mathcal{S}^0_{1,0}(\mathcal{M}_\ell)$ for the Galerkin finite-element solution in $\mathcal{S}^0_{1,0}(\mathcal{M}_\ell) := H^1_0(\Omega) \cap \mathcal{S}^0_1(\mathcal{M}_\ell)$,
- and $\widetilde{u}_\ell \in \mathcal{S}^0_{2,0}(\mathcal{M}_\ell)$ for the Galerkin finite-element solution in $\mathcal{S}^0_{2,0}(\mathcal{M}_\ell) := H^1_0(\Omega) \cap \mathcal{S}^0_2(\mathcal{M}_\ell)$.

(3-18.a) ⚀ (10 min.) What are the dimensions of $\mathcal{S}^0_{1,0}(\mathcal{M}_\ell)$ and $\mathcal{S}^0_{2,0}(\mathcal{M}_\ell)$? Express them in terms of N_*^ℓ, $* = V, E, C$, and $N_{\partial,*}^\ell$, $* = V, E$.

$$\dim \mathcal{S}^0_{1,0}(\mathcal{M}_\ell) = \boxed{}\,,$$
$$\dim \mathcal{S}^0_{2,0}(\mathcal{M}_\ell) = \boxed{}\,.$$

SOLUTION for (3-18.a) → 3-18-1-0:hems1.pdf ▲

(3-18.b) ⚁ (15 min.) You know that $u \in H^2(\Omega)$. Give a sharp characterization of the asymptotic behavior of $|\widetilde{u}_\ell - \widetilde{u}_{\ell-1}|_{H^1(\Omega)}$ for $h_\ell \to 0$ (which corresponds to $\ell \to \infty$):

$$|\widetilde{u}_\ell - \widetilde{u}_{\ell-1}|_{H^1(\Omega)} = O\Big(\boxed{}\Big) \quad \text{for} \quad h_\ell \to 0\,.$$

SOLUTION for (3-18.b) → 3-18-2-0:hem2p.pdf ▲

(3-18.c) ⚂ (15 min.) We write a for the bilinear form on the left-hand side of (3.18.1). Specify the largest test spaces such that the following variational equations hold true for any f:

$$\mathsf{a}(\widetilde{u}_\ell - u, v_h) = 0 \quad \forall v_h \in \boxed{}\,, \tag{3.18.2}$$

$$\mathsf{a}(u - u_\ell, v_h) = 0 \quad \forall v_h \in \boxed{}\,, \tag{3.18.3}$$

$$\mathsf{a}(\widetilde{u}_\ell - u_\ell, v_h) = 0 \quad \forall v_h \in \boxed{}\,. \tag{3.18.4}$$

SOLUTION for (3-18.c) → 3-18-3-0:hersp3.pdf ▲

(3-18.d) ⚄ (20 min.) [depends on Sub-problem (3-18.c)]

Write $\|\cdot\|_a$ for the energy norm on $H^1(\Omega)$ induced by the bilinear form a from (3.18.1). Motivated by the expectation that piecewise quadratic Lagrangian finite elements provide better accuracy than piecewise linears we make the so-called **saturation assumption**

$$\exists 0 \le \gamma < 1\colon \quad \|\widetilde{u}_\ell - u\|_a \le \gamma \|u_\ell - u\|_a \quad \forall \ell \in \mathbb{N}_0\,. \tag{3.18.5}$$

Determine sharp constants $C_1 = C_1(\gamma) > 0$ and $C_2 = C_2(\gamma) > 0$ such that

$$C_1 \|u_\ell - u\|_a \le \|\widetilde{u}_\ell - u_\ell\|_a \le C_2 \|u_\ell - u\|_a \quad \forall \ell \in \mathbb{N}_0\,. \tag{3.18.6}$$

$$C_1 = \boxed{}\,,$$

$$C_2 = \boxed{}\,.$$

SOLUTION for (3-18.d) → 3-18-4-0:hestsp4.pdf ▲

(3-18.e) ⚅ (60 min.) We consider a generic triangular mesh $\mathcal{M}$ of Ω. As $\mathcal{S}_1^0(\mathcal{M}) \subset \mathcal{S}_2^0(\mathcal{M})$ every finite-element function $v_h \in \mathcal{S}_1^0(\mathcal{M})$ can be written as a linear combination of basis functions of $\mathcal{S}_2^0(\mathcal{M})$. In the file `hierarchicalerrorestimator.cc` supplement the missing parts of the C++ function

```
Eigen::VectorXd trfLinToQuad(
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>
   fes_lin_p,
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>
   fes_quad_p,
const Eigen::VectorXd &mu);
```

that, when passed a vector `mu` of basis expansion coefficients in $\mathcal{S}_1^0(\mathcal{M})$ returns its vector of basis expansion coefficients in $\mathcal{S}_2^0(\mathcal{M})$. The standard choice of bases implemented in LEHRFEM++ is assumed. Both passed objects `fes_lin_p` and `fes_quad_p` must be built on the same *triangular* mesh $\mathcal{M}$.

SOLUTION for (3-18.e) → 3-18-5-0:tls.pdf ▲

Again, let $\mathcal{M}$ be a generic triangular mesh of Ω and write $\mathcal{V}$ for the set of its nodes. The quadratic hierarchical surplus finite-element space on $\mathcal{M}$ is defined as

$$\widehat{\mathcal{S}}_2^0(\mathcal{M}) := \left\{ v_h \in \mathcal{S}_2^0(\mathcal{M})\colon\ v_h(\boldsymbol{p}) = 0 \quad \forall \boldsymbol{p} \in \mathcal{V} \right\}. \tag{3.18.8}$$

(3-18.f) ⚄ (15 min.) Determine the dimension $M := \dim \widehat{\mathcal{S}}_2^0(\mathcal{M})$ in terms of the numbers N_*, $* = V, E, C$, of nodes, edges, and cells of $\mathcal{M}$. Characterize a basis $\widehat{\mathfrak{B}} := \left\{ \widetilde{b}_h^1, \ldots, \widetilde{b}_h^M \right\}$ of $\dim \widehat{\mathcal{S}}_2^0(\mathcal{M})$ comprising functions with minimal support.

$$M = \boxed{}\,.$$

SOLUTION for (3-18.f) → 3-18-6-0:hemsp7.pdf ▲

We set $\widehat{\mathcal{S}}^0_{2,0} := \widehat{\mathcal{S}}^0_2(\mathcal{M}) \cap H^1_0(\Omega)$. Given $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ we denote by $\widehat{w}_h \in \widehat{\mathcal{S}}^0_{2,0}$ the unique finite-element function in the quadratic hierarchical surplus finite-element space satisfying the **residual equation**

$$\mathsf{a}(\widehat{w}_h, v_h) = \ell(v_h) - \mathsf{a}(u_h, v_h) \quad \forall v_h \in \widehat{\mathcal{S}}^0_{2,0} \,. \tag{3.18.9}$$

(3-18.g) ⚄ (60 min.) [depends on Sub-problem (3-18.e)]

In the file `hierarchicalerrorestimator.h` implement the LEHRFEM++-based C++ function template

```
Eigen::VectorXd compHierSurplusSolution(
const MESHFUNCTION_ALPHA &mf_alpha, const MESHFUNCTION_F &mf_f,
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
   fes_lin_p,
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>>
   fes_quad_p,
const Eigen::VectorXd &mu);
```

that computes the $\mathcal{S}^0_2(\mathcal{M})$ basis expansion coefficient vector of the solution $\widetilde{w}_h$ of (3.18.9). The arguments have the following meanings

1. `alpha` and `f` are LEHRFEM++ **MeshFunction**-compatible objects providing the coefficient function α and the right-hand-side source function f.
2. `fes_lin_p` and `fes_quad_p` are pointers to object encapsulating the Lagrangian finite element spaces $\mathcal{S}^0_1(\mathcal{M})$ and $\mathcal{S}^0_2(\mathcal{M})$ built on the *same* mesh $\mathcal{M}$.
3. `mu` is the basis expansion coefficient vector for a function $u_h \in \mathcal{S}^0_1(\mathcal{M})$ using the global numbering of basis functions as fixed by `*fes_lin_p`. $u_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ can be taken for granted.

HIDDEN HINT 1 for (3-18.g) → 3-18-7-0:hhs1.pdf

HIDDEN HINT 2 for (3-18.g) → 3-18-7-1:hhs23.pdf

HIDDEN HINT 3 for (3-18.g) → 3-18-7-2:hhs4.pdf

SOLUTION for (3-18.g) → 3-18-7-3:hss.pdf

▲

End Problem 3-18, 195 min.

Problem 3-19: Convergence of Finite-Element Solutions

This problems revisits some of the numerical experiments reported in [Lecture → Section 3.2.3] and is about identifying the numerical experiment from error curves.

A purely theoretical problem based on [Lecture → Section 3.2.3]. Knowledge of [Lecture → Section 3.3.5] is certainly useful.

Throughout this problem we consider Dirichlet problems for $-\Delta$,

$$-\Delta u = f \quad \text{in} \quad \Omega \quad , \quad u = g \quad \text{on} \quad \partial\Omega \, , \tag{3.19.1}$$

on a 2D polygonal domain $\Omega \subset \mathbb{R}^2$ and with $f \in L^2(\Omega)$, $g \in H^{\frac{1}{2}}(\partial\Omega) \cap C^0(\partial\Omega)$.

(3-19.a) ⚂ (30 min.) We consider the following six numerical experiments. Throughout, the numerical experiments relied on *Lagrangian finite elements* on quasi-uniform and shape-regular sequences of triangular meshes.

(I) $\Omega :=]0,1[^2$, f, g such that $u(\boldsymbol{x}) = \sin(\pi x_1)\sin(\pi x_2)$, monitored $L^2(\Omega)$-norms of finite-element discretization errors for trial and test space $\mathcal{S}^0_{1,0}(\mathcal{M})$

(II) $\Omega :=]0,1[^2$, f, g such that $u(\boldsymbol{x}) = \sin(\pi x_1)\sin(\pi x_2)$, monitored $L^2(\Omega)$-norms of finite-element discretization errors for trial and test space $\mathcal{S}^0_{2,0}(\mathcal{M})$

(III) $\Omega :=]0,1[^2$, f, g such that $u(\boldsymbol{x}) = \sin(\pi x_1)\sin(\pi x_2)$, monitored: $H^1(\Omega)$-seminorms of finite-element discretization errors for trial and test space $\mathcal{S}^0_{1,0}(\mathcal{M})$

(IV) $\Omega :=]0,1[^2$, f, g such that $u(\boldsymbol{x}) = \sin(\pi x_1)\sin(\pi x_2)$, monitored: $H^1(\Omega)$-seminorms of finite-element discretization errors for trial and test space $\mathcal{S}^0_{2,0}(\mathcal{M})$

(V) $\Omega :=]-1,1[^2 \backslash [0,1]^2$, f, g such that $u(\boldsymbol{x}) = r^{2/3}\sin(\frac{2}{3}\varphi)$ (polar coordinates), monitored: $H^1(\Omega)$-seminorms of finite-element discretization errors for trial and test space $\mathcal{S}^0_{1,0}(\mathcal{M})$

(VI) $\Omega :=]-1,1[^2 \backslash [0,1]^2$, f, g such that $u(\boldsymbol{x}) = r^{2/3}\sin(\frac{2}{3}\varphi)$ (polar coordinates), monitored: $H^1(\Omega)$-seminorms of finite-element discretization errors for trial and test space $\mathcal{S}^0_{2,0}(\mathcal{M})$

The plots below display the monitored norms of the FE discretization error for the various experiments and sequences of seven meshes (magenta line).

Please label the plots with the number of the experiment(s) that could have produced them .

A single plot may have been obtained from several experiments. List them all.

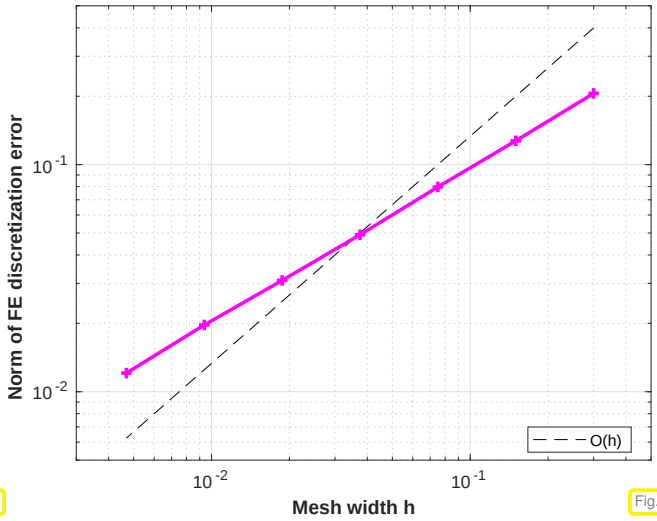


Fig. 73

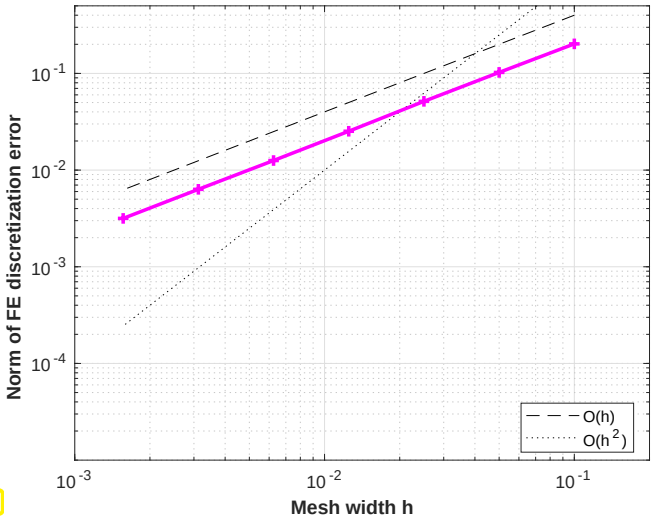


Fig. 74

Plot **A**: Exp(s)

Plot **B**: Exp(s)

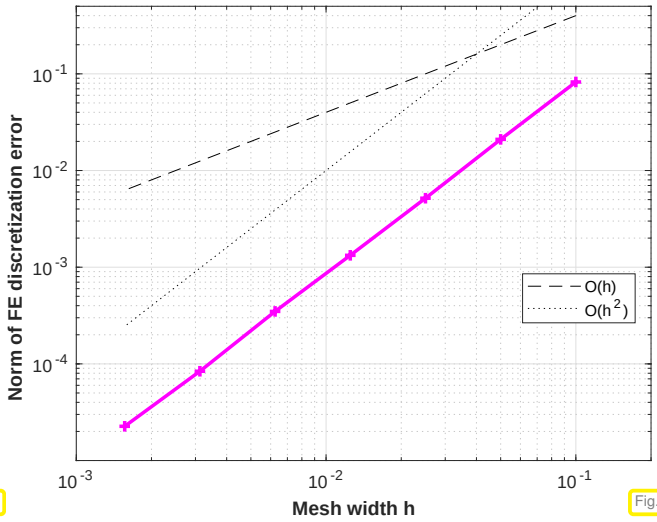


Fig. 75

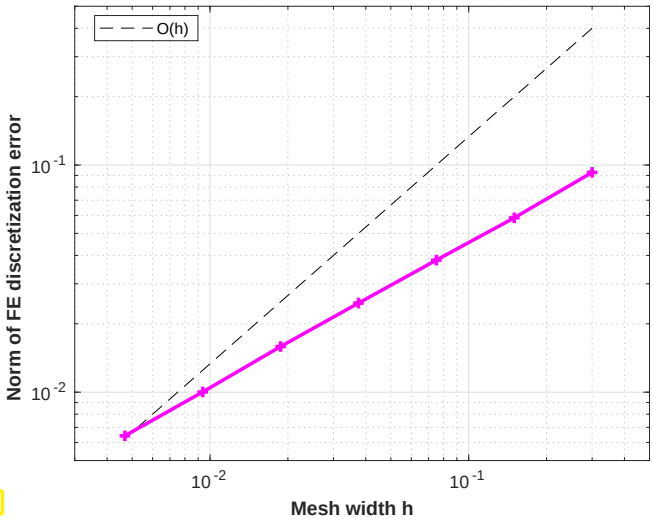


Fig. 76

Plot **C**: Exp(s)

Plot **D**: Exp(s)

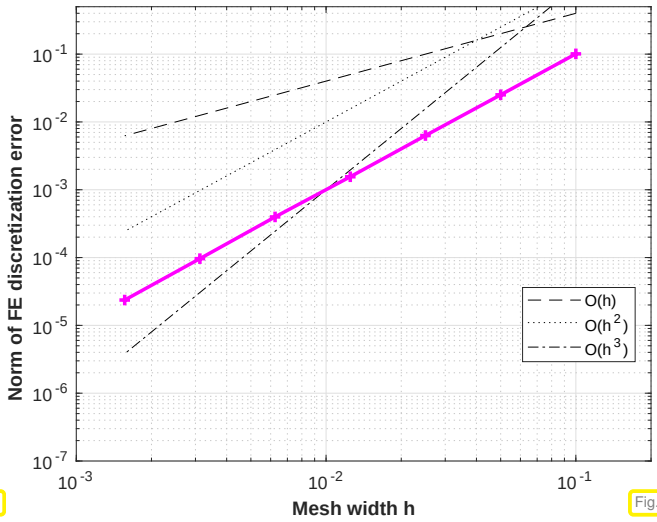


Fig. 77

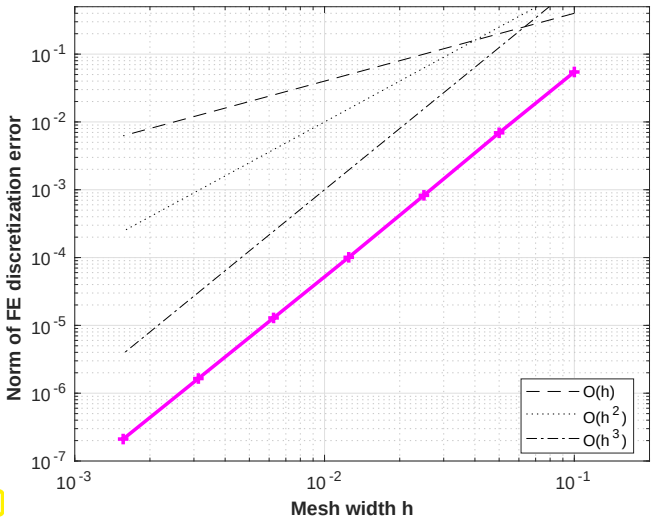


Fig. 78

Plot **E**: Exp(s)

Plot **F**: Exp(s)

SOLUTION for (3-19.a) → 3-19-1-0:ecfea.pdf ▲

(3-19.b) ⚀ (10 min.) How do you have to choose f and g in (3.19.1) for the unit square domain

$\Omega :=]0,1[^2$ such that

$$u(\boldsymbol{x}) = x_1(1-x_1)x_2(1-x_2)\,, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \text{ ?} \tag{3.19.2}$$

$$f(\boldsymbol{x}) = \boxed{}\,, \quad \boldsymbol{x} \in \Omega\,,$$

$$g(\boldsymbol{x}) = \boxed{}\,, \quad \boldsymbol{x} \in \partial\Omega\,.$$

SOLUTION for (3-19.b) → 3-19-2-0:ecfesb.pdf ▲

(3-19.c) ⚄ (15 min.) We consider (3.19.1) for $\Omega :=]0,1[^2$ and with manufactured solution (3.19.2). Given an arbitrary triangular mesh $\mathcal{M}$ of Ω, what is the *smallest* (= lowest-dimensional) Lagrangian finite element space V_h on $\mathcal{M}$ such that the discretization error of the finite-element Galerkin discretization vanishes?

$$V_h = \boxed{}\,.$$

SOLUTION for (3-19.c) → 3-19-3-0:ecfecs.pdf ▲

End Problem 3-19, 55 min.

Problem 3-20: First-order System Least-Squares Variational Formulation

This problem studies a non-standard variational formulation of a second-order elliptic boundary value problem.

This problem relies on parts of [Lecture → Chapter 1] and [Lecture → Chapter 3]. It is a purely theoretical problem.
▷ problem name/problem code folder: FOSLSEllipticBVP

Given a bounded computational domain $\Omega \subset \mathbb{R}^2$ we consider the quadratic minimization problem

$$(u, \mathbf{j}) = \operatorname*{argmin}_{(v,\mathbf{q})\in V} J\big((v,\mathbf{q})\big)\,, \tag{3.20.1}$$

$$J\big((v,\mathbf{q})\big) := \tfrac{1}{2}\left\| \sqrt{\alpha}\,\mathbf{grad}\, v - \frac{1}{\sqrt{\alpha}}\mathbf{q}\right\|^2_{L^2(\Omega)} + \tfrac{1}{2}\|\operatorname{div}\mathbf{q} - f\|^2_{L^2(\Omega)}\,, \tag{3.20.2}$$

(see [Lecture → Suppl. 1.5.2.3] for the definition of div) on the space

$$V := C^1_0(\overline{\Omega}) \times \Big(C^1(\overline{\Omega})\Big)^2\,. \tag{3.20.3}$$

Here, $\alpha \in C^0(\overline{\Omega})$ is a uniformly positive, real-valued function, and $f \in L^2(\Omega)$. Also recall the definition

$$C^1_0(\overline{\Omega}) := \{v : \overline{\Omega} \to \mathbb{R} \text{ continuously differentiable, } v|_{\partial\Omega} = 0\}\,. \tag{3.20.4}$$

(3-20.a) ⚄ (20 min.) State a second-order scalar linear elliptic Dirichlet boundary value problem on Ω, whose solution will provide the u-component of a solution of (3.20.2).

HIDDEN HINT 1 for (3-20.a) → 3-20-1-0:f0h1.pdf

SOLUTION for (3-20.a) → 3-20-1-1:fo0.pdf ▲

(3-20.b) ⚄ (30 min.) State a linear variational problem

(i) with a *symmetric* bilinear form $\mathsf{a}_F : V \times V \to \mathbb{R}$

(ii) for which every solution of the *quadratic minimization problem* (3.20.1) is also a solution.

SOLUTION for (3-20.b) → 3-20-2-0:fo1.pdf ▲

(3-20.c) ⚄ (15 min.) [depends on Sub-problem (3-20.b)]

Show that the bilinear form $\mathsf{a}_F : V \times V \to \mathbb{R}$ underlying the variational problem from Sub-problem (3-20.b) is continuous with respect to the energy norm

$$\|(v,\mathbf{q})\|^2_A := \|\mathbf{grad}\, v\|^2_{L^2(\Omega)} + \|\operatorname{div}\mathbf{q}\|^2_{L^2(\Omega)} + \|\mathbf{q}\|^2_{L^2(\Omega)}\,, \quad (v,\mathbf{q}) \in V\,. \tag{3.20.11}$$

HIDDEN HINT 1 for (3-20.c) → 3-20-3-0:fo2h2.pdf

SOLUTION for (3-20.c) → 3-20-3-1:fo2.pdf ▲

Estimates employing the Helmholtz decomposition of vector fields give the following result for the bilinear form found in Sub-problem (3-20.b):

Theorem 3.20.15. Ellipticity of FOSLS bilinear form [Cai+94]

There is a constant $\gamma > 0$ depending only on Ω such that

$$\mathsf{a}_F\big((v,\mathbf{q}),(v,\mathbf{q})\big) \geq \gamma\|(v,\mathbf{q})\|^2_A \quad \forall (v,\mathbf{q}) \in V\,.$$

We restrict ourselves to polygonal domains Ω. Next we perform a Galerkin finite element discretization of the variational problem from Sub-problem (3-20.b). Based on a triangular mesh $\mathcal{M}$ of Ω we use the finite element space

$$V_{0,h} := \mathcal{S}^0_{2,0}(\mathcal{M}) \times \left(\mathcal{S}^0_2(\mathcal{M})\right)^2 .$$

(3-20.d) ⚄ (15 min.) [depends on Sub-problem (3-20.b)]

Argue why $V_{0,h}$ can be used for the discretization of the variational problem from Sub-problem (3-20.b) or, equivalently, the minimization problem (3.20.2), though, owing to lack of smoothness, $V_{0,h} \not\subset V$ in general.

SOLUTION for (3-20.d) → 3-20-4-0:fo3.pdf ▲

(3-20.e) ⚅ (20 min.) For testing the Galerkin finite scheme introduced above we consider $\Omega =]0,1[^2$, $\alpha \equiv 1$, and choose f in (3.20.2) such that we obtain as exact solution

$$u(\boldsymbol{x}) = \sin(\pi x_1)\sin(\pi x_2) \quad , \quad \mathbf{j}(\boldsymbol{x}) = \pi \begin{bmatrix} \cos(\pi x_1)\sin(\pi x_2) \\ \sin(\pi x_2)\cos(\pi x_2) \end{bmatrix} .$$

Denote by $(u_h, \mathbf{j}_h) \in V_{0,h}$ the Galerkin finite element solution, given $N := \dim V_{0,h}$.

Describe qualitatively and quantitatively the asympotic convergence of

$$\|u - u_h\|_{H^1(\Omega)} \quad \text{and} \quad \|\mathbf{div}(\mathbf{j} - \mathbf{j}_h)\|_{L^2(\Omega)}$$

for $N \to \infty$, that one expects on a sequence of meshes obtained by regular uniform refinement of some coarse initial mesh.

HIDDEN HINT 1 for (3-20.e) → 3-20-5-0:fs4h1.pdf

SOLUTION for (3-20.e) → 3-20-5-1:.pdf ▲

References

[Cai+94] Z. Cai, R.D. Lazarov, T.A. Manteuffel, and S.F. McCormick. "First–order system least–squares for partial differential equations: Part I". In: *SIAM J. Numer. Anal.* 31 (1994), pp. 1785–1799 (cit. on p. 182).

End Problem 3-20 , 100 min.

Problem 3-21: Asymptotic Convergence of Finite-Element Solutions

For Lagrangian finite element solutions and derived quantities we expect asymptotic algebraic convergence of errors as the meshwidth of the underlying finite-element mesh tends to zero. This problem examines various such errors and requests predictions of rates of convergence.

The tools to solve this problem have been provided in [Lecture → Section 3.3.5], [Lecture → Section 3.6.1], and [Lecture → Section 3.6.3]. In addition, you will have to be able to read LEHRFEM++-based C++ code.
▷ problem name/problem code folder: AsymptoticCvgFEM

On a polygonally bounded domain $\Omega \subset \mathbb{R}^2$ we consider the variational problem: seek $u \in H^1(\Omega)$

$$\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \kappa(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \quad \forall v \in H^1(\Omega)\,. \tag{3.21.1}$$

Here, $f \in C^0(\overline{\Omega})$ and $h \in C^0_{\mathrm{pw}}(\partial\Omega)$ are given functions. The impendance coefficient $\kappa \in C^0_{\mathrm{pw}}(\partial\Omega)$ has to be bounded and uniformly positive.

(3-21.a) ⚀ (16 min.) State the second-order elliptic boundary value problem, whose weak/variational form is given by (3.21.1).

	in Ω,
	on $\partial\Omega$.

SOLUTION for (3-21.a) → 3-21-1-0:acvga.pdf ▲

We equip Ω with a triangular finite element mesh $\mathcal{M}$. We carry out a finite-element Galerkin discretization of (3.21.1) based on **quadratic Lagrangian finite elements** [Lecture → Ex. 2.6.1.2], that is, we use $\mathcal{S}^0_2(\mathcal{M})$ as discrete trial and test space.

(3-21.b) ⚁ (12 min.) In the context of the finite-element Galerkin discretization, what is the **minimal order** [Lecture → Def. 2.7.5.29] of the local quadrature rules used for the approximate evaluation of the integrals

- $\int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$,
- $\int_{\partial\Omega} \kappa(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})$,

so that the quadrature errors vanish for any mesh, whenever $f \in \mathcal{S}^0_1(\mathcal{M})$ and $\kappa \in \mathcal{S}^0_1(\mathcal{M})|_{\partial\Omega}$?

Minimal quadrature order for $\int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$	=	,
Minimal quadrature order for $\int_{\partial\Omega} \kappa(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})$	=	.

SOLUTION for (3-21.b) → 3-21-2-0:acvpb.pdf ▲

For testing purposes a CSE student relies on the following concrete setting, which will be used throughout the remainder of this problem.

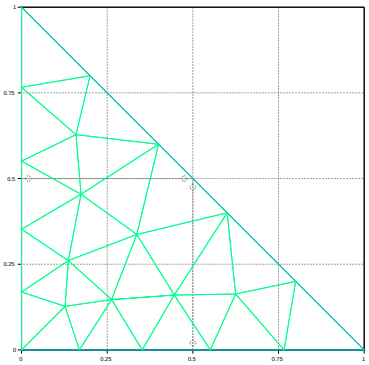


Fig. 79

- The computational domain is the "unit triangle"

$$\Omega := \operatorname{convex}\left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\} .$$

◁ domain Ω equipped with coarse FE mesh

- The impedance coefficient is constant $\kappa \equiv 1$.
- The data f and h are chosen so that (3.21.1) has the exact solution

$$u(\boldsymbol{x}) = \exp(\|\boldsymbol{x}\|^2) , \quad \|\boldsymbol{x}\| := \sqrt{x_1^2 + x_2^2} , \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} .$$

(3-21.c) ⚂ (16 min.) [depends on Sub-problem (3-21.a)]

We denote the sides of the unit triangle with Γ_1, Γ_2, and Γ_3 as in the figure beside.

Give an *explicit* expression of the function h in the student's setting:

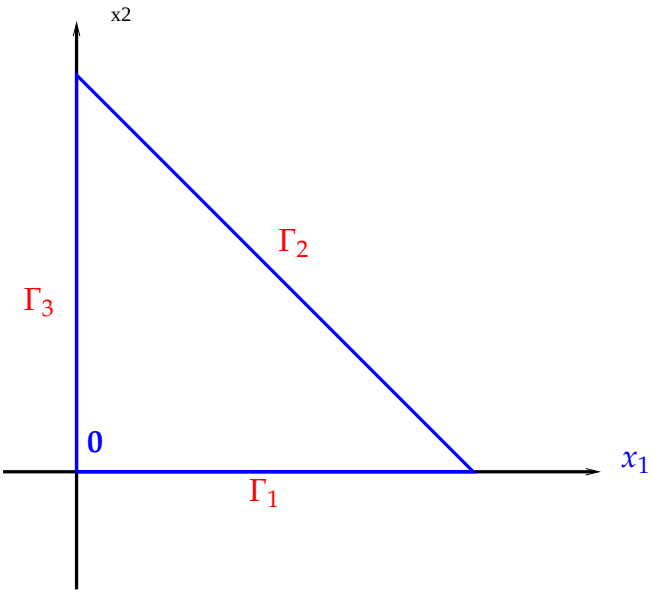


Fig. 80

$$h(\boldsymbol{x}) = \begin{cases} \qquad\qquad & \text{on } \Gamma_1 , \\ \qquad\qquad & \text{on } \Gamma_2 , \\ \qquad\qquad & \text{on } \Gamma_3 . \end{cases}$$

SOLUTION for (3-21.c) → 3-21-3-0:.pdf ▲

The student creates a sequence of triangular meshes $(\mathcal{M}_\ell)_{\ell=0}^7$ by successive regular refinement of the initial triangular mesh shown in Fig. 79. On those meshes he computes the finite-element Galerkin solutions $u_{h,\ell} \in \mathcal{S}_2^0(\mathcal{M}_{\ell\ll})$, $\ell = 0,\ldots,7$, of (3.21.1). Then he implements a number of functions `error_X()`, $X \in \{I, II, III, IV, V\}$, that compute various errors. He feeds them with $u_{h,\ell}$, $\ell = 0,\ldots,7$, and the exact solution $u(\boldsymbol{x}) = \exp \|\boldsymbol{x}\|^2$ of his test boundary value problem.

(3-21.d) ⚂ (12 min.) The function `error_II()` is listed in Code 3.21.4. The argument `fes_p` passes the usual information on the mesh $\mathcal{M}_\ell$ and the degrees of freedom, while `mu_vec` contains the basis expansion coefficient vector for the finite-element solution $u_{h,\ell}$. The argument `solg` contains a functor of type **std**`::function<Eigen::Vector2d(Eigen::Vector2d)>` and provides the

gradient of the exact solution u: `solg` $\leftrightarrow$ $\mathbf{grad}\, u$.

C++ code 3.21.4: Code for `error_II()`

```
template <typename FUNCTOR>
double error_II(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_p,
    const Eigen::VectorXd& mu_vec, FUNCTOR solg) {
  const lf::fe::MeshFunctionGradFE mf_fesolg(fes_p, mu_vec);
  const lf::mesh::utils::MeshFunctionGlobal mf_solg{solg};
  return std::sqrt(lf::fe::IntegrateMeshFunction(
      *(fes_p->Mesh()), lf::mesh::utils::squaredNorm(mf_fesolg - mf_solg), 4));
}
```

What mathematical expression (in terms of $u, u_{h,\ell}, \Omega$) describes the quantity returned by `error_II()`, if all numerical quadratures are supposed to provide the exact values?

Return value of `error_II()` = ____.

SOLUTION for (3-21.d) → `3-21-4-0:acvpI.pdf` ▲

(3-21.e) ⚄ (12 min.) [depends on Sub-problem (3-21.d)]

The function `error_II()` is called for $u_{h,0}, u_{h,1}, u_{h,2}, \ldots$. Make a *sharp* prediction of the rate of asymptotic algebraic convergence of the return values in terms of the meshwidth $h \to 0$.

Output of `error_II()` $= O(h^{\square})$ for $h \to 0$.

SOLUTION for (3-21.e) → `3-21-5-0:.pdf` ▲

(3-21.f) ⚄ (12 min.) The function `error_V()` is listed in Code 3.21.12. The argument `fes_p` passes the usual information on the mesh $\mathcal{M}_\ell$ and the degrees of freedom, while `mu_vec` contains the basis expansion coefficient vector for the finite-element solution $u_{h,\ell}$. The argument `sol` contains a functor of type `std::function<double(Eigen::Vector2d)>` and provides the exact solution $u(\boldsymbol{x}) = \exp(\|\boldsymbol{x}\|^2)$.

C++ code 3.21.12: Code for `error_V()`

```
template <typename FUNCTOR>
double error_V(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_p,
    const Eigen::VectorXd& mu_vec, FUNCTOR sol) {
  const lf::fe::MeshFunctionFE mf_fesol(fes_p, mu_vec);
  const lf::mesh::utils::MeshFunctionGlobal mf_sol{sol};
  return std::sqrt(lf::fe::IntegrateMeshFunction(
      *(fes_p->Mesh()), lf::mesh::utils::squaredNorm(mf_fesol - mf_sol), 6));
}
```

What mathematical expression (in terms of $u, u_{h,\ell}, \Omega$) describes the quantity returned by `error_V()`, if all numerical quadratures are supposed to provide the exact values?

Return value of `error_V()` = ____.

SOLUTION for (3-21.f) → 3-21-6-0:acvpI.pdf ▲

(3-21.g) ⚀ (12 min.) [depends on Sub-problem (3-21.f)]

The function `error_V()` is called for $u_{h,0}, u_{h,1}, u_{h,2}, \ldots$. Make a *sharp* prediction of the rate of asymptotic algebraic convergence of the return values in terms of the meshwidth $h \to 0$.

$$\text{Output of } \texttt{error_V()} = O\big(h^{\square}\big) \quad \text{for} \quad h \to 0\,.$$

HIDDEN HINT 1 for (3-21.g) → 3-21-7-0:acvpVah1.pdf

SOLUTION for (3-21.g) → 3-21-7-1:.pdf ▲

(3-21.h) ⚁ (12 min.) The listing Code 3.21.14 displays function `error_III()`. The argument `fes_p` passes the usual information on the mesh $\mathcal{M}_\ell$ and the degrees of freedom, while `mu_vec` contains the basis expansion coefficient vector for the finite-element solution $u_{h,\ell}$. The argument `sol` contains a functor of type **std::function<double(Eigen::Vector2d)>** and provides the exact solution u.

C++ code 3.21.14: Code for `error_III()`

```
2  template <typename FUNCTOR>
3  double error_III(
4      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_p,
5      const Eigen::VectorXd& mu_vec, FUNCTOR sol) {
6    const auto bd_flags{
7        lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 1)};
8    auto bd_sel = [&bd_flags](const lf::mesh::Entity& edge) -> bool {
9      return bd_flags(edge);
10   };
11   const lf::fe::MeshFunctionFE mf_fesol(fes_p, mu_vec);
12   const lf::mesh::utils::MeshFunctionGlobal mf_sol{sol};
13   return std::sqrt(lf::fe::IntegrateMeshFunction(
14       *(fes_p->Mesh()), lf::mesh::utils::squaredNorm(mf_fesol - mf_sol), 6,
15       bd_sel, 1));
16 }
```

What mathematical expression (in terms of $u, u_{h,\ell}, \Omega$) describes the quantity returned by `error_II()`, if all numerical quadratures are supposed to provide the exact values?

Return value of `error_III()` = ☐ .

SOLUTION for (3-21.h) → 3-21-8-0:acvpI.pdf ▲

(3-21.i) ⚂ (12 min.) [depends on Sub-problem (3-21.h)]

The function `error_III()` is called for $u_{h,0}, u_{h,1}, u_{h,2}, \ldots$. Make a *sharp* prediction of the rate of asymptotic algebraic convergence of the return values in terms of the meshwidth $h \to 0$.

$$\text{Output of } \texttt{error_III()} = O\big(h^{\square}\big) \quad \text{for} \quad h \to 0\,.$$

SOLUTION for (3-21.i) → 3-21-9-0:acvpsIII.pdf ▲

(3-21.j) ⚂ (12 min.) The function `error_I()` is shown in Code 3.21.15. The argument `fes_p` passes the usual information on the mesh $\mathcal{M}_\ell$ and the degrees of freedom, while `mu_vec` contains the basis expansion coefficient vector for the finite-element solution $u_{h,\ell}$. The argument `sol` contains a functor of type `std::function<double(Eigen::Vector2d)>` and provides the exact solution u.

C++ code 3.21.15: Code for `error_I()`

```
template <typename FUNCTOR>
double error_I(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_p,
    const Eigen::VectorXd& mu_vec, FUNCTOR sol) {
  const lf::fe::MeshFunctionFE mf_fesol(fes_p, mu_vec);
  const lf::mesh::utils::MeshFunctionGlobal mf_sol{sol};
  const auto bd_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 1)};
  auto bd_sel = [&bd_flags](const lf::mesh::Entity& edge) -> bool {
    return bd_flags(edge);
  };
  return std::abs(lf::fe::IntegrateMeshFunction(
      *(fes_p->Mesh()), mf_fesol - mf_sol, 4, bd_sel, 1));
}
```

What mathematical expression (in terms of $u, u_{h,\ell}, \Omega$) describes the quantity returned by `error_I()`, if all numerical quadratures are supposed to provide the exact values?

Return value of `error_I()` = ____ .

SOLUTION for (3-21.j) → 3-21-10-0:acvpI.pdf ▲

(3-21.k) ⚄ (12 min.) [depends on Sub-problem (3-21.j)]

The function `error_I()` is called for $u_{h,0}, u_{h,1}, u_{h,2}, \ldots$. Make a *sharp* prediction of the rate of asymptotic algebraic convergence of the return values in terms of the meshwidth $h \to 0$.

$$\text{Output of } \texttt{error_I()} = O\big(h^{\square}\big) \quad \text{for} \quad h \to 0\,.$$

HIDDEN HINT 1 for (3-21.k) → 3-21-11-0:acvpIah1.pdf

SOLUTION for (3-21.k) → 3-21-11-1:.pdf ▲

(3-21.l) ⚂ (12 min.) The function `error_IV()` is listed in Code 3.21.17. The argument `fes_p` passes the usual information on the mesh $\mathcal{M}_\ell$ and the degrees of freedom, while `mu_vec` contains the basis expansion coefficient vector for the finite-element solution $u_{h,\ell}$. The argument `sol` contains a functor of type `std::function<double(Eigen::Vector2d)>` and provides the exact solution u.

C++ code 3.21.17: Code for `error_IV()`

```
template <typename FUNCTOR>
double error_IV(
```

```
4     std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_p,
5     const Eigen::VectorXd& mu_vec, FUNCTOR sol) {
6   const lf::fe::MeshFunctionFE mf_fesol(fes_p, mu_vec);
7   const lf::mesh::utils::MeshFunctionGlobal mf_sol{sol};
8   return std::abs(
9       lf::fe::IntegrateMeshFunction(*(fes_p->Mesh()),
10                                    lf::mesh::utils::squaredNorm(mf_fesol), 6) -
11      lf::fe::IntegrateMeshFunction(*(fes_p->Mesh()),
12                                    lf::mesh::utils::squaredNorm(mf_sol), 6));
13  }
```

What mathematical expression (in terms of $u, u_{h,\ell}, \Omega$) describes the quantity returned by `error_IV()`, if all numerical quadratures are supposed to provide the exact values?

Return value of `error_IV()` = ____ .

SOLUTION for (3-21.l) → 3-21-12-0:acvpI.pdf ▲

(3-21.m) ⚂ (12 min.) [depends on Sub-problem (3-21.l)]

The function `error_IV()` is called for $u_{h,0}, u_{h,1}, u_{h,2}, \ldots$. Make a *sharp* prediction of the rate of asymptotic algebraic convergence of the return values in terms of the meshwidth $h \to 0$.

Output of `error_IV()` $= O(h^{\square})$ for $h \to 0$.

SOLUTION for (3-21.m) → 3-21-13-0:acvpsIVa.pdf ▲

End Problem 3-21, 164 min.

Problem 3-22: Coupled Elliptic Boundary Value Problems

Many physical models involve scalar second-order elliptic boundary value problems coupled in various ways. Here we consider a particularly simple case, namely two unidirectionally coupled BVPs: the solution of the second depends on that of the first, but not vice versa. So the BVPs can be solved in turns.

This problem connects to variational formulations of second-order elliptic BVPs [Lecture → Section 1.8] and also assumes familioarity with finite element implementation in LEHRFEM++ [Lecture → Section 2.7]. Knowledge of [Lecture → Section 3.7] will also be required.
▷ problem name/problem code folder: CoupledBVPs

For a particular application engineers employ a dimensionally reduced model for modulated heat transfer, which boils down to the following two coupled second-order elliptic Dirichlet boundary value problems

$$\left\{ \begin{array}{rcll} -\Delta w &=& f^2 & \text{in } \Omega\,, \\ w &=& 1 & \text{on } \partial\Omega\,, \end{array} \right. \tag{3.22.1a}$$

$$\left\{ \begin{array}{rcll} -\operatorname{div}(w\,\mathbf{grad}\, v) &=& w & \text{in } \Omega\,, \\ v &=& 0 & \text{on } \partial\Omega\,, \end{array} \right. \tag{3.22.1b}$$

Here, $f : \Omega \to \mathbb{R}$ is a real-valued source function and f^2 stands for its square: $f^2(\boldsymbol{x}) = f(\boldsymbol{x})^2$, $\boldsymbol{x} \in \Omega$.

The computational domain $\Omega \subset \mathbb{R}^2$ is chosen to be a hexagon with center $\mathbf{0}$,

$$\Omega := \operatorname{convex}\left\{ \begin{bmatrix} \cos(\pi/2j) \\ \sin(\pi/3j) \end{bmatrix} . j = 0, \ldots, 5 \right\}$$

Hexagonal computational domain equipped with a triangular mesh ▷

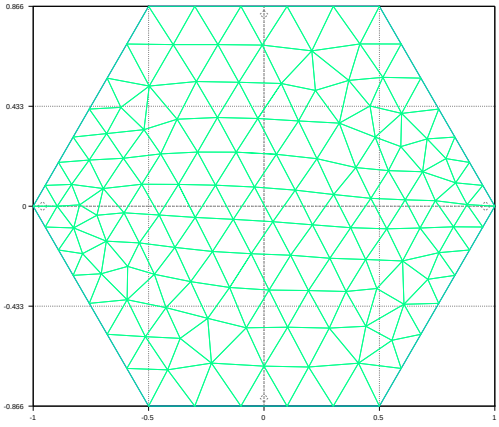


Fig. 86

(3-22.a) ⚂ (24 min.) The weak form of (3.22.1) ((3.22.1a) and (3.22.1b) together!) boils down to a linear variational problem [Lecture → Def. 1.4.1.6], which can be written as

$$\boxed{u \in \widehat{V}:\quad \mathsf{a}(u, \widetilde{u}) = \ell(\widetilde{u}) \quad \forall \widetilde{u} \in V_0}\,, \qquad \text{[Lecture → (1.4.1.7)]}$$

Specify concretely the affine space $\widehat{V}$, the vector space V_0, the bilinear form $\mathsf{a}(\cdot,\cdot)$, and the linear form

$\ell(\cdot)$ for the variational form of (3.22.1) (both (3.22.1a) and (3.22.1b)!).

$$\widehat{V} := \quad ,$$

$$V_0 := \quad ,$$

$$\mathsf{a}\left(\begin{bmatrix} w \\ v \end{bmatrix}, \begin{bmatrix} \widetilde{w} \\ \widetilde{v} \end{bmatrix}\right) := \quad ,$$

$$\ell\left(\begin{bmatrix} \widetilde{w} \\ \widetilde{v} \end{bmatrix}\right) := \quad .$$

SOLUTION for (3-22.a) → 3-22-1-0:cbvp0.pdf ▲

(3-22.b) ⚄ (16 min.) For which of the following source functions $f = f(\boldsymbol{x})$, $\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \Omega$, will there exist a **weak/variational solution** [Lecture → Rem. 1.5.3.10] $w \in H^1(\Omega)$ of (3.22.1a)? Tick the appropriate circles.

Function f	weak solution exists	weak solution does not exist
(i) $f(\boldsymbol{x}) := (\log\|\boldsymbol{x}\|)^{1/2}$	◯	◯
(ii) $f(\boldsymbol{x}) := \dfrac{1}{\|\boldsymbol{x}\|}$	◯	◯
(iii) $f(\boldsymbol{x}) := \sqrt[3]{\|\boldsymbol{x}\|}$	◯	◯
(iv) $f(\boldsymbol{x}) := \dfrac{\sin(x_1)}{x_1}$	◯	◯

SOLUTION for (3-22.b) → 3-22-2-0:cbvpa.pdf ▲

(3-22.c) ⚄ (24 min.) Assume that the source function is smooth: $f \in C^\infty(\overline{\Omega})$. Explain why in this case (3.22.1) has a unique weak solution $(w, v) \in H^1(\Omega) \times H^1(\Omega)$.

HIDDEN HINT 1 for (3-22.c) → 3-22-3-0:copsbsh.pdf

SOLUTION for (3-22.c) → 3-22-3-1:cbvpb.pdf ▲

We discretize both Dirichlet boundary value problems of (3.22.1) by means of piecewise linear Lagrangian finite elements on the same triangular mesh $\mathcal{M}$. All occurring integrals are (approximately) evaluated by the local midpoint quadrature rule [Lecture → (2.7.5.35)].

(3-22.d) ⚂ (12 min.) Formulate a requirement to be met by $\mathcal{M}$ such that existence and uniqueness of finite element solutions $w_h \in \mathcal{S}_1^0(\mathcal{M})$ and $v_h \in \mathcal{S}_1^0(\mathcal{M})$ of (3.22.1a) and (3.22.1b), respectively, is guaranteed for any $f \in C^\infty(\overline{\Omega})$.

HIDDEN HINT 1 for (3-22.d) → 3-22-4-0:cophsc.pdf

SOLUTION for (3-22.d) → 3-22-4-1:cbvpc.pdf ▲

(3-22.e) ⚄ (35 min.) In the file `coupledbvps.h` finish the implementation of the LEHRFEM++-based C++ function

```
template <typename F_FUNCTOR>
  std::pair<Eigen::VectorXd,Eigen::VectorXd>
    solveModulatedHeatFlow(
```

```
std::shared_ptr<const
   lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
F_FUNCTOR &&f);
```

which is supposed to return the basis expansion coefficient vectors of the finite-element solutions $w_h \in \mathcal{S}_1^0(\mathcal{M})$ and $v_h \in \mathcal{S}_1^0(\mathcal{M})$ of (3.22.1a) and (3.22.1b), respectively. The argument `fes_p` provides information about the mesh and the local→global index mapping via a **lf::assemble::DofHandler** object.

The functor type **F_FUNCTOR** is expected to provide an evaluation operator compatible with **std**::`function`<**double**(`Eigen::Vector2d`)>. The functor argument `f` passes the source function f.

HIDDEN HINT 1 for (3-22.e) → `3-22-5-0:cbvphx.pdf`

SOLUTION for (3-22.e) → `3-22-5-1:cbvpsx.pdf` ▲

End Problem 3-22 , 111 min.

Problem 3-23: Neumann Data by Projection

For a second-order scalar elliptic boundary value problem posed on a domain $\Omega \subset \mathbb{R}^2$ the Neumann data is the function $\mathbf{j} \cdot \boldsymbol{n}|_{\partial\Omega}$, where $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ is the exterior unit normal vector field and $\mathbf{j} : \Omega \to \mathbb{R}^2$ is the flux. This problem studies ways to approximate the Neumann data based on a finite-element solution of a Dirichlet BVP.

The problem requires intimate familiarity with LehrFEM++ and [Lecture → Section 2.7].
▷ problem name/problem code folder: NeumannDataRecovery

For a bounded computational domain $\Omega \subset \mathbb{R}^2$ and a source function $f \in L^2(\Omega)$ we consider the linear variational problem

$$u \in H^1_0(\Omega): \quad \underbrace{\int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = \underbrace{\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:\ell(v)} \quad \forall v \in H^1_0(\Omega)\,, \tag{3.23.1}$$

where $\boldsymbol{\alpha} : \Omega \to \mathbb{R}^{2,2}$ is a symmetric, bounded, uniformly positive definite diffusion coefficient.

(3-23.a) ⚀ (3 min.) Give the strong (PDE) form the the boundary value problem associated with (3.23.1).

$$\square = \square \quad \text{in } \Omega\,,$$
$$\square = \square \quad \text{on } \partial\Omega\,.$$

SOLUTION for (3-23.a) → 3-23-1-0:ndrs1.pdf ▲

(3-23.b) ⚀ (3 min.) Give an explicit formula for the energy norm induced by the bilinear form a on he left-hand side of (3.23.1).

$$\|v\|_a = \square\,, \quad v \in H^1(\Omega)\,. \tag{3.23.2}$$

SOLUTION for (3-23.b) → 3-23-2-0:ndrsol2.pdf ▲

(3-23.c) ⚁ (15 min.) In the file `neumanndatarecovery.cc` complete the implementation of the LehrFEM++-based C++ function

```
lf::mesh::utils::CodimMeshDataSet<double> getNeumannData(
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
   fe_space,
const Eigen::VectorXd &mu);
```

This function should return an **lf::mesh::utils::CodimMeshDataSet** object, which holds the value $\mathbf{grad}\,u_h(\boldsymbol{m}_e) \cdot \boldsymbol{n}_e$ for every edge e on the boundary of Ω_h. Here,

- Ω_h is the domain covered by the cells of the triangular mesh contained in the **lf::uscalfe::FeSpaceLagrangeO1** object,
- $u_h \in \mathcal{S}^0_1(\mathcal{M})$ is the piecewise linear function, whose basis expansion coefficient vector is passed through the argument `mu`,
- $\boldsymbol{m}_e$ is the midpoint of the straight edge e,

- and $\boldsymbol{n}_e$ is the is a unit normal for e pointing into the exterior of Ω_h.

For interior edges the **lf::mesh::utils::CodimMeshDataSet** object should just store 0.

HIDDEN HINT 1 for (3-23.c) → 3-23-3-0:ndr4h1.pdf

SOLUTION for (3-23.c) → 3-23-3-1:ndr4s.pdf ▲

(3-23.d) ⚄ (5 min.) [depends on Sub-problem (3-23.a)]

Let $u \in H^1_0(\Omega)$ be the solution of (3.23.1) and set $h(\boldsymbol{x}) := \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})$, $\boldsymbol{x} \in \partial\Omega$. Fill the blanks in the following identity.

$$\int_{\partial\Omega} h(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S = \mathsf{a}\Big(\boxed{}, \boxed{}\Big) + \ell\Big(\boxed{}\Big) \quad \forall v \in H^1(\Omega)\,, \tag{3.23.5}$$

where $\mathsf{a}(\cdot,\cdot)$ and $\ell(\cdot)$ are the bilinear form/linear form from (3.23.1).

SOLUTION for (3-23.d) → 3-23-4-0:ndrs4a.pdf ▲

Write $\mathfrak{B} = \{b_h^1, \ldots, b_h^M\}$, $M := \sharp\mathcal{V}(\mathcal{M})$, for the standard nodal basis (set of global shape functions) of $\mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh approximating Ω. We take for granted an ordering of the basis functions such that

$$\mathfrak{B} = \mathfrak{B}_0 \cup \mathfrak{B}_\partial\,, \quad \mathfrak{B}_0 = \left\{b_h^1, \ldots, b_h^N\right\}, \quad \mathfrak{B}_\partial = \left\{b_h^{N+1}, \ldots, b_h^M\right\}, \quad \mathcal{S}_{1,0}^0(\mathcal{M}) = \mathrm{Span}\{\mathfrak{B}_0\}\,. \tag{3.23.6}$$

Write $\mathbf{A} \in \mathbb{R}^{M,M}$ for the $\mathcal{S}_1^0(\mathcal{M})$ finite-element Galerkin matrix for the bilinear form

$$(u,v) \mapsto \mathsf{a}(u,v) := \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1(\Omega)\,, \tag{3.23.7}$$

using the basis $\mathfrak{B}$, and $\mathbf{Q} \in \mathbb{R}^{M,M}$ for the $\mathcal{S}_1^0(\mathcal{M})$ finite-element Galerkin matrix spawned by the bilinear form

$$(w,v) \mapsto \int_{\partial\Omega} w(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \quad w,v \in H^1(\Omega)\,, \tag{3.23.8}$$

using the same basis $\mathfrak{B}$. The partitioning of the basis $\mathfrak{B}$ according to (3.23.6) induces a block-partitioning of the Galerkin matrices

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{\partial,0} \\ \mathbf{A}_{0,\partial} & \mathbf{A}_{\partial,\partial} \end{bmatrix}\,, \quad \mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{0,0} & \mathbf{Q}_{\partial,0} \\ \mathbf{Q}_{0,\partial} & \mathbf{Q}_{\partial,\partial} \end{bmatrix}\,. \tag{3.23.9}$$

The **Neumann data projection** $\mathsf{N}_h : H^1(\Omega) \to \mathrm{Span}\{\mathfrak{B}_\partial\}$ is defined as

$$\int_{\partial\Omega_h} (\mathsf{N}_h u)(\boldsymbol{x})\,v_h(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\;\mathbf{grad}\,v_h(\boldsymbol{x}) - f(\boldsymbol{x})v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v_h \in \mathrm{Span}\{\mathfrak{B}_\partial\}\,. \tag{3.23.10}$$

(3-23.e) ⚄ (8 min.) Given $u_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$ with basis expansion coefficient vector $\vec{\boldsymbol{\mu}} \in \mathbb{R}^N$, that is, $u_h = \sum_{j=1}^N (\vec{\boldsymbol{\mu}})_j b_h^j$, derive a linear system of equations that yields the expansion coefficient vector $\vec{\boldsymbol{\eta}} \in \mathbb{R}^{M-N}$ of $\mathsf{N}_h u_h$ with respect to the basis $\mathfrak{B}_\partial$.

$$\boxed{}\,\vec{\boldsymbol{\eta}} = \boxed{}\,.$$

Use the notations for matrices introduced in (3.23.9).

SOLUTION for (3-23.e) → 3-23-5-0:ndrs4bs.pdf ▲

(3-23.f) ⚄ (15 min.) Code 3.23.12 displays the code of the C++ function

```
template <typename DIFFCOEFF, typename SOURCEFUNC>
std::pair<Eigen::VectorXd, Eigen::VectorXd> projectedNeumannData(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space,
    DIFFCOEFF &alpha, SOURCEFUNC &f);
```

The argument `fe_space` passes information about the finite element mesh $\mathcal{M}$, which is assumed to be triangular mesh with only straight edges, and the local→global index mapping (encoded in **lf::assemble::DofHandler** object). It also conveys that the finite element space $\mathcal{S}_1^0(\mathcal{M})$ is used for discretization.

The function computes the $\mathcal{S}_{1,0}^0(\mathcal{M})$ Galerkin solution of (3.23.1), where the coefficient α and the right-hand-side source function f are supplied as **MeshFunction** objects through the arguments `alpha` and `f`, respectively.

- The basis expansion coefficient vector of the Galerkin solution, as fixed by the **lf::assemble::DofHandler** object in `fe_space`, is the *first return value*.
- The *second return value* is the basis expansion coefficient vector of $\mathrm{N}_h u_h$ as defined in (3.23.10).

All returned basis expansion coefficient vector refer to the full basis of $\mathcal{S}_1^0(\mathcal{M})$. The coefficients corresponding to a priori non-contributing basis functions are to be set to zero.

Fill in the blanks in the following listing with valid C++/LehrFEM++ code to make the implementation of `projectedNeumannData()` match the specification. Complex control structures like **for**/**while**-loops and **if**/**else**-statements are not required.

C++ code 3.23.12: Incomplete code for the function `projectedNeumannData()`

```
template <typename DIFFCOEFF, typename SOURCEFUNC>
std::pair<Eigen::VectorXd, Eigen::VectorXd> projectedNeumannData(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    DIFFCOEFF &alpha, SOURCEFUNC &f) {
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space->Mesh();
  const lf::mesh::Mesh &mesh{*mesh_p};
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  const lf::base::size_type N_dofs(dofh.NumDofs());
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Right-hand side vector
  Eigen::VectorXd phi = Eigen::VectorXd::Zero(N_dofs);
  // Assembly of full Galerkin matrix and full right-hand side vector
  lf::mesh::utils::MeshFunctionConstant<double> mf_gamma{0.0};
  lf::uscalfe::ReactionDiffusionElementMatrixProvider elmat_builder(
      fe_space, alpha, mf_gamma);
  lf::uscalfe::ScalarLoadElementVectorProvider elvec_builder(fe_space, f);
  // Cell-oriented assembly of Galerkin matrix and r.h.s. vector
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);
  // Save full Galerkin matrix and full load vector
  const Eigen::SparseMatrix<double> A_full = A.makeSparse();
  const Eigen::VectorXd phi_full = phi;
  // Impose zero Dirichlet boundary conditions
  lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 2)};
  auto Dir_sel = [&](lf::assemble::gdof_idx_t idx) -> std::pair<bool, double> {
    const lf::mesh::Entity &node{dofh.Entity(idx)};
```

```
    return {bd_flags(node), 0.0};
  };
  lf::assemble::FixFlaggedSolutionComponents<double>(Dir_sel, A, phi);
  // Assembly completed: Convert COO matrix A into CRS format using Eigen's
  // internal conversion routines.
  const Eigen::SparseMatrix<double> A_crs = A.makeSparse();

  // Solve linear system using Eigen's sparse direct elimination
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_A;
  solver_A.compute(A_crs);  // LU decomposition
  LF_VERIFY_MSG(solver_A.info() == Eigen::Success, "LU decomposition of A failed");
  Eigen::VectorXd sol_vec = solver_A.solve(phi);
  LF_VERIFY_MSG(solver_A.info() == Eigen::Success, "Solving Galerkin LSE failed");

  // Edge mass matrix for edges on the boundary
  lf::mesh::utils::CodimMeshDataSet<bool> bded_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(mesh_p,                    )};
  lf::mesh::utils::MeshFunctionConstant<double> mf_bdcoeff{              };
  lf::uscalfe::MassEdgeMatrixProvider edmat_builder(fe_space,            ,
      [&bded_flags](const lf::mesh::Entity &edge) -> bool {
        return                                                           ;
      });
  lf::assemble::COOMatrix<double> M(          ,          );
  lf::assemble::AssembleMatrixLocally(1,          ,          ,          ,          );
  Eigen::VectorXd rho =                                                  ;
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&](lf::assemble::gdof_idx_t idx) -> std::pair<bool, double> {
        return {                                                  ,          };
      },
      ,          );
  const Eigen::SparseMatrix<double> M_crs = M.makeSparse();
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_M;
  solver_M.compute(M_crs);  // LU decomposition
  LF_VERIFY_MSG(solver_M.info() == Eigen::Success, "LU decomposition of M failed");
  Eigen::VectorXd neudof_vec = solver_M.solve(                );
  LF_VERIFY_MSG(solver_M.info() == Eigen::Success, "Solving LSE failed");
  return {                ,                };
}
```

SOLUTION for (3-23.f) → 3-23-6-0:ndrsol5.pdf ▲

(3-23.g) ⚂ (6 min.) As a test case we use $\Omega =]0,1[^2$, and choose f in (3.23.1) with $\alpha \equiv 1$ such that the exact solution becomes $u(x) = \sin(\pi x_1)\sin(\pi x_2)$, $x \in \Omega$. We consider the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin discretization of (3.23.1) on a sequence of $L \in \mathbb{N}$ triangular meshes $(\mathcal{M}_\ell)_{\ell=1}^L$ obtained by uniform regular refinement. Denoting the finite-element Galerkin solution by $u_\ell \in \mathcal{S}_{1,0}^0(\mathcal{M}_\ell)$ we monitor the following error quantities

(I) The error norms $\|u - u_\ell\|_{L^2(\Omega)}$,

(II) The error semi-norms $|u - u_\ell|_{H^1(\Omega)}$,

(III) the quantities

$$\mathcal{E}_\ell := \left(\sum_{e \in \mathcal{E}(\mathcal{M}_\ell), e \subset \partial\Omega} |e| (\mathbf{grad}\, u(\boldsymbol{m}_e) \cdot \boldsymbol{n}(\boldsymbol{m}_e) - (\mathsf{N}_\ell u_h)(\boldsymbol{m}_e))^2 \right)^{1/2}, \tag{3.23.14}$$

where N_ℓ is the Neumann data projection operator as defined in (3.23.10) based on the mesh $\mathcal{M}_\ell$.

ℓ	(A)	(B)	(C)
1	0.0500056	0.415862	0.663261
2	0.0141844	0.121605	0.351552
3	0.00369643	0.0440873	0.178733
4	0.000936174	0.0155509	0.0897867
5	0.000234942	0.00543724	0.0449509
6	5.88002e-05	0.00190468	0.0224832
7	1.47046e-05	0.00066951	0.0112426
8	3.67648e-06	0.000235938	0.00562145

Which error quantity is listed in which column of the table beside?

$\|u - u_\ell\|_{L^2(\Omega)}$ in column

$|u - u_\ell|_{H^1(\Omega)}$ in column

$\mathcal{E}_\ell$ in column

In addition, *describe qualitatively and quantitatively* the empiric asymptotic convergence of the quantities listed in the above table for $h_\ell \to 0$, where h_ℓ is the mesh width of $\mathcal{M}_\ell$.

HIDDEN HINT 1 for (3-23.g) → 3-23-7-0:ndrh6.pdf

SOLUTION for (3-23.g) → 3-23-7-1:ndrsp6s.pdf ▲

End Problem 3-23, 55 min.

Chapter 4

Beyond FEM: Alternative Discretizations

Problem 4-1: Mehrstellenverfahren for Poisson Equation

This problem deals with a special finite difference method for the Poisson equation [Lecture → (1.5.3.7)]. In line with the course's philosophy, we examine it from the perspective of a finite element method and derive the linear system of equation by means of local assembly.

You will need the techniques presented in [Lecture → Section 2.4.5] and [Lecture → Section 2.7.4.2]. Familiarity with [Lecture → Section 4.1] is an advantage. This problem involves coding in C++ using the EIGEN library.
▷ problem name/problem code folder: Mehrstellenverfahren

We consider the scalar second-order boundary value problem

$$\begin{aligned} -\Delta u &= f \quad \text{in} \quad \Omega :=]0,1[^2 \,, \\ u &= 0 \quad \text{on} \quad \partial\Omega \,, \end{aligned} \tag{4.1.1}$$

for a bounded and continuous function $f \in C^0(\overline{\Omega})$.

The so-called **Collatz Mehrstellenverfahren** [Col60], also known as compact 9-point scheme [Hac86, Sect. 4.6], can be viewed as a modified finite element Galerkin discretization using bilinear Lagrangian finite elements (→ [Lecture → Ex. 2.6.2.1]) on a uniform quadrilateral tensor-product mesh of Ω with meshwidth $h := (M+1)^{-1}$, $M \in \mathbb{N}$, see Fig. 90. We take for granted that standard nodal basis functions are used.

Then the Mehrstellenverfahren is obtained when using the following 4×4 element matrix on each cell K of the mesh

$$\mathbf{A}_K := \frac{1}{6}\begin{bmatrix} 5 & -2 & -1 & -2 \\ -2 & 5 & -2 & -1 \\ -1 & -2 & 5 & -2 \\ -2 & -1 & -2 & 5 \end{bmatrix}, \tag{4.1.2}$$

and the following element vector

$$\vec{\boldsymbol{\phi}}_K = \frac{h^2}{24}\begin{bmatrix} 4f_1 + f_2 + f_4 \\ 4f_2 + f_3 + f_1 \\ 4f_3 + f_4 + f_2 \\ 4f_4 + f_1 + f_3 \end{bmatrix}, \quad f_i := f(\boldsymbol{a}_i), \quad i = 1,2,3,4. \tag{4.1.3}$$

Here, we have denoted the vertices of the square cell K with length $h > 0$ of its sides by $\boldsymbol{a}_i$, $i = 1,2,3,4$, and have used the counter-clockwise local numbering indicated in Fig. 89.

Local numbering of vertices and associated local shape functions for a square cell K ▷

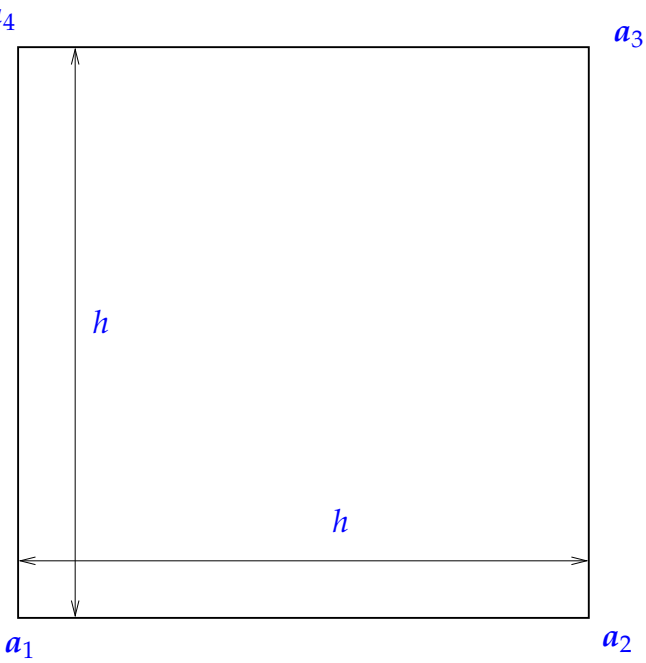


Fig. 89

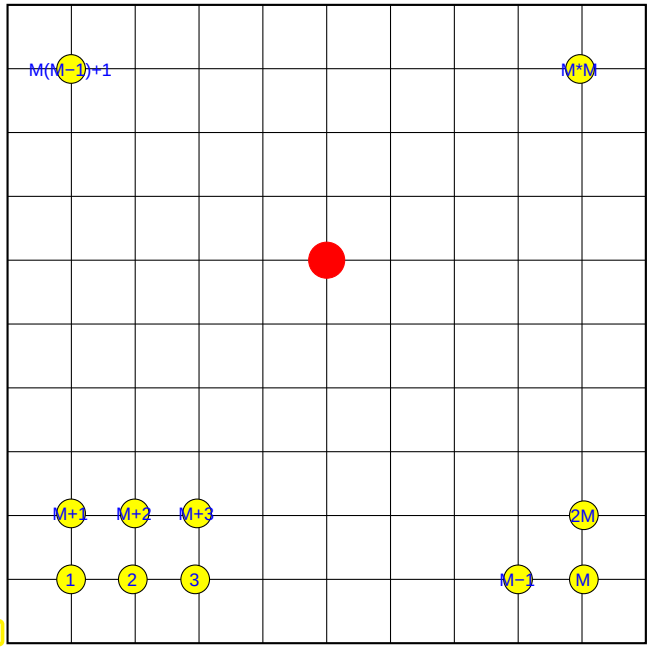


Fig. 90

Throughout this problem we assume a lexikographic numbering of the *interior* nodes of the mesh and of the associated basis functions, see Fig. 90 and [Lecture → § 4.1.2.1], [Lecture → Fig. 257]:

We number the interior nodes first from left to right and then from bottom to top, assigning them numbers $\in \{1,2,\ldots,M^2\}$. So the bottom row of interior nodes has the numbers $1,2,\ldots,M$, the top row $M(M-1)+1,\ldots,M^2$.

Eventually, we arrive at a linear system of equations $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\phi}}$ that can be obtained from (4.1.2) and (4.1.3) by the cell-oriented assembly scheme described in [Lecture → Section 2.7.4].

(4-1.a) ⚀ (10 min.) What is the size of the matrix $\mathbf{A}$ and of the right hand side vector $\vec{\boldsymbol{\phi}}$?

SOLUTION for (4-1.a) → 4-1-1-0:msts1.pdf ▲

(4-1.b) ⚁ (5 min.) What is the meaning of the entries of the solution vector $\vec{\boldsymbol{\mu}}$?

SOLUTION for (4-1.b) → 4-1-2-0:mst1sa.pdf ▲

(4-1.c) ⚂ (10 min.) What is the minimal and maximal number of non-zero entries in a row of $\mathbf{A}$?

SOLUTION for (4-1.c) → 4-1-3-0:msts2.pdf ▲

(4-1.d) ⚃ (30 min.) Single out an interior vertex of the mesh with index number k (maybe the vertex marked with • in Fig. 90). Write down the row of the linear system $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\phi}}$ that corresponds to this vertex. More precisely, specify the values and column indices of the non-zero matrix entries $(\mathbf{A})_{k,:}$, and the k-th component of the right hand side vector $\vec{\boldsymbol{\phi}}$, the latter in terms of certain point values of f.

HIDDEN HINT 1 for (4-1.d) → 4-1-4-0:mst3h1.pdf

SOLUTION for (4-1.d) → 4-1-4-1:msts3.pdf ▲

(4-1.e) ⚄ (10 min.) [depends on Sub-problem (4-1.d)]

Describe the matrix $\mathbf{A}$ in **stencil notation** as introduced in [Lecture → § 4.1.2.10].

SOLUTION for (4-1.e) → 4-1-5-0:.pdf ▲

(4-1.f) ⚄ Sketch the structure of the Galerkin matrix $\mathbf{A}$ analogous to how the structure of the Poisson matrix [Lecture → (4.1.2.8)] is depicted in [Lecture → Section 4.1].

HIDDEN HINT 1 for (4-1.f) → 4-1-6-0:msts4h1.pdf

SOLUTION for (4-1.f) → 4-1-6-1:msts4.pdf ▲

(4-1.g) ⚄ (30 min.) [depends on Sub-problem (4-1.c) and Sub-problem (4-1.d)]

In the file `mehrstellenverfahren.cc` implement an efficient C++ function

```
Eigen::SparseMatrix<double> compMehrstellenA(unsigned int M);
```

that assembles the *sparse* Galerkin matrix of the Mehrstellenverfahren on an $M \times M$ tensor-product mesh.

HIDDEN HINT 1 for (4-1.g) → 4-1-7-0:mst5h1.pdf

SOLUTION for (4-1.g) → 4-1-7-1:mst5s.pdf ▲

(4-1.h) ⚄ (20 min.) [depends on Sub-problem (4-1.d)]

In the file `mehrstellenverfahren.h` devise a C++ function

```
template <typename FUNCTOR>
Eigen::VectorXd compMehrstellenf(FUNCTOR &&f, unsigned int M);
```

that returns the right hand side vector $\vec{\phi}$ for the Mehrstellenverfahren. The argument `f` passes a functor object of type `std::function<double(Eigen::Vector2d)>` encoding the right-hand-side function f, while $M+1$ gives the number of grid cells in one direction.

SOLUTION for (4-1.h) → 4-1-8-0:msts6.pdf ▲

(4-1.i) ⚄ (20 min.) [depends on Sub-problem (4-1.g) and Sub-problem (4-1.h)]

In the file `mehrstellenverfahren.h` write a C++ function

```
template <typename FUNCTOR>
Eigen::VectorXd solveMehrstellen(FUNCTOR &&f, unsigned int M);
```

that computes the coefficient vector $\vec{\mu}$ for the Mehrstellen discretization of (4.1.1), when supplied with a functor object `f` to the source function f and the discretization parameter `M` (see Sub-problem (4-1.h)).

HIDDEN HINT 1 for (4-1.i) → 4-1-9-0:mst7h1.pdf

SOLUTION for (4-1.i) → 4-1-9-1:msts7.pdf ▲

(4-1.j) ⚄ (15 min.) [depends on Sub-problem (4-1.i)]

Now, for (4.1.1) consider the special data $f(x_1, x_2) := \sin(\pi x_1)\sin(\pi x_2)$, which yields the exact solution $u(x_1, x_2) = \frac{1}{2\pi^2}\sin(\pi x_1)\sin(\pi x_2)$.

Write a C++ function

```
double compgriderr(unsigned int M);
```

that computes the following "discrete norm" of the discretization error

$$\|u - u_M\|_{\mathcal{M},\infty} := \max\{|u(\boldsymbol{x}) - u_M(\boldsymbol{x})| : \boldsymbol{x} = (ih, jh),\ 1 \le i,j \le M,\ h := (M+1)^{-1}\}. \quad (4.1.10)$$

where u_M is the "Mehrstellen solution" for discretization parameter `M` defined at the vertices of the mesh.

SOLUTION for (4-1.j) → `4-1-10-0:msts8.pdf` ▲

(4-1.k) ⚂ (15 min.) [depends on Sub-problem (4-1.i) and Sub-problem (4-1.j)]

In the file `mehrstellenverfahren.cc` implement a C++ function

```
void tabulateMehrstellenError(void);
```

that tabulates the error norm (4.1.10) for the Mehrstellenverfahren with $M = 5, 10, 20, 40, 80, 160$ and for the specific source function $f(x,y) = \sin(\pi x)\sin(\pi y)$. Use the function `compgriderr()` from Sub-problem (4-1.j). Qualitatively and quantitatively describe the observed behavior of the error norm as a function of $h := \frac{1}{M+1}$.

SOLUTION for (4-1.k) → `4-1-11-0:msts9.pdf` ▲

References

[Col60] Lothar Collatz. *The numerical treatment of differential equations. 3d ed*. Die Grundlehren der mathematischen Wissenschaften, Band 60. Springer-Verlag, Berlin-Göttingen-Heidelberg, 1960, xv+568 pp. (1 plate) (cit. on p. 198).

[Hac86] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen*. Stuttgart: B.G. Teubner–Verlag, 1986 (cit. on p. 198).

End Problem 4-1 , 165 min.

Problem 4-2: Shifted Finite-Differences for Neumann BVP

[Lecture → Section 4.1.2] presented the standard finite-difference discretization of the Dirichlet problem for $-\Delta$ on a tensor-product domain. This was straightforward, because Dirichlet boundary data just provide the required solution values in mesh nodes on the boundary. For Neumann boundary conditions another finite-difference approximation of the normal component of the gradient in boundary points is required. The details will be elaborated in this problem.

The contents of [Lecture → Section 4.1.2] are required.
▷ problem name/problem code folder: ShiftedFiniteDifferences

We want to approximately solve the Neumann boundary value problem on the unit square $\Omega =]0,1[^2$

$$-\Delta u = f \quad \text{in} \quad \Omega \quad , \quad \mathbf{grad}\, u \cdot \boldsymbol{n} = g \quad \text{on} \quad \partial\Omega \, , \tag{4.2.1}$$

for given right-hand side source function $f \in C^0(\overline{\Omega})$ and Neumann data $g \in C^0(\partial\Omega)$. Here, $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ stands for the exterior unit vector field on $\partial\Omega$.

We employ the following finite-difference method: Fix $n \in \mathbb{N}$, set $h := \frac{1}{n}$, and let $\mathbb{L}_h := h\left(\mathbb{Z}^2 - \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix}\right)$ be a shifted lattice. Write $\Omega_h := \mathbb{L}_h \cap \Omega$ for the set of interior gridpoints, and

$$\Gamma_h := \left\{ \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} \in \partial\Omega : \, y_1/h - 1/2 \in \mathbb{Z} \quad \text{or} \quad y_2/h - 1/2 \in \mathbb{Z} \right\} . \tag{4.2.2}$$

for a set of gridpoints on the boundary.

Next, let $u_h : \mathbb{L}_h \to \mathbb{R}$ stand for a "generic grid function". The equations of the finite difference method are

$$\frac{4u_h(\boldsymbol{x}) - u_h(\boldsymbol{x} - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(\boldsymbol{x} + \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(\boldsymbol{x} - \begin{bmatrix} 0 \\ h \end{bmatrix}) - u_h(\boldsymbol{x} + \begin{bmatrix} 0 \\ h \end{bmatrix})}{h^2} = f(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Omega_h \, , \tag{4.2.3a}$$

$$\frac{u_h(\boldsymbol{x} + \boldsymbol{n}(\boldsymbol{x})\frac{h}{2}) - u_h(\boldsymbol{x} - \boldsymbol{n}(\boldsymbol{x})\frac{h}{2})}{h} = g(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Gamma_h \, . \tag{4.2.3b}$$

The unknowns of the finite-difference methods are all those values of u_h occuring in (4.2.3).

(4-2.a) ⚀ (20 min.) We consider the case $n = 8$. Mark the points in Ω_h with ● in the left figure, and the points of Γ_h with ✖ in the right figure (color is not important).

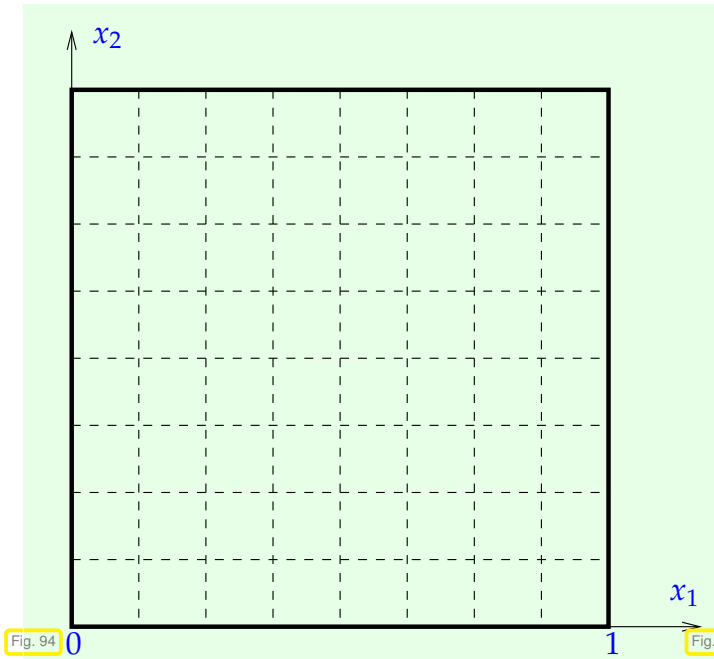


Fig. 94

Points of Ω_h

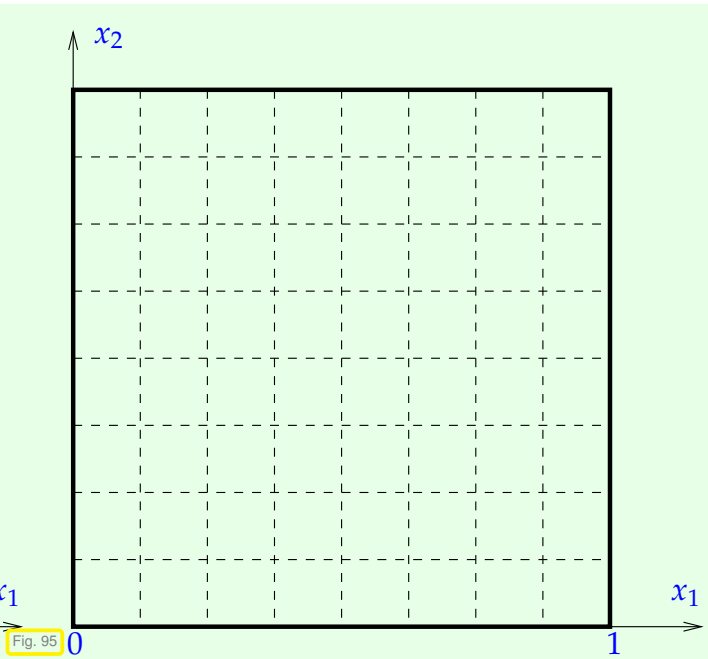


Fig. 95

Points of Γ_h

SOLUTION for (4-2.a) → 4-2-1-0:sfds1.pdf ▲

(4-2.b) ⚀ (15 min.) [depends on Sub-problem (4-2.a)]

What is the number of equations in (4.2.3), what is the number of unknowns in (4.2.3)?

No. of equations = , No. of unknowns = ,

SOLUTION for (4-2.b) → 4-2-2-0:sfds2.pdf ▲

(4-2.c) ⚁ (40 min.) From (4.2.3) eliminate all unknowns located at points outside Ω. This results in a linear system whose equations correspond to the points of Ω_h. State its equations in a fashion analogous to (4.2.3).

Distinguish equations associated with the point sets

$$\begin{aligned}
\widehat{\Omega}_h &:= \left\{ \boldsymbol{x} = \begin{bmatrix} (i-1/2)h \\ (j-1/2)h \end{bmatrix} : \; i,j \in \{2,\ldots,n-1\} \right\} , \\
\Omega_h^E &:= \{ \boldsymbol{x} \in \Omega_h : \; \boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix} \in \partial\Omega, \boldsymbol{x} \pm \begin{bmatrix} 0 \\ h \end{bmatrix} \in \Omega_h \} , \\
\Omega_h^N &:= \{ \boldsymbol{x} \in \Omega_h : \; \boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix} \in \partial\Omega, \boldsymbol{x} \pm \begin{bmatrix} h \\ 0 \end{bmatrix} \in \Omega_h \} , \\
\Omega_h^W &:= \{ \boldsymbol{x} \in \Omega_h : \; \boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix} \in \partial\Omega, \boldsymbol{x} \pm \begin{bmatrix} 0 \\ h \end{bmatrix} \in \Omega_h \} , \\
\Omega_h^S &:= \{ \boldsymbol{x} \in \Omega_h : \; \boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix} \in \partial\Omega, \boldsymbol{x} \pm \begin{bmatrix} h \\ 0 \end{bmatrix} \in \Omega_h \} .
\end{aligned} \tag{4.2.4}$$

$\forall \boldsymbol{x} \in \widehat{\Omega}_h$,

$\forall \boldsymbol{x} \in \widehat{\Omega}_h^E$,

$\forall \boldsymbol{x} \in \widehat{\Omega}_h^N$,

$\forall \boldsymbol{x} \in \widehat{\Omega}_h^W$,

$\forall \boldsymbol{x} \in \widehat{\Omega}_h^S$,

, $\boldsymbol{x} =$,

, $\boldsymbol{x} =$,

, $\boldsymbol{x} =$,

, $\boldsymbol{x} =$.

HIDDEN HINT 1 for (4-2.c) → 4-2-3-0:sfdh3.pdf

SOLUTION for (4-2.c) → 4-2-3-1:sfds3.pdf ▲

For the remainder of this problem we assume a lexikogaphic numbering of the points of Ω_h:

$$\left[\begin{array}{c}\frac{1}{2}\\ \frac{1}{2}\end{array}\right]h, \left[\begin{array}{c}\frac{3}{2}\\ \frac{1}{2}\end{array}\right]h, \left[\begin{array}{c}\frac{5}{2}\\ \frac{1}{2}\end{array}\right]h, \ldots, \left[\begin{array}{c}\frac{2n-1}{2}\\ \frac{1}{2}\end{array}\right]h, \left[\begin{array}{c}\frac{1}{2}\\ \frac{3}{2}\end{array}\right]h, \left[\begin{array}{c}\frac{3}{2}\\ \frac{3}{2}\end{array}\right]h, \ldots, \left[\begin{array}{c}\frac{2n-1}{2}\\ \frac{3}{2}\end{array}\right]h, \left[\begin{array}{c}\frac{1}{2}\\ \frac{5}{2}\end{array}\right]h, \ldots, \left[\begin{array}{c}\frac{1}{2}\\ \frac{2n-1}{2}\end{array}\right]h, \ldots, \left[\begin{array}{c}\frac{2n-1}{2}\\ \frac{2n-1}{2}\end{array}\right]h$$

(4-2.d) ⚄ (30 min.) [depends on Sub-problem (4-2.c)]

Write down the matrix $\mathbf{A}$ of the linear system of equations derived in Sub-problem (4-2.c)

HIDDEN HINT 1 for (4-2.d) → 4-2-4-0:sfdh4.pdf

SOLUTION for (4-2.d) → 4-2-4-1:.pdf ▲

(4-2.e) ⚄ (30 min.) [depends on Sub-problem (4-2.d)]

What sufficient and necessary condition do the data f and g have to satisfy to ensure

(I) uniqueness of solutions of the square linear system of equations derived in Sub-problem (4-2.c),

(II) or existence of solutions for that linear system?

SOLUTION for (4-2.e) → 4-2-5-0:sfds5.pdf ▲

End Problem 4-2 , 135 min.

Chapter 5

Non-Linear Elliptic Boundary Value Problems

Problem 5-1: A curve with tension

This problem studies a particular minimization problem on a function space, which can be regarded as a space of curves. In this task we practice the derivation of variational equations and of the associated two-point boundary value problems.

This problem heavily relies on the contents of [Lecture → Section 5.2].
▷ problem name/problem code folder: CurveWithTension

A **planar curve** can be described by a parameterization,

$$\mathbf{u} : [0,1] \to \mathbb{R}^2 , \quad \mathbf{u} \in (C^1_{\mathrm{pw}}([0,1]))^2 , \tag{5.1.1}$$

of which we assume a nowhere vanishing derivative

$$\mathbf{u}'(\xi) \neq \mathbf{0} \quad \forall \xi \in [0,1] . \tag{5.1.2}$$

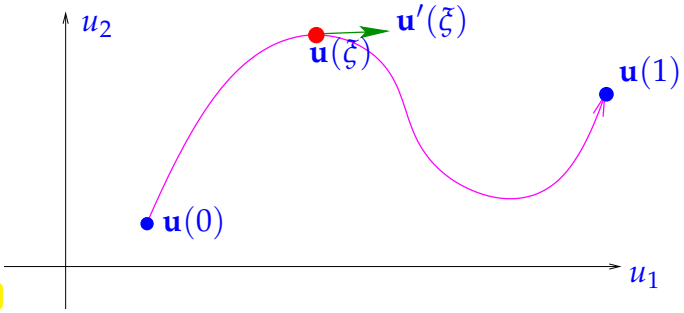


Fig. 101

Then the point set representing the curve is

$$\mathcal{C} := \left\{ \boldsymbol{x} \in \mathbb{R}^2 : \exists \xi \in [0,1] : \boldsymbol{x} = \boldsymbol{u}(\xi) \right\} . \tag{5.1.3}$$

Given a curve $\mathbf{u} \in V := (C^1_{\mathrm{pw}}([0,1]))^2$, we suppose that its tension energy is proportional to the square of its length:

$$J_{\mathrm{tens}}(\mathbf{u}) := \left(\int_0^1 \|\mathbf{u}'(\xi)\| \, \mathrm{d}\xi \right)^2 . \tag{5.1.4}$$

On the curve acts as an external force field $\mathbf{f} = \mathbf{f}(\xi)$, $\mathbf{f} \in (C^0([0,1]))^2$, which results in a potential energy contribution according to

$$J_{\mathrm{f}}(\mathbf{u}) := -\int_0^1 \mathbf{f}(\xi) \cdot \mathbf{u}(\xi) \, \mathrm{d}\xi . \tag{5.1.5}$$

The curve seeks to attain a shape such that its total energy $J(\mathbf{u}) = J_{\mathrm{tens}}(\mathbf{u}) + J_{\mathrm{f}}(\mathbf{u})$ becomes minimal. In addition, it is pinned at its ends, that is, we have to deal with the constraints

$$\mathbf{u}(0) = \mathbf{u}_0 \in \mathbb{R}^2, \qquad \mathbf{u}(1) = \mathbf{u}_1 \in \mathbb{R}^2 , \quad \mathbf{u}_0, \mathbf{u}_1 \text{ given} . \tag{5.1.6}$$

We follow the approach of [Lecture → Section 5.2.1] to determine which variational problem needs to be solved to compute the shape of the curve.

(5-1.a) ⚀ (10 min.) Determine the **test space** V_0, i.e. the functional space of all admissible variations $\mathbf{v}$.

HIDDEN HINT 1 for (5-1.a) → 5-1-1-0:cwth1.pdf

SOLUTION for (5-1.a) → 5-1-1-1:cwts1.pdf ▲

(5-1.b) ⚁ (10 min.) Following [Lecture → Section 5.2.1], compute the **directional derivative** of $J_{\mathrm{f}} = J_{\mathrm{f}}(\mathbf{u})$ in direction $\mathbf{v} \in V_0$ at a generic curve $\mathbf{u}$, that is, compute the limit of the difference quotient

$$\lim_{t\to 0} \frac{J_f(\mathbf{u}+t\mathbf{v}) - J_{\mathrm{f}}(\mathbf{u})}{t} \quad \text{for} \quad \mathbf{v} \in V_0 \,.$$

HIDDEN HINT 1 for (5-1.b) → 5-1-2-0:cwth2.pdf

SOLUTION for (5-1.b) → 5-1-2-1:cwts2.pdf ▲

(5-1.c) ⚂ (30 min.) [depends on Sub-problem (5-1.a)]

Compute the directional derivative of $J_{\mathrm{tens}} = J_{\mathrm{tens}}(\mathbf{u})$ in direction $\mathbf{v} \in V_0$ at a generic curve shape $\mathbf{u}$, that is, compute

$$\lim_{t\to 0} \frac{J_{\mathrm{tens}}(\mathbf{u}+t\mathbf{v}) - J_{\mathrm{tens}}(\mathbf{u})}{t} \,, \quad \mathbf{v} \in V_0 \,.$$

HIDDEN HINT 1 for (5-1.c) → 5-1-3-0:wt3h3.pdf

SOLUTION for (5-1.c) → 5-1-3-1:cwts3.pdf ▲

(5-1.d) ⚂ (15 min.) [depends on Sub-problem (5-1.a), Sub-problem (5-1.b), and Sub-problem (5-1.c)]

State the **variational equation** satisfied by any minimal-energy curve, that is, by any function $\mathbf{u}_*$ with

$$\mathbf{u}_* \in \operatorname*{argmin}_{\mathbf{v}\in\widehat{V}} J(\mathbf{v}) \,,$$

where $\widehat{V}$ is the space of curves compliant with the pinning conditions.

HIDDEN HINT 1 for (5-1.d) → 5-1-4-0:cwth4.pdf

SOLUTION for (5-1.d) → 5-1-4-1:cwts4.pdf ▲

(5-1.e) ⚂ Assuming that $\mathbf{u}_*$ is sufficiently smooth, find a **two-point boundary value problem**, whose solution provides the shape of the curve.

SOLUTION for (5-1.e) → 5-1-5-0:cwt5s.pdf ▲

End Problem 5-1 , 65 min.

Problem 5-2: The Brachistochrone Problem

Under the influence of gravity a ball is rolling on a track whose endpoints are fixed. Friction is neglected. The brachistochrone problem asks for the shape of the track, which minimizes the travel time of the ball, when released at the higher endpoint. We derive a variational equation and a two-point boundary value problem, whose solutions provide the desired shape. Then we solve the non-linear variational problem by means of a finite-element Galerkin discretization.

The problem broadly addresses the concepts and techniques treated throughout [Lecture → Chapter 5]. It retraces all the essential considerations employed in elastic string modeling in class for a different problem from classical mechanics. The purpose of this problem is to practice all the techniques introduced in [Lecture → Section 5.1.1.2], [Lecture → Section 5.1.1.3], and [Lecture → Section 5.2.1]. It involves some calculus drill, which will not do you any harm, and also asks for implementation in C++ based on EIGEN.
▷ problem name/problem code folder: Brachistochrone

The Brachistochrone Problem is a classical problem of variational calculus, already tackled by Newton and Bernoulli in the 17th century: Given two points $\boldsymbol{a} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix}, \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \in \mathbb{R}^2$, such that $a_2 > b_2$, we are looking for a curve $\mathbf{u} = [u_1, u_2]^\top : [0,1] \to \mathbb{R}^2$ connecting $\boldsymbol{a}$ and $\boldsymbol{b}$ ($\mathbf{u}(0) = \boldsymbol{a}$, $\mathbf{u}(1) = \boldsymbol{b}$) so that a ball rolling down the curve $\mathbf{u}$ reaches $\mathbf{b}$ in minimal time.

Different curves a ball may roll down. On which does it reach the bottom in shortest time? ▷

Note: You can experience brachistochrone curves at Technorama Winterthur, link.

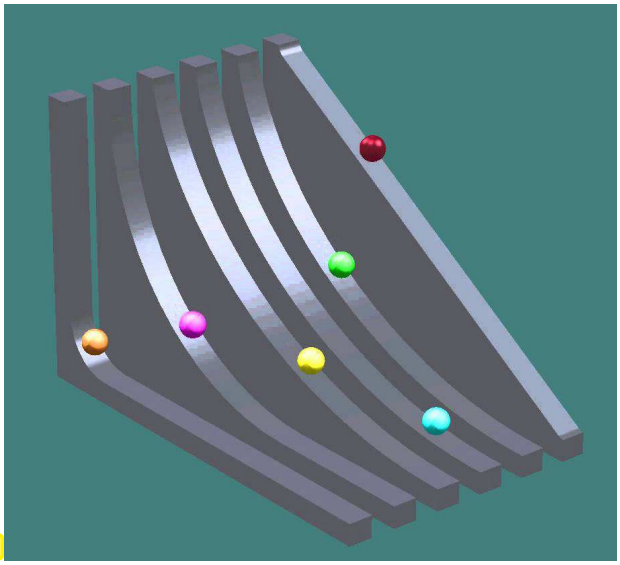


Fig. 102

Following our approach to the modeling of an elastic string in [Lecture → Section 5.1.1.2], we first consider an approximate **discrete model** in order to arrive at a continuous minimization problem describing the Brachistochrone Problem by a limit process.

To begin with, as in [Lecture → Section 5.1.1.2] we choose a Cartesian coordinate system whose vertical (x_2) axis is aligned with the direction of the gravity force.

Assumption 5.2.1.

We assume $0 > a_2 > b_2$ for the remainder of this problem.

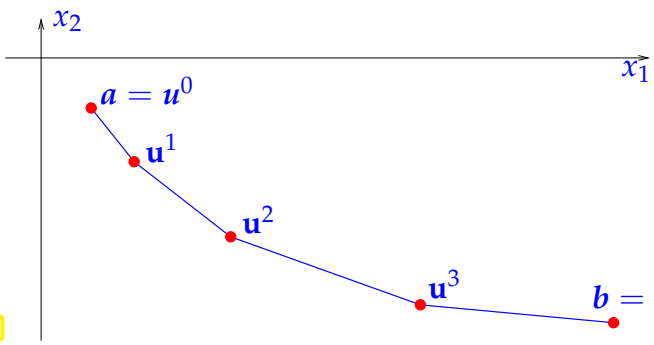


Fig. 103

The discrete model approximates the Brachistochrone curve by a polygon; let $\mathbf{u}^i \in \mathbb{R}^2$ for $i = 0, \ldots, N+1$, be points (knots) along the curve, with fixed and given endpoints $\mathbf{u}^0 = \boldsymbol{a}$ and $\mathbf{u}^{N+1} = \boldsymbol{b}$, so that there are N "free" points, which means that the configuration space is $(\mathbb{R}^2)^N \cong \mathbb{R}^{2N}$.

◁ $N = 3$, curve consisting of $N+1 = 4$ straight segments.

To find the speed $v(\mathbf{u})$ of the ball at a point $\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \in \mathbb{R}^2$, we assume that the ball is dropped onto the curved track from $\begin{bmatrix} a_1 \\ 0 \end{bmatrix}$ and then starts rolling down the curve without friction and air drag. Then we appeal to the conservation of total (kinetic and potential) energy. Thus, we have for the ball's speed in position $\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \in \mathbb{R}^2$

$$\text{[kinetic energy]} \quad v(\mathbf{u})^2 = -u_2 \quad \text{[potential energy]} \quad \Longrightarrow \quad v(\mathbf{u}) = \sqrt{-u_2}\,. \tag{5.2.2}$$

Note that we take for granted a suitable scaling that leads to $g = 1$ for earth's gravitational acceleration. The model is assumed to be non-dimensional already.

In order to ensure that the ball is always moving we demand that $(\mathbf{u}^i)_2 < 0$ for all $i = 1, \ldots, M$. We also make a simplifying assumption.

Assumption 5.2.3. Ball speed in discrete model

The speed of the ball on every straight segment $[\mathbf{u}^i, \mathbf{u}^{i+1}]$, $i = 0, \ldots, N$, is constant and equal to $v(\frac{1}{2}(\mathbf{u}^i + \mathbf{u}^{i+1}))$.

(5-2.a) ⚂ (15 min.) We consider the discrete model described above. Under Ass. 5.2.3 derive an expression $T_N(\mathbf{u}^1, \ldots, \mathbf{u}^N)$ for the time it takes the ball to roll from $\boldsymbol{a}$ to $\boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$, $b_2 < a_2 \leq 0$, on the polygon given by the $N+2$ knot positions $\boldsymbol{a}, \boldsymbol{u}^1, \ldots, \boldsymbol{u}^N, \boldsymbol{b}$. The endpoint positions $\boldsymbol{a}, \boldsymbol{b}$ can be considered as given fixed parameters.

SOLUTION for (5-2.a) → 5-2-1-0:bcs1.pdf ▲

(5-2.b) ⚄ (30 min.) [depends on Sub-problem (5-2.a)]

Write $\vec{\mathbf{u}} = [u_1^1, u_2^1, u_1^2, u_2^2, \ldots, u_1^N, u_2^N]^\top$ for elements of the configuration space $\mathbb{R}^{2N} = \mathbb{R}^2 \times \cdots \times \mathbb{R}^2$. Determine a non-linear system of equations $\boldsymbol{F}(\vec{\mathbf{u}}) = \mathbf{0}$ satisfied by every global minimizer $\vec{\mathbf{u}}^*$ of the total travel time $T_N = T_N(\vec{\mathbf{u}})$:

$$\vec{\mathbf{u}}^* \in \operatorname*{argmin}_{\vec{\mathbf{v}} \in \mathbb{R}^{2N}} T_N(\vec{\mathbf{v}}) \quad \Rightarrow \quad \boldsymbol{F}(\vec{\mathbf{u}}^*) = \mathbf{0}\,. \tag{5.2.7}$$

SOLUTION for (5-2.b) → 5-2-2-0:bcs2.pdf ▲

(5-2.c) ⚀ (15 min.) Now we tackle the continuum limit $N \to \infty$ for the polygon model. Please study again [Lecture → Section 5.1.1.3], where the corresponding considerations are pursued for the mass-spring model of an elastic string. ▲

(5-2.d) ⚅ (45 min.) [depends on Sub-problem (5-2.a)]

Following [Lecture → § 5.1.1.20] we assume that the knots $\mathbf{u}^i$ lie on a C^2-smooth curve $\mathbf{u} : [0,1] \to \mathbb{R}^2$, i.e. that

$$\mathbf{u}^i = \mathbf{u}(\xi^i)\,, \quad \xi^i := \frac{i}{N+1}\,, \quad i = 1, \ldots, N\,. \tag{5.2.10}$$

Of course, we demand that $\mathbf{u}^0 := \mathbf{u}(0) = \mathbf{0}$, $\mathbf{u}^{N+1} := \mathbf{u}(1) = \boldsymbol{b}$, $(\mathbf{u}(\xi))_2 < 0$ for all $\xi \in]0,1[$.

Compute the limit travel time

$$T(\mathbf{u}) := \lim_{N \to \infty} T_N(\mathbf{u}(\xi^1), \ldots, \mathbf{u}(\xi^N))\,.$$

✎ Notation: Throughout this problem use a $'$ to tag derivatives with respect to the curve parameter ξ, e.g., $\mathbf{u}'(\xi) := \frac{d\mathbf{u}}{d\xi}(\xi)$.

HIDDEN HINT 1 for (5-2.d) → 5-2-4-0:bcs4h1.pdf

SOLUTION for (5-2.d) → 5-2-4-1:bcs4.pdf ▲

(5-2.e) ⚄ (30 min.) [depends on Sub-problem (5-2.d)]

Analogous to [Lecture → Section 5.2.1] derive the variational problem arising from the minimization of the limit total travel $T(\mathbf{u})$ from (**??**) over the *affine space* of curves

$$\widehat{V} := \left\{ \mathbf{v} \in (C^1_{\mathrm{pw}}([0,1]))^2 : \mathbf{v}(0) = \boldsymbol{a},\ \mathbf{v}(1) = \boldsymbol{b} \right\}. \tag{5.2.15}$$

HIDDEN HINT 1 for (5-2.e) → 5-2-5-0:bcs5h1.pdf

SOLUTION for (5-2.e) → 5-2-5-1:bcs5.pdf ▲

(5-2.f) ⚂ (15 min.) [depends on Sub-problem (5-2.e)]

Using integration by parts as in [Lecture → Ex. 5.2.3.1] derive the ordinary differential equation (Euler-Lagrange equation) spawned by the variational problem obtained in Sub-problem (5-2.e).

SOLUTION for (5-2.f) → 5-2-6-0:bcs6.pdf ▲

(5-2.g) ⚂ (30 min.) [depends on Sub-problem (5-2.f)]

Show that the **cycloid curve**

$$\mathbf{u}(\xi) = \begin{bmatrix} \pi\xi - \sin(\pi\xi) \\ \cos(\pi\xi) - 1 \end{bmatrix}, \quad 0 \le \xi \le 1, \tag{5.2.22}$$

satisfies the differential equation found in sub-problem Sub-problem (5-2.f).

Cycloid curve connecting $\mathbf{0}$ and $\boldsymbol{b} := \left[\begin{smallmatrix} \pi \\ -2 \end{smallmatrix}\right]$. ▷

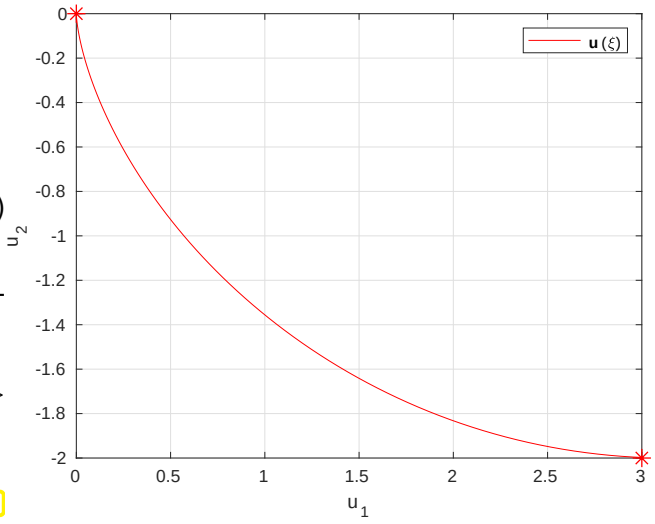


Fig. 104

SOLUTION for (5-2.g) → 5-2-7-0:bcs7.pdf ▲

(5-2.h) ⚂ We can recast the variational equation found in Sub-problem (5-2.e) in the form

$$\mathbf{u} \in \widehat{V}: \quad \int_0^1 \sigma(\mathbf{u}(\xi), \mathbf{u}'(\xi))\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)\,\mathrm{d}\xi = \int_0^1 \mathbf{f}(\mathbf{u}(\xi), \mathbf{u}'(\xi)) \cdot \mathbf{v}(\xi)\,\mathrm{d}\xi \quad \forall \mathbf{v} \in V_0 \tag{5.2.24}$$

with a suitable "$\mathbf{u}$-dependent scalar coefficient function" $\sigma : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$, and a "$\mathbf{u}$-dependent source function" $\mathbf{f} : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$. Give explicit formulas for σ and $\mathbf{f}$.

SOLUTION for (5-2.h) → 5-2-8-0:bras7a.pdf ▲

(5-2.i) ⚂ For the variational problem derived in Sub-problem (5-2.e) and based on (5.2.24) give the recursion for a suitable **fixed-point iteration in function space** in the spirit of [Lecture → Section 5.3.2.1].

SOLUTION for (5-2.i) → 5-2-9-0:brs8s.pdf ▲

We apply a Galerkin finite element Galerkin discretization to the linear variational problem obtained in Sub-problem (5-2.i) by the fixed-point method in function space. Concretely, we opt for piecewise linear

Lagrangian finite elements on an equidistant mesh of $[0,1]$ with $M \in \mathbb{N}$ cells, *cf.* [Lecture → § 2.3.1.3]:

$$\mathcal{M} := \{]x_{j-1}, x_j[:\ j \in \{1,\ldots,M\}\}\,, \quad x_j := \frac{j}{M}\,, \quad j = 0,\ldots,M\,. \tag{5.2.27}$$

This means the discrete variational problem is posed on the

- discrete trial space $\widehat{V}_h := \left\{ v_h \in (\mathcal{S}_1^0(\mathcal{M}))^2 :\ \mathbf{v}_h(0) = \boldsymbol{a},\ \mathbf{v}_h(1) = \boldsymbol{b} \right\}$,
- discrete test space $V_{0,h} := \left\{ \mathbf{v}_h \in (\mathcal{S}_1^0(\mathcal{M}))^2 :\ \mathbf{v}_h(0) = \mathbf{v}_h(1) = \mathbf{0} \right\} = (\mathcal{S}_{1,0}^0(\mathcal{M}))^2$.

In the context of the discrete variational problem, for the computation of any integral over $[0,1]$ we rely on the composite 2-point Gauss rule [Lecture → (2.3.3.13)] with respect to $\mathcal{M}$:

$$\int_0^1 \psi(\xi)\,\mathrm{d}\xi \approx \tfrac{1}{2}h \sum_{j=1}^{M} \psi(x_{j-1/2}^-) + \psi(x_{j-1/2}^+)\,, \quad \begin{aligned} x_{j-1/2}^- &:= h(j - \tfrac{1}{2} - \tfrac{1}{2\sqrt{3}})\,, \\ x_{j-1/2}^+ &:= h(j - \tfrac{1}{2} + \tfrac{1}{2\sqrt{3}})\,, \end{aligned} \quad h := \frac{1}{M}\,. \tag{5.2.28}$$

Throughout we use the tent-function basis $\{b_h^0, b_h^1, \ldots, b_h^M\}$ of $\mathcal{S}_1^0(\mathcal{M})$ introduced in [Lecture → Section 2.3.2] and number the basis functions consecutively from left to right as in [Lecture → (2.3.2.1)], that is, b_h^ℓ is associated with the mesh node x_ℓ. Then an ordered basis for the *vector-valued* finite-element space $V_{0,h}$ is

$$\mathfrak{B}_h = \mathfrak{B}_h^1 \cup \mathfrak{B}_h^2 = \left\{ \begin{bmatrix} b_h^1 \\ 0 \end{bmatrix}, \begin{bmatrix} b_h^2 \\ 0 \end{bmatrix}, \ldots, \begin{bmatrix} b_h^{M-1} \\ 0 \end{bmatrix} \right\} \cup \left\{ \begin{bmatrix} 0 \\ b_h^1 \end{bmatrix}, \begin{bmatrix} 0 \\ b_h^2 \end{bmatrix}, \ldots, \begin{bmatrix} 0 \\ b_h^{M-1} \end{bmatrix} \right\}. \tag{5.2.29}$$

Moreover, the essential Dirichlet-type boundary conditions are taken into account by he $\mathcal{M}$-piecewise linear minimally supported **offset function** from [Lecture → Rem. 2.3.3.15]:

$$\mathbf{u}_0(\xi) = \boldsymbol{a}\, b_h^0(\xi) + \boldsymbol{b}\, b_h^M(\xi)\,, \quad \xi \in [0,1]\,. \quad \Longrightarrow \quad \mathbf{u}_0 \in \mathcal{S}_1^0(\mathcal{M})\,. \tag{5.2.30}$$

Remark. The following indexing convention is adopted throughout this problem.

- Nodes of the mesh $\mathcal{M}$ and all node-associated entities are numbered with integer indices $\in \{0,\ldots,M\}$.
- Cells and all cell-related entities are numbered with integer$+\frac{1}{2}$ indices $^{2j-1}/_2$, $j \in \{1,\ldots,M\}$.

So cell $^{2j-1}/_2$ is $]x_{j-1}, x_j[$, and the cells $^{2j-1}/_2$ and $^{2j+1}/_2$ abut node x_j, $j = 1,\ldots,M-1$.

(5-2.j) ⚄ In the file `brachistochrone.cc` implement an EIGEN-based C++ function

```
Eigen::VectorXd coeff_sigma(
const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots);
```

that returns the vector

$$\vec{\sigma} := \left[\, \sigma_{j-1/2} \right]_{j=1}^{M} \in \mathbb{R}^M\,,$$

$$\sigma_{j-1/2} := \tfrac{1}{2}\Big(\sigma(\mathbf{u}_h(x_{j-1/2}^-), \mathbf{u}_h'(x_{j-1/2}^-)) + \sigma(\mathbf{u}_h(x_{j-1/2}^+), \mathbf{u}_h'(x_{j-1/2}^+)) \Big)\,, \quad j = 1,\ldots,M\,, \tag{5.2.31}$$

$x_{j-1/2}^\pm$ defined in (5.2.28), where the polygonal curve $\mathbf{u}_h \in (\mathcal{S}_1^0(\mathcal{M}))^2$ is defined through the argument `knots`, which passes the $2 \times (M+1)$ matrix

$$\texttt{knots} \quad \leftrightarrow \quad [\mathbf{u}_h(0), \mathbf{u}_h(^1/_M), \mathbf{u}_h(^2/_M), \ldots, \mathbf{u}_h(^{M-1}/_M), \mathbf{u}_h(1)] \in \mathbb{R}^{2,M+1}\,.$$

Here, σ is the coefficient function from Sub-problem (5-2.h).

SOLUTION for (5-2.j) → 5-2-10-0:brs8a.pdf ▲

(5-2.k) ⚂ Write a C++ function

```
Eigen::Matrix<double, Eigen::Dynamic, 2> sourcefn2(
const Eigen::Matrix<double,2,Eigen::Dynamic> &knots);
```

that, for the $\mathbf{u}$-dependent source function $\mathbf{f}$ found in Sub-problem (5-2.h), computes the $M \times 2$-matrix

$$\left[\left(\mathbf{f}(\mathbf{u}_h(x_{j-1/2}^-), \mathbf{u}_h'(x_{j-1/2}^-))\right)_2 \quad \left(\mathbf{f}(\mathbf{u}_h(x_{j-1/2}^+), \mathbf{u}_h'(x_{j-1/2}^+))\right)_2\right]_{j=1}^{M} \in \mathbb{R}^M ,$$

with $\mathbf{u}_h \in \left(\mathcal{S}_1^0(\mathcal{M})\right)^2$ standing for the finite element function defined through the `knots` matrix argument as in Sub-problem (5-2.j).

SOLUTION for (5-2.k) → 5-2-11-0:bras8b.pdf ▲

(5-2.l) ⚄ (60 min.) [depends on Sub-problem (5-2.i)]

After Galerkin discretization the fixed-point method in function space of Sub-problem (5-2.i) produces a sequence $\left(\mathbf{u}_h^{(k)} = \mathbf{u}_0 + \mathbf{w}_h^{(k)}\right)_{k\in\mathbb{N}_0}$, $\mathbf{w}_h^{(k)} \in V_{0,h}$, of iterates $\in \widehat{V}_h$. Let $\vec{\boldsymbol{\mu}}^{(k)}$ be the basis expansion coefficient vectors (with respect to $\mathfrak{B}_h$) of $\mathbf{w}_h^{(k)}$. It turns out that those can be computed by solving the following $(2M-2) \times (2M-2)$ linear system of equations

$$\begin{bmatrix} \mathbf{R}(\mathbf{u}_h^{(k)}) & \mathbf{O} \\ \mathbf{O} & \mathbf{R}(\mathbf{u}_h^{(k)}) \end{bmatrix} \vec{\boldsymbol{\mu}}^{(k+1)} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}_1(\mathbf{u}_h^{(k)}) \\ \vec{\boldsymbol{\varphi}}_2(\mathbf{u}_h^{(k)}) \end{bmatrix} , \tag{5.2.37}$$

with $\mathbf{R}(\mathbf{u}_h^{(k)}) \in \mathbb{R}^{M-1,M-1}$ and $\vec{\boldsymbol{\varphi}}_i(\mathbf{u}_h^{(k)}) \in \mathbb{R}^{M-1}$.

Give explicit formulas for $\mathbf{R}(\mathbf{u}_h)$, and $\vec{\boldsymbol{\varphi}}_i(\mathbf{u}_h)$, $\mathbf{u}_h \in \left(\mathcal{S}_1^0(\mathcal{M})\right)^2$.

HIDDEN HINT 1 for (5-2.l) → 5-2-12-0:braofsh1.pdf

SOLUTION for (5-2.l) → 5-2-12-1:bras9s.pdf ▲

(5-2.m) ⚂ (30 min.) [depends on Sub-problem (5-2.l),Sub-problem (5-2.j)]

In the file `brachistochrone.cc` implement the C++ function

```
Eigen::SparseMatrix<double> matR(
const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots);
```

which takes a function $\mathbf{u}_h \in \left(\mathcal{S}_1^0(\mathcal{M})\right)^2$ defined through its values $\mathbf{u}_h(x_i) \in \mathbb{R}^2$, $i = 0,\ldots,M$, in the nodes of $\mathcal{M}$, and returns the tridiagonal matrix $\mathbf{R}(\mathbf{u}_h)$. The matrix argument `knots` contains $\mathbf{u}_h(x_i) \in \mathbb{R}^2$ in its columns.

HIDDEN HINT 1 for (5-2.m) → 5-2-13-0:braAh1.pdf

HIDDEN HINT 2 for (5-2.m) → 5-2-13-1:braAh2.pdf

SOLUTION for (5-2.m) → 5-2-13-2:sbraA.pdf ▲

(5-2.n) ⚂ (30 min.) [depends on Sub-problem (5-2.l) and Sub-problem (5-2.k)]

In the file `brachistochrone.cc` complete the code of the C++ function

```
Eigen::VectorXd compute_rhs(
const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots,
Eigen::Vector2d a, Eigen::Vector2d b);
```

This function is supposed to compute the right-hand side vector $[\vec{\varphi}_1(\mathbf{u}_h), \vec{\varphi}_1(\mathbf{u}_h)]^\top \in \mathbb{R}^{2M-2}$ for the linear system of equations (5.2.37). The argument `knots` passes $\mathbf{u}_h \in \left(\mathcal{S}_1^0(\mathcal{M})\right)^2$ as in Sub-problem (5-2.m), the parameters `a` and `b` give the coordinates of the endpoints $\boldsymbol{a}$ and $\boldsymbol{b}$ of the curve.

HIDDEN HINT 1 for (5-2.n) → 5-2-14-0:braBh1.pdf

SOLUTION for (5-2.n) → 5-2-14-1:.pdf ▲

(5-2.o) ⚀ (30 min.) [depends on Sub-problem (5-2.m) and Sub-problem (5-2.n)]

In the file `brachistochrone.h` write a C++ function

```
template <typename RECORDER = std::function<
void(const Eigen::Matrix<double, 2, Eigen::Dynamic> &)>>
Eigen::Matrix<double, 2, Eigen::Dynamic> brachistochrone(
unsigned int M, Eigen::Vector2d a, Eigen::Vector2d b, double atol,
double rtol, unsigned int itmax,
RECORDER &&rec = [](const Eigen::Matrix<double, 2, Eigen::Dynamic>
    &)
-> void { return; });
```

that carries out the discrete fixed-point iteration of Sub-problem (5-2.l) with straight-line initial guess

$$\mathbf{u}_h^{(0)} = \{\xi \in [0,1] \mapsto (1.0 - xi)\boldsymbol{a} + \xi\,\boldsymbol{b}\} \in (\mathcal{S}_1^0(\mathcal{M}))^2 \,. \tag{5.2.50}$$

The function arguments have the following meaning:

1. `M` is the number of cells of the equidistant mesh $\mathcal{M}$, see (5.2.27).
2. `b` passes the coordinates of the endpoint $\boldsymbol{b}$ of the curve,
3. `atol`, `rtol` are the control parameters for correction-based termination of the iteration, see [NumCSE course → § 8.2.3.4]. Rely on the discrete $L^2(]0,1[)$-norm
$$\|\mathbf{v}_h\|_{0,h}^2 := M^{-1}\left(\tfrac{1}{2}\|\mathbf{v}_h(0)\|^2 + \sum_{j=0}^{M-1}\|\mathbf{v}_h(j/M)\|^2 + \tfrac{1}{2}\|\mathbf{v}_h(1)\|^2\right). \tag{5.2.51}$$
4. `itmax` is the maximal number of iterations,
5. `rec` is a functor object that can be given the $2\times(M+1)$-matrix
$$\left[\mathbf{u}_h^{(k)}(0), \mathbf{u}_h^{(k)}(1/M), \mathbf{u}_h^{(k)}(2/M), \ldots, \mathbf{u}_h^{(k)}(M-1/M), \mathbf{u}_h^{(k)}(1)\right] \in \mathbb{R}^{2,M+1} \tag{5.2.52}$$
in every step of the fixed-point iteration. This makes possible the optional storing of all intermediate iterates or simply the counting of number of iteration steps.

If the fixed-point iteration has converged, the function is expected to return an approximation of the finite-element Galerkin solution of the brachistochrone non-linear variational problem (**??**) in the format (5.2.50).

SOLUTION for (5-2.o) → 5-2-15-0:braCs.pdf ▲

(5-2.p) ⚁ (30 min.) [depends on Sub-problem (5-2.p)]

For $\boldsymbol{b} = \left[\begin{smallmatrix}\pi\\-2\end{smallmatrix}\right]$ and $M = 128$, investigate the convergence of the fixed-point iteration implemented in `brachistochrone()` by means of a C++ function (to be completed)

```
void tabiterr(
unsigned int M = 128, double rtol = 1.0E-4, double atol = 1.0E-6,
unsigned int maxit = 100);
```

that tabulates $\left\| \mathbf{u}_h^{(k)} - \mathbf{u}_h^{(100)} \right\|_{0,h}$ versus k for $k = 1, \ldots, 20$. The meaning of the arguments can be inferred from the variable names.

Describe qualitatively and quantitatively the observed convergence in the sense of [NumCSE course → Section 8.2.2].

SOLUTION for (5-2.p) → 5-2-16-0:braDs.pdf ▲

(5-2.q) ⚄ (30 min.) [depends on Sub-problem (5-2.o)]

Consider the setting $\boldsymbol{a} = \begin{bmatrix} \frac{1}{2}\pi - 1 \\ -1 \end{bmatrix}$, $\boldsymbol{b} = \begin{bmatrix} \pi \\ -2 \end{bmatrix}$, for which $\xi \mapsto \mathbf{u}(\frac{1}{2}(\xi + 1))$ with $\mathbf{u}$ as in (5.2.22) provides an exact solution.

Write a diagnostic C++ function

```
void brachistochrone_cvg(void);
```

that computes and tabulates $\|\mathbf{u}_h - \mathsf{I}_M \mathbf{u}\|_{0,h}$ as function of $M = 8, 26, 32, 64, 128, 258, 512, 1024$. Here $\mathbf{u}_h \in (\mathcal{S}_1^0(\mathcal{M}))^2$ is the Galerkin finite-element solution on an equidistant mesh $\mathcal{M}$ with M cells, which can be obtained up to high accuracy by running `brachistochrone()` with tight tolerances. $\mathsf{I}_M \mathbf{u} \in (\mathcal{S}_1^0(\mathcal{M}))^2$ is the piecewise linear interpolant of the exact solution on $\mathcal{M}$.

Esimate the rate of asymptotic algebraic convegence $\|\mathbf{u}_h - \mathsf{I}_M \mathbf{u}\|_{0,h} \to 0$ for $M \to \infty$ (in terms of the mesh width $h := M^{-1}$):

$$\|\mathbf{u}_h - \mathsf{I}_M \mathbf{u}\|_{0,h} = O\left(h^{\square} \right) \quad \text{for} \quad h \to 0 \,.$$

HIDDEN HINT 1 for (5-2.q) → 5-2-17-0:bracvgh1.pdf

SOLUTION for (5-2.q) → 5-2-17-1:.pdf ▲

(5-2.r) ⚄ (40 min.) Now we switch to a **graph description** of the curve $\mathbf{u} : [0,1] \to \mathbb{R}^2$, $\xi \mapsto \mathbf{u}(\xi)$. To that end we make the plausible assumption that $\xi \mapsto u_1(\xi) := (\mathbf{u}(\xi))_1$ is monotonic and we reparameterize the curve using the variable $x := u_1(\xi) := (\mathbf{u}(\xi))_1$:

$$\mathbf{u}(\xi) = \begin{bmatrix} u_1(\xi) \\ u_2(\xi) \end{bmatrix} \quad \leftrightarrow \quad \mathbf{u}(x) := \begin{bmatrix} x \\ y(x) \end{bmatrix}, \qquad \begin{aligned} &0 \le \xi \le 1 \,, \\ &a_1 \le x \le a_2 \,, \end{aligned} \tag{5.2.56}$$

with a continuous function $y : [a_1, b_1] \to \mathbb{R}$, $x \mapsto y(x)$

Express the expression $T(\mathbf{u})$ for the total travel time found in Sub-problem (5-2.d) in terms of the new parameterization, that is, find $T_g = T_g(y)$, $y \in C^1_{\mathrm{pw}}([a_1, b_1])$, such that $T(\mathbf{u}) = T_g(y)$ with $\mathbf{u}$ and y related as in (5.2.56).

SOLUTION for (5-2.r) → 5-2-18-0:.pdf ▲

(5-2.s) ⚄ (30 min.) Derive the variational equation satisfied by every minimizer of the functional T_g found in Sub-problem (5-2.r).

SOLUTION for (5-2.s) → 5-2-19-0:.pdf ▲

End Problem 5-2 , 460 min.

Problem 5-3: Minimal Surface Problem for Graphs

The graph minimal surface problem over a bounded domain $\Omega \subset \mathbb{R}^2$ seeks $u : \Omega \to \mathbb{R}$ with prescribed boundary values on $\partial\Omega$, whose graph has minimal surface among all those functions. We study a finite-element method for solving the associated non-linear variational equation.

[Lecture → Ex. 5.2.2.24] derives the surface functional and the variational equation. Otherwise this problem addresses many concepts and techniques from [Lecture → Section 5.3].
▷ problem name/problem code folder: MinimalGraphSurface

We consider the graph minimal surface problem presented in [Lecture → Ex. 5.2.2.24] over a polygonally bounded base domain $\Omega \subset \mathbb{R}^2$.

(5-3.a) ⚀ (15 min.) Study [Lecture → Ex. 5.2.2.24] again with focus on the derivation of the area functional [Lecture → (5.2.2.27)] and of the related variational equation [Lecture → (5.2.2.29)]. ▲

(5-3.b) ⚄ (15 min.) Formulate the recursion of a suitable fixed-point iteration in function space in the spirit of [Lecture → Def. 11.2.5.1] for the graph minimal surface variational equation

$$u \in C^1_{\mathrm{pw}}(\Omega)\,,\ u|_{\partial\Omega} = d \quad : \quad \int_\Omega \frac{\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1 + \|\mathbf{grad}\, u(\boldsymbol{x})\|_2^2}}\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\,, \quad \text{[Lecture → (5.2.2.29)]}$$

where $d \in C^1_{\mathrm{pw}}(\partial\Omega)$ are given boundary data.

SOLUTION for (5-3.b) → 5-3-2-0:mss2.pdf ▲

(5-3.c) ⚀ (15 min.) Examine [Lecture → Ex. 5.3.2.13], where the Newton iteration is derived for the following non-linear variational problem: Seek $u \in \widehat{V} := \{f \in H^1(\Omega) : f|_{\partial\Omega} = d\}$, such that

$$\int_\Omega \sigma(x)\mathbf{A}(\mathbf{grad}\, u(x))\,\mathbf{grad}\, u(x) \cdot \mathbf{grad}\, v(x)\,\mathrm{d}x = -\int_\Omega g\,\rho(x)v(x)\,\mathrm{d}x \quad \forall v \in H^1_0(\Omega)\,,$$

[Lecture → (5.2.2.17)]

$$\mathbf{A}\left(\begin{bmatrix} g_1 \\ g_2 \end{bmatrix}\right) := \frac{1}{2L}\begin{bmatrix} \left(\sqrt{1+g_1^2} - c\right)\dfrac{1}{\sqrt{1+g_1^2}} & 0 \\ 0 & \left(\sqrt{1+g_2^2} - c\right)\dfrac{1}{\sqrt{1+g_2^2}} \end{bmatrix}, \quad \begin{array}{l} L > 0, \\ c > 0\,. \end{array}$$

[Lecture → (5.2.2.15)]

Familiarize yourself with the single steps of the derivation of the linear variational equation

$$w \in V_0\text{:} \quad \mathsf{D}_u\mathsf{a}(u;v)w = 0 \quad \forall v \in V_0$$

for the Newton correction. ▲

(5-3.d) ⚅ (45 min.) Derive the recursion for the **Newton method in function space** of [Lecture → Section 5.3.2.2] for the non-linear variational problem [Lecture → (5.2.2.29)].

SOLUTION for (5-3.d) → 5-3-4-0:mgss4.pdf ▲

(5-3.e) ⚄ (20 min.) [depends on Sub-problem (5-3.d)]

As we have seen in [Lecture → Section 5.3.2.2], a single step $u^{(k)} \to u^{(k+1)}$ of the Newton iteration for the abstract variational equation

$$u \in \widehat{V} := u_0 + V_0\text{:} \quad \mathsf{a}(u;v) = 0 \quad \forall v \in V_0$$

can be written as

$$w \in V_0: \quad \mathsf{D}_u \mathsf{a}(u^{(k)};v)w = -\mathsf{a}(u^{(k)};v) \quad \forall v \in V_0 \quad , \quad u^{(k+1)} := u^{(k)} + w \, .$$

Show that in the case of [Lecture → (5.2.2.29)] the highlighted variational equation can be written in the form ($V_0 = C^1_{\mathrm{pw},0}(\overline{\Omega})$)

$$w \in V_0: \quad \int_\Omega \mathbf{A}(\mathbf{x}) \, \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \int_\Omega c(\boldsymbol{x}) \, \mathbf{grad}\, u^{(k)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \quad \forall v \in V_0 \, . \tag{5.3.6}$$

Give formulas for both "coefficient functions" $\mathbf{A}(\boldsymbol{x})$ and $c(\boldsymbol{x})$, $\boldsymbol{x} \in \Omega$. They may contain $u^{(k)}(\boldsymbol{x})$.

SOLUTION for (5-3.e) → 5-3-5-0:msg5s.pdf ▲

We employ a finite-element Galerkin discretization of the graph minimal surface variational equation [Lecture → (5.2.2.29)] based on the lowest-degree Lagrangian finite element space $\mathcal{S}^0_1(\mathcal{M})$, $\mathcal{M}$ a hybrid mesh of Ω. We end up with the discrete variational problem

$$u_h \in \widehat{V}_h: \quad \int_\Omega \frac{\mathbf{grad}\, u_h(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x})}{\sqrt{1 + \|\mathbf{grad}\, u_h(\boldsymbol{x})\|_2^2}} \, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v_h \in V_{0,h} := \mathcal{S}^0_{1,0}(\mathcal{M}) \, , \tag{5.3.9}$$

$$\widehat{V}_h := \{u_h \in \mathcal{S}^0_1(\mathcal{M}) : \; u_h|_{\partial\Omega} = d_h\} \, , \tag{5.3.10}$$

where $d_h \in \mathcal{S}^0_1(\mathcal{M})|_{\partial\Omega}$ is a piecewise linear interpolant of the boundary data d, *cf.* [Lecture → Ex. 2.7.6.5].

(5-3.f) ⚂ (30 min.) In the file `minimalgraphsurface.cc` complete the implementation of the LEHRFEM++-based C++ function

```
double computeGraphArea(
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
   fes_p,
const Eigen::VectorXd &mu_vec);
```

that computes and returns the area

$$A = \int_\Omega \sqrt{1 + \|\mathbf{grad}\, u_h(\boldsymbol{x})\|^2} \, \mathrm{d}\boldsymbol{x}$$

of the graph of the finite-element function $u_h \in \mathcal{S}^0_1(\mathcal{M})$ whose basis expansion coefficients are passed via the vector `mu_vec`. Information about the domain, the mesh, and the local→global index mapping for d.o.f.s can be retrieved from `fes_p`.

For the approximate evaluation of the integral use a composite quadrature rule on $\mathcal{M}$ of order 2.

HIDDEN HINT 1 for (5-3.f) → 5-3-6-0:mgs6h1.pdf

SOLUTION for (5-3.f) → 5-3-6-1:mgss6.pdf ▲

(5-3.g) ⚄ (30 min.) [depends on Sub-problem (5-3.e)]

In the file `minimalgraphsurface.cc` implement the constructors and the evaluation operators for the following two classes

```
class CoeffTensorA {
  public:
  CoeffTensorA(
```

```
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
       fes_p,
    const Eigen::VectorXd& mu);
    std::vector<Eigen::Matrix2d> operator()(
    const lf::mesh::Entity& e, const Eigen::MatrixXd& refc);
    private:
    lf::fe::MeshFunctionGradFE<double, double> graduh_;
  };

  class CoeffScalarc {
    public:
    CoeffScalarc(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
       fes_p,
    const Eigen::VectorXd& mu);
    std::vector<double> operator()(
    const lf::mesh::Entity& e, const Eigen::MatrixXd& refc);
    private:
    lf::fe::MeshFunctionGradFE<double, double> graduh_;
  };
```

They are meant to provide LEHRFEM++ **MeshFunction** s (→ documentation and [Lecture → Suppl. 2.8.3.32]) encoding the coefficient functions $\boldsymbol{x} \mapsto \mathbf{A}(\boldsymbol{x})$ and $\boldsymbol{x} \mapsto c(\boldsymbol{x})$ from (5.3.6). The constructors of these classes take

(i) a pointer to an object representing the finite element space $\mathcal{S}_1^0(\mathcal{M})$,

(ii) and the basis expansion coefficient vector for the finite element function $u_h^{(k)} \in \mathcal{S}_1^0(\mathcal{M})$.

SOLUTION for (5-3.g) → 5-3-7-0:mgss4.pdf ▲

(5-3.h) ⚄ (45 min.) [depends on Sub-problem (5-3.g)]

In the file `minimalgraphsurface.cc` implement the function

```
Eigen::VectorXd computeNewtonCorrection(
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
   fes_p,
const Eigen::VectorXd mu_vec);
```

that solves the discrete variational equation for the Newton correction in the $k+1$-th step of the Newton iteration

$$w_h \in \mathcal{S}_{1,0}^0(\mathcal{M}):$$
$$\int_\Omega \mathbf{A}(\mathbf{x})\,\mathbf{grad}\, w_h(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega c(\boldsymbol{x})\,\mathbf{grad}\, u_h^{(k)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$
$$\forall v_h \in \mathcal{S}_{1,0}^0(\mathcal{M})\,. \quad (5.3.14)$$

The previous iterate $u_h^{(k)} \in \mathcal{S}_1^0(\mathcal{M})$ is passed through its basis expansion coefficient vector `mu_vec`. The returned vector contains the basis expansion coefficients of w_h in $\mathcal{S}_1^0(\mathcal{M})$ (including zero d.o.f. associated with nodes on the boundary).

HIDDEN HINT 1 for (5-3.h) → 5-3-8-0:mgsh8.pdf

HIDDEN HINT 2 for (5-3.h) → 5-3-8-1:mgsh82.pdf

HIDDEN HINT 3 for (5-3.h) → 5-3-8-2:mgsh83.pdf

SOLUTION for (5-3.h) → 5-3-8-3:mgs83.pdf ▲

(5-3.i) ⚃ (30 min.) [depends on Sub-problem (5-3.h)]

In the file `minimalgraphsurface.h` complete the implementation of the C++ function

```
template<typename D_FUNCTOR, typename RECORDER
= std::function<void(const Eigen::VectorXd &)>>
Eigen::VectorXd graphMinimalSurface(
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
   fes_p,
D_FUNCTOR &&boundary_data,
double rtol, double atol, unsigned int itmax,
RECORDER &&rec = [](const Eigen::VectorXd &) -> void {});
```

that solves [Lecture → (5.2.2.29)] by means of **Newton's method**,

- using the Galerkin finite element solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ of the Diichlet problem

$$-\Delta u = 0 \quad \text{in} \quad \Omega \quad , \quad u = d \quad \text{on} \quad \partial\Omega$$

  as initial guess $u_h^{(0)}$,

- employing a correction-based stopping rule (→ [NumCSE course → § 8.2.3.4]) based on the scaled Euclidean norms of the basis expansion coefficient vectors,

$$\|\vec{\mu}\|_L^2 := \frac{1}{N}\|\vec{\mu}\|_2^2 \,, \quad N := \dim \mathcal{S}_1^0(\mathcal{M}) \,, \tag{5.3.16}$$

  relative tolerance `rtol`, and absolute tolerance `atol`.

The argument `boundary_data` is to supply a functor object describing the Dirichlet data $d : \partial\Omega \to \mathbb{R}$. The return value is the basis expansion coefficient vector of the $\mathcal{S}_1^0(\mathcal{M})$-approximation of the solution of [Lecture → (5.2.2.29)].

During the Newton iteration the basis expansion coefficient vectors of the iterates $u_h^{(k)} \in \mathcal{S}_1^0(\mathcal{M})$ should be made available to the recorder object `rec`.

SOLUTION for (5-3.i) → 5-3-9-0:mgss9.pdf ▲

(5-3.j) ⚂ (30 min.) In the file `minimalgraphsurface.cc` implement the C++ function

```
void graphMinSurfVis(std::string meshfile, std::string vtkfile);
```

that computes the finite-element solution of [Lecture → (5.2.2.29)] in the following setting:

- The finite element mesh is read from the `.msh`-file `meshfile`, which also defines the domain Ω. Use the provided mesh file `square.msh`.
- The parameters for the Newton iteration are `atol=1E-6`, `rtol=1E-4`.
- The boundary values are given by

$$d(\boldsymbol{x}) := |x_1 + x_2 - 1| \,, \quad \boldsymbol{x} \in \mathbb{R}^2 \,. \tag{5.3.18}$$

The computed finite-element solution should be stored in a `.vtk`-file for visualization with PARAVIEW.

SOLUTION for (5-3.j) → 5-3-10-0:mgsAs.pdf ▲

Remark. A priori error estimates for the $\mathcal{S}_1^0(\mathcal{M})$ finite element discretization of the graph minimal surface problem are presented in [Cia78, Sect. 5.2].

References

[Cia78] P.G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Vol. 4. Studies in Mathematics and its Applications. Amsterdam: North-Holland, 1978 (cit. on p. 217).

End Problem 5-3 , 275 min.

Problem 5-4: Iterative Methods for Non-Linear Variational Problems

Generically, non-linear problems have to be solved iteratively. This problem deals with a fixed-point iteration and Newton's method for a particular non-linear second-order elliptic boundary value problem.

The main reference for this problem is [Lecture → Section 5.3.2].

For a bounded convex domain $\Omega \subset \mathbb{R}^2$ we consider the non-linear variational problem: seek $u \in C^1_{\mathrm{pw},0}(\overline{\Omega})$:

$$\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\,, \tag{5.4.1}$$

with $f \in L^2(\Omega)$.

Recall that for all $\xi \in \mathbb{R}$

$$\sinh(\xi) = \tfrac{1}{2}(e^{\xi} - e^{-\xi}) \quad , \quad \sinh'(\xi) = \cosh(\xi) = \tfrac{1}{2}(e^{\xi} + e^{-\xi}) \quad , \quad \cosh'(\xi) = \sinh(\xi)\,.$$

(5-4.a) ⚄ (10 min.) There is a differentiable functional $J : C^1_{\mathrm{pw},0}(\overline{\Omega}) \to \mathbb{R}$, whose minimizers will solve (5.4.1). Tick the correct formula(s) for J.

$$J(v) = \int_\Omega \|\mathbf{grad}\, v(\boldsymbol{x})\| + \cosh(v(\boldsymbol{x})) - f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \qquad \bigcirc$$

$$J(v) = \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, v(\boldsymbol{x})\|^2 + \sinh(v(\boldsymbol{x}))^2 - f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \qquad \bigcirc$$

$$J(v) = \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, v(\boldsymbol{x})\|^2 + \cosh(v(\boldsymbol{x})) - f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \qquad \bigcirc$$

$$J(v) = \int_\Omega \tfrac{1}{2}\, \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(v(\boldsymbol{x})) - f(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \qquad \bigcirc$$

SOLUTION for (5-4.a) → 5-4-1-0:nlvps1.pdf ▲

(5-4.b) ⚄ (20 min.) We can iteratively solve (5.4.1) by means of a fixed-point iteration (in function space), which generates a sequence $\left(u^{(k)}\right)_{k\in\mathbb{N}_0}$ of approximate solutions $u^{(k)} \in C^1_{\mathrm{pw},0}(\overline{\Omega})$, where $u^{(k+1)}$ is computed from $u^{(k)}$ by solving a *linear* variational problem.

Which of the following variational problems for $u^{(k+1)} \in C^1_{\mathrm{pw},0}(\overline{\Omega})$ spawn a valid fixed-point iteration? Put ticks in the corresponding circle(s).

$$\int_\Omega \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k)}(\boldsymbol{x})) v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0\,, \qquad \bigcirc$$

$$\int_\Omega \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k)}(\boldsymbol{x})) u^{(k+1)}(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \qquad \bigcirc$$

$$\int_\Omega \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \frac{\sinh(u^{(k)}(\boldsymbol{x}))}{u^{(k)}(\boldsymbol{x})} u^{(k+1)}(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \qquad \bigcirc$$

$$\int_\Omega \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k)}(\boldsymbol{x})) u^{(k+1)}(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,. \qquad \bigcirc$$

Of course, all these equations are supposed to be satisfied for all $v \in C^1_{\mathrm{pw},0}(\overline{\Omega})$.

SOLUTION for (5-4.b) → 5-4-2-0:nlvps2.pdf ▲

(5-4.c) ⚄ (20 min.) Which of the following linear variational equations have to be solved in order to compute the Newton update $w \in C^1_{\mathrm{pw},0}(\overline{\Omega})$ in the k-th step of a Newton iteration (in function space) meant to solve (5.4.1) iteratively? Tick the associated circle(s).

(A) $\displaystyle\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k-1)}(\boldsymbol{x}))v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
$\displaystyle = -\int_\Omega \mathbf{grad}\, u^{(k-1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, w(\boldsymbol{x}) + \sinh(u^{(k-1)}(\boldsymbol{x}))v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,,$ ○

(B) $\displaystyle\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k-1)}(\boldsymbol{x}))v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
$\displaystyle = -\int_\Omega \mathbf{grad}\, u^{(k-1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k-1)}(\boldsymbol{x}))v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,,$ ○

(C) $\displaystyle\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k-1)}(\boldsymbol{x}))w(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
$\displaystyle = -\int_\Omega \mathbf{grad}\, u^{(k-1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k-1)}(\boldsymbol{x}))v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,,$ ○

(D) $\displaystyle\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k-1)}(\boldsymbol{x}))w(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
$\displaystyle = -\int_\Omega \mathbf{grad}\, u^{(k-1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k-1)}(\boldsymbol{x}))v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$ ○

Of course, each of these equations is to hold for all $v \in C^1_{\mathrm{pw},0}(\overline{\Omega})$.

SOLUTION for (5-4.c) → `5-4-3-0:nlvps3.pdf` ▲

End Problem 5-4, 50 min.

Problem 5-5: Solving a Semi-Linear Elliptic Boundary Value Problem

Semi-linear elliptic boundary value problems in PDE form are distinguished by a linear second-order part of the differential operator and a non-linearity only in the zero-order part. In this problem we examine a particular example, Galerkin finite-element discretization, and also study iterative methods for its solution.

This problem relies on the contents of [Lecture → Section 5.2] and [Lecture → Section 5.3.2]. It also expects the student to be familiar with LehrFEM++.
▷ problem name/problem code folder: SemilinearEllipticBVP

We are given an LehrFEM++-based code for solving a (variational) non-linear boundary-value problem by means of the finite-element method. The code involves three parts.

First, the following helper class:

C++ code 5.5.1: MeshFunction wrapper class FunctionMFWrapper

```
2  template <typename MESHFUNCTION, typename FUNCTION>
3  class FunctionMFWrapper {
4   public:
5    static_assert(lf::mesh::utils::MeshFunction<MESHFUNCTION>);
6    using mf_vector_t = decltype(std::declval<MESHFUNCTION>()(
7        std::declval<const lf::mesh::Entity>(),
8        std::declval<const Eigen::MatrixXd>()));
9    using mf_result_t = typename mf_vector_t::value_type;
10   // Return type of the function F
11   using F_result_t =
12       decltype(std::declval<FUNCTION>()(std::declval<mf_result_t>()));
13
14   explicit FunctionMFWrapper(MESHFUNCTION mf, FUNCTION F)
15       : mf_(std::move(mf)), F_(std::move(F)) {}
16   FunctionMFWrapper(const FunctionMFWrapper &) = default;
17   FunctionMFWrapper(FunctionMFWrapper &&) noexcept = default;
18   FunctionMFWrapper &operator=(const FunctionMFWrapper &) = delete;
19   FunctionMFWrapper &operator=(FunctionMFWrapper &&) = delete;
20   ~FunctionMFWrapper() = default;
21
22   std::vector<F_result_t> operator()(const lf::mesh::Entity &e,
23                                      const Eigen::MatrixXd &local) const {
24     LF_ASSERT_MSG(e.RefEl().Dimension() == local.rows(),
25                   "mismatch between entity dimension and local.rows()");
26     const std::vector<mf_result_t> mf_result = mf_(e, local);
27     std::vector<F_result_t> result(local.cols());
28     for (long i = 0; i < local.cols(); ++i) {
29       result[i] = F_(mf_result[i]);
30     }
31     return result;
32   }
33
34  private:
35   MESHFUNCTION mf_;
36   FUNCTION F_;
37 };
```

Recall that the evaluation operator of an **MeshFunction** object has the return type **std::vector<T>**. This type is accessible through `FunctionMFWrapper::mf_vector_t`, whereas `FunctionMFWrapper::mf_result_t` agrees with **T**.

Secondly, a function implementing the update of an approximate finite-element solution as required by a suitable fixed-point iteration.

C++ code 5.5.2: Function realizing fixed-point update

```
void fixedPointNextIt(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    Eigen::VectorXd &mu_vec, Eigen::VectorXd &rhs_vec) {
  // Set up mesh functions for diffusion coefficient and reaction
     coefficient
  lf::mesh::utils::MeshFunctionGlobal mf_one(
      [](Eigen::Vector2d /*x*/) -> double { return 1.0; });
  const lf::fe::MeshFunctionFE mf_uh_prev(fes_p, mu_vec);
  const FunctionMFWrapper mf_coeff(mf_uh_prev, [](double xi) -> double {
    return (std::abs(xi) < 1.0E-16) ? 1.0 : std::sinh(xi) / xi;
  });

  const lf::mesh::Mesh &mesh{*(fes_p->Mesh())};
  const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
  const std::size_t N_dofs(dofh.NumDofs());
  // Assemble Galerkin matrix
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double, decltype(mf_one),
                                                      decltype(mf_coeff)>
      elmat_provider(fes_p, mf_one, mf_coeff);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider, A);

  // Enforce homogeneous Dirichlet boundary conditions
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
  // Elimination of degrees of freedom on the boundary.  Also sets the
  // corresponding entries of rhs_vec to zero.
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&bd_flags,
       &dofh](lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
        const lf::mesh::Entity &node{dofh.Entity(gdof_idx)};
        return (bd_flags(node) ? std::make_pair(true, 0.0)
                               : std::make_pair(false, 0.0));
      },
      A, rhs_vec);
  // Solve linear system
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  mu_vec = solver.solve(rhs_vec);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
}
```

The argument `fes_p` passed an object describing the finite element space and its degrees of freedom. The vector `mu_vec` contains the basis expansion coefficient vector of the current iterate and *contains the basis expansion coefficient vector of the next iteration after execution of the function*. The other vector `rhs_vec` supplies the load vector. It may also be changed during the execution of the function.

HIDDEN HINT 1 for (5-5.) → `5-5-0-0:sleph0.pdf`

HIDDEN HINT 2 for (5-5.) → `5-5-0-1:slephx.pdf`

The third part of the code is a function that implements the set-up and the main loop of the fixed-point iteration:

C++ code 5.5.4: Function implementing the fixed-point iteration

```
template <typename FUNCTOR_F, typename RECORDER = std::function<
                                  void(const Eigen::VectorXd &, double)>>
Eigen::VectorXd solveSemilinearBVP(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    FUNCTOR_F f, double rtol = 1.0E-4, double atol = 1.0E-8,
    unsigned int itmax = 100,
    RECORDER &&rec = [](const Eigen::VectorXd &, double) -> void {}) {
  // Wrap right hand side source function into a Mesh Function
  lf::mesh::utils::MeshFunctionGlobal mf_f(f);
  // Reference to current mesh
  const lf::mesh::Mesh &mesh{*(fes_p->Mesh())};
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
  // Dimension of finite element space'
  const std::size_t N_dofs(dofh.NumDofs());

  // Assemble right-hand side vector
  Eigen::VectorXd phi(N_dofs);
  phi.setZero();
  // Assemble volume part of right-hand side vector depending on the
     source
  // function f.
  // Initialize object taking care of local computations on all cells.
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      elvec_builder(fes_p, mf_f);
  // Invoke assembly on cells (codim == 0)
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // Coefficient vector for approximate solution
  Eigen::VectorXd mu_vec(N_dofs);
  // Initial guesd = 0
  mu_vec.setZero();
  // Main fixed-point iteration loop
  Eigen::VectorXd mu_old(N_dofs);
  double diff_norm;  // Norm of correction
  unsigned int steps = 0;
  do {
    mu_old = mu_vec;
    // Solve linear fixed-point system and return solution in mu_vec
    fixedPointNextIt(fes_p, mu_vec, phi);
    diff_norm = (mu_vec - mu_old).norm();
    rec(mu_vec, diff_norm);
    steps++;
    // Correction-based termination
  } while ((diff_norm >= rtol * mu_vec.norm()) && (diff_norm >= atol) &&
           (steps < itmax));
  return mu_vec;
}
```

This function also expects a description of the finite element space via the `fes_p` argument. The argument `f` is supposed to contain a functor of type **std::**`function<`**double**`(Eigen::Vector2d)>`, which supplies a right-hand side source function $f \in C^0(\overline{\Omega})$, $\Omega \subset \mathbb{R}^2$ the computational domain covered by the mesh.

(5-5.a) ⚄ (30 min.) Writing $f : \Omega \to \mathbb{R}$ for the function represented by the functor argument `f`, $\mathcal{M}$ for the mesh on which the finite element space is based, and $\Omega \subset \mathbb{R}^2$ for the computational domain, write down the non-linear *discrete* variational problem, which `solveSemilinearBVP()` is meant to solve.

Seek $u_h \in$ [ ] such that

$$\int_\Omega \boxed{} \,\mathrm{d}\boldsymbol{x} = \int_\Omega \boxed{} \,\mathrm{d}\boldsymbol{x}$$

for all $v_h \in$ [ ].

SOLUTION for (5-5.a) → 5-5-1-0:slepsa.pdf ▲

(5-5.b) ⚂ (20 min.) [depends on Sub-problem (5-5.a)]

Again write $\mathcal{M}$ for the finite-element mesh underlying the `fes_p` argument of `solveSemilinearBVP()` and $\Omega \subset \mathbb{R}^2$ for the computational domain covered by it.

Find a suitable finite-element space V_h and a functional $J : V_h \to \mathbb{R}$ such that the function $u_h \in V_h$, whose basis expansion coefficient vector is computed by `solveSemilinearBVP()` satisfies

$$u_h = \operatorname*{argmin}_{v_h \in V_h} J(v_h) \,.$$

$V_h :=$ [ ] ,

$J(v_h) :=$ [ ] .

HIDDEN HINT 1 for (5-5.b) → 5-5-2-0:slephb.pdf

SOLUTION for (5-5.b) → 5-5-2-1:slepsb.pdf ▲

(5-5.c) ⚂ (16 min.) [depends on Sub-problem (5-5.a)]

Write down the strong/PDE form of the boundary value problem, for which `solveSemilinearBVP()` is meant to compute a finite element solution. As above, let Ω stand for the computational domain and $f : \Omega \to \mathbb{R}$ for the right-hand side source function.

[ ] = [ ] in Ω , (5.5.9)

[ ] = [ ] on $\partial\Omega$. (5.5.10)

SOLUTION for (5-5.c) → 5-5-3-0:slepsc.pdf ▲

(5-5.d) ⚄ (24 min.) [depends on Sub-problem (5-5.a)]

Instead of the fixed-point iteration we want to apply the **Newton method** to solve the same non-linear discrete variational problem as tackled by `solveSemilinearBVP()`. We write $u_h^{(k)}$ for the sequence of finite-element functions produced by the Newton iteration. Write down the linear discrete variational problem, whose solution provides the Newton update w_h such that $u_h^{(k+1)} = u_h^{(k)} + w_h$.

Seek $w_h \in C^1_{\mathrm{pw},0}(\overline{\Omega})$ such that

[ ] (5.5.13)

$$= \boxed{} \quad \text{for all} \quad w_h \in C^1_{\mathrm{pw},0}(\overline{\Omega}) \,. \qquad (5.5.14)$$

SOLUTION for (5-5.d) → 5-5-4-0:slepds.pdf ▲

(5-5.e) ⚄ (30 min.) [depends on Sub-problem (5-5.a) and Sub-problem (5-5.d)]

Fill the blanks in the following listing of the LEHRFEM++-based function

```
void newtonNextIt(std::shared_ptr<const
   lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    Eigen::VectorXd& mu_vec, Eigen::VectorXd& rhs_vec));
```

This function takes the basis expansion coefficient vector `mu_vec` of the current Newton iterate $u_h^{(k)}$ and overwrites it with the basis expansion coefficient vector of the next iterate $u_h^{(k+1)}$. The argument `rhs_vec` supplies the load vector for the linear functional $v \mapsto \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, where f is the source function given to `solveSemilinearBVP()` via the functor object `f`. Components of `rhs_vec` associated with d.o.f.s on $\partial\Omega$ may be set to zero.

C++ code 5.5.18: Incomplete listing of `newtonNextIt()`

```
void newtonNextIt(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    Eigen::VectorXd &mu_vec, Eigen::VectorXd &rhs_vec) {
  lf::mesh::utils::MeshFunctionConstant mf_one(1.);
  lf::mesh::utils::MeshFunctionConstant mf_zero(0.);
  const lf::fe::MeshFunctionFE mf_uh_prev(fes_p, mu_vec);

  const FunctionMFWrapper mf_coeff1_uh_prev(
      [          ], [                                                ]);
  const FunctionMFWrapper mf_coeff2_uh_prev(
      [          ], [                                                ]);
  // We define some variables for easy access
  const lf::mesh::Mesh &mesh{*(fes_p->Mesh())};
  const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
  const std::size_t N_dofs(dofh.NumDofs());
  // Assemble matrix A
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double>
      elmat_provider(fes_p, [          ], [          ]);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider, A);

  // Assemble Galerkin in Eigen::SparseMatrix format
  lf::assemble::COOMatrix<double> M(N_dofs, N_dofs);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double>
    elmat_provider_M(fes_p, mf_one, mf_zero);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider_M, M);
  Eigen::SparseMatrix<double> M_crs = M.makeSparse();

  // Construct the right-hand side of the system
  Eigen::VectorXd phi(N_dofs); phi.setZero();
  phi += [                                                ];
  phi -= [                                                ];
```

```
33    lf::uscalfe::ScalarLoadElementVectorProvider<double>
34        elvec_builder(fes_p, [redacted]);
35    lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);
36
37    // Enforce homogeneous Dirichlet boundary conditions
38    auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
39    // Elimination of degrees of freedom on the boundary.  Also sets the
40    // corresponding entries of rhs_vec to zero.
41    lf::assemble::FixFlaggedSolutionComponents<double>(
42        [&bd_flags,
43         &dofh](lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
44          const lf::mesh::Entity &node{dofh.Entity(gdof_idx)};
45          return (bd_flags(node) ? std::make_pair(true, 0.0)
46                                 : std::make_pair(false, 0.0));
47        },
48        A, phi);
49    // Convert matrix A in COO format to A_crs in Eigen::SparseMatrix
        format
50    Eigen::SparseMatrix<double> A_crs = A.makeSparse();
51    // Solve the system and add the solution to mu_vec
52    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
53    solver.compute(A_crs);
54    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
55    mu_vec += solver.solve(phi);
56    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
57  };
```

SOLUTION for (5-5.e) → prbfile.pdf

▲

End Problem 5-5, 120 min.

Problem 5-6: The P-Laplacian

As we have sen in [Lecture → Section 1.4.2] the Laplacian is connected to the minimization problem for the so-called Dirichlet energy $\int_\Omega \|\mathbf{grad}\, u\|^2 \,\mathrm{d}\boldsymbol{x}$. That functional can be regarded as one member of a family of (generally non-quadratic) energies, which give rise to other PDE operators. Those will be studied in this problem.

This problem covers the topics treated in [Lecture → Section 5.2] and [Lecture → Section 5.3].
▷ problem name/problem code folder: PLaplacian

On a two-dimensional polygonally bounded domain $\Omega \subset \mathbb{R}^2$ and for a parameter $p \in]1,\infty[$ we consider the following variational problem: Seek $u \in V_0$, $V_0 := C^1_{\mathrm{pw},0}(\overline{\Omega})$, such that

$$\int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\|^{p-2} \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\,, \tag{5.6.1}$$

where $f \in C^0(\overline{\Omega})$. The subscript "0" indicates that both trial and test functions have to vanish on $\partial\Omega$. For $p \neq 2$ (5.6.1) is a non-linear variational problem.

(5-6.a) ⚄ (16 min.) For given $p \in]1,\infty[$ state a functional $J_p : V_0 \to \mathbb{R}$, for which any global minimizer u is a solution of (5.6.1), that is,

$$u \in \operatorname*{argmin}_{v \in V_0} J_p(v) \quad \Rightarrow \quad u \quad \text{solves (5.6.1)}\,.$$

$$J_p(v) := \qquad\qquad , \quad v \in V_0\,.$$

HIDDEN HINT 1 for (5-6.a) → 5-6-1-0:Ph1.pdf

SOLUTION for (5-6.a) → 5-6-1-1:Psa.pdf ▲

(5-6.b) ⚄ (16 min.) Propose a **fixed-point iteration in function space** for the iterative solution of (5.6.1). More precisely, if that iteration produces the sequence $\left(u^{(k)}\right)_{k\in\mathbb{N}_0} \subset V_0$ of iterates, specify the *linear variational problem*, whose solution yields $u^{(k+1)}$ assuming that $u^{(k)}$ is given.

Seek $u^{(k+1)} \in V_0$ such that

$$\qquad\qquad = 0 \quad \forall v \in V_0\,.$$

SOLUTION for (5-6.b) → 5-6-2-0:Pbs.pdf ▲

For the Galerkin finite element discretization of (5.6.1) we rely on piecewise linear Lagrangian finite elements on a hybrid mesh $\mathcal{M}$ of Ω, that is, we use $\mathcal{S}^0_{1,0}(\mathcal{M})$ as discrete trial and test space. The standard tent function global shape functions are used throughout.

(5-6.c) ⚄ (40 min.) [depends on Sub-problem (5-6.b)]

Finish the implementation of the C++ function

```
template <typename F_FUNCTOR>
void fixedPointNextIt(
std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
   fes_p,
F_FUNCTOR f, double p, Eigen::VectorXd& mu_vec);
```

This function should carry out a single step of the fixed-point iteration you devised in Sub-problem (5-6.b) and applied to the discretized boundary value problem. The argument `mu_vec` passes the basis expansion coefficient vector for $u_h^{(k)}$, which is then to be *overwritten* with the coefficient vector for $u_h^{(k+1)}$. The first parameter `fes_p` contains all information about the finite-element space $\mathcal{S}_1^0(\mathcal{M})$, the mesh, and the local-to-global index mapping (**lf::assemble::DofHandler**). The arguments `f` and `p` provide a functor object for the right-hand side source function f and the parameter $p \in]1,\infty[$.

HIDDEN HINT 1 for (5-6.c) → 5-6-3-0:Pch1.pdf

HIDDEN HINT 2 for (5-6.c) → 5-6-3-1:Pch2.pdf

SOLUTION for (5-6.c) → 5-6-3-2:Pcs4.pdf ▲

(5-6.d) ⚄ (40 min.) Now we consider the **Newton method in function space** for (5.6.1). Now write $\left(u^{(k)}\right)_{k\in\mathbb{N}_0}$ for the sequence of approximate solutions $u^{(k)} \in V_0$ it generates. Explicitly state the $u^{(k)}$-dependent linear variational problem to be solved to compute the **Newton update** $w := u^{(k+1)} - u^{(k)} \in V_0$.

Seek $w \in V_0$ such that

$$\qquad\qquad\qquad

Chapter 6

Numerical Integration – Single Step Methods

Problem 6-1: Linear ODE in spaces of matrices

In this problem we consider initial value problems (IVPs) for linear ordinary differential equations (ODEs) whose state space is a vector space of $n \times n$ matrices. Such ODEs arise when modelling the dynamics of rigid bodies in classical mechanics. A related problem is Problem 6-5.

Mainly relies on the information contained in [Lecture → Section 6.2].
▷ problem name/problem code folder: MatODE

For $n \in \mathbb{N}$ we consider the autonomous *linear* matrix differential equation

$$\dot{\mathbf{Y}} = \mathbf{A}\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad \text{with} \quad \mathbf{A} \in \mathbb{R}^{n,n} , \tag{6.1.1}$$

whose solutions are *matrix-valued functions* $\mathbf{Y} : \mathbb{R} \to \mathbb{R}^{n,n}$, and whose right-hand-side function maps from $\mathbb{R}^{n,n}$ ijto $\mathbb{R}^{n,n}$

(6-1.a) ⚂ (15 min.) Show that for **skew-symmetric** $\mathbf{A}$, i.e. $\mathbf{A} = -\mathbf{A}^\top$, we have:

$$\mathbf{Y}(0) \text{ orthogonal} \implies \mathbf{Y}(t) \text{ orthogonal} \quad \forall t .$$

HIDDEN HINT 1 for (6-1.a) → 6-1-1-0:MatO1h1.pdf

HIDDEN HINT 2 for (6-1.a) → 6-1-1-1:MatO1h2.pdf

SOLUTION for (6-1.a) → 6-1-1-2:MatO1s.pdf ▲

(6-1.b) ⚂ (30 min.) In the file `matode.cc` implement three C++ functions realizing

1. a single step of the explicit Euler method, see [Lecture → Section 6.2.1]:

```
Eigen::MatrixXd eeulstep(const Eigen::Eigen::MatrixXd & A,
                         const Eigen::MatrixXd & Y0, double h);
```

2. a single step of the implicit Euler method, see [Lecture → Section 6.2.2],

```
Eigen::MatrixXd ieulstep(const Eigen::MatrixXd & A,
                         const Eigen::MatrixXd & Y0, double h);
```

3. a single step of the implicit mid-point method, see [Lecture → Section 6.2.3].

```
Eigen::MatrixXd impstep(const Eigen::MatrixXd & A,
                        const Eigen::MatrixXd & Y0, double h);
```

which compute, for a given initial value $\mathbf{Y}(t_0) = \mathbf{Y}_0$ and for given step size h, approximations for $\mathbf{Y}(t_0 + h)$ using one step of the corresponding method for the approximation of the ODE (6.1.1)

HIDDEN HINT 1 for (6-1.b) → 6-1-2-0:MatO2h.pdf

SOLUTION for (6-1.b) → 6-1-2-1:MatO2s.pdf ▲

(6-1.c) ⚄ (30 min.) [depends on Sub-problem (6-1.b)]

Investigate numerically, which one of the implemented methods preserves orthogonality for the ODE (6.1.1) and which one doesn't. To that end, consider the matrix

$$\mathbf{M} := \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 9 & 9 & 2 \end{bmatrix}$$

and use the matrix $\mathbf{Q}$ arising from the QR-decomposition (→ [NumCSE course → Section 3.3.3.4]) of $\mathbf{M}$ as initial data $\mathbf{Y}_0$. As matrix $\mathbf{A}$, use the skew-symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}.$$

Perform $M = 20$ time steps of size $h = 0.01$ with each method. Compute and tabulate the Frobenius norm of $\mathbf{Y}_k^\top \mathbf{Y}_k - \mathbf{I}$.

Do all this in `matode_main.cc` .

SOLUTION for (6-1.c) → 6-1-3-0:MatO3s.pdf ▲

End Problem 6-1 , 75 min.

Problem 6-2: Explicit Runge-Kutta Methods

The most widely used class of numerical integrators for IVPs is that of *explicit* Runge-Kutta (RK) methods as defined in [Lecture → Def. 6.4.0.9]. They are usually described by giving their coefficients in the form of a Butcher scheme [Lecture → (6.4.0.11)].

Relies on [Lecture → Section 6.4], [Lecture → Def. 6.4.0.9], and the concept of Butcher schemes as defined in [Lecture → (6.4.0.11)]
▷ problem name/problem code folder: RK3Prey

(6-2.a) ⚀ (30 min.) In the file `rk3prey.h`, implement a C++ class

```
class RKIntegrator {
 public:
  // Constructor for the RK method.
  RKIntegrator(const Eigen::MatrixXd &A, const Eigen::VectorXd &b);

  // Explicit Runge-Kutta numerical integrator
  template <class Function>
  std::vector<Eigen::VectorXd> solve(Function &&f, double T,
                                     const Eigen::VectorXd &y0,
                                     int M) const;
 private:
 // ...  stored data
};
```

which provides a generic **explicit Runge-Kutta single-step method** (→ [Lecture → Def. 6.4.0.9]) given by a Butcher scheme [Lecture → (6.4.0.11)] to solve the autonomous initial-value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$. The matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the vector $\mathbf{b} \in \mathbb{R}^s$ from the Butcher scheme are passed to the constructor, whereas the right-hand side vector field $\mathbf{f}$, the final time $T > 0$, the number M of *equidistant timesteps*, and the intial state $\mathbf{y}_0$ are arguments of the `solve()` method, which is supposed to return the sequence $(\mathbf{y}_k)_{k=0}^{M}$ of vectors generated by the explicit Runge-Kutta method.

SOLUTION for (6-2.a) → `6-2-1-0:RK3P1s.pdf` ▲

(6-2.b) ⚁ (30 min.) [depends on Sub-problem (6-2.a)]

In the main file `rk3prey_main.cc`, test your implementation of the RK methods with the following test case:

As autonomous initial value problem, consider the predator/prey model (*cf.* [Lecture → Ex. 6.1.2.5]):

$$\begin{aligned} \dot{y}_1(t) &= (\alpha_1 - \beta_1 y_2(t)) y_1(t) \,, \\ \dot{y}_2(t) &= (\beta_2 y_1(t) - \alpha_2) y_2(t) \,, \\ \mathbf{y}(0) &= [100, 5]^\top \,, \end{aligned} \tag{6.2.3}$$

with coefficients $\alpha_1 = 3, \alpha_2 = 2, \beta_1 = \beta_2 = 0.1$.

Use the explicit 3-stage Runge-Kutta single step method described by the following **Butcher scheme** (*cf.* [Lecture → Def. 6.4.0.9]):

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \tag{6.2.4}$$

Compute an approximated solution up to time $T = 10$ for the number of equidistant time steps $M = 2^j$, $j = 7, \ldots, 14$.

As reference solution, use $\mathbf{y}(10) = [0.319465882659820, 9.730809352326228]^\top$.

Tabulate the (modulus of the) *error at final time*.

SOLUTION for (6-2.b) → `6-2-2-0:RK3P2s.pdf` ▲

End Problem 6-2 , 60 min.

Problem 6-3: Extrapolation of evolution operators

In [Lecture → § 6.3.1.1] we have seen how discrete evolution operators can describe single-step methods for the numerical integration of ODEs. This task will study a way to combine discrete evolution operators of known order in order to build a single-step method with increased order. The method can be generalized to an extrapolation construction of higher-order single-step methods. These can be used for time-local stepsize control following the policy of [Lecture → § 6.5.2.11].

This problem assumes familiarity with the [Lecture → Section 6.3] and, in particular, the concept of discrete evolution, *cf.* [Lecture → Def. 6.3.1.4]. It also addresses the adaptive timestepping strategy presented in [Lecture → Section 6.5].
▷ problem name/problem code folder: ODESolve

Let $\boldsymbol{\Psi}^h$ define the discrete evolution of an order p Runge-Kutta single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subseteq \mathbb{R}^N \to \mathbb{R}^N$, $N \in \mathbb{N}$. We define a new discrete evolution operator [Lecture → Def. 6.1.4.3]:

$$\widetilde{\boldsymbol{\Psi}}^h := \frac{1}{1-2^p}\left(\boldsymbol{\Psi}^h - 2^p \cdot (\boldsymbol{\Psi}^{h/2} \circ \boldsymbol{\Psi}^{h/2})\right), \tag{6.3.1}$$

where $\circ$ denotes the composition of mappings.

(6-3.a) ⚀ (10 min.) If $\boldsymbol{\Psi}^h$ belongs to the explicit/forward Euler method as introduced in [Lecture → Section 6.2.1], give the explicit formula for $\widetilde{\boldsymbol{\Psi}}^h$.

SOLUTION for (6-3.a) → 6-3-1-0:ODESolve1s.pdf ▲

(6-3.b) ⚁ (15 min.) In `odesolve.h`, implement an EIGEN-based C++ function

```
template <class DiscEvlOp>
double PsiTilde(const DiscEvlOp& Psi, unsigned int p, double h,
   double y0);
```

that returns $\widetilde{\boldsymbol{\Psi}}^h \mathbf{y}_0$ when given the underlying $\boldsymbol{\Psi}$ through the functor `Psi`. Objects of type **DiscEvlOp** must provide an evaluation operator:

```
Eigen::VectorXd operator()(double h, const Eigen::VectorXd &y);
```

providing the result of $\boldsymbol{\Psi}^h(\mathbf{y})$. For instance, suitable C++ lambda functions satisfy this requirement, see [NumCSE course → Section 0.3.3].

SOLUTION for (6-3.b) → 6-3-2-0:ODESolve2s.pdf ▲

(6-3.c) ⚁ (15 min.) Implement a C++ function

```
template <class DiscEvlOp>
std::vector<double> OdeIntEqui(const DiscEvlOp& Psi, double T,
   double y0,int M);
```

in `odesolve.h` that carries out M equidistant steps of size $\tau := T/M$ of a single step method described by the discrete evolution $\boldsymbol{\Psi}$ passed through the functor object `Psi`. The function should return the sequence $(\mathbf{y}_k)_{k=0}^M$ produced by the single-step method when started with initial value $\mathbf{y}_0$ (given as argument `y0`).

HIDDEN HINT 1 for (6-3.c) → 6-3-3-0:ODESolve3h.pdf

SOLUTION for (6-3.c) → 6-3-3-1:ODESolve3s.pdf ▲

(6-3.d) ⚂ (20 min.) [depends on Sub-problem (6-3.c), Sub-problem (6-3.a)]

For the particular scalar initial-value problem (IVP)

$$\dot{y} = 1 + y^2 \quad , \quad y(0) = 0 \, , \tag{6.3.4}$$

with exact solution $y(t) = \tan(t)$, determine empirically the order of the single step method induced by $\widetilde{\Psi}^h$ from (6.3.1), when Ψ arises from the explicit Euler method, recall Sub-problem (6-3.a). For that purpose, write a C++ function

```
double TestCvgExtrapolatedEuler();
```

in `odesolve.cc` that monitors and tabulates the error $|y_M(1) - y_{ex}(1)|$ at final time $T = 1$ for M uniforms steps with $M = 2^q, q = 2, \ldots, 12$, also returns an estimate of the rate of algebraic convergence based on linear regression, for which an auxiliary function `polyfit()` is provided in the files `polyfit.h`/`polyfit.cc`.

HIDDEN HINT 1 for (6-3.d) → `6-3-4-0:ODESolve4h.pdf`

SOLUTION for (6-3.d) → `6-3-4-1:ODESolve4s.pdf` ▲

(6-3.e) ⚄ (40 min.) In general, the method defined by $\widetilde{\Psi}^h$ from (6.3.1) has order $p+1$. Thus, it can be used for adaptive timestep control and prediction.

Complete the implementation of a function

```
template <class DiscEvlOp>
std::pair<std::vector<double>, std::vector<double>> OdeIntSsCtrl(
    const DiscEvlOp& Psi, unsigned int p, double y0, double T,
       double h0,
    double reltol, double abstol, double hmin);
```

for the approximation of the solution of an IVP by means of adaptive timestepping based on Ψ and $\widetilde{\Psi}$, where Ψ is passed through the argument `Psi`.

Step rejection and stepsize correction and prediction as explained in [Lecture → Section 6.5] is to be employed as in [Lecture → Code 6.5.2.18]. The argument `T` supplies the final time, `y0` the initial state, `h0` an initial stepsize, `p` the order of the discrete evolution Ψ, `reltol` and `abstol` the respective tolerances, and `hmin` a minimal stepsize that will trigger premature termination.

The function should return a vector of tuples $(t_k, \mathbf{y}_k)$, $k = 0, \ldots, M$, where $M \in \mathbb{N}$ is the total number of timesteps, t_k are the knots of the temporal mesh created by the adaptive integrator, and $(\mathbf{y}_k)_k$ the sequence of states generated by the single-step method.

HIDDEN HINT 1 for (6-3.e) → `6-3-5-0:EPh.pdf`

SOLUTION for (6-3.e) → `6-3-5-1:ODESolve5s.pdf` ▲

(6-3.f) ⚂ (20 min.) [depends on Sub-problem (6-3.e)]

In `odesolve.cc`, implement a C++ function

```
std::pair<std::vector<double>, std::vector<double>>
   SolveTangentIVP();
```

that relies on `OdeIntSsCtrl` from Sub-problem (6-3.e) to compute the solution of the IVP (6.3.4) up to time $T = 1.5$ with the following data: `h0` $= 1/100$, `reltol` $= 10e-4$, `abstol` $= 10e-6$, `hmin` $= 10e-5$.

SOLUTION for (6-3.f) → `6-3-6-0:sx.pdf` ▲

End Problem 6-3 , 120 min.

Problem 6-4: System of Second-Order ODEs

In this problem we practise the conversion of a second-order ODE into a first-order system in the case of a large linear system of ODEs.

This problem assumes familiarity with Runge-Kutta single-step methods as introduced in [Lecture → Section 6.4]
▷ problem name/problem code folder: SystemODE

Consider the following initial value problem for an (implicit) second-order system of ordinary differential equations in the time interval $[0,T]$:

$$\begin{aligned} 2\ddot{u}_1 - \ddot{u}_2 &= u_1(u_2+u_1)\,, \\ -\ddot{u}_{i-1} + 2\ddot{u}_i - \ddot{u}_{i+1} &= u_i(u_{i-1}+u_{i+1})\,, \qquad i = 2,\ldots,n-1\,, \\ 2\ddot{u}_n - \ddot{u}_{n-1} &= u_n(u_n+u_{n-1})\,, \\ u_i(0) &= u_{0,i} \qquad i = 1,\ldots,n\,, \\ \dot{u}_i(0) &= v_{0,i} \qquad i = 1,\ldots,n\,. \end{aligned} \tag{6.4.1}$$

Here the notation $\ddot{w}$ designates the second derivative of a time-dependent function $t \mapsto w(t)$.

(6-4.a) ⚀ (20 min.) Write (6.4.1) as a first-order IVP of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

HIDDEN HINT 1 for (6-4.a) → 6-4-1-0:Syst1h1.pdf

HIDDEN HINT 2 for (6-4.a) → 6-4-1-1:Syst1h2.pdf

HIDDEN HINT 3 for (6-4.a) → 6-4-1-2:Syst1h3.pdf

SOLUTION for (6-4.a) → 6-4-1-3:Syst1s.pdf ▲

(6-4.b) ⚁ (15 min.) Write down the discrete evolution for the classical Runge-Kutta method of order 4, whose Butcher scheme is given in [Lecture → Ex. 6.4.0.15] for the concrete case of the autonomous linear ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{R}^{d,d}$.

HIDDEN HINT 1 for (6-4.b) → 6-4-2-0:h2.pdf

SOLUTION for (6-4.b) → 6-4-2-1:Syst2s.pdf ▲

(6-4.c) ⚂ (20 min.) [depends on Sub-problem (6-4.b)]

In the file `systemode.h` implement a C++ function

```
template <typename Function>
Eigen::VectorXd rk4step(Function &&f, double h, Eigen::VectorXd
    &y0)
```

that performs a single step of stepsize h of the classical Runge-Kutta method of order 4 for the first-order ODE obtained in Sub-problem (6-4.a) (associated right-hand side function passed via the functor `f`). Here, `y0` contains the vector before the step and the function returns the vector `y1` after the step.

SOLUTION for (6-4.c) → 6-4-3-0:Syst3s.pdf ▲

(6-4.d) ⚃ (40 min.) [depends on Sub-problem (6-4.a), Sub-problem (6-4.c)]

Apply the classical Runge-Kutta method of order 4 to solve a particular initial-value problem for the ODE derived in Sub-problem (6-4.a). Use the parameters and initial values

$$n = 5\,, \qquad u_i(0) = i/n\,, \qquad v_i(0) = -1\,, \qquad T = 1\,,$$

In the file `systemode.cc` implement a function

```
void testCvg(void)
```

that tabulates the Euclidean norm of the error at final time $T = 1$ for numbers $M = 2, 4, 8, \ldots, 1024$ of uniform timesteps. As reference solution use the value obtained with $M = 2^{12}$ equidistant timesteps. You can estimate the rate of algebraic convergence obtained by means of linear regression for which you can use the supplied function `polyfit()` from `polyfit.h`.

> Note that for the sake of efficiency you should use EIGEN's sparse matrix data type for any sparse matrix occurring. Refer to [NumCSE course → Section 2.7.2] for details.

SOLUTION for (6-4.d) → `6-4-4-0:Syst4s.pdf` ▲

End Problem 6-4 , 95 min.

Problem 6-5: Non-linear Evolutions in Spaces of Matrices

In this problem we consider initial value problems (IVPs) for ordinary differential equations whose state space is a vector space of $n \times n$ matrices. Such IVPs occur in mathematical models of discrete mechanical systems.

In this problem we practice the use of an adaptive Runge-Kutta-Fehlberg single-step method provided by the library class **Ode45** introduced in [Lecture → § 6.5.3.3] ➙ GitLab. Related to this problem is Problem 6-1.
▷ problem name/problem code folder: NLMatOde

We consider a *non-linear* ODE in the state space $D := \mathbb{R}^{n,n}$ of real $n \times n$ matrices and study the associated initial value problem

$$\dot{\mathbf{Y}} = -(\mathbf{Y} - \mathbf{Y}^\top)\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad , \quad \mathbf{Y}(0) = \mathbf{Y}_0 \in \mathbb{R}^{n,n}, \tag{6.5.1}$$

whose solution is given by a *matrix-valued function* $t \mapsto \mathbf{Y}(t) \in \mathbb{R}^{n \times n}$.

(6-5.a) ⚂ (30 min.) In `nlmatode.cc`, write a C++ function

```
Eigen::MatrixXd matode(const Eigen::MatrixXd & Y0, double T)
```

which solves (6.5.1) on $[0, T]$ using the C++ header-only class **Ode45** (in the file `ode45.hpp` in the `lecturecodes/Ode45` directory). The initial value should be given by a $n \times n$ EIGEN matrix `Y0`. Set the absolute tolerance to 10^{-10} and the relative tolerance to 10^{-8}. The output should be an approximation of $\mathbf{Y}(T) \in \mathbb{R}^{n \times n}$.

Instructions on the use of Ode45, see [Lecture → § 6.5.3.3], ➙ GitLab.

The class **Ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` ➙ GitLab defines the class **Ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 6.5.3] and [Lecture → Ex. 6.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 6.5.2.11].

1. Construct an object of type **Ode45**: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

```
template <class StateType,
class RhsType = std::function<StateType(const StateType &)>>
class Ode45 {
  public:
  Ode45(const RhsType &rhs);
  // ......
}
```

Template parameters are

- **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
- **RhsType**: type of rhs function (automatically deduced).

The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator()(const StateType & vec);
```

It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **Ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is `10e-6`)
- `atol`: absolute tolerance for error control (default is `10e-8`)

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

SOLUTION for (6-5.a) → 6-5-1-0:NMatO1s.pdf ▲

(6-5.b) ⚀ (10 min.) Show that the function $t \mapsto \mathbf{Y}^\top(t)\mathbf{Y}(t)$ is constant for the exact solution $\mathbf{Y}(t)$ of (6.5.1).

HIDDEN HINT 1 for (6-5.b) → 6-5-2-0:NMatO2h1.pdf

HIDDEN HINT 2 for (6-5.b) → 6-5-2-1:NMatO2h2.pdf

SOLUTION for (6-5.b) → 6-5-2-2:NMatO2s.pdf ▲

(6-5.c) ⚂ (15 min.) [depends on Sub-problem (6-5.a)]

In `nlmatode.cc` , write a C++ function

```
bool checkinvariant(const Eigen::MatrixXd & M, double T);
```

which (numerically) determines if the invariant $t \mapsto \mathbf{Y}(t)^\top\mathbf{Y}(t)$ is preserved for approximate solutions of (6.5.1) as output by `matode()`. The function's arguments should be the same as that of `matode()`.

Heed [NumCSE course → Rem. 1.5.3.15]: You must take into account round-off errors and cannot test for equality with zero.

SOLUTION for (6-5.c) → 6-5-3-0:NMatO3s.pdf ▲

(6-5.d) ⚄ (30 min.) [depends on Sub-problem (6-5.a)]

The so-called **discrete gradient method** for (6.5.1) reads

$$\mathbf{Y}_* = \mathbf{Y}_0 + \tfrac{1}{2}hf(\mathbf{Y}_0) \quad , \quad \mathbf{Y}_1 = (\mathbf{I} + \tfrac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*^\top))^{-1}(\mathbf{I} - \tfrac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*^\top))\mathbf{Y}_0 \,. \tag{6.5.5}$$

Using the solution produced by `matode()` from Sub-problem (6-5.a) as a substitute for an exact solution, determine the order of convergence of the discrete gradient rule in a numerical experiment that uses $M \in \{10, 20, 40, 80, 160, 320, 640, 1280\}$ equidistant integration steps for solving (6.5.1) approximately. As initial value $\mathbf{Y}_0$ use the *orthogonal* "left-shift" matrix

$$\mathbf{Y}_0 := \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} .$$

Concretely, implement a C++ function

```
double cvgDiscreteGradientMethod();
```

that tabulates the Frobenius norm of the discretization error at final time $T = 1$ and returns an estimate of the rate of algebraic convergence obtained by linear regression. You may use the C++ function `polyfit()` provided in `polyfit.h`.

SOLUTION for (6-5.d) → 6-5-4-0:s5.pdf ▲

End Problem 6-5, 85 min.

Problem 6-6: Order is not everything

In [Lecture → Section 6.3.2] we have seen that Runge-Kutta single step methods when applied to initial value problems with sufficiently smooth solutions will converge algebraically (with respect to the maximum error in the mesh points) with a rate given by their intrinsic order, see [Lecture → Def. 6.3.2.8].

This problem studies the convergence of some Runge-Kutta single-step methods as discussed in [Lecture → Section 6.3.2]. It relies on a class implemented in Problem 6-2.
▷ problem name/problem code folder: OrdNotAll

In this problem we perform empiric investigations of orders of convergence of several explicit Runge-Kutta single step methods. We rely on two IVPs, one of which has a perfectly smooth solution, whereas the second has a solution that is merely piecewise smooth. Thus in the second case the smoothness assumptions of the convergence theory for RK-SSMs might be violated and it is interesting to study the consequences.

Remark. For this problem you can reuse the class **RKIntegrator** implemented in Problem 6-2. You first need to construct an object of this class, passing as arguments the Butcher tableau matrices `A` and `b`, for instance:

```
RKIntegrator<double> rk(A,b);
```

After that, call the methods `solve`, with parameters: r.h.s. function, final time, initial value and number of steps. For instance:

```
rk.solve(f,T,y0,n);
```

The output of this function will be **std::vector<double>** containing the solution at each equidistant time step.

(6-6.a) ⚀ (30 min.) Consider the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{6.6.1}$$

where $\mathbf{f}: \mathbb{R} \to \mathbb{R}$ and $\mathbf{y}_0 \in \mathbb{R}$. Using the class **RKIntegrate** write a C++ function

```
template <class Function>
void testCvgRKSSM(const Function &f, double T, double y0,
const Eigen::MatrixXd &A, const Eigen::VectorXd &b)
```

in `ordnotall.cc` that computes an approximated solution $\mathbf{y}_M$ of (6.6.1) up to time T by means of an explicit Runge-Kutta method with $M = 2^k$, $k = 1, \ldots, 12$, uniform timesteps. The method is defined by the Butcher scheme described by the inputs `A` (Butcher matrix) and `b` (weight vector), see [Lecture → (6.4.0.11)]. The input `f` is an object with an evaluation operator (e.g. a lambda function) for arguments of type **double** representing $\mathbf{f}$. The input `y0` passes the initial value $\mathbf{y}_0$.

For each k, the function should output the error at the final point $E_M = \|\mathbf{y}_M(T) - \mathbf{y}_{2^{12}}(T)\|$, $M = 2^k$, $k = 1, \ldots, 12$, accepting $\mathbf{y}_{2^{12}}(T)$ as exact value. Assuming algebraic convergence for $E_M \approx CM^{-r}$, at each step also print an approximation of the order of convergence r_k (recall that $M = 2^k$). This will be an expression involving E_M and $E_{M/2}$.

Finally, compute and return an approximate order of convergence by linear regression. Only take into account results for which the error is larger than 10^{-14} in order not to be misled by the impact of round-off errors.

SOLUTION for (6-6.a) → 6-6-1-0:OrdN1h.pdf ▲

(6-6.b) ⚀ (30 min.) Calculate the analytical solutions of the logistic ODE (see [Lecture → Ex. 6.1.2.1])

$$\dot{y} = (1-y)y, \quad y(0) = 1/2, \tag{6.6.4}$$

and of the initial value problem

$$\dot{y} = |1.1 - y| + 1, \quad y(0) = 1. \tag{6.6.5}$$

HIDDEN HINT 1 for (6-6.b) → 6-6-2-0:ONA2s.pdf

HIDDEN HINT 2 for (6-6.b) → 6-6-2-1:OMA6s.pdf

HIDDEN HINT 3 for (6-6.b) → 6-6-2-2:OMA8s.pdf

SOLUTION for (6-6.b) → 6-6-2-3:OrdN2h.pdf ▲

(6-6.c) ⚁ (20 min.) [depends on Sub-problem (6-6.a)]

Using `testCvgRKSSM()` write in `ordnoatall.cc` a C++ function

```
void cmpCvgRKSSM(void);
```

that empirically determines and prints the rates of convergence of

- the explicit Euler method [Lecture → (6.2.1.4)], a RK single step method of order 1,
- the explicit trapezoidal rule [Lecture → (6.4.0.6)], a RK single step method of order 2,
- an RK method of order 3 given by the Butcher tableau [Lecture → (6.4.0.11)]

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 1 & -1 & 2 & \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

- the classical RK method of order 4, see [Lecture → Ex. 6.4.0.15] for details.

when applied for the numerical integration of the initial-value problems (6.6.4) and (6.6.5). Use final time $T = 1$ in each case.

Comment on the calculated order of convergence for the different methods and the two different initial value problems.

SOLUTION for (6-6.c) → 6-6-3-0:x1s.pdf ▲

End Problem 6-6, 80 min.

Problem 6-7: Initial Condition for Lotka-Volterra ODE

In this problem, we will face a situation where we need to compute the derivative of the solution of an initial value problem with respect to the initial state in order to apply Newton's method. So this exercise covers both numerical integration and the solution of non-linear systems of equations.

You should grasp the abstract view of ODEs from [Lecture → Section 6.1.4] and still remember Newton's method from [NumCSE course → Section 8.5]. You should also be familiar with the integrator **Ode45** from [Lecture → § 6.5.3.3] ➥ GitLab
▷ problem name/problem code folder: InitCondLV

A differential equation for the derivative w.r.t. initial state. We consider IVPs for the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{6.7.1}$$

with smooth right hand side $\mathbf{f}\colon D \to \mathbb{R}^N$, where $D \subseteq \mathbb{R}^N$ is the state space. We take for granted that for all initial states, solutions exist for all times (global solutions, see [Lecture → Ass. 6.1.4.1]).

By its very definition given in [Lecture → Def. 6.1.4.3], the evolution operator

$$\boldsymbol{\Phi}\colon \mathbb{R} \times D \to D, \quad (t, \mathbf{y}) \mapsto \boldsymbol{\Phi}(t, \mathbf{y})$$

satisfies

$$\frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y})) \,.$$

We can differentiate this identity with respect to the state variable $\mathbf{y}$. We assume that all derivatives can be interchanged, which can be justified by rigorous arguments (we won't do it here). Thus, by the chain rule, we obtain, after swapping partial derivatives $\frac{\partial}{\partial t}$ and $\mathrm{D}_{\mathbf{y}}$,

$$\frac{\partial \mathrm{D}_{\mathbf{y}} \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathrm{D}_{\mathbf{y}} \frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathrm{D}_{\mathbf{y}}(\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))) = \mathrm{D}\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))\mathrm{D}_{\mathbf{y}}\boldsymbol{\Phi}(t, \mathbf{y}) \,.$$

Abbreviating $\mathbf{W}(t, \mathbf{y}) := \mathrm{D}_{\mathbf{y}}\boldsymbol{\Phi}(t, \mathbf{y})$ we can rewrite this as the non-autonomous ODE, the **variational equation**[1]

$$\dot{\mathbf{W}} = \mathrm{D}\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))\mathbf{W} \,. \tag{6.7.2}$$

Here, the state $\mathbf{y}$ can be regarded as a parameter. Since $\boldsymbol{\Phi}(0, \mathbf{y}) = \mathbf{y}$, we also know $\mathbf{W}(0, \mathbf{y}) = \mathbf{I}$ (identity matrix), which supplies an initial condition for (6.7.2). In fact, we can even merge (6.7.1) and (6.7.2) into the ODE

$$\frac{d}{dt}[\mathbf{y}(\cdot) \,,\, \mathbf{W}(\cdot, \mathbf{y}_0)] = [\mathbf{f}(\mathbf{y}(t)) \,,\, \mathrm{D}\mathbf{f}(\mathbf{y}(t))\mathbf{W}(t, \mathbf{y}_0)] \,, \tag{6.7.3}$$

with state space $\mathbb{R}^N \times \mathbb{R}^{N,N}$, which is autonomous again.

Now let us apply (6.7.2)/(6.7.3). As in [Lecture → Ex. 6.1.2.5], we consider the following autonomous Lotka-Volterra differential equation of a predator-prey model

$$\begin{aligned} \dot{u} &= (2 - v)u \,, \\ \dot{v} &= (u - 1)v \,, \end{aligned} \tag{6.7.4}$$

on the state space $D = \mathbb{R}^2_+$, $\mathbb{R}_+ = \{\xi \in \mathbb{R} : \xi > 0\}$. All the solutions of (6.7.4) are periodic and their period depends on the initial state $[u(0), v(0)]^T$. In this exercise we want to develop a numerical method which computes a suitable initial condition for a given period.

[1] Please do not mix up with the other meaning of "variational equation" as introduced in [Lecture → Chapter 1], [Lecture → Def. 1.4.1.6]. This is an instance of ambiguous terminology, which as widespread as it is annoying in mathematics.

(6-7.a) ⚀ (5 min.) For fixed state $\mathbf{y} \in D$, (6.7.2) represents an ODE. What is its state space?

SOLUTION for (6-7.a) → 6-7-1-0:Init1h.pdf ▲

(6-7.b) ⚁ (10 min.) What is the right hand side function for the ODE (6.7.2), when the underlying autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is the Lotka-Volterra ODE (6.7.4)? You may write $u(t), v(t)$ for solutions of (6.7.4).

SOLUTION for (6-7.b) → 6-7-2-0:Init2h.pdf ▲

(6-7.c) ⚂ (10 min.) From now on we write $\boldsymbol{\Phi} : \mathbb{R} \times \mathbb{R}^2_+ \to \mathbb{R}^2_+$ for the evolution operator (→ [Lecture → Def. 6.1.4.3]) associated with (6.7.4). Based on $\boldsymbol{\Phi}$ derive a non-trivial function $\mathbf{F} : \mathbb{R}^2_+ \to \mathbb{R}^2$, $\mathbf{F} \not\equiv \mathbf{0}$, which evaluates to zero for the input $\mathbf{y}_0$ if the period of the solution of system (6.7.4) with initial value

$$\mathbf{y}_0 = \begin{bmatrix} u(0) \\ v(0) \end{bmatrix}$$

is equal to a given value $T_P > 0$.

SOLUTION for (6-7.c) → 6-7-3-0:Init4h.pdf ▲

(6-7.d) ⚁ (5 min.) [depends on Sub-problem (6-7.c)]

We write $\mathbf{W}(T, \mathbf{y}_0)$, $T \geq 0$, $\mathbf{y}_0 \in \mathbb{R}^2_+$ for the solution of (6.7.2) for the underlying ODE (6.7.4). Express the Jacobian of $\mathbf{F}$ by means of $\mathbf{W}$.

SOLUTION for (6-7.d) → 6-7-4-0:Init5h.pdf ▲

(6-7.e) ⚄ (10 min.) [depends on Sub-problem (6-7.c)]

Argue, why the solution of $\mathbf{F}(\mathbf{y}) = 0$ will, in general, not be unique. When will it be unique?

HIDDEN HINT 1 for (6-7.e) → 6-7-5-0:Init1s.pdf

SOLUTION for (6-7.e) → 6-7-5-1:Init6h.pdf ▲

A C++ implementation **Ode45** of an adaptive embedded Runge-Kutta method has been presented in [Lecture → § 6.5.3.3] and the code is available ⇝ GitLab.

Instructions on the use of Ode45, see [Lecture → § 6.5.3.3], ⇝ GitLab.

The class **Ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` ⇝ GitLab defines the class **Ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 6.5.3] and [Lecture → Ex. 6.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 6.5.2.11].

1. Construct an object of type **Ode45**: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

```
template <class StateType,
class RhsType = std::function<StateType(const StateType &)>>
class Ode45 {
  public:
  Ode45(const RhsType &rhs);
  // ......
}
```

   Template parameters are

   - **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`,

+= and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.

- **RhsType**: type of rhs function (automatically deduced).

The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator()(const StateType & vec);
```

It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **Ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is `10e-6`)
- `atol`: absolute tolerance for error control (default is `10e-8`)

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 = $ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

(6-7.f) ⚀ (45 min.) [depends on Sub-problem (6-7.b)]

Relying on **Ode45**, implement a C++ function

```
std::pair<Vector2d,Matrix2d> PhiAndW(double u0, double v0, double
  T)
```

that computes $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and $\mathbf{W}(T, [u_0, v_0]^T)$. The first component of the output pair should contain $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and the second component the matrix $\mathbf{W}(T, [u_0, v_0]^T)$.

Set relative and absolute tolerances of **Ode45** to 10^{-14} and 10^{-12}, respectively.

HIDDEN HINT 1 for (6-7.f) → `6-7-6-0:Init2s.pdf`

SOLUTION for (6-7.f) → `6-7-6-1:Init7h.pdf` ▲

(6-7.g) ⚄ (45 min.) [depends on Sub-problem (6-7.c), Sub-problem (6-7.f) and Sub-problem (6-7.d)]

Relying on `PhiAndW()`, write in `initcondlv_main.cc` a C++ code that determines and returns initial conditions $u(0)$ and $v(0)$ such that the solution of the system (6.7.4) has period $T_P = 5$. Use the multi-dimensional **Newton method** for $\mathbf{F}(\mathbf{y}) = 0$ with $\mathbf{F}$ from Sub-problem (6-7.c). As your initial approximation, use $[3,2]^T$. Terminate the Newton iteration as soon as $\|\mathbf{F}(\mathbf{y})\| \leq 10^{-5}$. Validate your implementation by comparing the obtained initial data $\mathbf{y}$ with $\mathbf{\Phi}(100,\mathbf{y})$.

Remark. The residual based termination criterion recommended above [NumCSE course → § 8.2.3.4] is appropriate for this particular application and, in general, should not be used for Newton's method. Better termination criteria are proposed in [NumCSE course → Section 8.5.3].

HIDDEN HINT 1 for (6-7.g) → `6-7-7-0:lvxh.pdf`

SOLUTION for (6-7.g) → `6-7-7-1:Init8h.pdf` ▲

End Problem 6-7 , 130 min.

Problem 6-8: Integrating ODEs using the Taylor expansion method

In [Lecture → § 10.2.2.7] of the course we studied single step methods for the integration of initial value problems for ordinary differential equations $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, [Lecture → Def. 6.3.1.4]. Explicit single step methods have the advantage that they only rely on *point evaluations* of the right hand side $\mathbf{f}$. However, if derivatives of $\mathbf{f}$ are also available, one have more options and this problem examines a class of single-step methods that also rely on $D\mathbf{f}$.

▷ problem name/problem code folder: TaylorODE

The new class of methods is obtained by the following reasoning: if the right hand side $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ of an autonomous initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) , \qquad \mathbf{y}(0) = \mathbf{y}_0 , \tag{6.8.1}$$

with solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is smooth, the solution $\mathbf{y}(t)$ will also be regular and it is possible to expand it into a Taylor sum at $t = 0$, see [NumCSE course → Thm. 8.3.2.12],

$$\mathbf{y}(t) = \sum_{k=0}^{m} \frac{\mathbf{y}^{(k)}(0)}{k!} t^k + R_m(t) , \tag{6.8.2}$$

with remainder term $R_m(t) = O(t^{m+1})$ for $t \to 0$.

A single step method for the numerical integration of (6.8.1) can be obtained by choosing $m = 3$ in (6.8.2), neglecting the remainder term and taking the remaining sum as an approximation of $\mathbf{y}(h)$, that is

$$\mathbf{y}(h) \approx \mathbf{y}_1 := \mathbf{y}(0) + \frac{d\mathbf{y}(0)}{dt} h + \frac{1}{2} \frac{d^2\mathbf{y}(0)}{dt^2} h^2 + \frac{1}{6} \frac{d^3\mathbf{y}(0)}{dt^3} h^3 .$$

Subsequently, one uses the ODE and the initial condition to replace the temporal derivatives $\frac{d^i\mathbf{y}}{dt^i}$ with expressions in terms of (derivatives of) $\mathbf{f}$. This yields a single step integration method called *Taylor (expansion) method*.

(6-8.a) ⚀ (10 min.) Express $\frac{d\mathbf{y}}{dt}(t)$ and $\frac{d^2\mathbf{y}}{dt^2}(t)$ in terms of $\mathbf{f}$ and its Jacobian $\mathbf{D}\mathbf{f}$.

HIDDEN HINT 1 for (6-8.a) → 6-8-1-0:Tayl1h.pdf

SOLUTION for (6-8.a) → 6-8-1-1:Tayl1s.pdf ▲

(6-8.b) ⚄ (45 min.) [depends on Sub-problem (6-8.a)]

Verify the formula

$$\frac{d^3\mathbf{y}}{dt^3}(0) = \mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{f}(\mathbf{y}_0), \mathbf{f}(\mathbf{y}_0)) + \mathbf{D}\mathbf{f}(\mathbf{y}_0)^2\mathbf{f}(\mathbf{y}_0) . \tag{6.8.3}$$

HIDDEN HINT 1 for (6-8.b) → 6-8-2-0:Tayl0h.pdf

HIDDEN HINT 2 for (6-8.b) → 6-8-2-1:Tayl2h.pdf

HIDDEN HINT 3 for (6-8.b) → 6-8-2-2:Tayl24h.pdf

SOLUTION for (6-8.b) → 6-8-2-3:Tayl2s.pdf ▲

(6-8.c) ⚄ (45 min.) Now apply the Taylor expansion method introduced above to the following ODE for the *predator-prey* model, as introduced in [Lecture → Ex. 6.1.2.5]:

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t)) y_1(t) , \tag{6.8.4}$$

$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2) y_2(t)\,, \tag{6.8.5}$$
$$\mathbf{y}(0) = [100, 5]^\top\,. \tag{6.8.6}$$

To this end, in `taylorode.cc` implement a function C++ function

```
std::vector<Eigen::Vector2d> solvePredPreyTaylor(
  double alpha1, double beta1, double alpha2, double beta2,
  double T, const Eigen::Vector2d& y0, unsigned int M);
```

for the numerical integration of initial-value problems for (6.8.4) using the Taylor expansion method with M, $M \in \mathbb{N}$ uniform timesteps on the temporal interval $[0, T]$. The arguments `alpha1`, `beta1`, `alpha2`, `beta2` provide the parameters of the model (6.8.4), while `y0` passes the initial value, and `T` the final time. `M` specifies the number of equidistant timesteps to be carried out. The function should return a sequence of approximate states $\mathbf{y}_k$, $k = 0, \ldots, M$.

SOLUTION for (6-8.c) → 6-8-3-0:Tayl3s.pdf ▲

(6-8.d) ⚄ (20 min.) [depends on Sub-problem (6-8.c)]

Based on the function `solvePredPreyTaylor()` implemented in Sub-problem (6-8.c) experimentally determine the order of convergence of the considered Taylor expansion method when it is applied to solve (6.8.4). Study the behaviour of the error at final time $t = 10$ for the initial data $\vec{y}(0) = [100, 5]^\top$. Implement a function

```
double testCvgTaylorMethod();
```

that generates a suitable error table and returns the empiric rate of algebraic convergence in terms of the (uniform) timestep $h \to 0$. This rate should be determined by linear regression.

As exact solution use $\mathbf{y}(10) = [0.319465882659820, 9.730809352326228]^\top$

SOLUTION for (6-8.d) → 6-8-4-0:Tayl4s.pdf ▲

(6-8.e) ⚄ (10 min.) What is the disadvantage of the Taylor's method compared with a Runge-Kutta method?

SOLUTION for (6-8.e) → 6-8-5-0:Tayl5s.pdf ▲

End Problem 6-8, 130 min.

Problem 6-9: Drawing Isolines of a Function

Isoline/countour plots of functions are a popular way to visualize real-valued functions or data defined on a subset of the plane. In this exercise we will use numerical integration for finding isolines.

This problem assumes familiarity with explicit adaptive Runge-Kutta single-step methods as introduced in [Lecture → Section 6.4] and [Lecture → Section 6.5]. It relies on the **Ode45** integrator class from [Lecture → § 6.5.3.3].
▷ problem name/problem code folder: ContourPlot

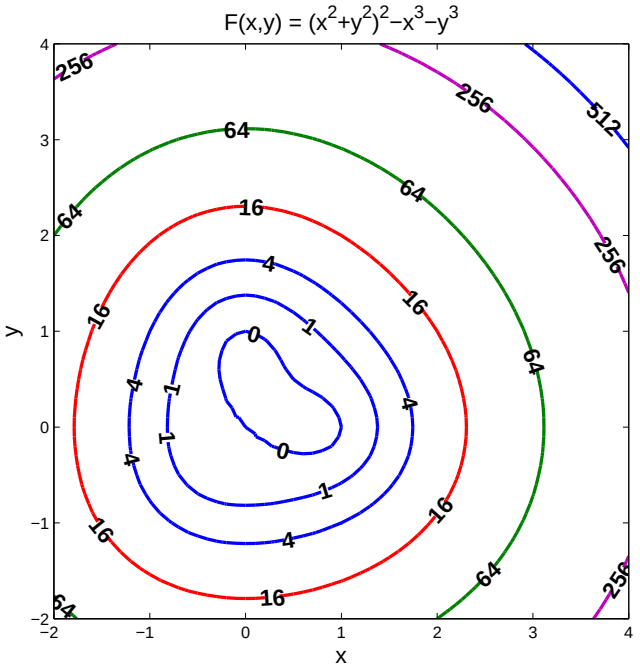


Fig. 112

An isoline is a curve in the plane $\mathbb{R}^2$ that can be defined as the set of zeros of a continuous function $F : \mathbb{R}^2 \mapsto \mathbb{R}$:

$$\mathcal{C} := \{\boldsymbol{x} \in \mathbb{R}^2 : \ F(\boldsymbol{x}) = 0\} \ .$$

In PYTHON (matplotlib) or MATLAB they can be plotted with the function `contour`

◁ Contour plot of the function

$$F(x,y) = (x^2+y^2)^2 - x^3 - y^3$$

If F is continuously differentiable, we can obtain connected components of $\mathcal{C}$ as solution curves of the initial value problems

$$\dot{\mathbf{y}}(t) = \frac{\mathbf{grad}\, F(\mathbf{y}(t))^{\perp}}{\|\mathbf{grad}\, F(\mathbf{y}(t))\|} \ , \quad F(\mathbf{y}(0)) = 0 \ . \tag{6.9.1}$$

Here $^{\perp}$ stands for the counterclockwise rotation of a vector $\in \mathbb{R}^2$ by $90°$, that is,

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\perp} = \begin{bmatrix} -y \\ x \end{bmatrix} \ , \quad x,y \in \mathbb{R} \ . \tag{6.9.2}$$

Then (6.9.1) is a consequence of the fact that an isoline always runs perpendicular to the direction of steepest ascent of F, which is provided by $\mathbf{grad}\, F$.

(6-9.a) ⚀ (10 min.) State the ordinary differential equation (6.9.1) for the concrete case of the function

$$F : \mathbb{R}^2 \to \mathbb{R} \ , \quad F(x,y) := (x^2+y^2)^2 - x^3 - y^3 \ . \tag{6.9.3}$$

SOLUTION for (6-9.a) → `6-9-1-0:s1.pdf` ▲

(6-9.b) ⚁ (30 min.) Use (with `rtol = 1e-6, atol = 1e-9`) the provided EIGEN-based integrator class **Ode45** [Lecture → § 6.5.3.3] that employs an embedded explicit Runge-Kutta pair of methods to implement (in the file `contourplot.h`) a C++ function

```
template<typename GradientFunctor>
Eigen::Matrix<double,2,Eigen::Dynamic>
computeIsolinePoints(GradientFunctor &&gradF,
Eigen::Vector2d y0, double T);
```

that solves an initial value problem for (6.9.1) with initial state $\mathbf{y}_0 \in \mathbb{R}^2$ passed in `y0` from intial time 0 up to final time T (given in `T`). The argument `gradF` supplies a functor object equipped with an evaluation operator

```
Eigen::Vector2d operator(Eigen::Vector2d x) const;
```

that returns $\mathbf{grad}\, F(\boldsymbol{x})$. The function `computeIsolinePoints()` should return the sequence of computed states $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_M$, $M \in \mathbb{N}$, in the columns of an $2 \times (M+1)$-matrix, where M is the number of steps taken by the adaptive integrator of **Ode45**.

Instructions on the use of Ode45, see [Lecture → § 6.5.3.3], ↬ GitLab.

The class **Ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` ↬ GitLab defines the class **Ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 6.5.3] and [Lecture → Ex. 6.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 6.5.2.11].

1. Construct an object of type **Ode45**: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

   ```
   template <class StateType,
   class RhsType = std::function<StateType(const StateType &)>>
   class Ode45 {
     public:
     Ode45(const RhsType &rhs);
     // ......
   }
   ```

   Template parameters are

   - **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
   - **RhsType**: type of rhs function (automatically deduced).

   The argument `rhs` must be of a functor type that provides an evaluation operator

   ```
   StateType operator()(const StateType & vec);
   ```

   It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **Ode45.options** to configure the solver:

   ```
   O.options.<option_you_want_to_set> = <value>;
   ```

   Examples:

   - `rtol`: relative tolerance for error control (default is `10e-6`)
   - `atol`: absolute tolerance for error control (default is `10e-8`)

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

SOLUTION for (6-9.b) → 6-9-2-0:s2.pdf ▲

(6-9.c) ⚀ (15 min.) [depends on Sub-problem (6-9.a) and Sub-problem (6-9.b)]

The 0-isoline of F from (6.9.3)

$$\mathcal{C}_{\text{egg}} := \{\boldsymbol{x} = (x,y) \in \mathbb{R}^2 \setminus \{\mathbf{0}\} : F(\boldsymbol{x}) := (x^2+y^2)^2 - x^3 - y^3 = 0\} .$$

is known as "crooked egg curve". Based on `computeIsolinePoints()` and with $\mathbf{y}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \in \mathcal{C}_{\text{egg}}$ and $T = 4$ implement a C++ function (in the file `contourplot.cc`)

```
Eigen::Matrix<double,2,Eigen::Dynamic> crookedEgg();
```

that returns coordinates of densely spaced points $\in \mathcal{C}_{\text{egg}}$ in the columns of the produced matrix.

HIDDEN HINT 1 for (6-9.c) → 6-9-3-0:s3h1.pdf

SOLUTION for (6-9.c) → 6-9-3-1:s3.pdf ▲

(6-9.d) ⚄ (30 min.) [depends on Sub-problem (6-9.b)]

Again rely on the integrator class **Ode45** to implement (in the file `contourplot.h`) a function

```
template<typename FFunctor>
Eigen::Matrix<double,2,Eigen::Dynamic>
computeIsolinePointsDQ(FFunctor &&F,
Eigen::Vector2d y0, double T);
```

that performs the same computation as `computeIsolinePoints()` from Sub-problem (6-9.b). However, this time the argument `F` merely contains a functor object providing the function F itself.

To obtain an approximation of $\mathbf{grad}\, F$ rely on an approximation by **symmetric difference quotients**,

e.g.,

$$\frac{\partial F}{\partial x_1}(\boldsymbol{x}) \approx \frac{F(\boldsymbol{x} + \begin{bmatrix} h \\ 0 \end{bmatrix}) - F(\boldsymbol{x} - \begin{bmatrix} h \\ 0 \end{bmatrix})}{2h} \quad \text{for some} \quad h > 0 \,. \tag{6.9.8}$$

Choose a suitable fixed $h > 0$ and explain your choice in a comment in your code.

HIDDEN HINT 1 for (6-9.d) → `6-9-4-0:h4.pdf`

HIDDEN HINT 2 for (6-9.d) → `6-9-4-1:h42.pdf`

SOLUTION for (6-9.d) → `6-9-4-2:s4.pdf` ▲

(6-9.e) ⚀ (10 min.) [depends on Sub-problem (6-9.c) and Sub-problem (6-9.d)]

The function `main()` in the file `contourplot_main.cc` uses your implemetation of `crookedEgg()` and `computeIsolinePointsDQ()` to plot some isolines of F. The image generated via the PYTHON script `contourplot.py` is called `contourplot.eps`. Compare the plot of the isolines with Fig. 112 to verify if your solution is correct.

SOLUTION for (6-9.e) → `6-9-5-0:s5.pdf` ▲

End Problem 6-9, 95 min.

Problem 6-10: Symplectic Timestepping for Equations of Motion

This problem examines a special class of timestepping schemes suitable for solving numerically equations of motions from classical mechanics. These timestepping schemes preserve profound stuctural properties of those equations and have been dubbed **symplectic**, [HLW06, Chapter VI] and [LR04].

This problem considers a particular instance of a single-step method as introduced in [Lecture → Section 6.3].
▷ problem name/problem code folder: SymplecticTimestepping

The authors of [RWR04] consider ordinary differential equations on the state space $\mathbb{R}^{2n}$, $n \in \mathbb{N}$, of the form

$$\frac{d}{dt}\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix} = \begin{bmatrix}\mathbf{f}(\mathbf{q},t)\\ \mathbf{g}(\mathbf{p})\end{bmatrix} \quad \text{with continuous functions} \quad \begin{aligned}&\mathbf{f}:\mathbb{R}^n\times\mathbb{R}\to\mathbb{R}^n\,,\\ &\mathbf{g}:\mathbb{R}^n\to\mathbb{R}^n\,.\end{aligned} \tag{6.10.1}$$

For the temporal discretization of (6.10.1), [RWR04, Sect. IV] proposes the following class of s-stage methods, $s \in \mathbb{N}$, with uniform timestep $h > 0$ for generating sequences $\left(\begin{bmatrix}\mathbf{p}_j\\ \mathbf{q}_j\end{bmatrix}\right)_{j=0,\ldots,M}$ of approximations of $\begin{bmatrix}\mathbf{p}(jh)\\ \mathbf{q}(jh)\end{bmatrix}$, starting from initial values $\mathbf{p}_0 := \mathbf{p}(0)$ and $\mathbf{q}_0 := \mathbf{q}(0)$:

$$\begin{aligned}
&\textbf{for}\ \ j := 1\ \ \textbf{to}\ \ M\ \ \textbf{do}\\
&\quad \mathbf{p}_{\mathrm{in}} := \mathbf{p}_{j-1}\,,\quad \mathbf{q}_{\mathrm{in}} := \mathbf{q}_{j-1}\\
&\quad \textbf{for}\ \ k := 1\ \ \textbf{to}\ \ s\ \ \textbf{do}\\
&\qquad t_k := jh + h\sum_{\ell=1}^{k-1} a_\ell\,;\\
&\qquad \mathbf{p}_{\mathrm{out}} := \mathbf{p}_{\mathrm{in}} + h\,b_k\,\mathbf{f}(\mathbf{q}_{\mathrm{in}}, t_k)\,;\\
&\qquad \mathbf{q}_{\mathrm{out}} := \mathbf{q}_{\mathrm{in}} + h\,a_k\,\mathbf{g}(\mathbf{p}_{\mathrm{out}})\,;\\
&\qquad \mathbf{p}_{\mathrm{in}} := \mathbf{p}_{\mathrm{out}}\,,\quad \mathbf{q}_{\mathrm{in}} := \mathbf{q}_{\mathrm{out}}\,;\\
&\quad \textbf{end}\\
&\quad \mathbf{p}_j := \mathbf{p}_{\mathrm{out}}\,,\quad \mathbf{q}_j := \mathbf{q}_{\mathrm{out}}\,;\\
&\textbf{end}
\end{aligned} \tag{6.10.2}$$

Here $a_1,\ldots,a_s$ and $b_1,\ldots,b_s$ are given coefficients, carefully chosen such that the single-step method achieves a certain order. A possible choice discussed in [RWR04] is, for $s = 3$,

$$a_1 := \tfrac{2}{3}\,,\quad a_2 := -\tfrac{2}{3}\,,\quad a_3 := 1\,,\quad b_1 := \tfrac{7}{24}\quad b_2 := \tfrac{3}{4}\,,\quad b_3 := -\tfrac{1}{24}\,. \tag{6.10.3}$$

(6-10.a) ⚄ (10 min.) [depends on Sub-problem (6-10.a)]

In `symplectictimestepping.cc`, write a C++ function

```
void sympTimestep(double tau, Eigen::Vector2d& pq_j);
```

that implements a single timestep $\begin{bmatrix}\mathbf{p}_j\\ \mathbf{q}_j\end{bmatrix} \mapsto \begin{bmatrix}\mathbf{p}_{j+1}\\ \mathbf{q}_{j+1}\end{bmatrix}$ of the method (6.10.2) with $s = 3$ and coefficients according to (6.10.3) for the "harmonic oscillator" ODE ($n = 1$)

$$\dot{p} = q \quad,\quad \dot{q} = -p\,. \tag{6.10.4}$$

The components of the argument vector supply p_j and q_j, and will be replaced with p_{j+1} and q_{j+1}.

SOLUTION for (6-10.a) → `6-10-1-0:symp4.pdf` ▲

(6-10.b) ⚁ (15 min.) [depends on Sub-problem (6-10.a)]

Experimentally determine the order of convergence of the single-step method implemented in Sub-problem (6-10.a) in terms of the number of timesteps by

- integrating (6.10.4) over the period $[0, 2\pi]$,
- with initial values $p(0) = 0$, $q(0) = 1$,
- and using $M = 10, 20, 40, 80, 160, 320, 640$ timesteps.

Tabulate the following errors at the final time:

$$\mathrm{err}(M) := |p(2\pi) - p^{(M)}| + |q(2\pi) - q^{(M)}| , \quad M = 10, 20, 40, 80, 160, 320, 640 , .$$

where $p^{(M)}$ and $q^{(M)}$ are the approximations of $p(2\pi)$ and $q(2\pi)$ produced by the single-step method. To that end complete the function `sympTimesteppingODETest()` in the file `symplectictimestepping.cc` , which is called from `main()`.

HIDDEN HINT 1 for (6-10.b) → `6-10-2-0:smp5h1.pdf`

SOLUTION for (6-10.b) → `6-10-2-1:symp5.pdf` ▲

The symplectic timestepping method has especially been designed for a special and very important class of evolution problems which arose from classical mechanics.

Definition 6.10.7. Hamiltonian ODEs

An ODE with state space $\mathbb{R}^{2n} \cong \mathbb{R}^n \times \mathbb{R}^n$ is called **Hamilonian**, if there is a splitting of solution components and a continuously differentiable function $H : \mathbb{R}^n \times \mathbb{R}^n$, $H = H(\mathbf{p}, \mathbf{q})$, such that it can be written as

$$\begin{aligned} \dot{\mathbf{p}}(t) &= -\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}(t), \mathbf{q}(t)) , \\ \dot{\mathbf{q}}(t) &= \frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}(t), \mathbf{q}(t)) \end{aligned} \quad \text{with} \quad H : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} . \tag{6.10.8}$$

The smooth function H is called a **Hamiltonian**.

Notation: The partial derivative vector $\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}_0, \mathbf{q}_0) \in \mathbb{R}^n$ stands for the gradient of the function $\mathbf{q} \mapsto H(\mathbf{p}_0, \mathbf{q})$ in $\mathbf{q} = \mathbf{q}_0$. $\frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}_0, \mathbf{q}_0)$ is defined analogously, see [Lecture → § 0.3.2.11].

(6-10.c) ⚂ (15 min.) Show that for every solution $t \mapsto (\mathbf{p}(t), \mathbf{q}(t))$, $t \in I \subset \mathbb{R}$, of (6.10.8) the function

$$t \mapsto E(t) := H(\mathbf{p}(t), \mathbf{q}(t)) , \quad t \in I , \tag{6.10.9}$$

is constant.

SOLUTION for (6-10.c) → `6-10-3-0:s6.pdf` ▲

(6-10.d) ⚁ (10 min.) Write down explicitly in the form (6.10.1) the Hamiltonian ODE (6.10.8) for the particular Hamiltonian

$$H(\mathbf{p}, \mathbf{q}) = \tfrac{1}{2}\|\mathbf{p}\|_2^2 + \|\mathbf{q}\|_2^4 , \quad \mathbf{p}, \mathbf{q} \in \mathbb{R}^n , \quad n \in \mathbb{N} . \tag{6.10.10}$$

SOLUTION for (6-10.d) → `6-10-4-0:s7.pdf` ▲

(6-10.e) ⚄ (30 min.) In `symplectictimestepping.cc`, implement a C++ function

```
Eigen::MatrixXd simulateHamiltonianDynamics(
  const Eigen::VectorXd &p0, const Eigen::VectorXd &q0,
  double T, unsigned int M);
```

that uses M equidistant steps of the sympletic timestepping scheme (6.10.2) + (6.10.3) in order solve the Hamilonian ODE (6.10.8) with Hamiltonian H as in (6.10.10) up to final time $T > 0$ and with initial values $\mathbf{p}_0$ and $\mathbf{q}_0$ passed in the argument vectors `p0` and `q0` of equal length $n \in \mathbb{N}$.

The function should return the numerical solution

$$\begin{bmatrix}\mathbf{p}_0\\ \mathbf{q}_0\end{bmatrix}, \begin{bmatrix}\mathbf{p}_1\\ \mathbf{q}_1\end{bmatrix}, \ldots, \begin{bmatrix}\mathbf{p}_M\\ \mathbf{q}_M\end{bmatrix},$$

in the columns of a $2n \times (M+1)$-matrix.

SOLUTION for (6-10.e) → `6-10-5-0:s8.pdf` ▲

(6-10.f) ⚄ (10 min.) The following table gives the quantities

$$\mathcal{E}(M) := \max\{E(\mathbf{p}_k, \mathbf{q}_k),\ k = 0, \ldots, M\}\,, \tag{6.10.13}$$

where E is the "energy" as defined in (6.10.9) and $\left(\begin{bmatrix}\mathbf{p}_k\\ \mathbf{q}_k\end{bmatrix}\right)_{k=0}^{M}$ is the sequence produced by `simulateHamiltonianDynamics()` for $n = 2$, $\mathbf{p}_0 = \mathbf{0}$, $\mathbf{q}_0 = \begin{bmatrix}1\\ 0\end{bmatrix}$, $T = 5$ and M timesteps.

M	$\mathcal{E}(M)$
10	1.136158638333363
20	1.009854310088357
40	1.001187287267320
80	1.000149549345033
160	1.000018794627444
320	1.000002358738605
640	1.000000295580299
1280	1.000000036994599

Based on the data in table describe qualitatively and quantitatively the expected asymptotic behavior of $\mathcal{E}(M)$ for $M \to \infty$.

SOLUTION for (6-10.f) → `6-10-6-0:s9.pdf` ▲

References

[HLW06] E. Hairer, C. Lubich, and G. Wanner. *Geometric numerical integration*. 2nd ed. Vol. 31. Springer Series in Computational Mathematics. Heidelberg: Springer, 2006 (cit. on p. 253).

[LR04] B. Leimkuhler and S. Reich. *Simulating Hamiltonian Dynamics*. Vol. 14. Cambridge Monographs on Applied and Computational Mathematics. Cambridge, UK: Cambridge University Press, 2004 (cit. on p. 253).

[RWR04] R. Rieben, D. White, and G. Rodrigue. "High Order Symplectic Integration Methods for Finite Element Solutions to Time Dependent Maxwell Equations". In: *IEEE Trans. Antennas and Propagation* 52.8 (2004), pp. 2190–2195 (cit. on p. 253).

End Problem 6-10, 90 min.

Chapter 7

Single Step Methods for Stiff Initial Value Problems

Problem 7-1: Implicit Runge-Kutta method

This problem addresses the implementation of general implicit Runge-Kutta methods [Lecture → Def. 7.3.3.1]. We will adapt an integrator class so far available for explicit Runge-Kutta single-step methods to the implicit case.

Related to Problem 6-2. This problem assumes familiarity with [Lecture → Section 7.3], and, especially, [Lecture → Section 7.3.3] and [Lecture → Rem. 7.3.3.9]. It practices the implementation of a full implicit RK-SSM in C++ based on EIGEN.
▷ problem name/problem code folder: ImplRK3Prey

Problem 6-2 introduced the class **RKIntegrator** that implemented the timestepping for a general explicit Runge-Kutta method according to [Lecture → Def. 6.4.0.9]. Keeping the interface we now extend this class so that it realizes a general Runge-Kutta timestepping method as given in [Lecture → Def. 7.3.3.1] for solving an autonomous initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

(7-1.a) ⚀ (15 min.) Rederive the **stage form**

$$\begin{aligned} \mathbf{g}_i &= h \sum_{j=1}^{s} a_{ij} \mathbf{f}(t_0 + c_j h, \mathbf{y}_0 + \mathbf{g}_j) \,, \\ \mathbf{y}_1 &= \mathbf{y}_0 + h \sum_{i=1}^{s} b_i \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_i) \,, \end{aligned} \qquad \text{[Lecture → (7.3.3.8)]}$$

of a general (implicit) Runge-Kutta method from the increment equations as given in [Lecture → Def. 7.3.3.1].

SOLUTION for (7-1.a) → 7-1-1-0:0s.pdf ▲

(7-1.b) ⚁ (15 min.) Let $\mathbf{g}_i$, $i = 1, \ldots, s$, be the so-called stages of a general Runge-Kutta method as defined in [Lecture → (7.3.3.7)]. In [Lecture → Rem. 7.3.3.9] the stages are recovered by solving a non-linear system of equations $F(\mathbf{g}) = \mathbf{0}$ with a suitable function $F : \mathbb{R}^{sN} \to \mathbb{R}^{sN}$. Formulate the Newton iteration for it when only autonomous ODEs $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with differentiable right-hand-side functions $\mathbf{f} : D \subset \mathbb{R}^N \to \mathbb{R}^N$ are considered.

SOLUTION for (7-1.b) → 7-1-2-0:0as.pdf ▲

(7-1.c) ⚂ (45 min.) [depends on Sub-problem (7-1.b)]

By modifying the class **RKIntegrator** for the implementation of explicit Runge-Kutta methods, design in `implicit_rkintegrator.hpp` a similar header-only C++ class **implicitRKIntegrator** which implements a general implicit RK method given through a Butcher scheme [Lecture → (7.3.3.3)] to solve the autonomous initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

```
class implicitRKIntegrator {
 public:
  implicitRKIntegrator(const Eigen::MatrixXd &A, const
     Eigen::VectorXd &b);
  template <class Function, class Jacobian>
  std::vector<Eigen::VectorXd> solve(Function &&f, Jacobian &&Jf,
                                     double T,
                                    const Eigen::VectorXd &y0,
                                    unsigned int N) const;
 private:
  .......
};
```

This class should implement a generic implicit Runge-Kutta single-step method for an autonomous ODE according to [Lecture → Def. 7.3.3.1]. The method is specified through its Butcher matrix $\mathcal{A} \in \mathbb{R}^{s,s}$ and the weight vector $\mathbf{b} \in \mathbb{R}^s$ that are passed to the constructor as arguments `A` and `b`.

The `solve()` method carries out N equidistant timesteps of the method for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on the time interval $[0, T]$, $T > 0$, and with initial value $\mathbf{y}_0$. The right-hand side function $\mathbf{f}$ and its Jacobian $D\mathbf{f}$ are passed through the functor objects `f` and `Jf` equipped with suitable evaluation operators.

In the `solve()` method the stages $\mathbf{g}_i$ as introduced in ((7-1.a)) are to be computed with the **damped Newton method** (see [NumCSE course → Section 8.5.4]) applied to the nonlinear system of equations satisfied by the stages (see [Lecture → Rem. 7.3.3.6] and [Lecture → Rem. 7.3.3.9]). Use the provided function

```
template <typename FuncType, typename JacType>
void dampnewton(const FuncType &F, const JacType &DF,
                Eigen::VectorXd &x,
                double rtol = 1e-4,
                double atol = 1e-6);
```

from the file `dampnewton.hpp`, that is a simplified version of [NumCSE course → Code 8.5.4.5]. Note that we do not use the simplified Newton method as discussed in [Lecture → Rem. 7.3.3.9].

In the code template you will find large parts of **implicitRKIntegrator** already implemented. In fact, you only have to write the method `step()` for the actual implicit RK-SSM timestepping.

SOLUTION for (7-1.c) → `7-1-3-0:Impl1h.pdf` ▲

(7-1.d) ⚀ (15 min.) Examine the following code

C++11-code 7.1.5: Initialization of implicitRKIntegrator object ↬ GitLab

Write down the complete Butcher scheme according to [Lecture → (7.3.3.3)] for the implicit Runge-Kutta method now available through `RK`. Which method is it? Is it A-stable [Lecture → Def. 7.3.4.9], L-stable [Lecture → Def. 9.2.7.46]?

HIDDEN HINT 1 for (7-1.d) → `7-1-4-0:Impl1s.pdf` ▲

SOLUTION for (7-1.d) → 7-1-4-1:Impl2h.pdf

(7-1.e) ⚄ (15 min.) [depends on Sub-problem (7-1.e)]

The file `main.cpp` uses your implementation in **implicitRKIntegrator** of general implicit RK-SSMs to solve an initial value problem for the Lotka-Volterra ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_1(\alpha_1 - \beta y_2) \\ y_2(-\alpha_2 + \beta y_2) \end{bmatrix} . \tag{7.1.6}$$

with the particular paramters $\alpha_1 = 3$, $\alpha_2 = 2$, $\beta = 0.1$. The right-hand-side function and its Jacobian are implemented as suitable lambda functions. Initial state is $\mathbf{y}_0 = [100\ 5]^\top$ and final time $T = 10$. The norm of the error at final time is tabulated.

Run the code. What information can be gleaned from the table?

SOLUTION for (7-1.e) → 7-1-5-0:Impl3h.pdf ▲

End Problem 7-1 , 105 min.

Problem 7-2: Damped precession of a magnetic needle

This problem deals with a dynamical system from mechanics describing the movement of a rod-like magnet in a strong magnetic field. This can be modelled by an ODE with a particular invariant that can become stiff in the case of large friction.

Assumes an idea about the notion of stiffness [Lecture → Notion 7.2.0.7] and knowledge of heuristic criteria for predicting stiffness of an IVP [Lecture → § 7.2.0.12]. Also looks at simple implicit and semi-implicit [Lecture → Section 7.4] single-step methods for a concrete ODE.
▷ problem name/problem code folder: CrossProd

We consider the initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c\mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1,1,1]^\top , \tag{7.2.1}$$

where $c > 0$ and $\mathbf{a} \in \mathbb{R}^3$, $\|\mathbf{a}\|_2 = 1$.

Note: $\mathbf{x} \times \mathbf{y}$ denotes the **cross product** of the two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$. It is defined by

$$\mathbf{x} \times \mathbf{y} = \begin{bmatrix} x_2y_3 - x_3y_2 \\ x_3y_1 - x_1y_3 \\ x_1y_2 - x_2y_1 \end{bmatrix} .$$

It satisfies $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$. In Eigen, it is available as `x.cross(y)` → EIGEN documentation, **#include** `<Eigen/Geometry>` is required.

(7-2.a) ⚀ (10 min.) Show that $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}_0\|_2$ for every solution $\mathbf{y}$ of the ODE.

HIDDEN HINT 1 for (7-2.a) → 7-2-1-0:Cros1s.pdf

SOLUTION for (7-2.a) → 7-2-1-1:Cros1h.pdf ▲

(7-2.b) ⚁ (20 min.) Compute the Jacobian $D\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{3,3}$ and its spectrum (= set of eigenvalues) $\sigma(D\mathbf{f}(\mathbf{y}))$ in the stationary state $\mathbf{y} = \mathbf{a}$, for which $\mathbf{f}(\mathbf{y}) = 0$. For simplicity, you may consider only the case $\mathbf{a} = [1,0,0]^\top$.

SOLUTION for (7-2.b) → 7-2-2-0:Cros2h.pdf ▲

(7-2.c) ⚀ (10 min.) [depends on Sub-problem (7-2.b)]

Discuss whether and when IVP for (7.2.1) will be stiff in the case $\mathbf{y}_0 \approx \mathbf{a}$.

HIDDEN HINT 1 for (7-2.c) → 7-2-3-0:stiffcrit.pdf

SOLUTION for (7-2.c) → 7-2-3-1:s22.pdf ▲

(7-2.d) ⚀ (10 min.) [depends on Sub-problem (7-2.b)]

For $\mathbf{a} = [1,0,0]^\top$, (7.2.1) was solved with the standard explicit adaptive Runge-Kutta numerical integrator **Ode45** and another A-stable semi-implicit adaptive numerical integrator up to the point $T = 10$ (same tolerances for the stepsize control). Explain the different dependence of the total number of steps from the parameter c observed in the figure.

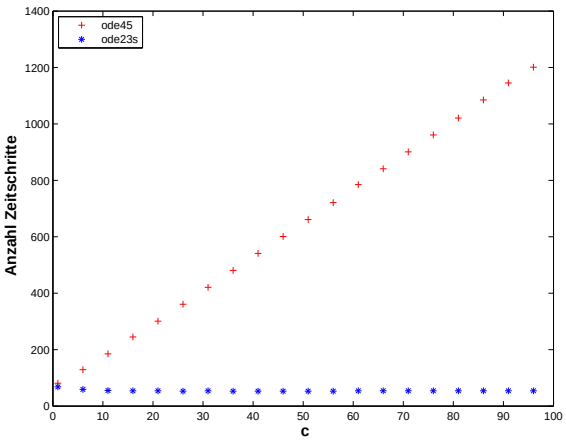


Fig. 115

SOLUTION for (7-2.d) → 7-2-4-0:Cros3h.pdf ▲

(7-2.e) ⚀ (10 min.) Formulate the non-linear equation resulting from the implicit mid-point method from [Lecture → Section 6.2.3] for the initial value problem (7.2.1).

SOLUTION for (7-2.e) → 7-2-5-0:Cros4h.pdf ▲

(7-2.f) ⚁ (20 min.) By means of the C++ function

```
void tab_crossprod(void);
```

in `crossprod.cc` , solve (7.2.1) with $\mathbf{a} = [1,0,0]^\top$, $c = 1$ up to $T = 10$. Use $M = 128$ uniform time steps of the implicit mid-point method. Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the implicit midpoint method. What do you observe?

For this task rely on the helper class **implicitRKIntegrator**:

```
class implicitRKIntegrator {
public:
  implicit_RKIntegrator(
    const Eigen::MatrixXd &A,
    const Eigen::VectorXd &b);
  template <class Function, class Jacobian>
  std::vector<Eigen::VectorXd> solve(
    Function &&f, Jacobian &&Jf, double T,
    const Eigen::VectorXd &y0,
    unsigned int M) const;
private:
  .......
};
```

It implements a generic implicit Runge-Kutta single-step method for an autonomous ODE according to [Lecture → Def. 7.3.3.1]. The method is specified through its Butcher matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the weight vector $\mathbf{b} \in \mathbb{R}^s$ that are passed to the constructor as arguments `A` and `b`.

The `solve()` method carries out M equidistant timesteps of the method for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on the time interval $[0,T]$, $T > 0$, and with initial value $\mathbf{y}_0$. The right-hand side function $\mathbf{f}$ and its Jacobian $D\mathbf{f}$ are passed through the functor objects `f` and `J` equipped with suitable evaluation operators. The `solve()` method relies on the damped Newton iteration to solve the stage equations, see [Lecture → Rem. 7.3.3.6] and [Lecture → Rem. 7.3.3.9].

SOLUTION for (7-2.f) → 7-2-6-0:Cros5h.pdf ▲

(7-2.g) ⚁ (20 min.) The so-called **linear-implicit mid-point method** can be obtained by a simple *linearization* of the (single) increment equation of the implicit mid-point method around the current state.

Equivalently, we can obtain it from the standard implicit midpoint method by carrying out a single step of the Newton iteration for the increment equations with initial guess $\mathbf{0}$.

Give the defining equation of the linear-implicit mid-point method for the general autonomous differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$$

with smooth right-hand-side function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$, $N \in \mathbb{N}$.

HIDDEN HINT 1 for (7-2.g) → 7-2-7-0:linhi.pdf

SOLUTION for (7-2.g) → 7-2-7-1:Cros6h.pdf ▲

(7-2.h) ⚂ (20 min.) [depends on Sub-problem (7-2.g)]

Implement the linear-implicit midpoint method in the C++ function

```
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solve_lin_mid(
  Function &&f, Jacobian &&Jf,
  double T, const Eigen::VectorXd &y0,
  unsigned int M);
```

The signature of this function is the same as that of the `solve()` method of **implicitRKIntegrator**.

Extend your implementation of `tab_crossprod()` so that it uses `solve_lin_mid()` to solve (7.2.1) with $\mathbf{a} = [1,0,0]^\top$, $c = 1$ up to $T = 10$ and $M = 128$. Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the linear implicit midpoint method. What do you observe?

SOLUTION for (7-2.h) → 7-2-8-0:Cros7h.pdf ▲

End Problem 7-2 , 120 min.

Problem 7-3: Singly Diagonally Implicit Runge-Kutta Method

SDIRK methods (**S**ingly **D**iagonally **I**mplicit **R**unge-**K**utta methods) are distinguished by Butcher schemes with a lower triangular coefficient matrix $\mathfrak{A}$ whose diagonal entries are all the same:

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} = \begin{array}{c|ccccc} c_1 & \gamma & & \cdots & & 0 \\ c_2 & a_{21} & \ddots & & & \vdots \\ \vdots & \vdots & & & & \vdots \\ \vdots & \vdots & & \ddots & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ c_s & a_{s1} & \cdots & & a_{s,s-1} & \gamma \\ \hline & b_1 & \cdots & & b_{s-1} & b_s \end{array} , \tag{7.3.1}$$

with $\gamma \neq 0$. Their advantage is that systems of equations of the same structure have to be solved in order to compute the s increments sequentially.

Familiarity with [Lecture → Section 7.3.3] is required. This problem has a focus on stability theory.
▷ problem name/problem code folder: SDIRK

In this problem, the scalar linear initial value problem of second order

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{7.3.2}$$

should be solved numerically using a family of 2-stage SDIRK method described by the Butcher scheme

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array} , \tag{7.3.3}$$

where $\gamma > 0$ is a parameter.

(7-3.a) ⚀ (5 min.) Assume you want to solve an initial-value problem for an ordinary differential equation $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{f} : I \times D \subset \mathbb{R} \times \mathbb{R}^N \to \mathbb{R} \times \mathbb{R}^N$, numerically. Explain the benefit of using SDIRK-SSMs compared to using Gauss-Radau RK-SSMs as introduced in [Lecture → Ex. 7.3.4.21]. In what situations will this benefit matter much?

HIDDEN HINT 1 for (7-3.a) → 7-3-1-0:SDIR0h.pdf

SOLUTION for (7-3.a) → 7-3-1-1:SDIR0s.pdf ▲

(7-3.b) ⚁ (10 min.) State the equations for the increments $\mathbf{k}_1$ and $\mathbf{k}_2$ of the Runge-Kutta method (7.3.3) applied to the initial value problem corresponding to the differential equation $\dot{\boldsymbol{y}} = \mathbf{f}(t, \boldsymbol{y})$.

SOLUTION for (7-3.b) → 7-3-2-0:SDIR1s.pdf ▲

(7-3.c) ⚁ (20 min.) Show that the stability function $S(z)$ of the SDIRK method (7.3.3) is given by

$$S(z) = \frac{1 + z(1-2\gamma) + z^2(1/2 - 2\gamma + \gamma^2)}{(1-\gamma z)^2}$$

and plot the stability domain $\mathcal{S}_\Psi$ for the parameter value $\gamma = 1$ using the supplied PYTHON script `stabdomSDIRK.py`

SOLUTION for (7-3.c) → 7-3-3-0:SDIR2s.pdf ▲

(7-3.d) ⚂ (25 min.) Find out whether for $\gamma = 1$ the SDIRK RK-SSM (7.3.3) is

- A–stable,
- L–stable.

Argue, based on the following version of the maximum principle for analytic functions:

Lemma 7.3.4. Maximum principle for rational functions

*If a rational function $z \in \mathbb{C} \mapsto R(z)$ has no pole in the left half plane $\mathbb{C}^- := \{z \in \mathbb{C} : \operatorname{Re} z \leq 0\}$, then **either***

$$\max\{|R(z)| : z \in \mathbb{C}^-\} = \max\{|R(z)| : \operatorname{Re} z = 0\}$$

or

$$\min\{|R(z)| : z \in \mathbb{C}^-\} = \min\{|R(z)| : \operatorname{Re} z = 0\}\,,$$

that is, the function attains on the imaginary axis either its maximum or its minimum over the whole left half plane.

SOLUTION for (7-3.d) → 7-3-4-0:SDIRxs.pdf ▲

(7-3.e) ⚀ (10 min.) Formulate (7.3.2) as an initial value problem for a linear first order system for the function $\boldsymbol{z}(t) = (y(t), \dot{y}(t))^\top$.

SOLUTION for (7-3.e) → 7-3-5-0:SDIR3s.pdf ▲

(7-3.f) ⚂ (20 min.) [depends on Sub-problem (7-3.e)]

Implement a C++ function

```
Eigen::Vector2d SdirkStep(const Eigen::Vector2d &z0, double h,
  double gamma);
```

that realizes one step of the method (7.3.3) for the linear differential equation (7.3.2), starting from the initial state `z0` and returning the state after a single step of size `h`.

SOLUTION for (7-3.f) → 7-3-6-0:SDIR4s.pdf ▲

(7-3.g) ⚂ (25 min.) [depends on Sub-problem (7-3.f)]

Realize a C++ function

```
double CvgSDIRK();
```

to conduct a numerical experiment, which gives an indication of the order of the method (with $\gamma = \frac{3+\sqrt{3}}{6}$) for the initial value problem from (7.3.3). Choose $\mathbf{z}_0 = [1,0]^\top$ as initial value, `T=10` as end time and `M=20,40,80,...,10240` as numbers of equidistant timesteps. Tabulate the error $|y(T) - y_M|$ ($y := (\mathbf{z})_1$) at final time and use it as a basis for the estimation of the rate of algebraic convergence by means of the supplied function `polyfit()`.

HIDDEN HINT 1 for (7-3.g) → 7-3-7-0:sd5h1.pdf

SOLUTION for (7-3.g) → 7-3-7-1:SDIR5s.pdf ▲

End Problem 7-3, 115 min.

Problem 7-4: Semi-implicit Runge-Kutta SSM

General implicit Runge-Kutta methods as introduced in [Lecture → Section 7.3.3] entail solving systems of non-linear equations for the increments, see [Lecture → Rem. 7.3.3.9]. Semi-implicit Runge-Kutta single step methods, also known as Rosenbrock-Wanner (ROW) methods [Lecture → (7.4.0.9)] just require the solution of linear systems of equations. This problem deals with a concrete ROW method, its stability and aspects of its implementation.

Relies on the contents of [Lecture → Section 7.1] and also depends on [Lecture → Section 7.3.4].
▷ problem name/problem code folder: SemImpRK

We consider the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \tag{7.4.1}$$

and discretize it with a *semi-implicit* Runge-Kutta SSM: a so-called **Rosenbrock method** given by

$$\begin{cases} \mathbf{W}\mathbf{k}_1 = \mathbf{f}(\mathbf{y}_0)\,, \\ \mathbf{W}\mathbf{k}_2 = \mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\,\mathbf{k}_1) - ah\,\mathbf{J}\,\mathbf{k}_1\,, \\ \mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}_2\,, \end{cases} \tag{7.4.2}$$

where

$$\mathbf{J} = D\mathbf{f}(\mathbf{y}_0) \quad , \quad \mathbf{W} = \mathbf{I} - ah\,\mathbf{J} \quad , \quad a := \frac{1}{2+\sqrt{2}}\,.$$

(7-4.a) ⚀ (10 min.) In [Lecture → § 7.4.0.8] the general form of s-stage Rosenbrock-Wanner single-step methods for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ was given as

$$(\mathbf{I} - ha_{ii}\mathbf{J})\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h\sum_{j=1}^{i-1}(a_{ij}+d_{ij})\mathbf{k}_j) - h\mathbf{J}\sum_{j=1}^{i-1} d_{ij}\mathbf{k}_j\,, \quad \mathbf{J} = \mathrm{D}\mathbf{f}(\mathbf{y}_0)\,,$$
$$\mathbf{y}_1 := \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{k}_j\,. \qquad \text{[Lecture → (7.4.0.9)]}$$

What is s and are the coefficients a_{ij}, d_{ij}, and b_j for the method (7.4.2)?

SOLUTION for (7-4.a) → 7-4-1-0:simplA.pdf ▲

(7-4.b) ⚀ (15 min.) Refresh your knowledge about how you can compute the stability function of a Runge-Kutta single-step function.

SOLUTION for (7-4.b) → 7-4-2-0:.pdf ▲

(7-4.c) ⚀ (15 min.) [depends on Sub-problem (7-4.b)]

Compute the stability function S of the Rosenbrock method (7.4.2) for the linear scalar model ODE $\dot{y} = \lambda y, \lambda \in \mathbb{C}$. In other words, compute the (rational) function $S(z)$, such that

$$y_1 = S(z)y_0, \quad z := h\lambda,$$

when the method is applied to perform a single setp of size h starting from y_0.

SOLUTION for (7-4.c) → 7-4-3-0:SemI1s.pdf ▲

(7-4.d) ⚀ (15 min.) [depends on Sub-problem (7-4.c)]

Compute the first 4 terms of the Taylor expansion of $S(z)$ around $z = 0$. What is the maximal $q \in \mathbb{N}$ such that

$$|S(z) - \exp(z)| = O(|z|^q)$$

for $|z| \to 0$? Deduce the maximal possible order of the method (7.4.2).

HIDDEN HINT 1 for (7-4.d) → 7-4-4-0:SemI2h.pdf

SOLUTION for (7-4.d) → 7-4-4-1:SemI2s.pdf ▲

(7-4.e) ⚁ (30 min.) In `semimprk.h`, implement a C++ function:

```
template <class Func, class Jac>
std::vector<Eigen::VectorXd>
 SolveRosenbrock(Func &&f, Jac &&df,
                 const Eigen::VectorXd &y0,
                 unsigned int M, double T) ;
```

that applies the Rosenbrock method (7.4.2) for solving an initial-value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

It takes as input functor objects for $\mathbf{f}$ and $D\mathbf{f}$ (e.g., as lambda functions), an initial data (vector or scalar) `y0` $= \mathbf{y}(0)$, a number of steps M and a final time T. The function returns the sequence of states generated by the single step method up to $t = T$, using M equidistant steps of the Rosenbrock method.

SOLUTION for (7-4.e) → 7-4-5-0:SemI3s.pdf ▲

(7-4.f) ⚂ (15 min.) [depends on Sub-problem (7-4.e)]

Explore the order of the method (7.4.2) empirically by applying it to the IVP for the limit cycle [Lecture → Ex. 7.2.0.4]:

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\mathbf{y}\,, \tag{7.4.5}$$

with $\lambda = 1$ and initial state $\mathbf{y}_0 = [1,1]^\top$ on $[0,10]$. Use uniform timesteps of size $h = 2^{-k}, k = 4,\ldots,10$ and compute a reference solution $\mathbf{y}_{\mathrm{ref}}$ with timestep size $h = 2^{-12}$. Monitor the maximal error on the temporal mesh

$$\max_j \|\mathbf{y}_j - \mathbf{y}_{\mathrm{ref}}(t_j)\|_2\,,$$

and use it to estimate a rate of algebraic convergence by means of linear regression.

To do this write in `semimprk.cc` a C++ function

```
double CvgRosenbrock();
```

that tabulates the errors and returns the estimated rate of convergence.

SOLUTION for (7-4.f) → 7-4-6-0:SemI4s.pdf ▲

In complex analysis you might have hear about the maximum principle for holomorphic/analytic functions. The following is a special version of it.

Theorem 7.4.7. Maximum principle for holomorphic functions

Let

$$\mathbb{C}^- := \{z \in \mathbb{C} \mid Re(z) < 0\} .$$

Let $f : D \subset \mathbb{C} \to \mathbb{C}$ be non-constant, defined on $\overline{\mathbb{C}^-}$, and analytic in $\mathbb{C}^-$. Furthermore, assume that $w := \lim_{|z|\to\infty} f(z)$ exists and $w \in \mathbb{C}$, then:

$$\forall z \in \mathbb{C}^- : \quad |f(z)| < \sup_{\tau\in\mathbb{R}} |f(i\tau)| .$$

(7-4.g) ⚄ (20 min.) Appealing to Thm. 7.4.7 show that the method (7.4.2) is **L-stable** (*cf.* [Lecture → § 9.2.7.42]).

HIDDEN HINT 1 for (7-4.g) → 7-4-7-0:SemI5h.pdf

SOLUTION for (7-4.g) → 7-4-7-1:SemI5s.pdf ▲

End Problem 7-4 , 120 min.

Problem 7-5: Exponential integrator

The exponential integrators are a modern class of single step methods developed for special initial value problems (problems that can be regarded as perturbed linear ODEs), see

M. HOCHBRUCK AND A. OSTERMANN, *Exponential integrators*, Acta Numerica, 19 (2010), pp. 209–286.

These methods fit the concept of single step methods as introduced in [Lecture → Def. 6.3.1.4] and, usually, converge algebraically according to [Lecture → (6.3.2.7)].

You have to be familiar with fundamental concepts for single-step methods as introduced in [Lecture → Section 6.3.1], their asymptic convergence properties from [Lecture → Section 6.3.2] and the notion of a stability domain [Lecture → Def. 7.1.0.51].

▷ problem name/problem code folder: ExponentialIntegrator

A step with size h of the so-called **exponential Euler** single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with continuously differentiable $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$ reads:

$$\mathbf{y}_1 = \mathbf{y}_0 + h\, \varphi\big(h\mathrm{D}\mathbf{f}(\mathbf{y}_0)\big)\, \mathbf{f}(\mathbf{y}_0), \tag{7.5.1}$$

where $\mathrm{D}\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{N,N}$ is the Jacobian of $\mathbf{f}$ at $\mathbf{y} \in \mathbb{R}^N$, and the matrix function $\varphi : \mathbb{R}^{N,N} \to \mathbb{R}^{N,N}$ is defined as $\varphi(\mathbf{Z}) = (\exp(\mathbf{Z}) - \mathrm{Id})\, \mathbf{Z}^{-1}$. Here, $\exp(\mathbf{Z})$ is the matrix exponential of $\mathbf{Z}$, a special function $\exp : \mathbb{R}^{N,N} \to \mathbb{R}^{N,N}$, see [Lecture → (7.1.0.34)].

The function φ is implemented in the file `exponentialintegrator.cc` as the function

```
Eigen::MatrixXd phim(const Eigen::MatrixXd &Z);
```

When plugging in the exponential series, it is clear that the function $z \mapsto \varphi(z) := \frac{\exp(z)-1}{z}$ is analytic on $\mathbb{C}$ with $\varphi(0) = 1$. Thus, $\varphi(\mathbf{Z})$ is well defined for all matrices $\mathbf{Z} \in \mathbb{R}^{N,N}$.

(7-5.a) ⚀ (15 min.) Is the exponential Euler single step method defined in (7.5.1) consistent with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (see [Lecture → Def. 6.3.1.11])? Explain your answer.

SOLUTION for (7-5.a) → 7-5-1-0:Expo1h.pdf ▲

(7-5.b) ⚁ (15 min.) Show that the exponential Euler single step method defined in (7.5.1) solves the linear initial value problem

$$\dot{\mathbf{y}} = \mathbf{A}\,\mathbf{y}\,, \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^d\,, \qquad \mathbf{A} \in \mathbb{R}^{d,d}\,, \tag{7.5.2}$$

exactly.

HIDDEN HINT 1 for (7-5.b) → 7-5-2-0:Expo1s.pdf

SOLUTION for (7-5.b) → 7-5-2-1:Expo2h.pdf ▲

(7-5.c) ⚁ (15 min.) [depends on Sub-problem (7-5.b)]

Determine the region of stability of the exponential Euler single step method defined in (7.5.1) (see [Lecture → Def. 7.1.0.51]).

SOLUTION for (7-5.c) → 7-5-3-0:Expo3h.pdf ▲

(7-5.d) ⚁ (15 min.) In `exponentialintegrator.h`, write a C++ function

```
template <class Function, class Jacobian>
VectorXd exponentialEulerStep(const VectorXd &y0, Function &&f,
   Jacobian &&df, double h)
```

that implements (7.5.1). Here `f` and `df` are objects with evaluation operators representing the ODE right-hand side function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$ and its Jacobian, respectively.

SOLUTION for (7-5.d) → 7-5-4-0:Expo4h.pdf ▲

(7-5.e) ⚂ (15 min.) What is the order of the single step method (7.5.1)?

To investigate it empirically, in the file `exponentialintegrator.cc` , implement a C++ function

```
void testExpEulerLogODE();
```

that applies the method to the scalar logistic ODE

$$\dot{y} = y\,(1-y)\,, \quad y(0) = 0.1\,,$$

in the time interval $[0,1]$. Tabulate the error at the final time against the stepsize $h = T/M$, $M = 2^k$ for $k = 1,\ldots,15$. Discuss qualitatively and quantitative the empiric asymptotic convergence of the single-step method.

HIDDEN HINT 1 for (7-5.e) → 7-5-5-0:Expo3s.pdf

SOLUTION for (7-5.e) → 7-5-5-1:Expo5h.pdf ▲

End Problem 7-5 , 75 min.

Problem 7-6: Mono-implicit Runge-Kutta single step method

As explained in [Lecture → Section 7.3.4] we must resort to implicit single-step methods as solvers for stiff initial-value problems for ODEs. The big challenge about *implicit* Runge-Kutta single-step methods (RK-SSMs) is the need to solve possibly big non-linear systems of equations. Many attempts have been made to alleviate this drawback and this problem presents one of them.

This problem needs the linear model problem analysis as introduced in [Lecture → Section 7.1] and [Lecture → Section 7.3.4]. It practices the implementation of implicit single step methods with a focus on solving the non-linear systems of equations, *cf.* [Lecture → Rem. 7.3.3.9]. A little C++ implementation based on EIGEN is requested.
▷ problem name/problem code folder: MIRK

A so-called s-stage **mono-implicit Runge-Kutta single step method (MIRK)** for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subset \mathbb{R}^N \to \mathbb{R}^N$, is defined as:

$$\begin{aligned} \mathbf{g}_i &:= (1-v_i)\mathbf{y}_0 + v_i\mathbf{y}_1 + h\sum_{j=1}^{i-1} d_{i,j}\mathbf{f}(\mathbf{g}_j), \quad i = 1,\dots s\,, \\ \mathbf{y}_1 &:= \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{f}(\mathbf{g}_j) \end{aligned} \tag{7.6.1}$$

for suitable coefficients $v_i, b_i, d_{i,j} \in \mathbb{R}$, fixed to achieve a desired order.

(7-6.a) ⚀ (20 min.) Single step methods defined as in (7.6.1) belong to the class of implicit Runge-Kutta methods [Lecture → Def. 7.3.3.1]. Write down the corresponding Butcher scheme [Lecture → (7.3.3.3)] in terms of the coefficients $v_i, b_i, d_{i,j}$.

HIDDEN HINT 1 for (7-6.a) → 7-6-1-0:mirk0.pdf

SOLUTION for (7-6.a) → 7-6-1-1:MIRK1h.pdf

▲

(7-6.b) ⚁ (20 min.) Compute the stability function of a MIRK scheme defined as in (7.6.1) for $s = 2$.

HIDDEN HINT 1 for (7-6.b) → 7-6-2-0:MIRK1.pdf

SOLUTION for (7-6.b) → 7-6-2-1:MIRK2h.pdf

▲

(7-6.c) ⚂ (10 min.) Now, we consider the special case of a scalar ODE ($N = 1$) and $s = 2$. Abbreviating $\mathbf{z} := [g_1, g_2, y_1]^\top$, rewrite (7.6.1) as a non-linear system of equations in the form $\mathbf{F}(\mathbf{z}) = \mathbf{0}$ for an explicitly specified suitable function $\mathbf{F} : \mathbb{R}^3 \to \mathbb{R}^3$.

SOLUTION for (7-6.c) → 7-6-3-0:MIRK3h.pdf

▲

(7-6.d) ⚂ (10 min.) [depends on Sub-problem (7-6.c)]

Find the Jacobian $D\mathbf{F}(\mathbf{z})$ of the function $\mathbf{F}$ (in terms of $d_{i,j}, v_i, b_i$ and the derivative of f) from the previous sub-problem.

SOLUTION for (7-6.d) → 7-6-4-0:MIRK4h.pdf ▲

In the following sub-problems, we will implement the 2-stage MIRK scheme for the scalar case: $N = 1, s = 2$.

(7-6.e) ⚄ (25 min.) [depends on Sub-problem (7-6.c) and Sub-problem (7-6.c)]

In `mirk.h` , implement a function

```
template <class Func, class Jac>
Eigen::VectorXd Newton2Steps(Func &&F, Jac &&DF,
               Eigen::VectorXd z);
```

that approximates the solution of $\mathbf{F}(\mathbf{z}) = \mathbf{0}$ by performing two steps of the Newton method applied to $\mathbf{F}(\mathbf{z}) = \mathbf{0}$. Use z to pass the initial guess. Objects of type `Func` and `Jac` have to supply suitable evaluation operators **operator** `()` , which, for the type `Jac` must return an EIGEN-compatible matrix.

SOLUTION for (7-6.e) → `7-6-5-0:MIRK5h.pdf`

▲

(7-6.f) (15 min.) [depends on Sub-problem (7-6.e)]

Consider the particular MIRK scheme given by the coefficients:

$$v_1 = 1, \quad v_2 = \frac{344}{2025}, \quad d_{21} = -\frac{164}{2025}, \quad b_1 = \frac{37}{82}, \quad b_2 = \frac{45}{82}\,. \tag{7.6.6}$$

Using the function `Newton2Steps`, implement in `mirk.h` a function

```
template <class Func, class Jac>
double MIRKStep(Func &&f, Jac &&df, double y0, double h);
```

that realizes one step of the MIRK scheme defined by (7.6.1) for $s = 2$ and a scalar ODE, that is, $d = 1$. The right hand side function $\mathbf{f}$ and its Jacobian are passed through f and df. The solution of the nonlinear system arising from (7.6.1) is approximated using two Newton steps, namely by using the function `Newton2Steps()`. The initial guess has to be chosen appropriately!

SOLUTION for (7-6.f) → `7-6-6-0:MIRK6h.pdf` ▲

(7-6.g) ⚄ (15 min.) [depends on Sub-problem (7-6.f)]

Implement in `mirk.h` a function

```
template <class Func, class Jac>
double MIRKSolve(Func &&f, Jac &&df, double y0,
    double T, unsigned int M);
```

for the solution of a scalar ODE up to time T, using M equidistant steps of the mono-implicit Runge-Kutta single step method defined by (7.6.6). The initial value is passed in `y0`.

SOLUTION for (7-6.g) → `7-6-7-0:MIRK7h.pdf` ▲

(7-6.h) In `mirk_main.cc` , apply your implementation to the IVP

$$\dot{y} = 1 + y^2 \quad , \quad y(0) = 0\,, \tag{7.6.9}$$

on $[0,1]$. The exact solution of (7.6.9) is $y_{ex}(t) := \tan t$. Compute the solution y_M at $T = 1$ with a sequence of uniform temporal meshes with $M = 4, \ldots, 512$ intervals. Compute and output the error $|y_M(1) - y_{ex}(1)|$ and determine the rate of convergence of the scheme.

SOLUTION for (7-6.h) → `7-6-8-0:MIRK8h.pdf` ▲

End Problem 7-6 , 115 min.

Problem 7-7: Stability of a Runge-Kutta method

This problem is devoted to an empirical study of stability problems haunting explicit Runge-Kutta single-step methods, whose stability functions invariably are polynomials.

This problem is meant to supplement the discussion in [Lecture → Section 7.1].
▷ problem name/problem code folder: StabRK3

We focus on a 3-stage Runge-Kutta single step method described by the following Butcher-Tableau [Lecture → (6.4.0.11)].

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ \hline & 1/6 & 1/6 & 2/3 \end{array} \tag{7.7.1}$$

We also consider the following concrete case of the prey/predator model as introduced in [Lecture → Ex. 6.1.2.5]:

$$\begin{aligned} \dot{y}_1(t) &= (1 - y_2(t))y_1(t)\,, \\ \dot{y}_2(t) &= (y_1(t) - 1)y_2(t)\,. \end{aligned} \tag{7.7.2}$$

(7-7.a) ⚀ (30 min.) Create a C++ function

```
Eigen::Vector2d PredPrey(Eigen::Vector2d y0, double T, unsigned M);
```

that uses the RK-SSM (7.7.1) to solve an initial value problem for (7.7.2) with initial value $\mathbf{y}_0$ and M equidistant timestep up to final time $T > 0$. It should return $\mathbf{y}_N \approx \mathbf{y}(T)$.

SOLUTION for (7-7.a) → 7-7-1-0:Stab0s.pdf ▲

(7-7.b) ⚁ (20 min.) [depends on Sub-problem (7-7.a)]

Write a C++ function

```
void SimulatePredPrey();
```

to approximate the solution at time $T = 1$ of the IVP for (7.7.2) with initial value $\mathbf{y}_0 = \mathbf{y}(0) = [100, 1]^\top$. Use the RK-SSM (7.7.1) with uniform timestep $h > 0$ and the function from Sub-problem (7-7.a).

Qualitatively and quantitatively determine the type of asymptotic convergence of the method for uniform steps of size 2^{-j}, $j = 2, \ldots, 13$. As a reference solution, use an approximation obtained with 2^{14} steps. Tabulate the norm of the error at final time.

What do you notice for big step sizes? Try to find the maximum step size for which blow-up of the numerical solution can still be avoided.

HIDDEN HINT 1 for (7-7.b) → 7-7-2-0:Stab1h2.pdf

SOLUTION for (7-7.b) → 7-7-2-1:Stab1s.pdf ▲

(7-7.c) ⚁ (15 min.) Calculate the stability function $S = S(z)$, $z \in \mathbb{C}$, of the method given by the Butcher scheme (7.7.1).

SOLUTION for (7-7.c) → 7-7-3-0:Stab2s.pdf ▲

End Problem 7-7, 65 min.

Problem 7-8: Implicit Runge-Kutta Method for Gradient-Flow ODEs

In this problem we apply a special class of *implicit* Runge-Kutta single-step methods to a special class of ODEs, many of which give rise to *stiff* initial-value problems.

You need to master [Lecture → Section 7.3.3] and [Lecture → Section 7.3.4] and should also be familiar with the notion of a "stiff" IVP, see [Lecture → Section 7.2].
▷ problem name/problem code folder: GradientFlow

We consider the autonomous **gradient-flow ODE**

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}) := -\mathbf{grad}\, V(\mathbf{y}(t))\,, \tag{7.8.1}$$

where $V : D \subset \mathbb{R}^N \to \mathbb{R}$ is a continuously differentiable function.

(7-8.a) ⚄ (15 min.)

Show that for a solution $t \mapsto \mathbf{y}(t)$ of (7.8.1) defined on $I \subset \mathbb{R}$ the function $t \in I \mapsto V(\mathbf{y}(t))$ is non-increasing.

SOLUTION for (7-8.a) → 7-8-1-0:s1.pdf ▲

(7-8.b) ⚄ (10 min.)

The scalar ($N = 1$) linear ODE $\dot{y} = -\lambda y$, $\lambda \in \mathbb{R}$, belongs to the class of gradient-flow ODEs of the form (7.8.1). What is V in this case?

SOLUTION for (7-8.b) → 7-8-2-0:s2.pdf ▲

For the numerical integration of gradient-flow ODEs we opt for a so-called **SDIRK** (singly diagonally implicit Runge-Kutta) method, an implicit 5-stage Runge-Kutta method described by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^\top \end{array} \;\widehat{=}\; \begin{array}{c|ccccc} \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & 0 & 0 \\ \frac{1}{2} & \frac{371}{1360} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & 0 \\ 1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\ \hline & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \end{array} \tag{7.8.2}$$

In the file `gradientflow.cc` you find a function

```
Eigen::MatrixXd ButcherMatrix();
```

that gives you the 6×5-matrix $\begin{bmatrix} \mathfrak{A} \\ \mathbf{b}^\top \end{bmatrix}$ of the Butcher scheme (7.8.2). Its bottom row is the vector $\mathbf{b}^\top$.

(7-8.c) ⚄ (20 min.)

State the non-linear systems of equations of size $N \times N$ that have to be solved when applying the implicit Runge-Kutta single-step method (with timestep $h > 0$) described by the Butcher scheme (7.8.2) to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$.

SOLUTION for (7-8.c) → 7-8-3-0:s3.pdf ▲

(7-8.d) ⚄ (15 min.) [depends on Sub-problem (7-8.c)]

Give the **stage form** [Lecture → (7.3.3.8)] of the increment equations for the implicit Runge-Kutta single-step method (timestep $h > 0$) described by the Butcher scheme (7.8.2) and applied to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$.

SOLUTION for (7-8.d) → 7-8-4-0:s4.pdf ▲

(7-8.e) ⚄ (30 min.) [depends on Sub-problem (7-8.d)]

In the file `gradientflow.h` implement a C++ function

```
template <typename Func, typename Jac>
std::array<Eigen::VectorXd, 5> ComputeStages(
Func &&f, Jac &&df,
const Eigen::VectorXd &y, double h,
double rtol = 1E-6, double atol = 1E-8);
```

that uses a standard (undamped) *Newton iteration* (correction-based termination with relative tolerance `rtol` and absolute tolerance `atol`) to solve for the *stages* $\mathbf{g}_i$ (→ [Lecture → (7.3.3.8)])required for one step of size $h > 0$ of the implicit Runge-Kutta single-step method described by the Butcher scheme (7.8.2) and applied to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$.

The right-hand side function $\mathbf{f}$ and its Jacobian are passed through suitable functor objects `f` and `J` respectively, whereas `y` supplies the initial state for the step.

HIDDEN HINT 1 for (7-8.e) → 7-8-5-0:newt.pdf

SOLUTION for (7-8.e) → 7-8-5-1:s5.pdf ▲

(7-8.f) ⚄ (20 min.) [depends on Sub-problem (7-8.e)]

Based on `ComputeStages()` in `gradientflow.h` realize a a C++ function

```
template <typename Func, typename Jac>
Eigen::VectorXd DiscEvolSDIRK(
Func &&f, Jac &&df, const Eigen::VectorXd &y,
double h, double rtol = 1E-6, double atol = 1E-8);
```

that evaluates the discrete evolution operator $\Psi^h\mathbf{y}$ for the implicit Runge-Kutta single-step method described by the Butcher scheme (7.8.2) and applied to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$. Of course, the parameters `y` and `h` supply the arguments $\mathbf{y}$ and h for Ψ. The meaning of `f`, `J`, `rtol` and `atol` has been explained before

SOLUTION for (7-8.f) → 7-8-6-0:s6.pdf ▲

Finally we consider the concrete gradient-flow ODE (7.8.1) spawned by the potential function

$$V(\mathbf{y}) = \sin(\|\mathbf{y}\|^2) + \lambda(\mathbf{d}^\top\mathbf{y})^2\,, \quad \lambda \in \mathbb{R}\,, \quad \mathbf{y},\mathbf{d} \in \mathbb{R}^N\,, \quad \|\mathbf{d}\| = 1\,. \tag{7.8.8}$$

We tackle associated initial value-problems with initial state vector $\mathbf{y}_0$, $\|\mathbf{y}_0\| = 1$, and want to solve them on the time interval $[0, 0.1]$.

(7-8.g) ⚄ (10 min.)

Write down the autonomous gradient-flow ODE induced by $\mathbf{y} \mapsto V(\mathbf{y})$ from (7.8.8).

SOLUTION for (7-8.g) → 7-8-7-0:s6a.pdf ▲

(7-8.h) ⚄ (20 min.) [depends on Sub-problem (7-8.g)]

For what values of the parameter $\lambda \in \mathbb{R}$ do we face a *stiff* ODE for states $\mathbf{y}$ close to zero: $\|\mathbf{y}\| \ll 1$?

HIDDEN HINT 1 for (7-8.h) → 7-8-8-0:h7s.pdf

SOLUTION for (7-8.h) → 7-8-8-1:s7.pdf ▲

(7-8.i) ⚅ (15 min.) [depends on Sub-problem (7-8.f) and Sub-problem (7-8.g)]

Based on your implementation of `discEvolSDIRK()` in `gradientflow.cc` write a C++ function

```
std::vector<Eigen::VectorXd>
SolveGradientFlow(const Eigen::VectorXd &d,
double lambda,
const Eigen::VectorXd &y0,
double T, unsigned int M);
```

that uses M equidistant steps of the SDIRK RK-SSM with the Butcher scheme (7.8.2) to solve the gradient-flow ODE (7.8.1) spawned by V from (7.8.8) up to final time $T > 0$. The arguments `d` and `lambda` supply the parameters, and `y0` the initial state $\mathbf{y}^{(0)}$. Use the default tolerance parameters for Newton's method as specified in the signature of `ComputeStages()`.

HIDDEN HINT 1 for (7-8.i) → 7-8-9-0:h8lf.pdf

SOLUTION for (7-8.i) → 7-8-9-1:s8.pdf ▲

(7-8.j) ⚂ (5 min.)

We have solved the gradient-flow ODE (7.8.1) with V from (7.8.8) and $N = 2$, $\mathbf{d} = \begin{bmatrix}1\\0\end{bmatrix}$, $\lambda = 10$, $\mathbf{y}_0 = \begin{bmatrix}1\\0\end{bmatrix}$, $T = 0.1$.

N	error norm
10	5.99348e-07
20	3.65190e-08
40	2.25261e-09
80	1.39860e-10
160	8.71007e-12
320	5.41081e-13

The table list the norm of the error $\mathbf{y}_N - \mathbf{y}(T)$ at final time for different numbers N of equidistant timesteps.

Which empiric order of convergence of the SDIRK single-step method do the data suggest?

SOLUTION for (7-8.j) → 7-8-10-0:s9.pdf ▲

End Problem 7-8, 160 min.

Chapter 8

Structure-Preserving Numerical Integration

Chapter 9

Second-Order Linear Evolution Problems

Problem 9-1: Implicit Two-Stage Radau Runge-Kutta Single-Step Method for Parabolic IBVPs

This problem studies the full discretization of a linear second-order parabolic initial-boundary value problem as discussed in [Lecture → Section 9.2.1]. The focus will be on the implementation of higher-order implicit Runge-Kutta timestepping in the framework offered by LEHRFEM++.

This problem requires skills in LEHRFEM++. It assumes familiarity with the contents of [Lecture → Section 9.2.2], [Lecture → Section 9.2.4], [Lecture → Section 9.2.6.5], and [Lecture → Section 9.2.7.1].
▷ problem name/problem code folder: RadauThreeTimestepping

We consider the following parabolic initial-boundary value problem (IBVP) with homogeneous Dirichlet boundary conditions and zero initial conditions

$$\begin{aligned} \dot{u} - \Delta u &= f(\boldsymbol{x},t) && \text{in } \Omega\times]0,1[\,, \\ u &= 0 && \text{on } \partial\Omega\times]0,1[\,, \\ u &= 0 && \text{on } \Omega\times\{0\}\,, \end{aligned} \tag{9.1.1}$$

where the spatial domain Ω is the square $]-1,1[^2$. The time-dependent source function is given by

$$f(\boldsymbol{x},t) = \begin{cases} 1 & \text{, if } \left\| \boldsymbol{x} - \frac{1}{2}\begin{pmatrix}\cos \pi t \\ \sin \pi t\end{pmatrix}\right\| < \frac{1}{2}\,, \\ 0 & \text{elsewhere,} \end{cases} \qquad \boldsymbol{x} \in \Omega\,. \tag{9.1.2}$$

This models heating by a moving laser beam with circular cross section.

We pursue the method of lines approach, see [Lecture → Section 9.2.4]. Spatial discretization relies on linear Lagrangian finite elements on a *triangular* mesh $\mathcal{M}$, that is, $V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M})$, using a polygonal approximation of $\partial\Omega$.

RK-SSM for method-of-lines ODE

In [Lecture → Section 9.2.7.2] we learned that, when higher order timestepping schemes are desired for the initial value problem arising from the method of lines approach, L(π)-stable implicit Runge-Kutta single-step methods (→ [Lecture → Def. 9.2.7.46]) are the methods of choice.

One specimen of an L(π)-stable RK-SSM is the Radau-3 scheme, a 2-stage implicit Runge-Kutta method,

whose description via a Butcher scheme is given in [Lecture → (9.2.7.50)]:

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \qquad \text{[Lecture → (9.2.7.50)]}$$

(9-1.a) ⚀ (10 min.)

Following [Lecture → § 9.2.2.2] establish the the spatial variational formulation of the evolution problem (9.1.1).

SOLUTION for (9-1.a) → 9-1-1-0:m1s.pdf ▲

(9-1.b) ⚁ (20 min.) Derive the discrete evolution operator as introduced in

Definition [Lecture → Def. 9.2.6.2]. Discrete evolution operator

A discrete evolution operator belonging to a non-autonomous ODE $\dot{\mathbf{u}} = \mathbf{f}(t,\mathbf{u})$ on $I \times \mathbb{R}^N$ is a mapping $\boldsymbol{\Psi} : I \times I \times \mathbb{R}^N \to \mathbb{R}^N$ such that for all $t \in I$ and $\mathbf{u} \in \mathbb{R}^N$

$$\exists q \in \mathbb{N}_0\,,\ \tau_0 > 0: \quad \left\| \boldsymbol{\Psi}^{t,t+\tau}\mathbf{u} - \boldsymbol{\Phi}^{t,t+\tau}\mathbf{u} \right\| \le C(t,\mathbf{u})\tau^{q+1} \quad \forall |\tau| < \tau_0\,, \qquad \text{[Lecture → (9.2.6.3)]}$$

with $C = C(t,\mathbf{u})$ depending on $t, \mathbf{u}$ continuously, and $\boldsymbol{\Phi}$ the evolution operator (→ [Lecture → § 9.2.5.8]) associated with the ODE.
Terminology: The largest possible q in [Lecture → (9.2.6.3)] is called the order of the discrete evolution.

for the 2-stage Radau method, a Runge-Kutta single-step method (RK-SSM) defined by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \hat{=} \begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}\,, \qquad \text{[Lecture → (9.2.7.50)]}$$

when applied to the linear scalar ordinary differential equation $\dot{y} = -y$.

SOLUTION for (9-1.b) → 9-1-2-0:0as.pdf ▲

(9-1.c) ⚂ (20 min.) [depends on Sub-problem (9-1.b)]

Based on EIGEN implement a C++ function (in the file `radauthreetimesteppingode.cc`)

```
std::vector<double> twoStageRadauTimeSteppingLinScalODE(unsigned
    int m);
```

that uses $m \in \mathbb{N}$ equidistant steps of the 2-stage Radau method [Lecture → (9.2.7.50)] to solve the linear scalar initial value problem

$$\dot{y} = -y \quad , \quad y(0) = 1 \qquad (9.1.10)$$

on the time interval $[0,5]$ and returns the generated sequence $y^{(0)}, y^{(1)}, \ldots, y^{(m)}$, *cf.* [Lecture → Section 9.2.6.3].

SOLUTION for (9-1.c) → 9-1-3-0:s0b.pdf ▲

(9-1.d) ⚃ (15 min.) [depends on Sub-problem (9-1.c)]

Devise and implement a C++ function (also in the file `radauthreetimesteppingode.cc`)

```
void testConvergenceTwoStageRadauLinScalODE();
```

that produces an output (e.g., a table of the maximal error in the points of the temporal mesh as a function of the stepsize τ) that permits you to qualitatively and quantitatively assess the convergence of the 2-stage Radau implicit Runge-Kutta single step method for the initial-value problem (9.1.10). One way to proceed is to produce a sequence of N solutions with $m = 10 \cdot 2^k$ equidistant timesteps, $k = 1, ..., N$. What is the observed order of this single-step method, *cf.* [Lecture → Thm. 9.2.6.14]?

SOLUTION for (9-1.d) → 9-1-4-0:.pdf ▲

(9-1.e) ⚄ (20 min.) According to [Lecture → Def. 7.3.3.1], the application of s-stage one-step Runge-Kutta methods entails the computation of s increments in each timestep. For the linear ordinary differential equation

$$\mathbf{M}\left\{\frac{d}{dt}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu}(t) = \vec{\varphi}(t) \quad \text{for } 0 < t < T\,, \qquad \text{[Lecture → (9.2.4.4)]}$$

arising from the method of lines approach applied to a linear parabolic evolution problem, the increments can be obtained by solving a linear system of equations, see [Lecture → § 9.2.7.6] and [Lecture → (9.2.7.7)].

Describe this linear system of equations (in block form) for the RADAU-3 method of (9.1.1) using, as in [Lecture → (9.2.4.4)], the notations $\mathbf{A}$ and $\mathbf{M}$ for the $N \times N$ finite element Galerkin matrices associated with the bilinear form for $-\Delta$ and the $L^2(\Omega)$-inner product, respectively. Write $\vec{\varphi}(t)$ for the time-dependent right hand side vector obtained by Galerkin discretization of the source term.

SOLUTION for (9-1.e) → 9-1-5-0:.pdf ▲

Our next goal is to implement a class **Radau3MOLTimestepper**

```
class Radau3MOLTimestepper {
public:
// Disabled constructors
Radau3MOLTimestepper() = delete;
Radau3MOLTimestepper(const Radau3MOLTimestepper &) = delete;
Radau3MOLTimestepper(Radau3MOLTimestepper &&) = delete;
Radau3MOLTimestepper &operator=(const Radau3MOLTimestepper &) = delete;
Radau3MOLTimestepper &operator=(const Radau3MOLTimestepper &&) = delete;

// Main constructor; precomputations are done here
Radau3MOLTimestepper(const lf::assemble::DofHandler &dofh);

// Destructor
virtual ~Radau3MOLTimestepper() = default;

/* Class member functions */
// Discrete evolution operator for Radau IIA 3rd order
Eigen::VectorXd discreteEvolutionOperator(double time, double tau,
const Eigen::VectorXd &mu) const;

private:
const lf::assemble::DofHandler &dofh_;  // dangerous
//====================
// Your code goes here
//====================
};
```

The member function `Radau3MOLTimestepper::discreteEvolutionOperator()` is supposed to evaluate the discrete evolution operator $\boldsymbol{\Psi}^{t,t+\tau}$ [Lecture → Def. 9.2.6.2] for the 2-stage Radau RK-SSM, when applied to the method-of-lines ODE for the parabolic evolution problem (9.1.1). The argument `time` passes t, `tau` the timestep size $\tau > 0$, and `mu` the state vector, which a basis expansion coefficient vector for $\mathcal{S}_1^0(\mathcal{M})$ in this case.

For the sake of efficiency it is essential to *precompute* all the algebraic objects that are needed to evaluate the discrete evolution operator and are independent of τ. These precomputations have to be done in the constructor.

(9-1.f) ⚀ (45 min.) Using the LehrFEM++ library, in the file `radauthreetimestepping.cc` implement a function

```
Eigen::VectorXd rhsVectorheatSource(
    const lf::assemble::DofHandler &dofh,
    double time);
```

that computes the right-hand-side vector arising from the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin finite element discretization of the linear functional $v \mapsto \int_\Omega f(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ on the right-hand-side of the spatial variational formulation of (9.1.1), where $f(\boldsymbol{x},t)$ is given by (9.1.2). As usual, the argument `dofh` supplies information about the mesh and the local-to-global index mapping. Of course, local quadrature should has to be employed and we opt for the 2D trapezoidal rule

$$\int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{\sharp\mathcal{V}(K)} \sum_{\ell=1}^{\sharp\mathcal{V}(K)} \psi(\boldsymbol{a}_K^\ell) \tag{9.1.17}$$

for a cell $K \in \mathcal{M}$ with vertices $\boldsymbol{a}_K, \ldots, \boldsymbol{a}_K^{\sharp\mathcal{V}(K)}$ and area $|K|$.

As part of the specification we demand that all entries of the return vector corresponding to basis functions associated with nodes on $\partial\Omega$ are zero.

Requirement. Your code should rely on the LehrFEM++ assembly function `assembleVectorLocally()` [Lecture → Code 2.7.4.28]. To use it you have to write a simple class **TrapRuleLinFEElemVecProvider** compliant with **ENTITY_VECTOR_PROVIDER** in `radauthreetimestepping.h` ➙ GitLab

HIDDEN HINT 1 for (9-1.f) → `9-1-6-0:rhsh1.pdf`

SOLUTION for (9-1.f) → `9-1-6-1:srsh.pdf` ▲

(9-1.g) ⚀ (15 min.) Peruse again [Lecture → Ex. 2.7.4.22] and study [Lecture → Ex. 2.8.3.29] along with the documentation of **lf::uscalfe::ReactionDiffusionElementMatrixProvider**. Compare the latter with the LehrFEM++ class **lf::uscalfe::LinearFELaplaceElementMatrix**. Together with the implementation of **LinFEMassMatrixProvider** supplied in `radauthreetimestepping.cc` ➙ GitLab, this is an alternative to the more general **lf::uscalfe::ReactionDiffusionElementMatrixProvider**. ▲

(9-1.h) ⚁ (15 min.) Study the documentation of **lf::assemble::COOMatrix** and try to understand the following function, which follows the implementation of `lf::assemble::fix_flagged_solution_components()`.

C++11 code 9.1.21: Function `dropMatrixRowsColumns` for the isolation of solution components

```
1 template <typename SCALAR, typename SELECTOR>
2 void dropMatrixRowsColumns(SELECTOR &&selectvals,
3 lf::assemble::COOMatrix<SCALAR> &A) {
```

```
  const lf::assemble::size_type N(A.cols());
  LF_ASSERT_MSG(A.rows() == N, "Matrix must be square!");
  // Set the rows and columns of boundary DOFs to zero
  A.setZero([&selectvals](lf::assemble::gdof_idx_t i,
                          lf::assemble::gdof_idx_t j) {
            return (selectvals(i) || selectvals(j)); });
  // Set the diagonal entries of zeroed out rows and columns to 1
  for (lf::assemble::gdof_idx_t dofnum = 0; dofnum < N; ++dofnum) {
    const auto selval{selectvals(dofnum)};
    if (selval) A.AddToEntry(dofnum, dofnum, 1.0);
  }
}
```

SOLUTION for (9-1.h) → 9-1-8-0:s21.pdf ▲

(9-1.i) ⚄ (50 min.) [depends on Sub-problem (9-1.e) and Sub-problem (9-1.h)]

Decide which objects are worth precomputing and in which format, and then, in the file `radauthreetimestepping.cc`, implement the constuctor of the class **Radau3MOLTimestepper**.

A very handy tool is the function `Eigen::kroneckerProduct()`, see ➙ documentation, because the linear system of equations for the increments can be expressed using the Kronecker product of matrices as in [Lecture → (9.2.7.9)].

You may rely on the class **lf::uscalfe::ReactionDiffusionElementMatrixProvider** or use the simpler classes **lf::uscalfe::LinearFELaplaceElementMatrix** (provided by LehrFEM++) and **LinFEMassMatrixProvider** (provided in `radauthreetimestepping.cc` ➙ GitLab).

HIDDEN HINT 1 for (9-1.i) → 9-1-9-0:2h1.pdf

SOLUTION for (9-1.i) → 9-1-9-1:s5.pdf ▲

(9-1.j) ⚄ (35 min.) [depends on Sub-problem (9-1.e) and Sub-problem (9-1.f)]

In the file `radauthreetimestepping.cc` supply an efficient implementation of the member function `discreteEvolutionOperator()` of **Radau3MOLTimestepper**.

HIDDEN HINT 1 for (9-1.j) → 9-1-10-0:eh1.pdf

SOLUTION for (9-1.j) → 9-1-10-1:s3.pdf ▲

(9-1.k) ⚂ (15 min.) [depends on Sub-problem (9-1.j)]

Based on the class **Radau3MOLTimestepper** implement a C++ function (in the file `radauthreetimestepping.cc`)

```
Eigen::VectorXd solveHeatEvolution(
  const lf::assemble::DofHandler &dofh,
  unsigned int m, double final_time);
```

that solves the IBVP (9.1.1) over the time interval $[0,1]$ using m equidistant timesteps and the finite element space $\mathcal{S}^0_{1,0}(\mathcal{M})$ as encoded in the argument `dofh`.

SOLUTION for (9-1.k) → 9-1-11-0:s4.pdf ▲

(9-1.l) ⚂ (30 min.) [depends on Sub-problem (9-1.k)]

In the beginning of the `main()` function in `radauthreetimestepping_main.cc` a triangular mesh of the square domain Ω is built.

Augment the `main()` function such that it calls the function `solveHeatEvolution()` from Sub-problem (9-1.k) with $m = 50$ and outputs the final temperature distribution in a VTK-file `discrete_heat_solution.vtk` , which can be visualized using PARAVIEW. The `final_time` argument can be set to 1.

HIDDEN HINT 1 for (9-1.l) → `9-1-12-0:pvh1.pdf`

SOLUTION for (9-1.l) → `9-1-12-1:.pdf`

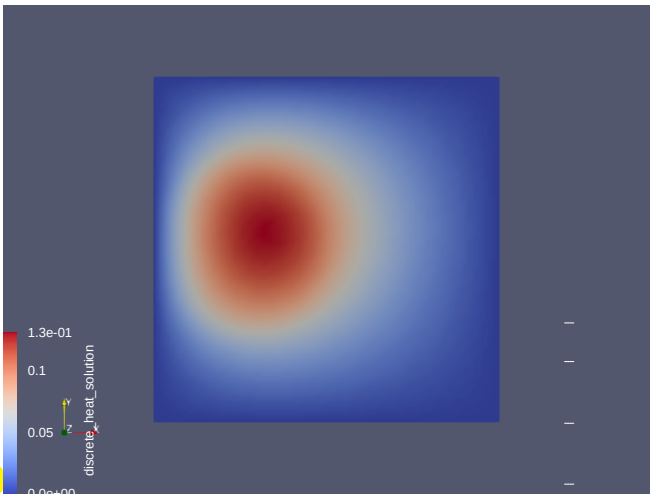


Fig. 117

◁ Visualization of the finite element/2-stage Radau RK-SSM method-of-lines solution of the transient heat conduction problem (9.1.1) at $T = 1$.

Visualization with PARAVIEW

▲

(9-1.m) ⚄ (10 min.) [depends on Sub-problem (9-1.e)]

Elaborate how the implementation of the 2-stage Radau RK-SSM timestepping for the method-of-linear ODE for (9.1.1) can be made more efficient when a uniform timestep is used.

SOLUTION for (9-1.m) → `9-1-13-0:s6.pdf` ▲

End Problem 9-1 , 300 min.

Problem 9-2: Implicit Timestepping for Parabolic IBVP

[Lecture → Section 9.2.7.2] and [Lecture → Rem. 9.2.8.10] in combination with sent the key message that *$L(\pi)$-stable Runge-Kutta single-step method are the right choice* as regards timestepping methods for parabolic initial-boundary value problems (IBVP) in a method-of-lines (→ [Lecture → Section 9.2.4]) setting. In this problem we will see a special class of $L(\pi)$-stable RK-SSMs, which are very popular for parabolic evolutions, applied to a somewhat non-standard transient heat conduction problem.

This problem involes implementation in LehrFEM++ and also relies on the material in [Lecture → Section 9.2.2], [Lecture → Section 9.2.4], [Lecture → Section 9.2.6.5], [Lecture → Section 9.2.7.1], and [Lecture → Section 9.2.7.2].
▷ problem name/problem code folder: SDIRKMethodOfLines

The evolution of the temperature distribution $u = u(\mathbf{x},t)$ in a homogeneous "2D body" (occupying the area $\Omega \subset \mathbb{R}^2$) with convective cooling (→[Lecture → Ex. 1.7.0.5]) is modelled by the linear second-order parabolic initial-boundary value problem (IBVP) with flux (spatial) boundary conditions (→ [Lecture → § 9.2.1.10])

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u = 0 \quad & \text{in } \Omega \times [0,T]\,, \quad \text{(evolution equation)} \\ -\mathbf{grad}\, u \cdot n = cu \quad & \text{on } \partial\Omega \times [0,T]\,, \quad \text{(boundary conditions)} \\ u(\cdot,0) = u_0 \quad & \text{in } \Omega\,, \quad \text{(initial conditions)} \end{aligned} \tag{9.2.1}$$

with a "cooling coefficient" $c > 0$ and given initial data $u_0 \in H^1(\Omega)$.

Remark. Of course, the model (9.2.1) is non-dimensional and obtained by suitable scaling, which makes the heat capacity and heat conductivity attain value 1. To see this, compare with the unscaled model

$$\frac{\partial}{\partial t}(\rho u) - \operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, u) = f \quad \text{in} \quad \widetilde{\Omega} := \Omega \times]0,T[\,. \qquad \text{[Lecture → (9.2.1.6)]}$$

In this problem, we pursue a **method-of-lines** approach, see [Lecture → Section 9.2.4]. For the spatial Galerkin semi-discretization of (9.2.1) we employ linear finite elements on a triangular mesh $\mathcal{M}$ of Ω (FE space $\mathcal{S}_1^0(\mathcal{M})$) with polygonal boundary approximation.

(9-2.a) ⚀ (10 min.) Derive the spatial variational formulation of (9.2.1) in the form

$$\mathsf{m}(\dot{u},v) + \mathsf{a}(u,v) = \ell(v)\,, \tag{9.2.2}$$

with suitable bilinear forms a and m, and linear form ℓ (to be stated explicitly). Do not forget to specify the function spaces for $\{\boldsymbol{x} \to u(t,\boldsymbol{x})\}$ and the test function v.

HIDDEN HINT 1 for (9-2.a) → 9-2-1-0:rcth1.pdf

SOLUTION for (9-2.a) → 9-2-1-1:rct1.pdf ▲

(9-2.b) ⚄ (15 min.) Argue why the total thermal energy

$$E(t) := \int_\Omega u(\mathbf{x},t)\,\mathrm{d}\mathbf{x}\,, \tag{9.2.5}$$

decreases with time, if $u_0(\mathbf{x}) > 0$ for all $\mathbf{x} \in \Omega$.

HIDDEN HINT 1 for (9-2.b) → 9-2-2-0:rct21.pdf

SOLUTION for (9-2.b) → 9-2-2-1:rct2.pdf ▲

(9-2.c) ⚀ (5 min.) Compute the $\mathcal{S}_1^0(\mathcal{M})$ element mass matrix $\mathbf{M}_{\widehat{K}}$ spawned by the bilinear form $\mathsf{m}(\cdot,\cdot)$ found in Sub-problem (9-2.a) on the unit triangle $\widehat{K}$ with vertices $\begin{bmatrix}0\\0\end{bmatrix}$, $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$.

HIDDEN HINT 1 for (9-2.c) → 9-2-3-0:rct30.pdf

SOLUTION for (9-2.c) → 9-2-3-1:s20.pdf ▲

(9-2.d) ⚅ (20 min.) [depends on Sub-problem (9-2.a)]

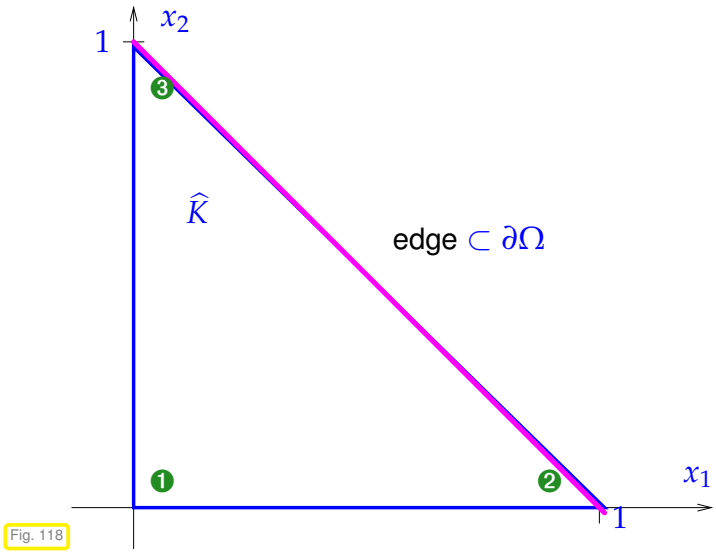


Fig. 118

Compute the $\mathcal{S}_1^0(\mathcal{M})$ element (stiffness) matrix $\mathbf{A}_{\widehat{K}}$ corresponding to $\mathsf{a}(\cdot,\cdot)$ for the unit triangle $\widehat{K}$ with vertices $\begin{bmatrix}0\\0\end{bmatrix}$, $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$. Assume that the edge connecting $\begin{bmatrix}1\\0\end{bmatrix}$ and $\begin{bmatrix}0\\1\end{bmatrix}$ (marked in the figure) forms part of $\partial\Omega$ and that the coefficient c is constant along this edge.

HIDDEN HINT 1 for (9-2.d) → 9-2-4-0:rct31.pdf

SOLUTION for (9-2.d) → 9-2-4-1:rct3.pdf ▲

Now we turn to the full $\mathcal{S}_1^0(\mathcal{M})$-based finite element spatial semi-discretization of (9.2.1), and its implementation.

(9-2.e) ⚀ (15 min.) Study [Lecture → Ex. 2.7.4.41] and the codes [Lecture → Code 2.7.4.43], [Lecture → Code 2.7.4.45]. Understand why they are relevant for the present problem.

SOLUTION for (9-2.e) → 9-2-5-0:s30.pdf ▲

(9-2.f) (30 min.) [depends on Sub-problem (9-2.a)]

Implement a LEHRFEM++-based C++ function (in the file `sdirkmethodoflines.cc`)

```
std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>>
assembleGalerkinMatrices(const lf::assemble::DofHandler
   &dofh,double cool_coeff);
```

that assembles the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrices for the bilinear forms $\mathsf{a}(\cdot,\cdot)$ and $\mathsf{m}(\cdot,\cdot)$ found in Sub-problem (9-2.a). The argument `dofh` is to provide a hybrid mesh $\mathcal{M}$ and information about the numbering of local and global basis functions. The coefficient $c > 0$ is given through `cool_coeff)`.

Your implementation can rely on the LEHRFEM++ library class **lf::uscalfe::LinearFELaplaceElementMatrix** and, of course, on the related function `lf::assemble::assembleMatrixLocally()` (see [Lecture → Code 2.7.4.23]) which even accepts a co-dimension argument! To solve this problem using the later LEHRFEM++ functions, you will also have to implement you own slightly extended version of the class **LinFEMassEdgeMatProvider** from [Lecture → Code 2.7.4.43]. To do so, complete the function `LinFEMassMatrixProvider::Eval` found in `sdirkmethodoflines.cc`. Moreover, you will need to complete the `Eval` function of the **LinearMassEdgeMatrixProvider** found in that same file.

An alternative is the use of **lf::uscalfe::ReactionDiffusionElementMatrixProvider** with suitable coefficient functions, see [Lecture → Ex. 2.8.3.29].

SOLUTION for (9-2.f) → 9-2-6-0:rct4.pdf ▲

In [Lecture → Ex. 9.2.7.49] you have seen the L-stable SDIRK-2 implicit 2-stage Runge-Kutta method described by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \hat{=} \quad \begin{array}{c|cc} \zeta & \zeta & 0 \\ 1 & 1-\zeta & \zeta \\ \hline & 1-\zeta & \zeta \end{array} \quad , \quad \zeta := 1 - \tfrac{1}{2}\sqrt{2} > 0 \, . \tag{9.2.11}$$

Note that the 2×2 "Butcher matrix" $\mathfrak{A}$ of this RK-SSM is *lower-triangular*, with important consequences for implementation.

(9-2.g) ⚀ (15 min.)

Derive the recursion generating the sequence $(y^{(j)})_{j \in \mathbb{N}_0}$ obtained by applying the SDIRK-2 method to the linear scalar ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{R}$.

Remark. That ODE $\dot{y} = \lambda y$ should ring a bell: As we have seen in [Lecture → Section 7.1]/[NumCSE course → Section 12.1], this is the "model problem" used for investigating the response of a single-step method to the "stiffness" [Lecture → Notion 7.2.0.7]/[NumCSE course → Notion 12.2.0.7] of an initial-value problem.

SOLUTION for (9-2.g) → 9-2-7-0:rct6.pdf ▲

(9-2.h) ⚁ (20 min.) [depends on Sub-problem (9-2.g)]

In this Sub-Problem, we want to determine the **order** [Lecture → Def. 6.3.2.8]/[NumCSE course → Def. 11.3.2.8] of the SDIRK-2 single step method empirically by applying it to the initial value problem

$$\dot{y} = -y \quad \text{on} \quad [0,2] \quad , \quad y(0) = 1 \, . \tag{9.2.16}$$

To that end, complete in the file `sdirkmethodoflines_ode.cc` the C++ function

```
std::vector<double> sdirk2SteppingLinScalODE(unsigned int m)
```

that approximately solves (9.2.16) using the SDIRK-2 method from (9.2.11). Then, the function `sdirk2ScalarODECvTest()` (which is already implemented for you) will call `sdirk2SteppingLinScalODE` with $m = 10, 20, 40, 80, \ldots, 2560, 5120$ equidistant timesteps and output a table of the timestep size τ vs. the error

$$\mathrm{err}(\tau) := \max_{j=1,\ldots,m} |y^{(j)} - y(\tau j)| \, ,$$

where $t \mapsto y(t)$ is the exact solution of (9.2.16).

SOLUTION for (9-2.h) → 9-2-8-0:rct7.pdf ▲

(9-2.i) ⚂ (30 min.) Give a rigorous proof that the SDIRK-2 method is **L-stable** [Lecture → Def. 9.2.7.46]/[NumCSE course → Def. 12.3.4.15].

Remark. The numerical tests in [NumCSE course → § 12.3.4.14] convey the importance of L-stability, if a single-step methods is to perform well for extremely stiff IVPs.

HIDDEN HINT 1 for (9-2.i) → 9-2-9-0:rct81.pdf

SOLUTION for (9-2.i) → 9-2-9-1:rct8.pdf ▲

(9-2.j) ⚂ (15 min.) [Lecture → § 9.2.7.6] presents the linear system of equations for the increments of an s-stage Runge-Kutta method when applied to the method-of-lines ODE

$$\mathbf{M}\dot{\vec{\mu}} + \mathbf{A}\vec{\mu} = \mathbf{0}$$

arising from the semi-discrete evolution for (9.2.1). State this linear system explicitly in block form for the special case of the SDIRK-2 method from (9.2.11).

HIDDEN HINT 1 for (9-2.j) → 9-2-10-0:rct101.pdf

SOLUTION for (9-2.j) → 9-2-10-1:rct9.pdf ▲

(9-2.k) (30 min.) In the file `sdirkmethodoflines.cc` code a routine

```
double thermalEnergy(const lf::assemble::DofHandler &dofh,
                     const Eigen::VectorXd &temperature_vec)
```

that computes $\int_\Omega v_h \,\mathrm{d}\boldsymbol{x}$ for $v_h \in \mathcal{S}_1^0(\mathcal{M})$. The argument `temperature_vec` passes the coefficients of v_h w.r.t. the standard "tent function" basis of $\mathcal{S}_1^0(\mathcal{M})$.

SOLUTION for (9-2.k) → 9-2-11-0:BETL-rad-cooling11.pdf ▲

(9-2.l) ⚂ (30 min.) [depends on Sub-problem (9-2.j) and Sub-problem (9-2.k)]

In the file `sdirkmethodoflines.cc` implement an efficient C++ function

```
std::pair<Eigen::VectorXd, Eigen::VectorXd>
  solveTemperatureEvolution(const lf::assemble::DofHandler &dofh,
  unsigned int m, double cool_coeff,
  Eigen::VectorXd initial_temperature_vec)
```

that carries out m *equidistant timesteps* of the SDIRK-2 in order to solve (9.2.1) over the time interval $[0,1]$. The argument `initial_temperature_vec` provides the initial conditions for the method-of-lines ODE, *cf.* [Lecture → (9.2.4.4)], while the parameter `cool_coeff` provides the coefficient $c \in \mathbb{R}$. The discretization in space relies on $\mathcal{S}_1^0(\mathcal{M})$-FEM.

The function should return the basis expansion ocefficient vector of the solution at final time $T = 1$ and the "energies" (9.2.32) for times τj, $j = 0,\ldots,m$, where $\tau := 1/m$ is the constant timestep size.

HIDDEN HINT 1 for (9-2.l) → 9-2-12-0:sp9h1.pdf

SOLUTION for (9-2.l) → 9-2-12-1:BETL-rad-cooling10.pdf ▲

(9-2.m) ⚂ (20 min.) [depends on Sub-problem (9-2.l)]

For the evolution problem (9.2.1) on $\Omega =]0,1[^2$, we track the behavior of the thermal energy

$$E(t) = \int_\Omega u(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} \tag{9.2.32}$$

over the period $[0,1]$ for $u_0 \equiv 1$, $c = 1$.

In order to accomplish this perform the following steps:

1. Complete the `main()` function (in the file `sdirkmethodoflines_main.cc`) so that it reads the mesh from `square64_bnd.msh` using **lf::io::GmshReader**, and calls `solveTemperatureEvolution()` with a **lf::assemble::DofHandler** object associated with that mesh.
2. Run the executable and, for $m = 100$ timesteps, create a plot of $(t_k, E_h(t_k))$, $k = 0,\ldots,m$, in regular scale using the plotting facilities of either MATHGL, PYTHON, or MATLAB.

SOLUTION for (9-2.m) → 9-2-13-0:rct12.pdf ▲

(9-2.n) ⚀ (15 min.) Study the use of the LEHRFEM++ class **lf::io::VtkWriter** and the demo code `io_demo.cc` ➙ GITHUB. ▲

(9-2.o) ⚁ (20 min.) The setting of Sub-problem (9-2.m) still applies. Extend the `main()` function in the file `sdirkmethodoflines_main.cc` so that it writes the $\mathcal{S}_1^0(\mathcal{M})$-approximation $u(\boldsymbol{x},1)$ to the VTK file `discrete_temperature_sol.vtk`.

HIDDEN HINT 1 for (9-2.o) → 9-2-15-0:spxh1.pdf

SOLUTION for (9-2.o) → 9-2-15-1:sx.pdf ▲

References

[Eva98] L.C. Evans. *Partial differential equations*. Vol. 19. Graduate Studies in Mathematics. Providence, RI: American Mathematical Society, 1998.

[Han02] M. Hanke-Bourgeois. *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens*. Mathematische Leitfäden. Stuttgart: B.G. Teubner, 2002.

End Problem 9-2 , 290 min.

Problem 9-3: Decaying Method-of-Lines Solution with Implicit-Euler Timestepping

In this problem we practice the diagonalization technique that was a key tool in the stability analysis of single step methods for semi-discrete parabolic evolution problems, see the presentation in [Lecture → Section 9.2.7.2]

This is a purely theoretical problem building upon [Lecture → Section 9.2.7.1] and connected with [Lecture → Section 9.2.3].
▷ problem name/problem code folder: DecayingSolutionsByImplicitEulerTimestepping

In class we learned that solutions of the abstract variational linear parabolic evolution problem

$$\begin{aligned} \mathsf{m}(\dot{u},v) + \mathsf{a}(u,v) = 0 \quad \forall v \in V_0, \quad 0 < t < T, \\ u(0) = u_0 \in V_0 \,, \end{aligned} \tag{9.3.1}$$

where both m and a are symmetric positive definite bilinear forms on V_0 that satisfy, see [Lecture → (9.2.3.6)],

$$\exists \gamma > 0 : \quad \mathsf{a}(v,v) \geq \gamma \mathsf{m}(v,v) \quad \forall v \in V_0 \,, \tag{9.3.2}$$

display an exponential decay of their m-norm and energy norm,

A simple L(π)-stable [Lecture → Def. 9.2.7.46] implicit timestepping scheme for the semi-discrete linear parabolic evolution problem

$$\begin{aligned} \mathbf{M}\left\{\frac{d}{dt}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu} = 0, \quad 0 < t < T, \\ \vec{\mu}(0) = \vec{\mu}_0 \,, \end{aligned} \tag{9.3.3}$$

arising from the *method of lines* (→ [Lecture → Section 9.2.4]) is the implicit Euler method, see [Lecture → (9.2.7.3)]. Here, $\mathbf{M}$ and $\mathbf{A}$ denote the Galerkin matrices (→ [Lecture → Section 2.2]) associated with m and a w.r.t. an ordered basis of a finite-dimensional trial and test space $V_{0,h} \subset V_0$.

The question is, to what extent the sequence $\vec{\mu}^{(j)}$ generated by the implicit Euler method inherits the exponential decay of the norms stated in [Lecture → Lemma 9.2.3.8].

(9-3.a) ⚀ (10 min.) Write down the formula for a step of the implicit Euler method producing $\vec{\mu}^{(j)}$ from $\vec{\mu}^{(j-1)}$. Write $\tau > 0$ for the size of the timestep.

SOLUTION for (9-3.a) → 9-3-1-0:exp-decay2.pdf ▲

(9-3.b) ⚁ (15 min.) Refresh the material of [Lecture → § 9.2.7.15] and of [Lecture → § 9.2.7.23]. ▲

(9-3.c) ⚂ (15 min.) From the generalized eigenvalue problem $\mathbf{A}\vec{\mu} = \lambda\mathbf{M}\vec{\mu}$ we deduced the existence of a regular matrix $\mathbf{T} \in \mathbb{R}^{N\times N}$ such that (*cf.* the diagonalization technique discussed in [Lecture → Section 9.2.7.2], and, in particular [Lecture → (9.2.7.19)], [Lecture → Suppl. 9.2.7.20])

$$\mathbf{AT} = \mathbf{MTD} \,, \quad \mathbf{D} := \operatorname{diag}(\lambda_1, \ldots, \lambda_N) \,, \tag{9.3.4a}$$

$$\mathbf{T}^\top \mathbf{M} \mathbf{T} = \mathbf{I} \,. \tag{9.3.4b}$$

Rewrite the implicit Euler step from Sub-problem (9-3.a) in terms of the transformed coefficient vectors $\vec{\eta}^{(k)} := \mathbf{T}^\top \mathbf{M} \vec{\mu}^{(k)}$, $k = j-1, j$.

SOLUTION for (9-3.c) → 9-3-3-0:exp-decay2a.pdf ▲

(9-3.d) ⚀ (15 min.) The s.p.d. matrices $\mathbf{A} \in \mathbb{R}^{N,N}$ and $\mathbf{M} \in \mathbb{R}^{N,N}$ define norms on $\mathbb{R}^N$:

$$\left\|\vec{\xi}\right\|_{\mathbf{M}} := (\vec{\xi}^{\top}\mathbf{M}\vec{\xi})^{\frac{1}{2}} \quad , \quad \left\|\vec{\xi}\right\|_{\mathbf{A}} := (\vec{\xi}^{\top}\mathbf{A}\vec{\xi})^{\frac{1}{2}} \, . \tag{9.3.5}$$

What will the norms $\|\cdot\|_{\mathbf{A}}$ and $\|\cdot\|_{\mathbf{M}}$ of a coefficient vector $\vec{\mu}$ become for the transformed vector $\vec{\eta} := \mathbf{T}^{\top}\mathbf{M}\vec{\mu}$?

SOLUTION for (9-3.d) → 9-3-4-0:exp-decay2b.pdf ▲

(9-3.e) ⚀ (10 min.) Express $\mathsf{m}(u_h, u_h)$ and $\mathsf{a}(u_h, u_h)$ through the coefficient vector $\vec{\mu}$ associated with a function $u_h \in V_{0,h}$ from the discrete Galerkin trial/test space and through the Galerkin matrices $\mathbf{A}$ and $\mathbf{M}$. How does (9.3.2) read when stated in terms of matrices and coefficient vectors? What is the relationship of γ and the generalized eigenvalues λ_i?

SOLUTION for (9-3.e) → 9-3-5-0:exp-decay1.pdf ▲

(9-3.f) ⚁ (25 min.) In the sequel we assume a uniform timestep $\tau > 0$. Show that

$$\left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}} \leq \frac{1}{1+\gamma\tau}\left\|\vec{\mu}^{(j-1)}\right\|_{\mathbf{M}} , \tag{9.3.8}$$

where $\gamma > 0$ is the constant from (9.3.2).

HIDDEN HINT 1 for (9-3.f) → 9-3-6-0:decay3hx.pdf

HIDDEN HINT 2 for (9-3.f) → 9-3-6-1:exp-decay31.pdf

SOLUTION for (9-3.f) → 9-3-6-2:exp-decay3.pdf ▲

(9-3.g) ⚁ (20 min.) Now show (9.3.8) with $\|\cdot\|_{\mathbf{M}}$ replaced with $\|\cdot\|_{\mathbf{A}}$.

HIDDEN HINT 1 for (9-3.g) → 9-3-7-0:exp-decay41.pdf

SOLUTION for (9-3.g) → 9-3-7-1:exp-decay4.pdf ▲

(9-3.h) ⚁ (20 min.) [depends on Sub-problem (9-3.f), Sub-problem (9-3.g)]

What is the relationship of the estimates obtained in subproblems (9-3.f) and (9-3.g) with the following result, which is a "semi-discrete version" of [Lecture → Lemma 9.2.3.8]:

Lemma 9.3.13. Decay of solutions of parabolic evolutions

Assuming (9.3.2)*, the solution* $t \mapsto \vec{\mu}(t)$ *of the method-of-lines ODE* (9.3.3) *satisfies*

$$\|\vec{\mu}(t)\|_{\mathbf{M}} \leq e^{-\gamma t}\|\vec{\mu}(0)\|_{\mathbf{M}} \quad , \quad \|\vec{\mu}(t)\|_{\mathbf{A}} \leq e^{-\gamma t}\|\vec{\mu}(0)\|_{\mathbf{A}} \quad \forall t \geq 0 \, ,$$

with $\|\cdot\|_{\mathbf{A}}$ *and* $\|\cdot\|_{\mathbf{M}}$ *defined in* (9.3.5)*.*

HIDDEN HINT 1 for (9-3.h) → 9-3-8-0:exp-decay51.pdf

SOLUTION for (9-3.h) → 9-3-8-1:exp-decay5.pdf ▲

End Problem 9-3 , 130 min.

Problem 9-4: The One-Dimensional Wave Equation with Absorbing Boundary Condition

This problem studies the wave equation [Lecture → Section 9.3.2] in one spatial dimension, now endowed with a special boundary condition known as **absorbing boundary condition**. This boundary condition will allow outgoing waves to leave the computational domain without any reflections! We treat absorbing boundary conditions in the context of the method of lines [Lecture → Section 9.3.3] using simple piecewise linear finite elements in space [Lecture → Section 2.3] and a leapfrog-like timestepping method [Lecture → § 9.3.4.16]. Finally we study the behavior of the total energy.

This problem involves implementation in C++ based on EIGEN. Familiarity with [Lecture → Section 9.3.4] and [Lecture → Section 9.3.3] is important.
▷ problem name/problem code folder: 1DWaveAbsorbingBC

In this problem, we consider the wave equation in one spatial dimension

$$\frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0, \quad \text{on }]0,1[\times]0,T[\ . \tag{9.4.1}$$

with the following initial and boundary conditions

$$c\frac{\partial u}{\partial x}(0,t) - \frac{\partial u}{\partial t}(0,t) = 0 \ , \tag{9.4.2a}$$

$$u(1,t) = g(t) := \begin{cases} \sin t & 0 \le t \le \pi, \\ 0 & \text{otherwise,} \end{cases} \tag{9.4.2b}$$

$$u(x,0) = 0, \quad \frac{\partial u}{\partial t}(x,0) = 0 \ , \quad 0 < x < 1 \ . \tag{9.4.2c}$$

The radiation-type boundary condition (9.4.2a) imposed at $x = 0$ is called an *exact absorbing boundary condition*, for reasons which will hopefully become clear soon.

Remark. When posed on $\mathbb{R} \times [0,T]$, the so-called Cauchy problem [Lecture → § 9.3.2.1] for the 1D linear wave equation (9.4.1) has the **d'Alembert solution**

$$u(x,t) = \tfrac{1}{2}(u_0(x+ct) + u_0(x-ct)) + \tfrac{1}{2c}\int_{x-ct}^{x+ct} v_0(s)\,\mathrm{d}s \ . \qquad \text{[Lecture → Ex. 11.6.1.23]}$$

Its partial derivatives are, by virtue of the fundamental theorem of calculus,

$$\frac{\partial u}{\partial x}(x,t) = \tfrac{1}{2}\big(u_0'(x+ct) + u_0'(x-ct)\big) + \tfrac{1}{2}\big(v_0(x+ct) - v_0(x-ct)\big) \ ,$$
$$\frac{\partial u}{\partial t}(x,t) = \tfrac{1}{2}c\,\big(u_0'(x+ct) - u_0'(x-ct)\big) + \tfrac{1}{2}c\,\big(v_0(x+ct) + v_0(x-ct)\big) \ .$$

Let us assume that $\mathrm{supp}(u_0) \subset [0,1]$ and $\mathrm{supp}(v_0) \subset [0,1]$. Then we find

$$\frac{\partial u}{\partial x}(0,t) = \tfrac{1}{2}\big(u_0'(ct) + v_0(ct)\big) \quad , \quad \frac{\partial u}{\partial t}(0,t) = \tfrac{1}{2}c\,\big(u_0'(ct) + v_0(ct)\big) \ .$$

Obviously, the d'Alembert solution satisfies the boundary condition (9.4.2a). In a sense, the boundary condition at $x = 0$ lets $u(x,t)$ evolve as if the spatial domain was the whole real axis. In other words, the solution is just swallowed by the boundary at $x = 0$ as if it was not there. ⌟

We consider a spatial finite element Galerkin discretization with piecewise linear Lagrangian finite elements on equidistant meshes, see [Lecture → Section 2.3]. The non-homogeneous Dirichlet boundary condition at $x = 1$ can be taken into account through the use of a locally supported offset function as explained in [Lecture → Rem. 2.3.3.15], see also [Lecture → Section 2.7.6].

(9-4.a) ⚄ (15 min.) What is the (spatial) variational formulation for this problem?

HIDDEN HINT 1 for (9-4.a) → 9-4-1-0:1D-wave-absorb11.pdf

SOLUTION for (9-4.a) → 9-4-1-1:1D-wave-absorb1.pdf ▲

We pursue the method-of-lines approach of [Lecture → Section 9.3.3] in the case of (9.4.2). For the spatial finite element Galerkin semi-discretization we employ the space $\mathcal{S}_1^0(\mathcal{M})$ on an equidistant spatial mesh with N cells and meshwidth $h := 1/N$, see [Lecture → § 2.3.1.3], and the customary tent function basis $\mathfrak{B} := \{b_h^1, \ldots, b_h^{N+1}\}$, see [Lecture → § 2.3.1.4]. The nodes of the mesh and the associated tent functions are numbered lexikographically "from left to right". Thus the basis function b_h^1 is associated with the node $x = 0$ and b_h^N with $x = 1 - h$. Details can be found in [Lecture → Section 2.3]. We can write the Galerkin test space as

$$V_{0,h} := \mathcal{S}_1^0(\mathcal{M}) \cap V_0 = \operatorname{Span}\{b_h^1, \ldots, b_h^N\} \quad \Rightarrow \quad \dim V_{0,h} = N\,, \tag{9.4.5}$$

whereas the time-dependent Galerkin trial space will be

$$\widehat{V}_h(t) := \mathcal{S}_1^0(\mathcal{M}) \cap \widehat{V}(t) = \underbrace{g(t) b_h^{N+1}}_{\text{offset function}} + V_0\,. \tag{9.4.6}$$

Moreover, we use the 1D composite trapezoidal quadrature rule [Lecture → (2.3.3.8)] to evaluate all integrals.

(9-4.b) ⚄ (20 min.) [depends on Sub-problem (9-4.a)]

By the method-of-lines approach as outlined above we arrive at an ordinary differential equation of the form

$$\mathbf{M}\ddot{\vec{\mu}}(t) + \mathbf{B}\dot{\vec{\mu}}(t) + \mathbf{A}\vec{\mu}(t) = \vec{\varphi}(t)\,, \quad t \in [0, T]\,, \tag{9.4.7}$$

with matrices $\mathbf{M}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{N,N}$ and a time-dependent vector $t \mapsto \vec{\varphi}(t) \in \mathbb{R}^N$.

Give concrete formulas for the entries of $\mathbf{M}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{N,N}$ and the components of $\vec{\varphi}$.

HIDDEN HINT 1 for (9-4.b) → 9-4-2-0:1D-wave-absorb20.pdf

HIDDEN HINT 2 for (9-4.b) → 9-4-2-1:1D-wave-absorb21.pdf

HIDDEN HINT 3 for (9-4.b) → 9-4-2-2:wabh2.pdf

SOLUTION for (9-4.b) → 9-4-2-3:1D-wave-absorb2.pdf ▲

To discretize the method-of-lines ODE (9.4.7) in time we use an extension of the leapfrog method as introduced in [Lecture → § 9.3.4.16]. For uniform timestep $\tau > 0$ it is defined by the recursion

$$\begin{aligned} \mathbf{M}\frac{\vec{\nu}^{(j+1/2)} - \vec{\nu}^{(j-1/2)}}{\tau} &= -\mathbf{A}\mu^{(j)} - \mathbf{B}\frac{\vec{\nu}^{(j+1/2)} + \vec{\nu}^{(j-1/2)}}{2} + \vec{\varphi}(t_j)\,, \\ \frac{\vec{\mu}^{(j+1)} - \vec{\mu}^{(j)}}{\tau} &= \vec{\nu}^{(j+1/2)}\,. \end{aligned} \tag{9.4.8}$$

It has to be supplemented with a special initial step, which is the same as for the standard leapfrog method from [Lecture → (9.3.4.17)]:

$$\vec{\nu}^{(-1/2)} = 2\vec{\nu}_0 - \vec{\nu}^{(1/2)}\,. \tag{9.4.9}$$

Thus we arrive at a full discretization of (9.4.2) is space and time.

(9-4.c) (45 min.) [depends on Sub-problem (9-4.b)]

In the file `1dwaveabsorbingbc.cc` implement a C++ function

```
Eigen::MatrixXd waveLeapfrogABC(
    double c,double T,
    unsigned int N,unsigned int m);
```

with

- `c` the wave speed parameter $c \in \mathbb{R}$ in (9.4.1),
- `T` the final time, that is the computation should cover the time interval $[0,T]$,
- `N` the number of cells of the equidistant spatial mesh of $]0,1[$,
- `m` the number $m \in \mathbb{N}$ of equal timesteps of size $\tau := T/m$.

This function should implement the extended leapfrog timestepping scheme (9.4.8) for (9.4.1), (9.4.2) using mass-lumped $\mathcal{S}_1^0(\mathcal{M})$ finite-element Galerkin discretization in space. It should return the matrix $\mathbf{R} \in \mathbb{R}^{m+1,N+1}$ with

$$(\mathbf{R})_{i,j} \approx u_h(h(i-1),\tau(j-1))\,, \quad j \in \{1,\ldots,m+1\}\,, \quad i \in \{1,\ldots,N+1\}\,.$$

Note that the values of the solution at $x = 1$ are included.

HIDDEN HINT 1 for (9-4.c) → 9-4-3-0:3h1.pdf

SOLUTION for (9-4.c) → 9-4-3-1:1D-wave-absorb3.pdf ▲

(9-4.d) ⚀ (30 min.) [depends on Sub-problem (9-4.c)]

Extend the `main()` function in the file `1dwaveabsorbingbc_main.cc` so that it outputs the fully discrete solution of (9.4.1), (9.4.2) to file in a suitable format that can be read by a PYTHON script, which visualizes it by means of a suitable 3D plot. Create such a script `viswave.py`.

HIDDEN HINT 1 for (9-4.d) → 9-4-4-0:3bh1.pdf

SOLUTION for (9-4.d) → 9-4-4-1:3bs.pdf ▲

(9-4.e) ⚁ (25 min.) [depends on Sub-problem (9-4.c)]

In the file `1dwaveabsorbingbc.cc` write a C++ function

```
std::pair<Eigen::VectorXd, Eigen::VectorXd> computeEnergies(
    const Eigen::MatrixXd &full_solution,
    double c, double tau);
```

that accepts as an argumment the output matrix of `waveLeapfrogABC()` from Sub-problem (9-4.c) plus the constant c and the timestep size $\tau > 0$, and computes the discrete energies

$$\textbf{potential energy}: \quad E_{\mathrm{pot}}^{(j)} := \tfrac{1}{2}\int_0^1 c^2 \left|\frac{du_h^{(j)}}{dx}\right|^2 \mathrm{d}x\,, \qquad j = 0,\ldots m\,,$$

$$\textbf{kinetic energy}: \quad E_{\mathrm{kin}}^{(j)} := \tfrac{1}{2}\int_0^1 \left|v_h^{(j+\frac{1}{2})}\right|^2 \mathrm{d}x\,, \qquad j = 0,\ldots,m-1\,.$$

Here $u_h^{(j)} \in \mathcal{S}_1^0(\mathcal{M}) \cap \widehat{V}(t_j)$ is the fully discrete finite-element solution after the j-th timestep and $v_h^{(j+\frac{1}{2})}$ is an approximation of $\frac{\partial u}{\partial t}(x,t_{j+\frac{1}{2}})$.

The energies are returned as a pair of vectors, the potential energy in the first, the kinetic energy in the second.

HIDDEN HINT 1 for (9-4.e) → 9-4-5-0:s40h1.pdf

SOLUTION for (9-4.e) → 9-4-5-1:s40.pdf ▲

(9-4.f) ⚀ (25 min.) As functions of time t, plot the (discrete) potential, kinetic and total energies for the case of $N = 100$ interior mesh points, $m = 2000$ timesteps, $T = 7$ and $c = 1$. To this end, add a few lines to the `main()` function that write the discrete energies to file. From there the data should be read by a PYTHON or MATLAB script, which generates the plot.

HIDDEN HINT 1 for (9-4.f) → 9-4-6-0:4h1.pdf

SOLUTION for (9-4.f) → 9-4-6-1:1D-wave-absorb.pdf ▲

(9-4.g) ⚀ (5 min.) What do you observe? Explain why (9.4.2a) have been dubbed "absorbing boundary conditions".

SOLUTION for (9-4.g) → 9-4-7-0:1D-wave-absorb5.pdf ▲

(9-4.h) ⚄ (15 min.) Now we consider the method-of-lines ODE (9.4.7) without excitation, that is, in the case $\vec{\varphi} \equiv 0$.

Show that even in this case he total energy

$$E(t) = \frac{1}{2}\dot{\vec{\mu}}(t)^\top \mathbf{M}\dot{\vec{\mu}}(t) + \frac{1}{2}\vec{\mu}(t)^\top A\vec{\mu}(t) \tag{9.4.23}$$

will not be conserved, but decay, that is, $t \mapsto E(t)$ is decreasing, if $\vec{\mu}(t)$ solves (9.4.7).

HIDDEN HINT 1 for (9-4.h) → 9-4-8-0:1D-wave-absorb61.pdf

SOLUTION for (9-4.h) → 9-4-8-1:1D-wave-absorb.pdf ▲

References

[CDI20] Juliette Chabassier, Julien Diaz, and Sébastien Imperiale. "Construction and analysis of fourth order, energy consistent, family of explicit time discretizations for dissipative linear wave equations". In: *ESAIM Math. Model. Numer. Anal.* 54.3 (2020), pp. 845–878. DOI: 10.1051/m2an/2019079.

End Problem 9-4 , 180 min.

Problem 9-5: Symplectic Timestepping for Wave Equations

This problem examines a special class of timestepping schemes suitable for the wave equation in a method-of-lines context. It is related to [Lecture → Section 9.3] and based on the research article [RWR04].

Involves implementation in LEHRFEM++ and requires familiarity with [Lecture → Section 9.3.3].
▷ problem name/problem code folder: SymplecticTimesteppingWaves

On a 2D polygonal domain $\Omega \subset \mathbb{R}^2$ we consider the following wave equation for a scalar unknown function $u = u(\boldsymbol{x}, t)$

$$\frac{\partial^2 u}{\partial t^2} - \Delta u + c(\boldsymbol{x})u = 0 \quad \text{on} \quad \Omega \times [0,T]\,, \tag{9.5.1a}$$

$$\mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega \times [0,T]\,, \tag{9.5.1b}$$

$$u(0) = u_0\,, \quad \frac{\partial u}{\partial t}(0) = v_0 \quad \text{on} \quad \Omega\,, \tag{9.5.1c}$$

where $\boldsymbol{x} \mapsto c(\boldsymbol{x}) \in C^0(\overline{\Omega})$ is a non-negative bounded function on Ω and the initial values u_0 and v_0 belong to $H^1(\Omega)$.

(9-5.a) ⚀ (10 min.) State the spatial variational formulation of (9.5.1).

HIDDEN HINT 1 for (9-5.a) → 9-5-1-0:symp1h1.pdf

SOLUTION for (9-5.a) → 9-5-1-1:symp1.pdf ▲

(9-5.b) ⚁ (10 min.) Write down a non-trivial quadratic functional $E : H^1(\Omega) \to \mathbb{R}$, which is conserved during the evolution, that is, for which

$$t \mapsto E(\{\boldsymbol{x} \mapsto u(\boldsymbol{x}, t)\}) \equiv \text{const} \quad \text{if } u \text{ solves (9.5.1).}$$

The notion of a quadratic functional was introduced in [Lecture → Def. 1.2.3.2].

HIDDEN HINT 1 for (9-5.b) → 9-5-2-0:2h1.pdf

SOLUTION for (9-5.b) → 9-5-2-1:symp2.pdf ▲

Following the method-of-lines policy we opt for a spatial finite element discretization of (9.5.1) based on

- the Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$, where $\mathcal{M}$ is a triangular mesh of Ω,
- the standard nodal "tent function" basis $\{b_h^\ell\}_{\ell=1}^N$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$, of $\mathcal{S}_1^0(\mathcal{M})$.

This leads to the following initial value problem for a second-order ordinary differential equation on the state space $\mathbb{R}^N$:

$$\mathbf{M}\frac{d^2\vec{\mu}}{dt^2}(t) + \mathbf{A}\vec{\mu}(t) = 0\,, \quad \text{and} \quad \vec{\mu}(0) = \mu_0\,, \quad \frac{d\vec{\mu}}{dt}(0) = \vec{\eta}_0\,. \tag{9.5.4}$$

(9-5.c) ⚂ (10 min.) [depends on Sub-problem (9-5.a)]

Give formulas for the entries of the matrices $\mathbf{M} \in \mathbb{R}^{N,N}$ and $\mathbf{A} \in \mathbb{R}^{N,N}$ occuring in (9.5.4).

SOLUTION for (9-5.c) → 9-5-3-0:symp3.pdf ▲

The authors of [RWR04] consider ordinary differential equations on the state space $\mathbb{R}^{2n}$, $n \in \mathbb{N}$, of the form

$$\frac{d}{dt}\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix} = \begin{bmatrix}\mathbf{f}(\mathbf{q},t)\\ \mathbf{g}(\mathbf{p})\end{bmatrix} \quad \text{with continuous functions} \quad \begin{aligned}&\mathbf{f} : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}^n\,,\\ &\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n\,.\end{aligned} \tag{9.5.5}$$

For the temporal discretization of (9.5.5), [RWR04, Sect. IV] proposes the following class of s-step methods, $s \in \mathbb{N}$, with uniform timestep $\tau > 0$ for generating sequences $\begin{bmatrix} \mathbf{p}^{(j)} \\ \mathbf{q}^{(j)} \end{bmatrix}$, $j = 0, \ldots, M$, of approximations of $\begin{bmatrix} \mathbf{p}(j\tau) \\ \mathbf{q}(j\tau) \end{bmatrix}$, starting from initial values $\mathbf{p}^{(0)} := \mathbf{p}(0)$ and $\mathbf{q}^{(0)} := \mathbf{q}(0)$:

$$
\begin{array}{l}
\textbf{for} \quad j := 1 \quad \textbf{to} \quad M \quad \textbf{do} \\
\left[\begin{array}{l}
\quad \mathbf{p}_{\text{in}} := \mathbf{p}^{(j-1)} \,, \quad \mathbf{q}_{\text{in}} := \mathbf{q}^{(j-1)} \\
\quad \textbf{for} \quad k := 1 \quad \textbf{to} \quad s \quad \textbf{do} \\
\quad \left[\begin{array}{l}
\qquad t_k := j\tau + \tau \sum_{\ell=1}^{k-1} a_\ell \,; \\
\qquad \mathbf{p}_{\text{out}} := \mathbf{p}_{\text{in}} + \tau\, b_k\, \mathbf{f}(\mathbf{q}_{\text{in}}, t_k) \,; \\
\qquad \mathbf{q}_{\text{out}} := \mathbf{q}_{\text{in}} + \tau\, a_k\, \mathbf{g}(\mathbf{p}_{\text{out}}) \,; \\
\qquad \mathbf{p}_{\text{in}} := \mathbf{p}_{\text{out}} \,, \quad \mathbf{q}_{\text{in}} := \mathbf{q}_{\text{out}} \,;
\end{array} \right. \\
\quad \textbf{end} \\
\quad \mathbf{p}^{(j)} := \mathbf{p}_{\text{out}} \,, \quad \mathbf{q}^{(j)} := \mathbf{q}_{\text{out}} \,;
\end{array} \right. \\
\textbf{end}
\end{array}
\tag{9.5.6}
$$

Here $a_1, \ldots, a_s$ and $b_1, \ldots, b_s$ are given coefficients, carefully chosen such that the single-step method achieves a certain order. A possible choice discussed in [RWR04] is, for $s = 3$,

$$
a_1 := \tfrac{2}{3} \,, \quad a_2 := -\tfrac{2}{3} \,, \quad a_3 := 1 \,, \quad b_1 := \tfrac{7}{24} \quad b_2 := \tfrac{3}{4} \,, \quad b_3 := -\tfrac{1}{24} \,. \tag{9.5.7}
$$

(9-5.d) ⚄ (30 min.) In the file `symplectictimesteppingwaves_ode.cc` write a C++ function

```
sympTimestep(double tau, Eigen::Vector2d& pq_j)
```

that implements a single timestep $\begin{bmatrix} \mathbf{p}^{(j)} \\ \mathbf{q}^{(j)} \end{bmatrix} \mapsto \begin{bmatrix} \mathbf{p}^{(j+1)} \\ \mathbf{q}^{(j+1)} \end{bmatrix}$ of the method (9.5.6) with $s = 3$ and coefficients according to (9.5.7) for the "harmonic oscillator" ODE ($n = 1$)

$$
\dot{p} = q \quad , \quad \dot{q} = -p \,. \tag{9.5.8}
$$

The components of the argument vector supply $p^{(j)}$ and $q^{(j)}$, and will be replaced with $p^{(j+1)}$ and $q^{(j+1)}$.

SOLUTION for (9-5.d) → `9-5-4-0:symp4.pdf` ▲

(9-5.e) ⚄ (20 min.) [depends on Sub-problem (9-5.d)]

Experimentally determine the order of convergence of the single-step method implemented in Sub-problem (9-5.d) in terms of the number of timesteps by

- integrating (9.5.8) over the period $[0, 2\pi]$,
- with initial values $p(0) = 0$, $q(0) = 1$,
- and using $M = 10, 20, 40, 80, 160, 320, 640$ timesteps.

Tabulate the following errors at the final time:

$$
\mathrm{err}(M) := |p(2\pi) - p^{(M)}| + |q(2\pi) - q^{(M)}| \,, \quad M = 10, 20, 40, 80, 160, 320, 640 \,.
$$

To that end complete the function `sympTimesteppingODETest()` in the file `symplectictimesteppingwaves_ode.cc`, which is called from `main()`.

HIDDEN HINT 1 for (9-5.e) → `9-5-5-0:smp5h1.pdf`

SOLUTION for (9-5.e) → `9-5-5-1:symp5.pdf` ▲

(9-5.f) ⚁ (15 min.) Rewrite (9.5.4) in the form (9.5.5) with $\mathbf{g} = \mathrm{Id}$. Give an expression for the function $\mathbf{f}$ based on the matrices $\mathbf{M}$ and $\mathbf{A}$.

HIDDEN HINT 1 for (9-5.f) → 9-5-6-0:6h1.pdf

SOLUTION for (9-5.f) → 9-5-6-1:symp6.pdf ▲

(9-5.g) ⚁ (45 min.) [depends on Sub-problem (9-5.c)]

In the file `symplectictimesteppingwaves_assemble.h` create a LEHRFEM++-based C++ function

```
template <typename FUNC_ALPHA, typename FUNC_GAMMA, typename
   FUNC_BETA>
Eigen::SparseMatrix<double> assembleGalerkinMatrix(
    std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
       fe_space_p,
    FUNC_ALPHA &&alpha, FUNC_GAMMA &&gamma, FUNC_BETA &&beta)
```

that, given a mesh $\mathcal{M}$, (approximately, using "safe" local numerical quadrature) computes the $\mathcal{S}_1^0(\mathcal{M})$-based finite element Galerkin matrix for the left-hand side of the variational boundary value problem

$$u \in H^1(\Omega): \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\beta(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{9.5.12}$$

where $\alpha, \gamma : \Omega \to \mathbb{R}$, $\beta : \partial\Omega \to \mathbb{R}$ are bounded coefficient functions, and $f \in L^2(\Omega)$. The use of the standard nodal basis consisting of "tent functions" is assumed.

The argument `fe_space_p` passes a pointer to a **lf::uscalfe::UniformScalarFESpace** object, which grants access to the underlying mesh, the local-to-global index mapping for shape functions, and the definition of the local shape functions on the reference elements [Lecture → § 2.8.3.28].

Through `alpha`, `gamma`, and `beta` we have to pass functors for the coefficient functions $\boldsymbol{x} \mapsto \alpha(\boldsymbol{x})$, $\boldsymbol{x} \mapsto \gamma(\boldsymbol{x})$, and $\boldsymbol{x} \mapsto \beta(\boldsymbol{x})$. These functors have to supply an evaluation operator

```
double operator ()(const Eigen::Vector2d x);
```

accepting point (world) coordinates as argument `x`.

You may use the facilities available in the LEHRFEM++ library, in particular the class **lf::uscalfe::ReactionDiffusionElementMatrixProvider**, **lf::fe::MassEdgeMatrixProvider**, and `lf::assemble::AssembleMatrixLocally()`, see the LEHRFEM++ documentation and [Lecture → Ex. 2.8.3.29] for further information.

SOLUTION for (9-5.g) → 9-5-7-0:s6a.pdf ▲

(9-5.h) ⚂ (45 min.) [depends on Sub-problem (9-5.f) and Sub-problem (9-5.g)]

Based on the function `assembleGalerkinMatrix()`, complete the implementation (except for the method `computeEnergies()`, which will be dealt with in Sub-problem (9-5.j)) of a C++ class (in the files `symplectictimesteppingwaves.h` and `symplectictimesteppingwaves.cc`)

```
template <typename FUNCTION>
class SympTimestepWaveEq
{
public:
  SympTimestepWaveEq(
```

```
        std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
           fe_space_p,
        FUNCTION c);
    void compTimestep(double tau, Eigen::VectorXd &p,
       Eigen::VectorXd &q) const;
    double computeEnergies(const Eigen::VectorXd &p,
                           const Eigen::VectorXd &q) const;

  private:
  // suitable data member here
  }; // class SympTimestepWaveEq
```

whose method `compTimestep` realizes a single timestep of length $\tau > 0$ of the method (9.5.6) with $s = 3$ and coefficients according to (9.5.7) for the first-order method-of-lines ODE found in Sub-problem (9-5.f). As arguments it takes two vectors `p` and `q` of basis expansion coefficients corresponding to $\mathbf{p}^{(j)}, \mathbf{q}^{(j)}$ from the previous timestep and *replaces* them with the counterparts of $\mathbf{p}^{(j+1)}, \mathbf{q}^{(j+1)}$.

The constructor should initialize suitable member variables based on the coefficient function c. Take pains to achieve an efficient implementation by precomputing as much as possible, in particular a factored form of the matrix $\mathbf{M}$ to speed up the solution of linear systems in each timestep.

HIDDEN HINT 1 for (9-5.h) → 9-5-8-0:symp7h.pdf

SOLUTION for (9-5.h) → 9-5-8-1:symp7.pdf ▲

(9-5.i) ⚁ For (9.5.4) rewritten in the form (9.5.5) express the semi-discrete counterpart of the conserved "energy" $E(u)$ found in Sub-problem (9-5.b) in terms of $\mathbf{p}$ and $\mathbf{q}$.

SOLUTION for (9-5.i) → 9-5-9-0:s80.pdf ▲

(9-5.j) ⚁ (30 min.) [depends on Sub-problem (9-5.i) and Sub-problem (9-5.h)]

Augment the class **SympTimestepWaveEq** by a member function

```
double SympTimestepWaveEq::computeEnergies(
    const Eigen::VectorXd &p,const Eigen::VectorXd &q);
```

that returns the "energy" quantity found in Sub-problem (9-5.i). The arguments `p` and `q` pass the basis expansion coefficient vectors corresponding to $\mathbf{p}$ and $\mathbf{q}$ in the first-order form of the method-of-lines ODE derived in Sub-problem (9-5.f).

HIDDEN HINT 1 for (9-5.j) → 9-5-10-0:8h1.pdf

SOLUTION for (9-5.j) → 9-5-10-1:s3.pdf ▲

(9-5.k) ⚁ (30 min.) In the file `symplectictimesteppingwaves.h` write a C++ function

```
template <typename FUNCTION>
std::pair<Eigen::VectorXd, Eigen::VectorXd>
  solvewave(
    std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
       fes_p,
    FUNCTION c,
    const Eigen::VectorXd& u0_vec,
    const Eigen::VectorXd& v0_vec,
    double T, unsigned int m);
```

the solves the fully discrete evolution problem for (9.5.1) over the time interval $[0, T]$ using the class

SympTimestepWaveEq and m uniform timesteps of the symplectic timestepping scheme (9.5.6) with $s = 3$.

The coefficient function c is specified through the functor argument `c`, the basis expansion coefficients of (approximations in $\mathcal{S}_1^0(\mathcal{M})$ of) u_0 and v_0 are passed through `u0_vec` and `v0_vec`. Finally, `fes_p` gives all information about the mesh and local-to-global index mapping for the spatial finite-element methods.

The function should also write the tuples $(\tau j, E(\tau j))$, $j = 0, \ldots, m$, $\tau := T/m$ to a table, where E is the "conserved energy" quantity considered in Sub-problem (9-5.i).

SOLUTION for (9-5.k) → 9-5-11-0:s9.pdf ▲

(9-5.l) ⚂ (30 min.) [depends on Sub-problem (9-5.k)]

In the file `symplectictimesteppingwaves.cc` implement a C++ function

```
void wavePropSimulation(unsigned int m);
```

that uses the scheme implemented in `solveWave()` to tackle (9.5.1) for $T = 10$ and

$$u_0 \equiv 1 \quad , \quad v_0 \equiv 0 \quad , \quad c(\boldsymbol{x}) = 1 + \|\boldsymbol{x}\|^2 \, . \tag{9.5.21}$$

Use the mesh given in the file `hex4.msh` in GMSH format and read it via LEHRFEM++'s **lf::io::GmshReader**. **lf::uscalfe::FeSpaceLagrangeO1** can be used to define the finite element space, see [Lecture → § 2.8.3.28].

For $m = 2000$ timesteps create a plot of the "conserved energy" E at the discrete times $j\tau$, $\tau := T/m$, versus time. Output the solution $u_h(T) \in \mathcal{S}_1^0(\mathcal{M})$ in VTK format in the file `ufinal.vtk` using **lf::io::VtkWriter** and visualize this function by means of PARAVIEW.

SOLUTION for (9-5.l) → 9-5-12-0:s10.pdf ▲

(9-5.m) ⚂ (20 min.) The symplectic timestepping (9.5.6) for (9.5.4) is only *conditionally stable* and blow-up will inevitably occur for large timesteps. Based on **SympTimestepWaveEq** and for the setting (9.5.21) implement a C++ function (in the file `symplectictimesteppingwaves.cc`)

```
double testStab();
```

that determines a crude estimate for the maximal uniform timestep possible, if blow-up is to be avoided. The spatial finite element discretization should rely on the mesh given in the file `hex4.msh` in GMSH format.

HIDDEN HINT 1 for (9-5.m) → 9-5-13-0:sp11h1.pdf

SOLUTION for (9-5.m) → 9-5-13-1:s11s.pdf ▲

References

[RWR04] R. Rieben, D. White, and G. Rodrigue. "High Order Symplectic Integration Methods for Finite Element Solutions to Time Dependent Maxwell Equations". In: *IEEE Trans. Antennas and Propagation* 52.8 (2004), pp. 2190–2195 (cit. on pp. 293, 294).

End Problem 9-5 , 295 min.

Problem 9-6: A mixed elliptic-hyperbolic linear evolution problem

This problem studies a special 2nd-order-in-time scalar linear evolution problem and its discretization by the method of lines. It revisits energy conservation and implicit timestepping.

This problem relates to [Lecture → Section 9.3] and relies on implementation in LEHRFEM++.
▷ problem name/problem code folder: BoundaryWave

On a polygonal domain $\Omega \subset \mathbb{R}^2$ we consider the following evolution in variational form: seek $u : [0,T] \to H^1(\Omega)$ such that it satisfies the initial conditions

$$u(0) = u_0 \quad , \quad \frac{du}{dt}(0) = v_0 \quad \text{in} \quad \Omega \, , \tag{9.6.1}$$

for given functions $u_0, v_0 \in H^1(\Omega)$ and such that for all $t \in [0,T]$

$$\frac{d^2}{dt^2} \int_{\partial\Omega} u(\boldsymbol{x},t) v(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) + \int_\Omega (1 + \|\boldsymbol{x}\|^2) \, \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H^1(\Omega) \, . \tag{9.6.2}$$

(9-6.a) ⚀ (10 min.)

State a non-constant quadratic functional (→ [Lecture → Def. 1.2.3.2]) $E : H^1(\Omega) \times H^1(\Omega) \to \mathbb{R}$ so that $t \mapsto E(u(t), \frac{du}{dt}(t))$ is constant, if $t \mapsto u(t)$ solves (9.6.2).

SOLUTION for (9-6.a) → 9-6-1-0:s1.pdf ▲

(9-6.b) ⚅ (15 min.) [depends on Sub-problem (9-6.a)]

Argue, why the solution of the initial-boundary value problem (9.6.1)–(9.6.2) is unique.

SOLUTION for (9-6.b) → 9-6-2-0:sigma2.pdf ▲

(9-6.c) ⚂ (15 min.)

State the evolution problem (9.6.2) in strong (PDE) form.

SOLUTION for (9-6.c) → 9-6-3-0:s3.pdf ▲

We tackle (9.6.1) and (9.6.2) in the spirit of the method of lines. The spatial finite-element Galerkin discretization of (9.6.2) relies on $\mathcal{S}_1^0(\mathcal{M})$, where $\mathcal{M}$ is a 2D hybrid mesh covering a polygonal approximation of Ω. The standard basis of $\mathcal{S}_1^0(\mathcal{M})$ comprised of "tent functions" is employed throughout and we denote it by $\mathfrak{B}_h := \{b_h^1, \ldots, b_h^N\}$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$.

Spatial semidiscretization leads to the method-of-lines ordinary differential equation

$$\mathbf{M} \tfrac{d^2}{dt^2} \vec{\boldsymbol{\mu}}(t) + \mathbf{A} \vec{\boldsymbol{\mu}}(t) = 0 \tag{9.6.9}$$

for the time-dependent basis expansion coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t) \in \mathbb{R}^N$ for a finite-element approximation of $t \mapsto u(t)$.

(9-6.d) ⚂ (10 min.)

Give formulas for the entries of the matrices $\mathbf{M} \in \mathbb{R}^{N,N}$ and $\mathbf{A} \in \mathbb{R}^{N,N}$ in (9.6.9).

SOLUTION for (9-6.d) → 9-6-4-0:s4.pdf ▲

(9-6.e) ⚅ (15 min.) [depends on Sub-problem (9-6.d)]

Why is it **not** possible to apply the leapfrog timestepping scheme introduced in [Lecture → § 9.3.4.16] to (9.6.9)?

SOLUTION for (9-6.e) → 9-6-5-0:s5.pdf ▲

(9-6.f) ⚄ (30 min.) [depends on Sub-problem (9-6.d)]

In the file `boundarywave.cc` complete the implementation of a LEHRFEM++-based function

```
lf::assemble::COOMatrix<double>
  buildM(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
      &fe_space_p);
```

that initializes an object storing the matrix $\mathbf{M}$ in triplet (COO) format. The (pointer) argument `fe_space_p` also passes information about the mesh and the numbering of basis functions.

HIDDEN HINT 1 for (9-6.f) → 9-6-6-0:h61.pdf

HIDDEN HINT 2 for (9-6.f) → 9-6-6-1:h62.pdf

SOLUTION for (9-6.f) → 9-6-6-2:s6.pdf ▲

(9-6.g) ⚂ (20 min.) [depends on Sub-problem (9-6.d)]

Also in the file `boundarywave.cc` supplement the missing parts of the function

```
lf::assemble::COOMatrix<double>
  buildA(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
      &fe_space_p);
```

which performs the assembly of the finite-element Galerkin matrix $\mathbf{A}$ in triplet format based on LEHRFEM++'s auxiliary class **lf::uscalfe::ReactionDiffusionElementMatrixProvider**. You may rely on the automatic selection of local quadrature rules offered by that class.

SOLUTION for (9-6.g) → 9-6-7-0:s7.pdf ▲

(9-6.h) ⚂ (20 min.) Initial values $\vec{\mu}(0)$ and $\dot{\vec{\mu}}(0)$ for the method-of-lines ODE (9.6.9) can be obtained by *linear interpolation* of the initial data u_0 and v_0.

In the file `boundarywave.h` complete the implementation of the function

```
template <typename FUNCTOR_U, typename FUNCTOR_V>
std::pair<Eigen::VectorXd, Eigen::VectorXd>
  interpolateInitialData(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
       &fe_space_p,
    FUNCTOR_U &&u0, FUNCTOR_V &&v0);
```

that uses the LEHRFEM++ function `lf::fe::NodalProjection()` to compute the expansion coefficient vectors of the linear interpolants of functions u_0 and v_0 (returned in that order). The arguments `u0` and `v0` pass functors of signature

```
std::function<double(Eigen::Vector2d)>;
```

These functors provide the initial data u_0 and v_0 in procedural form.

SOLUTION for (9-6.h) → 9-6-8-0:s8.pdf ▲

(9-6.i) ⚅ (60 min.) [depends on Sub-problem (9-6.f), Sub-problem (9-6.g), Sub-problem (9-6.h)]

In the file `boundarywave.h` supply the missing parts of the function

```
template <typename FUNCTOR_U, typename FUNCTOR_V>
Eigen::VectorXd solveBoundaryWave(
```

```
const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
   &fe_space_p,
FUNCTOR_U &&u0, FUNCTOR_V &&v0, double T, unsigned int N);
```

that computes the basis expansion coefficients of an approximation $u_h^{(M)} \in \mathcal{S}_1^0(\mathcal{M})$ of the solution $t \to u(t)$ of (9.6.1) and (9.6.2) at final time $T > 0$. The initial data are passed through the functors `u0` and `v0` as in Sub-problem (9-6.h).

Use $M \in \mathbb{N}$ equidistant steps of **Crank-Nicolson timestepping** as introduced in [Lecture → § 9.3.4.7]. Pay attention to the efficiency of your code.

SOLUTION for (9-6.i) → `9-6-9-0:s9.pdf` ▲

(9-6.j) ⚄ (15 min.) [depends on Sub-problem (9-6.i)]

We assume that the solution of (9.6.1) & (9.6.2) is smooth enough to ensure the validity of the asymptotic estimates given in [Lecture → Thm. 9.3.5.10] also for Crank-Nicolson timestepping.

Asymptotically, how will the computational effort involved in a call to `solveBoundaryWave()` that you implemented in Sub-problem (9-6.i) increase, if the error norm

$$\epsilon(\mathcal{M},\tau) := \left(\tau \sum_{j=1}^{M} \left\| u - u_h^{(j)} \right\|_{H^1(\Omega)}^2\right)^{\frac{1}{2}},$$

where $u_h^{(j)} \in \mathcal{S}_1^0(\mathcal{M})$ is the approximation of $u(\tau j)$ in the j-th timestep, is to be *halved*. Assume that you work with quasi-uniform and shape-regular families of finite element meshes and that the LU/Cholesky factorization of sparse $N \times N$ finite-element Galerkin matrices involves an asymptotic effort of $O(N^2)$, whereas forward and backward substitution incur asymptotic cost of $O(N^{\frac{3}{2}})$.

HIDDEN HINT 1 for (9-6.j) → `9-6-10-0:h101.pdf`

SOLUTION for (9-6.j) → `9-6-10-1:s10.pdf` ▲

End Problem 9-6, 210 min.

Problem 9-7: Absorbing Boundary Conditions (ABCs) for the 2D Wave Equation

The finite element spatial discretization of a time-domain wave propagation problem on an unbounded domain entails truncating that domain. On the truncation boundary absorbing boundary conditions (ABCs) are imposed that are to be transparent for outgoing waves. In this problem we study local low-order approximate ABCs in the context of the method of lines for the linear wave equations in two spatial dimensions.

Familiarity with the method of lines for the wave equation [Lecture → Section 9.3.4] and special timestepping schemes is required. Implementation is based on LEHRFEM++
▷ problem name/problem code folder: WaveABC2D

Let $\Omega \subset \mathbb{R}^2$ stand for a polygonally bounded 2D domain. We consider the following initial-boundary value problem for the linear wave equation with constant coefficients: For given initial value $u_0 \in C^1(\overline{\Omega})$, a non-negative bounded coefficient function $\rho : \Omega \to \mathbb{R}$, and final time $T > 0$, find $u : \Omega \times [0,T] \to \mathbb{R}$ such that

$$\rho(\boldsymbol{x})\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) - \Delta u(\boldsymbol{x},t) = 0 \quad \text{in} \quad \Omega\times]0,T[\,, \tag{9.7.1a}$$

$$\mathbf{grad}_{\boldsymbol{x}}\, u(\boldsymbol{x},t)\cdot \boldsymbol{n}(\boldsymbol{x}) + \frac{\partial u}{\partial t}(\boldsymbol{x},t) = 0 \quad \text{on} \quad \partial\Omega\times]0,T[\,, \tag{9.7.1b}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad , \quad \frac{\partial u}{\partial t}(\boldsymbol{x},0) = 0 \quad \forall \boldsymbol{x} \in \Omega\,. \tag{9.7.1c}$$

The spatial boundary condition (9.7.1b) is known as first-order approximate absorbing boundary condition.

(9-7.a) ⚂ (15 min.) What is the spatial variational formulation of the IVP (9.7.1) (in suitable Sobolev spaces)?

SOLUTION for (9-7.a) → 9-7-1-0:s1.pdf ▲

We pursue the **method-of-lines** approach of [Lecture → Section 9.3.3] to perform the spatial semi-discretization of (9.7.1). For spatial finite element Galerkin discretization we employ the lowest-degree Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$ on a hybrid mesh $\mathcal{M}$ of Ω consisting of triangles and quadrilaterals. We rely on the customary tent function basis, here denoted by $\{b_N^1,\ldots,b_N^N\}$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$.

(9-7.b) ⚂ (10 min.) [depends on Sub-problem (12-4.a)]

By the method-of-lines approach to (9.7.1) as outlined before we arrive at an ordinary differential equation for the time-dependent basis expansion coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t)$ of the semi-discrete solution, of the form

$$\boxed{\mathbf{M}\ddot{\vec{\boldsymbol{\mu}}}(t) + \mathbf{B}\dot{\vec{\boldsymbol{\mu}}}(t) + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \mathbf{0}} \quad , \quad t \in [0,T]\,, \tag{9.7.2}$$

with matrices $\mathbf{M}, \mathbf{A}, \mathbf{B} \in \mathbb{R}^{N,N}$.

Give formulas for the entries of $\mathbf{M}, \mathbf{A}, \mathbf{B}$, as concrete as possible.

SOLUTION for (9-7.b) → 9-7-2-0:s2.pdf ▲

For the method-of-lines ODE (9.7.2) we opt for the following single-step timestepping scheme with uniform

time step size $\tau := T/N > 0$:

$$\begin{aligned} \mathbf{M}\frac{\vec{v}^{(k+1)}-\vec{v}^{(k)}}{\tau} &= -\tfrac{1}{2}\mathbf{B}\big(\vec{v}^{(k+1)}+\vec{v}^{(k)}\big) - \tfrac{1}{2}\mathbf{A}\big(\vec{\mu}^{(k+1)}+\vec{\mu}^{(k)}\big)\,, \\ \frac{\vec{\mu}^{(k+1)}-\vec{\mu}^{(k)}}{\tau} &= \tfrac{1}{2}\big(\vec{v}^{(k+1)}+\vec{v}^{(k)}\big)\,, \end{aligned} \qquad k \in \{0,\ldots,N-1\}\,, \tag{9.7.5}$$

where $\vec{v}^{(k)} \in \mathbb{R}^N$ are auxiliary vectors. The recusion is started with initial values $\vec{\mu}^{(0)}$ and $\vec{v}^{(0)}$.

(9-7.c) ⚀ (10 min.) In (9.7.5) the new vectors $\vec{v}^{(k+1)}$ and $\vec{\mu}^{(k+1)}$ have to be computed by solving a linear system of equations. State that linear system of equations.

SOLUTION for (9-7.c) → 9-7-3-0:s2m.pdf ▲

(9-7.d) ⚄ (20 min.) [depends on Sub-problem (9-7.c) and Sub-problem (12-4.b)]

Show that the linear system of equations derived in Sub-problem (9-7.c) will always have a solution for the method-of-lines ODE arising from the spatial semi-discretization of (9.7.1).

HIDDEN HINT 1 for (9-7.d) → 9-7-4-0:2nh1.pdf

HIDDEN HINT 2 for (9-7.d) → 9-7-4-1:2nh2.pdf

SOLUTION for (9-7.d) → 9-7-4-2:s2n.pdf ▲

(9-7.e) ⚀ (10 min.) When using the scheme (9.7.5) for the temporal discretization of (9.7.1), what is the interpretation of the vectors $\vec{v}^{(k)}$?

SOLUTION for (9-7.e) → 9-7-5-0:s2x.pdf ▲

(9-7.f) ⚂ (15 min.) By eliminating the vectors $\vec{v}^{(k)}$, $k \in \mathbb{N}_0$, rewrite (9.7.5) as a **2-step timestepping method** of the form

$$\vec{\mu}^{(k+1)} = T(\vec{\mu}^{(k)}, \vec{\mu}^{(k-1)})\,,$$

with a suitable term $T(\cdot,\cdot)$.

SOLUTION for (9-7.f) → 9-7-6-0:s2y.pdf ▲

(9-7.g) ⚄ (10 min.) [depends on Sub-problem (9-7.e)]

If the scheme (9.7.5) is used for the temporal discretization of (9.7.1), how do you have to choose the intial values $\vec{\mu}^{(0)}$ and $\vec{v}^{(0)}$?

SOLUTION for (9-7.g) → 9-7-7-0:s2a.pdf ▲

(9-7.h) ⚄ (15 min.) [depends on Sub-problem (12-4.b)]

Using the notations from (9.7.5) we define the time-discrete **energy**

$$E^{(k)} := \big(\vec{\mu}^{(k)}\big)^\top \mathbf{A}\vec{\mu}^{(k)} + \big(\vec{v}^{(k)}\big)^\top \mathbf{M}\vec{v}^{(k)}\,, \quad k \in \mathbb{N}_0\,. \tag{9.7.11}$$

Show that

$$E^{(k)} \le E^{(0)} \quad \forall k \in \mathbb{N}\,.$$

SOLUTION for (9-7.h) → 9-7-8-0:2ms.pdf ▲

(9-7.i) ⚂ (15 min.) [depends on Sub-problem (9-7.c)]

Empririically, we want to determine the order of the timestepping scheme (9.7.5) by applying it to the initial-value problem for a scalar-valued function $\mu : \mathbb{R} \to \mathbb{R}$,

$$\epsilon\ddot{\mu} + (1-\epsilon)\dot{\mu} + \mu = 0\,, \quad y(0) = 1\,, \quad \dot{y}(0) = 1\,, \tag{9.7.12}$$

with parameter $\epsilon \in [0,1]$. To that end, implement a C++ function

```
Eigen::VectorXd scalarImplicitTimestepping(
  double epsilon, unsigned int M);
```

that carries out $M \in \mathbb{N}$ equidistant timesteps to solve (9.7.12) on the time interval $[0,1]$ and returns the sequence $\left(\mu^{(k)}\right)_{k=0}^{M}$ thus generated. The parameter ϵ is passed through `epsilon`.

SOLUTION for (9-7.i) → 9-7-9-0:3is.pdf ▲

(9-7.j) ⚂ (10 min.)

M	$\mathrm{err}(M)$
10	2.82397e-04
20	6.93351e-05
30	3.07126e-05
40	1.72556e-05
50	1.10376e-05
60	7.66274e-06
80	4.30903e-06
100	2.75740e-06
160	1.07695e-06
200	6.89227e-07
320	2.69219e-07
500	1.10271e-07
640	6.73036e-08

The table beside reports the error

$$\mathrm{err}(M) := \max_{k=0,\ldots,M} |\mu(k/M) - \mu^{(k)}|$$

of the timestepping scheme (9.7.5) applied to the initial-value problem (9.7.12) on $[0,1]$. M denotes the number of equidistant timesteps, $t \mapsto \mu(t)$ the exact solution, and $\left(\mu^{(k)}\right)_{k=0}^{M}$ the sequence produced by the single-step method.
Determine the empiric rate of algebraic convergence of the implicit timestepping method implemented in `scalarImplicitTimestepping()` from these data.

SOLUTION for (9-7.j) → 9-7-10-0:s3f.pdf ▲

The following class defined in the file `waveabc2d.h` is meant to solve (9.7.1) by the method of lines using a spatial finite-element Galerkin discretization based on $\mathcal{S}_1^0(\mathcal{M})$ and the timestepping (9.7.5).

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
class WaveABC2DTimestepper {
public:
  // Main constructor; precomputations are done here
  WaveABC2DTimestepper(
      const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
        &fe_space_p,
      FUNC_RHO rho, unsigned int M, double T);
  // Public member functions
  Eigen::VectorXd solveWaveABC2D(FUNC_MU0 mu0, FUNC_NU0 nu0);
  double energies();
private:
  double T_;                    // final time
  unsigned int M_;              // nb of steps
  double step_size_;            // time inverval
  std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
    fe_space_p_;
```

```
  lf::uscalfe::size_type N_dofs_; // nb of degrees of freedom
  // to ensure that energies are computed only after timestepping
  bool timestepping_performed_;
  // Introduce suitable precomputed objects
}; // class WaveABC2DTimestepper
```

For the requested implementations in LehrFEM++ you can rely on the C++ function (provided in the file `waveabc2d.h`)

```
template <typename FUNC_ALPHA, typename FUNC_BETA, typename
   FUNC_GAMMA>
lf::assemble::COOMatrix<double> computeGalerkinMat(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
       &fe_space_p,
    FUNC_ALPHA alpha, FUNC_GAMMA gamma, FUNC_BETA beta);
```

that, given a mesh $\mathcal{M}$, (approximately, using "safe" local numerical quadrature,) computes the $\mathcal{S}_1^0(\mathcal{M})$-based finite element Galerkin matrix, in COO/triplet format, for the left-hand side of the variational boundary value problem

$$u \in H^1(\Omega): \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\beta(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{9.7.14}$$

where $\alpha,\gamma : \Omega \to \mathbb{R}$, $\beta : \partial\Omega \to \mathbb{R}$ are bounded coefficient functions, and $f \in L^2(\Omega)$. The use of the standard nodal basis consisting of "tent functions" is assumed.

The argument `fe_space_p` implicitly passes the local-to-global index mapping for $\mathcal{S}_1^0(\mathcal{M})$ and the mesh $\mathcal{M}$, while `alpha`, `gamma`, and `beta` are functors for the coefficient functions $\boldsymbol{x}\mapsto\alpha(\boldsymbol{x})$, $\boldsymbol{x}\mapsto\gamma(\boldsymbol{x})$, and $\boldsymbol{x}\mapsto\beta(\boldsymbol{x})$.

(9-7.k) ⚄ (60 min.) [depends on Sub-problem (9-7.c)]

Based on the LehrFEM++ library, in the file `waveabc2d.cc` complete the implementation of the following methods of **WaveABC2DTimestepper**:

(I) The constructor

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
WaveABC2DTimestepper<FUNC_RHO, FUNC_MU0,
   FUNC_NU0>::WaveABC2DTimestepper(
     const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
        &fe_space_p,
    FUNC_RHO rho, unsigned int M, double T);
```

of the class **WaveABC2DTimestepper**, which takes as arguments

- `fe_space_p`, a pointer to an object describing a Lagrangian finite element space of lowest order, see [Lecture → § 2.8.3.28],
- `rho`, a functor object for the coefficient function $\boldsymbol{x}\mapsto\rho(\boldsymbol{x})$,
- `M`, the number of equidistant timesteps,
- and the final time $T > 0$ in `T`.

(II) The member function

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
Eigen::VectorXd
    WaveABC2DTimestepper<FUNC_RHO, FUNC_MU0,
       FUNC_NU0>::solveWaveABC2D(
    FUNC_MU0 mu0, FUNC_NU0 nu0);
```

This function should return the basis expansion coefficient vector of the finite-element approximation of $u(T)$. The parameters `mu0` and `nu0` are functor objects providing the initial data.

HIDDEN HINT 1 for (9-7.k) → 9-7-11-0:h3x.pdf

SOLUTION for (9-7.k) → 9-7-11-1:s3.pdf ▲

(9-7.l) ⚄ (20 min.) Implement in the class **WaveABC2DTimestepper** the method

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
double WaveABC2DTimestepper<FUNC_RHO, FUNC_MU0,
   FUNC_NU0>::energies()
```

that returns the energy (9.7.11) of the approximate solution at the time currently stored in the data member `T_` and based on the approximation computed in the last call of `WaveABC2DTimestepper::solveWaveABC2D()` and stored in the variable `full_sol_` of the class **WaveABC2DTimestepper**.

SOLUTION for (9-7.l) → 9-7-12-0:s6.pdf ▲

End Problem 9-7 , 210 min.

Problem 9-8: Non-linear Schrödinger Equation with Cubic Non-Linearity

The non-linear Schrödinger equation (NLSE) with cubic non-linearity in the de-focusing setting

$$\imath \dot{u} = -\Delta_{\boldsymbol{x}} u + |u|^2 u \quad \text{in} \quad \Omega \times [0,T] \tag{NLSE}$$

is a fundamental model for many non-linear evolution problems, see Wikipedia. In one dimension it features a host of very special solutions called *solitons* and *breathers*.
In this problem we are going to consider the NLSE in two dimensions and study its discretization in space and time. We will learn about the important class of split-step timestepping methods.

This problem assumes familiarity with the variational formulation of elliptic boundary value problems as taught in [Lecture → Section 1.8] and [Lecture → Section 9.2.4]. It involves implementation in LehrFEM++ and also has a focus on structure-preserving timestepping by single-step method applied in the context of the method of lines.
▷ problem name/problem code folder: NonLinSchroedingerEquation

On a bounded computational domain $\Omega \subset \mathbb{R}^2$ (with exterior unit normal vector field $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$) we consider the following initial-boundary value problem (IBVP) for the *non-linear* Schrödinger equation (NLSE) over the time interval $[0,T]$:

$$\begin{aligned} \imath \frac{\partial u}{\partial t} &= -\Delta_{\boldsymbol{x}} u + |u|^2 u \quad \text{in} \quad \Omega \times [0,T] \,, \\ u(\boldsymbol{x},0) &= u_0(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,, \\ \mathbf{grad}_{\boldsymbol{x}}\, u \cdot \boldsymbol{n} &= 0 \quad \text{on} \quad \partial\Omega \times [0,T] \,, \end{aligned} \tag{NLSE}$$

where the unknown is a complex-valued function $u : \Omega \to \mathbb{C}$ and $\imath$ is the imaginary unit. Note that the spatial Laplacian applied to a complex-valued function is separately acting on the real and imaginary part.

Remark. Owing to the term marked in green, in contrast to the heat equation and wave equation, (NLSE) is a non-linear evolution problem.

Remark. Both spaces, $\mathbb{R}^N$ and $\mathbb{C}^N$, $N \in \mathbb{N}$, can be endowed with the *same* basis of *real* Cartesian unit vectors. The difference is that for vectors $\in \mathbb{R}^n$ we restrict ourselves to linear combinations of the basis vectors with real coefficients, while in the case of $\mathbb{C}^N$ we admit complex coefficients. In a similar fashion one can understand the complexification of function spaces.

(9-8.a) ⚁ (15 min.) Split the PDE in (NLSE) into two equations, one for the real part v and the other for the imaginary part w of $u = v + \imath w$.

SOLUTION for (9-8.a) → 9-8-1-0:nlse1.pdf ▲

(9-8.b) ⚂ (20 min.) Using the techniques of [Lecture → Section 1.8] and [Lecture → Section 9.2.2] state the spatial variational formulation of the of (NLSE). Do not forget to specify appropriate function spaces.

HIDDEN HINT 1 for (9-8.b) → 9-8-2-0:nlse2h1.pdf

SOLUTION for (9-8.b) → 9-8-2-1:nlse2.pdf ▲

(9-8.c) ⚁ (10 min.) [depends on Sub-problem (9-8.b)]

Show that for every complex-valued solution $u : [0,T] \to H^1(\Omega)$ of (NLSE) its $L^2(\Omega)$-norm is a tem-

poral invariant:

$$\frac{d}{dt}\int_\Omega |u(\boldsymbol{x},t)|^2\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall t \in [0,T]\,. \tag{9.8.6}$$

Remark. In physical models $\boldsymbol{x} \mapsto |u(\boldsymbol{x},t)|^2$ often has the interpretation of a time-dependent probability density, which has to satisfy $\int_\Omega |u(\boldsymbol{x},t)|^2\,\mathrm{d}\boldsymbol{x} = 1$ for all times.

HIDDEN HINT 1 for (9-8.c) → 9-8-3-0:lse3h1.pdf

HIDDEN HINT 2 for (9-8.c) → 9-8-3-1:lse3h2.pdf

SOLUTION for (9-8.c) → 9-8-3-2:nlse3.pdf ▲

(9-8.d) ⚄ (15 min.) [depends on Sub-problem (9-8.b)]

Show that the **Hamiltonian**

$$H(u) := \frac{1}{2}\int_\Omega \|\mathbf{grad}\, u\|^2\,\mathrm{d}\boldsymbol{x} + \frac{1}{4}\int_\Omega |u|^4\,\mathrm{d}\boldsymbol{x}\,, \quad u \in H^1_0(\Omega)\,, \tag{9.8.8}$$

is a temporal invariant for every solution of the spatial variational form of (NLSE).

SOLUTION for (9-8.d) → 9-8-4-0:lse4.pdf ▲

In the spirit of the method of lines (→ [Lecture → Section 9.2.4]), we perform a Galerkin semi-discretization of the spatial variational formulation of (NLSE) based on the finite element space $\mathcal{S}^0_1(\mathcal{M})$, where $\mathcal{M}$ is a triangular mesh of Ω.

We also employ a policy called mass lumping [Lecture → Rem. 9.3.4.18]: for the approximate evaluation of *all* integrals we use the composite 2D trapezoidal rule introduced in [Lecture → § 2.4.6.9]. We also use the standard "tent function" basis $\mathfrak{B} := \{b^1_h,\ldots,b^N_h\}$, $N := \dim \mathcal{S}^0_1(\mathcal{M})$, as presented in [Lecture → Section 2.4.3].

Starting from the complex-valued variational formulation found in Sub-problem (9-8.b) the method-of-lines spatial semi-discretization leads to an ODE for the complex-valued basis expansion coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t) = \big(\mu_j(t)\big)_{j=1}^N \in \mathbb{C}^N$ of the form

$$\mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) = \mathbf{A}\vec{\boldsymbol{\mu}}(t) + \mathbf{f}(\vec{\boldsymbol{\mu}})\,, \quad \mathbf{f}(\vec{\boldsymbol{\mu}}) := \big[f_j(\mu_j(t))\big]_{j=1}^N\,, \tag{9.8.10}$$

with matrices $\mathbf{M},\mathbf{A} \in \mathbb{C}^{N,N}$ and suitable functions $f_j : \mathbb{C} \to \mathbb{C}$, $j = 1,\ldots,N$.

(9-8.e) ⚄ (15 min.) [depends on Sub-problem (9-8.b)]

Give formulas for the entries of the matrices $\mathbf{A}$ and $\mathbf{M}$ from (9.8.10).

SOLUTION for (9-8.e) → 9-8-5-0:s5a.pdf ▲

(9-8.f) ⚄ (15 min.) [depends on Sub-problem (9-8.a)]

Give the concrete formulas for the functions f_j occurring in the method-of-lines ODE (9.8.10).

HIDDEN HINT 1 for (9-8.f) → 9-8-6-0:nlse5h1.pdf

SOLUTION for (9-8.f) → 9-8-6-1:nlse5.pdf ▲

(9-8.g) ⚄ (15 min.) [depends on Sub-problem (9-8.f) and ((9-8.e))]

Write $t \mapsto \vec{\boldsymbol{\mu}}$ for the solution of the method-of-lines ODE (9.8.10). Show that $t \mapsto \vec{\boldsymbol{\mu}}(t)^{\mathrm{H}}\mathbf{M}\vec{\boldsymbol{\mu}}(t)$ is constant, where $\mathbf{M} \in \mathbb{C}^{N,N}$ is the matrix from (9.8.10).

SOLUTION for (9-8.g) → 9-8-7-0:nlse7.pdf ▲

For the method-of-lines ODE (9.8.10) arising from (NLSE) and established in Sub-problem (9-8.e), Sub-problem (9-8.f), we write

$$\mathbf{M}\dot{\vec{\mu}}(t) = \mathbf{A}\vec{\mu}(t) + \mathbf{f}(\vec{\mu}(t)) \,, \tag{9.8.17}$$

with suitable matrices $\mathbf{A}, \mathbf{M}$ and a function $\mathbf{f} : \mathbb{C}^N \to \mathbb{C}^N$.

We opt for a **split-step single-step timestepping** scheme with uniform timestep $\tau := T/M$, $M \in \mathbb{N}$. It is also known as **Strang splitting scheme**, see [NumCSE course → Section 12.5] for details. A single timestep $\vec{\mu}^{(k)} \mapsto \vec{\mu}^{(k+1)}$ involves three separate phases:

(I) First linear half-step:

$$\vec{\mu}^{(k+1/2)} := \Psi_{\mathrm{IMP}}^{0,\tau/2}(\vec{\mu}^{(k)}) \,, \tag{9.8.18}$$

where Ψ_{IMP} is the discrete evolution operator of the implicit midpoint rule [Lecture → (9.2.6.7)] for the linear ODE $\mathbf{M}\dot{\vec{\mu}}(t) = \mathbf{A}\vec{\mu}(t)$.

(II) Non-linear complementary step: Obtain $\mu^{(k,*)} \in \mathbb{C}$ as the *exact solution* at time τ of the initial value problem

$$\mathbf{M}\dot{\vec{\xi}}(t) = \mathbf{f}(\vec{\xi}(t)) \quad , \quad \vec{\xi}(0) = \vec{\mu}^{(k+1/2)} \quad \Rightarrow \quad \mu^{(k,*)} := \vec{\xi}(\tau) \,. \tag{9.8.19}$$

(III) Second linear half-step:

$$\vec{\mu}^{(k+1)} := \Psi_{\mathrm{IMP}}^{0,\tau/2}(\vec{\mu}^{(k,*)}) \,.$$

We carry out M such timestep starting from an initial value $\vec{\mu}^{(0)}$ ($\leftrightarrow u_0$) and ending with $\vec{\mu}^{(M)}$ ($\leftrightarrow u(\cdot, T)$).

Remark. The linear half-steps of consecutive timesteps can be merged into one linear full step, which results in an alternative more efficient implementation, saving half the number of linear system solves.

(I) Initial half-step:

$$\vec{\mu}^{(1/2)} := \Psi_{\mathrm{IMP}}^{0,\tau/2}(\vec{\mu}^{(0)}) \,.$$

(II) Carry out M interleaved linear and non-linear steps: for $k = 1, 2, \ldots, M$ do

(i) Obtain $\mu^{(k)}$ as the exact solution at time τ of the initial value problem

$$\mathbf{M}\dot{\vec{\xi}}(t) = \mathbf{f}(\vec{\xi}(t)) \quad , \quad \vec{\xi}(0) = \vec{\mu}^{(k-1/2)} \,. \tag{9.8.19}$$

(ii) Compute

$$\vec{\mu}^{(k+1/2)} := \Psi_{\mathrm{IMP}}^{0,\tau}(\vec{\mu}^{(k)}) \,.$$

(III) Final half-step:

$$\vec{\mu}^{(M)} := \Psi_{\mathrm{IMP}}^{0,\tau/2}(\vec{\mu}^{(M+1/2)}) \,.$$

(9-8.h) ⚄ (20 min.) [depends on Sub-problem (9-8.f)]

Determine the exact solution of the initial value problem

$$\mathbf{M}\dot{\vec{\xi}}(t) = \mathbf{f}(\vec{\xi}(t)) \quad , \quad \vec{\xi}(0) = \vec{\xi}_0 \in \mathbb{C}^N \, . \tag{9.8.20}$$

HIDDEN HINT 1 for (9-8.h) → 9-8-8-0:s9h1.pdf

HIDDEN HINT 2 for (9-8.h) → 9-8-8-1:nlse9h1.pdf

SOLUTION for (9-8.h) → 9-8-8-2:.pdf ▲

(9-8.i) ⚂ (15 min.) [depends on Sub-problem (9-8.h)]

In the file `propagator.h` you find the definition of the class

```
class InteractionPropagator : public Propagator {
blic:
InteractionPropagator(double tau);
Eigen::VectorXcd operator()(const Eigen::VectorXcd &mu) const
   override;
ivate:
// Suitable data
```

Its purpose is to implement the exact evolution operator (→ [Lecture → (9.2.5.9)]) $\boldsymbol{\Phi} : \mathbb{R} \times \mathbb{C}^N \to \mathbb{C}^N$, $\boldsymbol{\Phi} = \boldsymbol{\Phi}(\tau, \xi)$ for the autonomous ODE $\mathbf{M}\dot{\vec{\xi}}(t) = \mathbf{f}(\vec{\xi}(t))$ from (9.8.20). The timestep size τ is passed to the constructor, whereas the member function **operator**`()` realizes the mapping $\vec{\xi} \mapsto \boldsymbol{\Phi}^\tau \vec{\xi} := \boldsymbol{\Phi}(\tau, \vec{\xi})$.

Complete the implementation of the class **InteractionPropagator** in the file `propagator.cc`

Remark. In theoretical physics the term propagator is used to refer to evolution operator, because this operator "propagates" the state of a system to the future.

SOLUTION for (9-8.i) → 9-8-9-0:s9a.pdf ▲

(9-8.j) ⚂ (10 min.) Give the concrete update formula of (9.8.18) in terms of the Galerkin matrices $\mathbf{M} \in \mathbb{C}^{N,N}$ and $\mathbf{A} \in \mathbb{C}^{N,N}$.

SOLUTION for (9-8.j) → 9-8-10-0:nlse8.pdf ▲

(9-8.k) ⚂ (15 min.) [depends on Sub-problem (9-8.j)]

The following class (defined in `propagator.h`)

```
class KineticPropagator : public Propagator {
  using SparseMatrixXcd =
     Eigen::SparseMatrix<std::complex<double>>;
  using SparseMatrixXd = Eigen::SparseMatrix<double>;
public:
  KineticPropagator(const SparseMatrixXd &A,
                    const SparseMatrixXcd &M,
                    double tau);
  Eigen::VectorXcd operator()(const Eigen::VectorXcd &mu) const
     override;
private:
  // Suitable data members
};
```

is supposed to implement the discrete evolution operator Ψ_{IMP} for the implicit midpoint rule applied to the linear autonomous ODE $\mathbf{M}\dot{\vec{\mu}} = \mathbf{A}\vec{\mu}$, *cf.* (9.8.18), with matrices $\mathbf{M}, \mathbf{A}$ as in (9.8.17). The matrices $\mathbf{M}, \mathbf{A} \in \mathbb{C}^{N,N}$ and the timestep $\tau > 0$ are passed through the constructor, whereas the member function **operator**`()` provides the mapping $\vec{\mu} \in \mathbb{C}^N \mapsto \Psi_{\mathrm{IMP}}^{0,\tau}\vec{\mu}$.

Deliver an *efficicient* implementation of the class **KineticPropagator** in the file `propagator.cc`.

Remark. The Laplacian is related to the so-called momentum operator in quantum mechanics. This accounts for the term "kinetic" in the name of the class.

HIDDEN HINT 1 for (9-8.k) → 9-8-11-0:s8bh1.pdf

SOLUTION for (9-8.k) → 9-8-11-1:s8b.pdf ▲

(9-8.l) ⚀ (10 min.) [depends on Sub-problem (9-8.k) and Sub-problem (9-8.i)]

In the file `propagator.cc` complete the implementation of the class

```
class SplitStepPropagator : public Propagator {
  using SparseMatrixXcd =
     Eigen::SparseMatrix<std::complex<double>>;
  using SparseMatrixXd = Eigen::SparseMatrix<double>;
public:
  SplitStepPropagator(const SparseMatrixXd &A,
                      const SparseMatrixXcd &M,
                      double tau);
  Eigen::VectorXcd operator()(const Eigen::VectorXcd &mu) const
     override;
private:
  // Suitable data members
};
```

Its member function **operator**`()` is meant to realize the discrete evolution operator for the split-step single-step timestepping scheme (I), (II), (III) as introduced above. The argument `tau` passes the uniform timestep to the constructor, which also takes the Galerkin matrices $\mathbf{A}$ and $\mathbf{M}$.

SOLUTION for (9-8.l) → 9-8-12-0:s8a.pdf ▲

(9-8.m) ⚀ In the file `nonlinschroedingerequation.cc` complete the implementation of the two C++ functions

```
double KineticEnergy(const Eigen::VectorXcd &mu,
                     const Eigen::SparseMatrix<double> &A);
double InteractionEnergy(const Eigen::VectorXcd &mu,
                     const Eigen::SparseMatrix<double> &D);
```

that are meant to compute the two contributions to the Hamiltonian $H(u)$

$$\begin{aligned} \text{"kinetic energy"}: \quad & H_{\mathrm{kin}}(u) := \tfrac{1}{2}\int_\Omega \|\mathbf{grad}\, u\|^2 \,\mathrm{d}\boldsymbol{x}\,, \\ \text{"interaction energy"}: \quad & H_{\mathrm{coll}}(u) := \tfrac{1}{4}\int_\Omega |u|^4 \,\mathrm{d}\boldsymbol{x}\,, \end{aligned} \tag{9.8.31}$$

for a finite-element function $u \in \mathcal{S}_1^0(\mathcal{M})$ passed through its tent-function basis coefficient expansion vector $\vec{\mu} \in \mathbb{C}^N$ (argument `mu`). The matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ is supposed to provide the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix for $-\Delta$, whereas $\mathbf{D} \in \mathbb{R}^{N,N}$ is such that $\mathbf{M} = \imath\mathbf{D}$, $\mathbf{M}$ as in Sub-problem (9-8.e).

SOLUTION for (9-8.m) → 9-8-13-0:s10a.pdf ▲

(9-8.n) ⚄ (30 min.) [depends on Sub-problem (9-8.l)]

In the file `nonlinschroedingerequation_main.cc` elaborate a LEHRFEM++-based implementation of the split-step timestepping using the given triangular mesh `square_64.msh` of $\Omega =]0,1[^2$, initial data

$$u_0(\boldsymbol{x}) = 4\cos(\pi x_1)\cos(\pi x_2)\,, \quad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \Omega\,,$$

final time $T = 1$, and a number of timesteps $M = 100$. Output (in a table) approximate values of the Hamiltonian $H(u_h(t_j))$, $t_j = \frac{j}{M}$, $j = 0,\ldots,M$, as defined in (9.8.8) and based on the 2D trapezoidal rule for each timestep.

Use the LEHRFEM++ auxiliary function `lf::fe::NodalProjection()` to obtain the finite-element coefficient vector $\vec{\boldsymbol{\mu}}^{(0)}$ corresponding to the initial state u_0.

SOLUTION for (9-8.n) → `9-8-14-0:nlse120.pdf` ▲

(9-8.o) ⚄ (30 min.) [depends on Sub-problem (9-8.g)]

Write $\vec{\boldsymbol{\mu}}^{(j)}$ for the sequence of finite-element basis expansion coefficient vector produced by the split-step timestepping method applied to the method-of-lines ODE (9.8.10). Show that $(\vec{\boldsymbol{\mu}}^{(k)})^{\mathrm{H}}\mathbf{M}\vec{\boldsymbol{\mu}}^{(k)}$ is the same for all $j = 0,\ldots,M$.

HIDDEN HINT 1 for (9-8.o) → `9-8-15-0:nlse11h1.pdf`

HIDDEN HINT 2 for (9-8.o) → `9-8-15-1:nlse11h2.pdf`

SOLUTION for (9-8.o) → `9-8-15-2:nlse11.pdf` ▲

(9-8.p) ⚀ (20 min.) [depends on Sub-problem (9-8.n)]

Given your solution of Sub-problem (9-8.n), plot the approximate value of the Hamiltonian and the contributing "energies" according to (9.8.31) for every timestep.

The plotting can be done with a suitable PYTHON script that reads the table created as output of your code from Sub-problem (9-8.n).

SOLUTION for (9-8.p) → `9-8-16-0:nlse12.pdf` ▲

End Problem 9-8, 255 min.

Problem 9-9: The Fisher KPP Equation for Migration Modeling

This problem is dedicated to the numerical solution of a non-linear reaction-diffusion evolution initial-boundary value problem, supposed to be a model for the migration of animal or prehistoric human populations.

This problem relies on the techniques treated in [Lecture → Section 9.2] and involves sbstantial coding in LEHRFEM++
▷ problem name/problem code folder: FisherKPP

In this problem, we study the evolution of the population density $u = u(\boldsymbol{x},t) : \Omega \times [0,T] \to \mathbb{R}$ over time and space, where the final time $T > 0$ denotes a finite time horizon, and where $\Omega \subset \mathbb{R}^2$ represents the earth's surface. We model the evolution with Fisher's equation, a special case of the Kolmogorov-Petrovsky-Piscounov (KPP) equation. Belonging to the class of non-linear **reaction-diffusion equations**, it can exhibit traveling wave solutions.

The strong form of the evolution equation reads

$$\frac{\partial u}{\partial t} = \operatorname{div}(c(t,\boldsymbol{x})\ \mathbf{grad}\ u) + \lambda\,(1 - \frac{u}{K(\boldsymbol{x},t)})\,u \qquad \text{in} \quad \Omega \times [0,T]\,, \tag{9.9.1}$$

where $c : [0,T] \times \Omega \to \mathbb{R}$ denotes the diffusion coefficient, $\lambda \in \mathbb{R}$ is a constant growth factor, and $K : [0,T] \times \Omega \to \mathbb{R}$ describes the carrying capacity.

Modeling crossings over the sea requires non-local boundary conditions to account for the dispersal over straits, whenever two land territories are close enough such that humans indeed crossed the sea. However, for the sake of simplicity, we may assume that our ancestors avoided the sea and remained ashore. Thus, we simply impose homogeneous Neumann boundary conditions:

$$-\mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \qquad \text{on} \quad [0,T] \times \partial\Omega\,. \tag{9.9.2}$$

Thus, finally, with an initial population denoted by $u_0 : \Omega \to \mathbb{R}$, we obtain the following initial boundary value problem:

$$\begin{aligned} \frac{\partial u}{\partial t} - \operatorname{div}(\,c(t,\boldsymbol{x})\ \mathbf{grad}\ u) - \lambda\,(1 - \frac{u}{K(\boldsymbol{x},t)})\,u = 0 \qquad & \text{in } \Omega \times [0,T], \\ -\mathbf{grad}\ u \cdot \boldsymbol{n} = 0 \qquad & \text{on } \partial\Omega \times [0,T], \\ u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \qquad & \text{in } \Omega\,. \end{aligned} \tag{9.9.3}$$

Simulating this evolution gives a crude idea how modern humans settled the globe during the last ice age. Of course, the estimation of the parameters c and K faces huge uncertainty and so the results are mainly qualitative in nature.

Coarse triangular mesh used for the simulations, $N = 15463$ nodes ▷

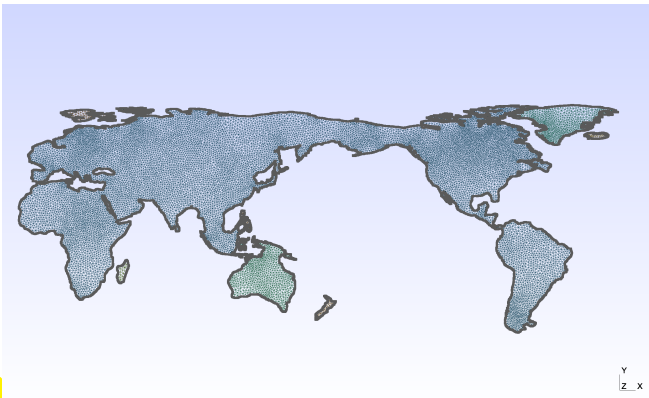


Fig. 126

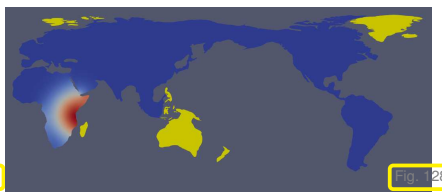


Fig. 127

200,000 years ago

Fig. 128

150,000 years ago

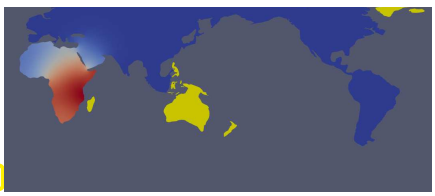
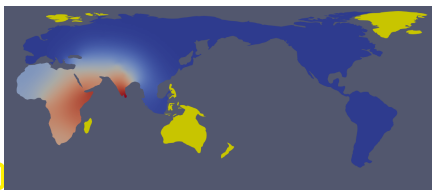
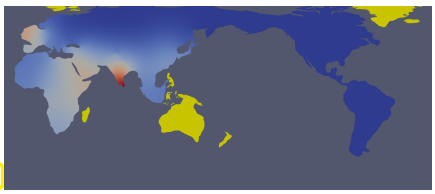


Fig. 129

130,000 years ago



100,000 years ago 70,000 years ago 65,000 years ago



50,000 years ago 45,000 years ago 25,000 years ago

(9-9.a) ⚂ (15 min.) Derive the spatial variational formulation of (9.9.3) in the form

$$u : [0,T[\to V_0 : \quad \mathsf{m}(\dot{u}(t),v) + \mathsf{a}(u(t),v) - r(u(t);v) = \ell(v) \quad \forall v \in V_0 \,, \tag{9.9.4}$$

with suitable bilinear forms m and a on V_0, with a functional $r : V_0 \times V_0 \to \mathbb{R}$ for the nonlinear term (linear only in the second argument!), and with a linear form ℓ. You should also specify the function space V_0 for $\boldsymbol{x} \to u(\boldsymbol{x},t)$ and for the test function v.

SOLUTION for (9-9.a) → 9-9-1-0:sa.pdf ▲

In the context of the method-of-lines semi-discretization we need the Galerkin matrices $\mathbf{M}, \mathbf{A} \in \mathbb{R}^{N,N}$, defined as

$$\begin{aligned} \mathbf{M} &:= \left[\mathsf{m}(b_h^j, b_h^i)\right]_{i,j=1}^N = \left[\int_\Omega b_h^j(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right]_{i,j=1}^N, \\ \mathbf{A} &:= \left[\mathsf{a}(b_h^j, b_h^i)\right]_{i,j=1}^N = \left[\int_\Omega c\ \mathbf{grad}\ b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\ b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right]_{i,j=1}^N, \end{aligned} \tag{9.9.9}$$

where we have chosen the ordered tent-function basis $\mathfrak{B}_h = \{b_h^1, \ldots, b_h^N\}$ of our Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$.

(9-9.b) ⚂ (15 min.) [depends on Sub-problem (9-9.a)]

The matrices $\mathbf{M}$ and $\mathbf{A}$ are assembled based on the entries of the corresponding element matrices.

Compute the $\mathcal{S}_1^0(\mathcal{M})$ element matrices $\mathbf{M}_{\hat{K}}$ and $\mathbf{A}_{\hat{K}}$ spawned by the bilinear forms $\mathsf{m}(\cdot,\cdot)$ and $\mathsf{a}(\cdot,\cdot)$ from Sub-problem (9-9.a) on the unit triangle $\hat{K}$ with vertices $\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. You may assume that the diffusion coefficient c is constant.

SOLUTION for (9-9.b) → 9-9-2-0:sb.pdf ▲

(9-9.c) ⚄ (30 min.) [depends on Sub-problem (9-9.a)]

Implement a LEHRFEM++-based C++ function in the file `fisherkpp.cc`

```
template <typename DIFF_COEFF>
std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>>
assembleGalerkinMatrices(const lf::assemble::DofHandler& dofh,
   DIFF_COEFF &&c);
```

that assembles the Galerkin matrices for the bilinear forms $\mathsf{m}(\cdot,\cdot)$ and $\mathsf{a}(\cdot,\cdot)$ from Sub-problem (9-9.a). The argument `dofh` provides the d.o.f.-handler for the triangular mesh $\mathcal{M}$, and `c` provides a function handle for the diffusion coefficient.

Your implementation may rely on the LehrFEM++ library class **lf::uscalfe::ReactionDiffusionElementMatrixProvider**.

SOLUTION for (9-9.c) → 9-9-3-0:sc.pdf ▲

For the spatial Galerkin $\mathcal{S}_1^0(\mathcal{M})$-finite-element discretization of the non-linear term $r(u;v)$, $u,v \in H^1(\Omega)$, we rely on the composite trapezoidal rule for numerical quadrature, see [Lecture → § 2.4.6.9], [Lecture → (2.4.6.10)]. This and the use of the standard tent-function basis of $\mathcal{S}_1^0(\mathcal{M})$ will be taken for granted in the sequel.

Using $t \mapsto \vec{\mu}(t)$ for the time-dependent vector of basis expansion coefficients, $\mathbf{M}, \mathbf{A}$ to denote the Galerkin matrices associated with m and a, respectively, and writing $\vec{\rho} : \mathbb{R}^N \times [0,T] \to \mathbb{R}^N$ for the nonlinear term, we can state the initial-value problem arising from the method-of-lines Galerkin semi-discretization of (9.9.4) as

$$\begin{cases} \mathbf{M}\{\dfrac{d\vec{\mu}}{dt}(t)\} + \mathbf{A}\vec{\mu}(t) - \vec{\rho}(\vec{\mu}(t),t) = 0\,, \\ \vec{\mu}(0) = \vec{\mu}_0\,. \end{cases} \tag{9.9.14}$$

(9-9.d) ⚂ (20 min.) [depends on Sub-problem (9-9.a)]

Develop a detailed expression for $\vec{\rho}$ from (9.9.14) taking into account the use of the composite trapezoidal quadrature rule.

SOLUTION for (9-9.d) → 9-9-4-0:sd.pdf ▲

To solve the ordinary differential equation (9.9.14) we employ the Strang splitting method, see [Hip15]. To this end, first note that since the bilinear form m in (**??**) is symmetric and positive definite, we have that $\mathbf{M}$ is invertible. Thus, we may rewrite (9.9.14) as

$$\begin{cases} \dfrac{d\vec{\mu}}{dt}(t) = f(t,\vec{\mu}) + g(t,\vec{\mu}) \\ \vec{\mu}(0) = \vec{\mu}_0 \end{cases} \tag{9.9.16}$$

where

$$f(t,\vec{\mu}) := -\mathbf{M}^{-1}\mathbf{A}\vec{\mu}(t)\,,$$
$$g(t,\vec{\mu}) := \mathbf{M}^{-1}\vec{\rho}(\vec{\mu}(t),t)\,.$$

Splitting equation (9.9.16) into two parts,

$$\frac{d\vec{\mu}}{dt}(t) = f(t,\vec{\mu})\,,$$
$$\frac{d\vec{\mu}}{dt}(t) = g(t,\vec{\mu})\,,$$

permits us to solve each part separately. We proceed by introducing an equidistant temporal mesh with nodes $0 = t_0 < t_1 < \ldots < t_{M-1} < t_M = T$, $M \in \mathbb{N}$. Let $\tau := t_1 - t_0$ be the step size.

Let us first consider the pure diffusion problem, represented by

$$\frac{d\vec{\mu}}{dt}(t) = f(t,\vec{\mu})$$

with initial condition $\vec{\mu}(0) = \vec{\mu}_0$. In [Lecture → Ex. 9.2.7.49], [Lecture → (9.2.7.51)] of the lecture notes, you have seen the L(π)-stable SDIRK-2 implicit 2-stage Runge-Kutta method described by the Butcher tableau [Lecture → (7.3.3.3)]

$$\begin{array}{c|cc} \xi & \xi & 0 \\ 1 & 1-\xi & \xi \\ \hline & 1-\xi & \xi \end{array}$$

where $\xi := 1 - \frac{1}{2}\sqrt{2}$. To see that the method is indeed of order 2 and L(π)-stable, study homework problem (6-2).

(9-9.e) ⚀ (20 min.) State the linear system of equation obtained from the Butcher scheme for the SDIRK-2 method applied to the semi-discrete evolution $f(t,\vec{\mu}) := -\mathbf{M}^{-1}\mathbf{A}\vec{\mu}(t)$.

SOLUTION for (9-9.e) → 9-9-5-0:se.pdf ▲

(9-9.f) ⚄ (20 min.) For the reaction term, taken into account through the partial evolution

$$\frac{d\vec{\mu}}{dt}(t) = g(t,\vec{\mu})$$

with initial condition $\vec{\mu}(0) = \vec{\mu}_0$, we can implement the exact evolution operator, since the analytical solution to this ordinary differential equation is known. Thus the discrete evolution operator coincides with the exact evolution.

Solve the following logistic ordinary differential equation analytically:

$$\begin{aligned} \frac{\partial u}{\partial t} &= \lambda\,(1 - \frac{u}{K(\boldsymbol{x},t)})\,u \qquad \text{in } [T_0,T_1]\times\Omega, \\ u(\boldsymbol{x},T_0) &= u_0(\boldsymbol{x}) \qquad\qquad \text{in } \Omega, \end{aligned} \tag{9.9.19}$$

where $\mathbb{R}_{\geq 0} \ni T_0 < T_1 < \infty$. Note that λ is constant. Moreover, you may assume that the carrying capacity is constant over the time interval $[T_0,T_1]$.

SOLUTION for (9-9.f) → 9-9-6-0:sf.pdf ▲

(9-9.g) ⚄ (25 min.) Revise the notions of (discrete) evolution operators from [Lecture → § 9.2.5.8], [Lecture → Def. 9.2.6.2], and state the evolution operator for the reaction term in (9.9.3), that is for (9.9.19) with $T_0 = 0, T_1 = T$. Note that since we know the analytic solution, the evolution operator coincides with the discrete evolution operator. Moreover, state the discrete evolution operator for the diffusion term as well.

SOLUTION for (9-9.g) → 9-9-7-0:sg.pdf ▲

The crux of the Strang splitting method is to combine the evolution operators for the diffusion and the reaction term as $\boldsymbol{\Psi}^\tau := \boldsymbol{\Psi}_f^{\tau/2} \circ \boldsymbol{\Phi}_g^\tau \circ \boldsymbol{\Psi}_f^{\tau/2}$. This is what we are going to implement in the next subtasks.

(9-9.h) ⚄ (30 min.) In the file `fisherkpp.cc` you find an implementation of a class named **StrangSplit**. It constitutes the implementation for the evolution of our population density. Complete the implementation for the constructor in the file `fisherkpp.h`.

```
/* Main constructor */
template <typename DIFF_COEFF>
explicit StrangSplit(
  const std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>>
    fe_space,
  double T, unsigned int m, double lambda, DIFF_COEFF &&c);
```

The argument `fe_space` stores the finitea element space for our mesh $\mathcal{M}$, `T` and `m` denotes the final time and the number of timesteps, respectively, `lambda` represents the growth factor, and `c` is the function handle for the diffusion coefficient.

SOLUTION for (9-9.h) → 9-9-8-0:sh.pdf ▲

(9-9.i) ⚄ (30 min.) Complete the member function `diffusionEvolutionOperator()` in the class **StrangSplit** in the file `fisherkpp.cc`.

```
Eigen::VectorXd diffusionEvolutionOperator(double tau,
  const Eigen::VectorXd &mu);
```

Here, `tau` denotes the time step size, and `mu` is the basis expansion coefficient vector for the finite-element solution $u_N \in V_{0,h}$.

SOLUTION for (9-9.i) → `9-9-9-0:si.pdf` ▲

(9-9.j) ⚂ (20 min.) [depends on Sub-problem (9-9.i) and Sub-problem (9-9.g)]

Implement the member function `Evolution()` in the class **StrangSplit** which combines the evolution operators for the diffusion and the reaction term according to the Strang splitting method

$$\Psi^\tau := \Psi_f^{\tau/2} \circ \Phi_g^\tau \circ \Psi_f^{\tau/2} .$$

```
Eigen::VectorXd Evolution(Eigen::VectorXd &cap, const
                          Eigen::VectorXd &mu);
```

Again, the input parameter `mu` is the discrete approximation for the solution vector $u_N \in V_{0,h}$. Moreover, `cap` is an N-dimensional vector storing the values for the carrying capacity depending on the coordinates in the discrete mesh. Note that we solve the evolution for several time horizons, in each of which the carrying capacities are constant in time.

SOLUTION for (9-9.j) → `9-9-10-0:sj.pdf` ▲

(9-9.k) ⚄ (30 min.) Solutions of the Fisher/KPP evolution develop traveling wave fronts. Now we want to see this type of solution in a model problem on an island-shaped domain $\Omega \subset \mathbb{R}^2$.

The geometric information is provided in the form of a planar triangular mesh $\mathcal{M}$ in GMSH format in the file `island.msh` in the directory `meshes`.
In the file `fisherkpp_main.cc` you find the skeleton of a function `modelproblem()`, which uses your implementation of **StrangSplit** to simulation a Fisher/KPP evolution using piecewise linear Lagrangian finite elements $\mathcal{S}_1^0(\mathcal{M})$.

The mesh for the simulation run comprising $N = 6474$ nodes ▷

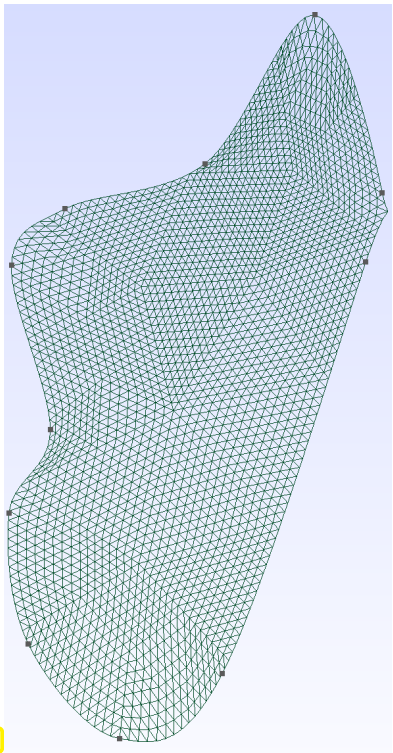


Fig. 136

To generate traveling wave fronts, you will find a setting of parameters provided in the function `modelproblem()`. The initial population is represented by two point loads, one on the upper part of the domain, one on the lower part. The growth factor and the diffusion coefficient are both set to be constant. Moreover, the carrying capacity is set to be constant as well.

You are asked to implement a visualization by means of ParaView [**ParaView**] of the solution after a total of 500 time steps by using the VTK-writer class **lf::io::VtkWriter** of LEHRFEM++. To this end, complete

the part on the visualization in the function `modelproblem` (file `fisherkpp_main.cc`) such that after every 100 time steps, you generate a VTK-file `sol<k>.vtk`, `k=1,2,3,4,5`. Finally, visualize the generated data with PARAVIEW.

SOLUTION for (9-9.k) → `9-9-11-0:sx.pdf` ▲

Acknowledgment. This homework problem has been prepared by A. Loher, student of CSE, who did a BSc project on "Solving the Fisher/KPP equation on the globe".

References

[Hip15] R. Hiptmair. "Maxwell's Equations: Continuous and Discrete". In: *Topics in Computational Electromagnetism*. Ed. by A. Bermúdez and A. Valli. Vol. 2148. Lecture Notes in Mathematics. Springer, 2015, pp. 1–58. DOI: 10.1007/978-3-319-19306-9_1 (cit. on p. 314).

End Problem 9-9, 255 min.

Problem 9-10: Mixed-Hybrid Finite-Element Method for the Wave Equation

This problem examines a non-standard spatial discretization of the 2D wave equation, called a **mixed-hybrid** method. It involves the implementation of the method of lines and of a fully discrete version.

This problem is based on [Lecture → Section 9.3.3] and [Lecture → Section 9.3.4].
▷ problem name/problem code folder: MixedFEMWave

In this problem the computational domain $\Omega \subset \mathbb{R}^2$ represents the surface of a water basin, on which we want to simulation the propagation of ripples. The shape of Ω is that of the letter **H**, see Fig. 142.

Ripples are excited by a cylinder periodically touching the water surface in the circular region B, see Fig. 142. This is modeled by the time-dependent function

$$f(\boldsymbol{x},t) = \begin{cases} \sin(t) & \text{for } \boldsymbol{x} \in B\,, \\ 0 & \text{elsewhere,} \end{cases}\,, \tag{9.10.1}$$

$$B := \{\boldsymbol{x} \in \Omega : \ \left\| \boldsymbol{x} - \begin{bmatrix} 1 \\ 2.5 \end{bmatrix} \right\|_2 < 1/2\}\,. \tag{9.10.2}$$

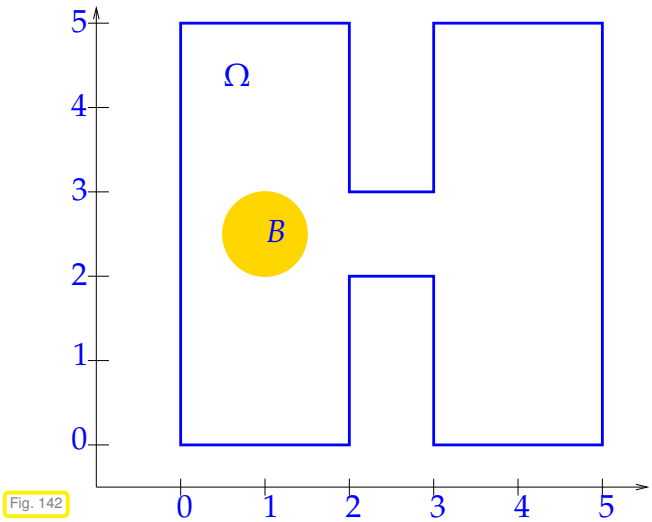


Fig. 142

A mathematical model boils down to the following evolution problem in variational form: Given a final time $T > 0$ seek $(u, \mathbf{j}) : [0,T] \to H^1_0(\Omega) \times (L^2(\Omega))^2$ such that for all $v \in H^1_0(\Omega)$ and $\mathbf{q} \in (L^2(\Omega))^2$

$$\begin{aligned} \int_\Omega \rho(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;-\; \int_\Omega \mathbf{j}(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_\Omega f(\boldsymbol{x},t)\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \frac{\partial \mathbf{j}}{\partial t}(\boldsymbol{x},t) \cdot \mathbf{q}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \mathbf{q}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} &= 0\,, \end{aligned} \tag{9.10.3}$$

$$u(\boldsymbol{x},0) = 0 \quad , \quad \mathbf{j}(\boldsymbol{x},0) = \mathbf{0} \quad \forall \boldsymbol{x} \in \Omega \tag{9.10.4}$$

Here $\rho \in C^0(\overline{\Omega})$ is a uniformly positive scalar-valued coefficient function.

(9-10.a) ⚀ (15 min.) Let $(u, \mathbf{j}) : [0,T] \to H^1_0(\Omega) \times (L^2(\Omega))^2$ be a solution of (9.10.3). Show that in the case $f \equiv 0$, the total **energy**

$$\mathcal{E}(t) := \tfrac{1}{2} \int_\Omega \rho(\boldsymbol{x}) u^2(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} + \tfrac{1}{2} \int_\Omega \|\mathbf{j}\|^2\,\mathrm{d}\boldsymbol{x} \tag{9.10.5}$$

is a *constant* function of time.

SOLUTION for (9-10.a) → 9-10-1-0:s2.pdf ▲

(9-10.b) ⚄ (15 min.) Determine two bilinear forms $\mathsf{m} : H^1_0(\Omega) \times H^1_0(\Omega) \to \mathbb{R}$, $\mathsf{a} : H^1_0(\Omega) \times H^1_0(\Omega) \to \mathbb{R}$, and a time-dependent linear form $\ell(t) : H^1_0(\Omega) \to \mathbb{R}$ such that the solution $u = u(t)$ of (9.10.3) is a solution of

$$u : [0,T] \to H^1_0(\Omega): \quad \mathsf{m}\big(\frac{d^2u}{dt^2}(t), v\big) + \mathsf{a}(u(t), v) = \ell(t)(v) \quad \forall v \in H^1_0(\Omega)\,. \tag{9.10.7}$$

HIDDEN HINT 1 for (9-10.b) → 9-10-2-0:s1h1.pdf

SOLUTION for (9-10.b) → 9-10-2-1:s1.pdf ▲

We equip Ω with a *triangular* finite-element mesh $\mathcal{M}$. In the spirit of the method of lines we perform a Galerkin finite-element semi-discretization of (9.10.3) relying on finite-dimensional subspaces $V_h \subset H^1_0(\Omega)$ and $Q_h \subset \left(L^2(\Omega)\right)^2$, which are endowed with ordered bases $\mathfrak{B}_V := \{b_h^1, \ldots, b_h^N\} \subset V_h$, $N := \dim V_h$, and $\mathfrak{B}_Q := \{\mathbf{q}_h^1, \ldots, \mathbf{q}_h^M\} \subset Q_h$, $M := \dim Q_h$.

(9-10.c) ⚂ (15 min.) The Galerkin semi-discretization leads to a system of ODEs of the form

$$\begin{aligned} \mathbf{M}_V \frac{d\vec{\boldsymbol{\mu}}}{dt}(t) \quad - \quad \mathbf{B}^\top \vec{\boldsymbol{\kappa}}(t) \quad &= \quad \vec{\boldsymbol{\varphi}}(t)\,, \\ \mathbf{M}_Q \frac{d\vec{\boldsymbol{\kappa}}}{dt}(t) \quad + \quad \mathbf{B}\vec{\boldsymbol{\mu}}(t) \quad &= \quad \mathbf{0}\,, \end{aligned} \tag{9.10.13}$$

where $\vec{\boldsymbol{\mu}}(t)$ and $\vec{\boldsymbol{\kappa}}(t)$ are the basis expansion coefficient vectors of approximations of $u(t)$ and $\mathbf{j}(t)$, respectively.

Give formulas for the entries of the matrices $\mathbf{B}$, $\mathbf{M}_V$, $\mathbf{M}_Q$, and of the vector $\vec{\boldsymbol{\varphi}}(t)$.

SOLUTION for (9-10.c) → 9-10-3-0:ss3.pdf ▲

Concretely, we use the following finite-element spaces

- $V_h := \mathcal{S}^0_{1,0}(\mathcal{M})$, the lowest-order Lagrangian finite-element space $\subset H^1_0(\Omega)$,
- $Q_h := (\mathcal{S}_0^{-1}(\mathcal{M}))^2$, the space of $\mathcal{M}$-piecewise constant vector fields on Ω.

For V_h we use the standard tent-function basis from [Lecture → Section 2.4.3], whereas the basis for Q_h is

$$\mathfrak{B}_Q := \left\{\mathbf{q}_h^1, \ldots, \mathbf{q}_h^M\right\}, \quad M := 2\sharp\mathcal{M}\,, \tag{9.10.14}$$

$$\mathbf{q}_h^{2j-1}(\boldsymbol{x}) = \begin{cases} \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] & \text{for } \boldsymbol{x} \in K_j\,, \\ \mathbf{0} & \text{elsewhere} \end{cases}, \quad \mathbf{q}_h^{2j}(\boldsymbol{x}) = \begin{cases} \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right] & \text{for } \boldsymbol{x} \in K_j\,, \\ \mathbf{0} & \text{elsewhere,} \end{cases} \quad j = 1, \ldots, \sharp\mathcal{M}\,, \tag{9.10.15}$$

where $K_1, \ldots, K_{\sharp\mathcal{M}}$ is some numbering of the cells of the mesh.

A LEHRFEM++-based implementation of the function

```
template <typename FFUNCTION>
Eigen::VectorXd computeRHS(
const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
   fe_space_V,
FFUNCTION &&f, double t);
```

is *already provided* in the file `mixedfemwave.h`. It computes the vector $\vec{\boldsymbol{\varphi}}(t) \in \mathbb{R}^{\widetilde{N}}$, $\widetilde{N} := \dim \mathcal{S}^0_1(\mathcal{M})$, as introduced in (9.10.13). The argument `fe_space_V` must contain a pointer to a **lf::uscalfe::FeSpaceLagrangeO1** object for the *full* finite-element space $\mathcal{S}^0_1(\mathcal{M})$ (including tent functions sitting on $\partial\Omega$). The functor `f` provides $f(\boldsymbol{x}, t)$. Finally, vector entries corresponding to basis functions on $\partial\Omega$ are set to zero and the resulting vector is returned.

For the approximate evaluation of the integrals it relies on the 2D local trapezoidal quadrature rule

$$\int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \tfrac{1}{3}|K|\left(\psi(\boldsymbol{a}_K^1) + \psi(\boldsymbol{a}_K^2) + \psi(\boldsymbol{a}_K^3)\right), \quad K \in \mathcal{M}\,, \tag{9.10.16}$$

where $\boldsymbol{a}_K^\ell$, $\ell = 1, 2, 3$, are the vertices of K.

(9-10.d) ⚂ (8 min.) Complete the following LEHRFEM++-based C++ code for the initialization of a **lf::assemble::DofHandler**-type that fits $Q_h := (\mathcal{S}_0^{-1}(\mathcal{M}))$ on a 2D hybrid mesh and the basis $\mathfrak{B}_Q$ introduced before.

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(),  },
  {lf::base::RefEl::kSegment(),  },
  {lf::base::RefEl::kTria(),  },
  {lf::base::RefEl::kQuad(),  }});
```

SOLUTION for (9-10.d) → 9-10-4-0:sdh.pdf ▲

(9-10.e) ⚂ (15 min.) [depends on Sub-problem (9-10.c)]

In the file `mixedfemwave.cc` implement an efficient LEHRFEM++-based function

```
Eigen::SparseMatrix<double>
computeMQ(const lf::assemble::DofHandler &dofh_Q) {
```

that accepts a **lf::assemble::DofHandler** object for the finite-element space Q_h and returns the matrix $\mathbf{M}_Q$ (in EIGEN's CRS format) as introduced in (9.10.13).

HIDDEN HINT 1 for (9-10.e) → 9-10-5-0:h51.pdf

SOLUTION for (9-10.e) → 9-10-5-1:sl5.pdf ▲

(9-10.f) ⚄ (20 min.) [depends on Sub-problem (9-10.c)]

In the file `mixedfemwave.h` implement an efficient LEHRFEM++-based function

```
template <typename RHOFUNCTION>
Eigen::SparseMatrix<double>
computeMV(std::shared_ptr<
lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space_V,
RHOFUNCTION &&rho);
```

that takes a pointer to a **lf::uscalfe::FeSpaceLagrangeO1** object for the finite-element space $\mathcal{S}_1^0(\mathcal{M})$, a functor object of signature **std**::function<**double**(Eigen::Vector2D)> passing the coefficient function $\boldsymbol{x} \mapsto \rho(\boldsymbol{x})$ and returns the finite-element Galerkin matrix $\widetilde{\mathbf{M}}_V \in \mathbb{R}^{\widetilde{N},\widetilde{N}}$, $\widetilde{N} := \dim \mathcal{S}_1^0(\mathcal{M})$ (in EIGEN's CRS format) for the bilinear form $(u,v) \mapsto \int_\Omega \rho(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$ for the trial/test space $\mathcal{S}_1^0(\mathcal{M})$ equipped with tent-function global shape functions.

For the evaluation of the local integrals over triangles rely on the 2D local trapezoidal quadrature rule

$$\int_K \psi(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \approx \tfrac{1}{3}|K|\big(\psi(\boldsymbol{a}_K^1) + \psi(\boldsymbol{a}_K^2) + \psi(\boldsymbol{a}_K^3)\big)\,, \quad K \in \mathcal{M}\,, \tag{9.10.18}$$

where $\boldsymbol{a}_K^\ell$, $\ell = 1,2,3$, are the vertices of K.

> The function is to return a matrix of size $\widetilde{N} \times \widetilde{N}$, where $\widetilde{N} := \dim \mathcal{S}_1^0(\mathcal{M})$ is the dimension of the *full* lowest-order Lagrangian finite element space including basis functions associated with nodes on the boundary.

HIDDEN HINT 1 for (9-10.f) → 9-10-6-0:lumping.pdf

SOLUTION for (9-10.f) → 9-10-6-1:sl6.pdf ▲

(9-10.g) ⚄ (15 min.) [depends on Sub-problem (9-10.c)]

In the file `mixedfemwave.cc` complete the implementation of the `Eval()` method of the class

```
class BElemMatProvider {
  public:
  explicit BElemMatProvider() = default;
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) {
     return true; }
  Eigen::Matrix<double, 2, 3> Eval(const lf::mesh::Entity &tria);
};
```

that complies with the LEHRFEM++ concept of **ENTITY_MATRIX_PROVIDER** and can be used together with LEHRFEM++'s assembly utility functions `AssembleMatrixLocally()` to compute the entries of the matrix $\mathbf{B}$ as introduced in (9.10.13) for the finite element spaces $\mathcal{S}_1^0(\mathcal{M})$ (no boundary conditions taken into account!) and $Q_h := (\mathcal{S}_0^{-1}(\mathcal{M}))^2$.

HIDDEN HINT 1 for (9-10.g) → `9-10-7-0:h71fe.pdf`

SOLUTION for (9-10.g) → `9-10-7-1:sepm.pdf` ▲

(9-10.h) ⚅ (15 min.) [depends on Sub-problem (9-10.g)]

In the file `mixedfemwave.cc` finish the implementation of the C++ function

```
Eigen::SparseMatrix<double>
computeB(
const lf::assemble::DofHandler &dofh_V,
const lf::assemble::DofHandler &dofh_Q);
```

that expects two **lf::assemble::DofHandler** objects for *full* the finite-element spaces $\mathcal{S}_1^0(\mathcal{M})$ and $Q_h := (\mathcal{S}_0^{-1}(\mathcal{M}))^2$, respectively, and returns the "truncated" matrix $\widetilde{\mathbf{B}} \in \mathbb{R}^{M,\widetilde{N}}$, $M := \dim Q_h$, $\widetilde{N} := \dim \mathcal{S}_1^0(\mathcal{M})$, in EIGEN's CRS sparse matrix format. Of course, $\widetilde{\mathbf{B}}$ is the $\mathcal{S}_1^0(\mathcal{M}) \times Q_h$ Galerkin matrix for the bilinear form $(v, \mathbf{q}) \mapsto \int_\Omega \mathbf{grad}\, v(\boldsymbol{x}) \cdot \mathbf{q}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$.

> By "truncated" we mean that all columns of $\widetilde{\mathbf{B}}$ corresponding to nodal basis functions of $\mathcal{S}_1^0(\mathcal{M})$ belonging to nodes on $\partial\Omega$ are set to zero.

HIDDEN HINT 1 for (9-10.h) → `9-10-8-0:h18.pdf`

SOLUTION for (9-10.h) → `9-10-8-1:sl8.pdf` ▲

(9-10.i) ⚅ (20 min.) [depends on Sub-problem (9-10.e), Sub-problem (9-10.f), and Sub-problem (9-10.g)]

Based on the functions

- `computeRHS()`
- `computeMQ()` from Sub-problem (9-10.e)
- `computeMV()` from Sub-problem (9-10.f)
- `computeB()` from Sub-problem (9-10.g)

complete the code of the function

```
template <typename RHOFUNCTION, typename FFUNCTION,
typename RECORDER = std::function<void(double, double)>>
```

```
std::pair<Eigen::VectorXd, Eigen::VectorXd> leapfrogMixedWave(
const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
   &fe_space_V,
const lf::assemble::UniformFEDofHandler &dofh_Q,
RHOFUNCTION &&rho, FFUNCTION &&f,
double T, unsigned int nb_timesteps,
RECORDER &&rec = [](double, double) {});
```

that implements leapfrog timestepping for (9.10.3)/(9.10.13) with uniform timestep $\tau := T/n$, n passed in `no_timesteps`:

$$\begin{aligned} \mathbf{M}_V \frac{\vec{\boldsymbol{\mu}}^{(j+1)} - \vec{\boldsymbol{\mu}}^{(j)}}{\tau} - \mathbf{B}^\top \vec{\boldsymbol{\kappa}}^{(j+\frac{1}{2})} &= \vec{\boldsymbol{\varphi}}(\tau(j+\tfrac{1}{2}))\,, \\ \mathbf{M}_Q \frac{\vec{\boldsymbol{\kappa}}^{(j+\frac{3}{2})} - \vec{\boldsymbol{\kappa}}^{(j+\frac{1}{2})}}{\tau} + \mathbf{B}\vec{\boldsymbol{\mu}}^{(j+1)} &= 0\,, \end{aligned} \qquad j = 0,\ldots,n\,, \tag{9.10.27}$$

$$\vec{\boldsymbol{\mu}}^{(0)} = \mathbf{0} \quad , \quad \vec{\boldsymbol{\kappa}}^{(\frac{1}{2})} = \mathbf{0}\,. \tag{9.10.28}$$

The source function $f = f(\boldsymbol{x}, t)$ is available through the lambda function `f()`. The function should return the basis expansion coefficient vectors of the finite-element approximations of $u(T)$ (first return value) and $\mathbf{j}(T)$ (second return value). In order to obtain the latter, compute $\vec{\boldsymbol{\kappa}}^{(n+\frac{1}{2})}$ and then take the average of that vector and $\vec{\boldsymbol{\kappa}}^{(n-\frac{1}{2})}$.

SOLUTION for (9-10.i) → `9-10-9-0:sol9.pdf` ▲

(9-10.j) ⚄ (10 min.) The function `leapfrogMixedWave()` also takes an argument `rec`, which passes a function object of signature `std::function<void(double, double)>`. Extend your implementation of `leapfrogMixedWave()` such that suitable approximations of

$$\int_\Omega \rho(\boldsymbol{x}) u(\boldsymbol{x}, j\tau)^2 \,\mathrm{d}\boldsymbol{x} \quad \text{and} \quad \int_\Omega \|\mathbf{j}(\boldsymbol{x}, j\tau)\|^2 \,\mathrm{d}\boldsymbol{x}\,, \quad j = 0,\ldots,n\,, \tag{9.10.30}$$

are passed to this function object as first and second argument, while the main timestepping loop is running.

Obtain an approximation of $u(\boldsymbol{x}, j\tau)$ from $\vec{\boldsymbol{\mu}}^{(j)}$, and one for $\mathbf{j}(\boldsymbol{x}, j\tau)$ from averaging $\vec{\boldsymbol{\kappa}}^{(j-\frac{1}{2})}$ and $\vec{\boldsymbol{\kappa}}^{(j+\frac{1}{2})}$.

HIDDEN HINT 1 for (9-10.j) → `9-10-10-0:half.pdf`

SOLUTION for (9-10.j) → `9-10-10-1:solaf.pdf` ▲

End Problem 9-10, 148 min.

Problem 9-11: Gauss-Lobatto IIIC Timestepping

This problem deals with a linear second-order parabolic evolution problem with Dirichlet boundary conditions. Method-of-lines semi-discretization in space is followed by discretization in time by means of an implicit Runge-Kutta single-step method of class Gauss-Lobatto IIIC.

This problem requires familiarity with [Lecture → Section 9.2.2], [Lecture → Section 9.2.4], [Lecture → Section 9.2.6.5], and [Lecture → Section 9.2.7.1] and proficiency in LEHRFEM++.
▷ problem name/problem code folder: GaussLobattoParabolic

Let $\Omega \subset \mathbb{R}^2$ be a domain with polygonal boundary. On it we consider the following parabolic initial-boundary value problem (IBVP) in variational form and zero initial conditions:

For a given final time $T > 0$ seek $u :]0,T[\to u(t) \in H^1(\Omega)$, $u(t)|_{\partial\Omega} = g(t)\mathbf{1}$, such that

$$\int_\Omega \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \qquad \forall v \in H^1_0(\Omega)\,, \tag{9.11.1a}$$

$$u(0) = 0 \qquad \text{in} \quad \Omega\,, \tag{9.11.1b}$$

where $\mathbf{1}$ stands for the constant function $\{\boldsymbol{x} \in \partial\Omega \mapsto 1\}$, and $g : [0,T] \to \mathbb{R}$ is continuously differentiable and satisfies $g(0) = 0$. This is a model for transient heat conduction in a body immersed in an environment with uniform but changing temperature.

(9-11.a) ⚂ (10 min.) Write the the strong form of the evolution problem, whose weak form is given in (9.11.1).

SOLUTION for (9-11.a) → 9-11-1-0:s1.pdf ▲

We pursue the method-of-lines approach according to [Lecture → Section 9.2.4]. We equip Ω with a 2D hybrid mesh $\mathcal{M}$. For spatial discretization we employ a Lagrangian finite element space $\mathcal{S}^0_p(\mathcal{M})$ of degree $p \in \mathbb{N}$ and use a standard finite element basis $\mathfrak{B}_h \subset \mathcal{S}^0_p(\mathcal{M})$, that is, the global shape functions are associated with nodes, edges, and cells of the mesh and they form a cardinal basis with respect to suitable sets of interpolation nodes.

For the theoretical parts of this problem we assume the following ordering of the basis functions: We first number the $N_0 \in \mathbb{N}$ global shape functions belonging to mesh entities located in the interior of Ω, and then number the $N_\partial \in \mathbb{N}$ basis functions associated with mesh entities on $\partial\Omega$:

$$\mathfrak{B}_h = \{b_h^1,\ldots,b_h^{N_0}\} \cup \{b_h^{N_0+1},\ldots,b_h^N\}\,, \quad N := N_0 + N_\partial = \dim \mathcal{S}^0_p(\mathcal{M})\,.$$

This induces a (block) partitioning of all basis expansion coefficient vectors and Galerkin matrices, e.g., $\vec{\boldsymbol{\mu}} = \begin{bmatrix} \vec{\boldsymbol{\mu}}_0 \\ \vec{\boldsymbol{\mu}}_\partial \end{bmatrix} \in \mathbb{R}^N$, $\vec{\boldsymbol{\mu}}_0 \in \mathbb{R}^{N_0}$, $\vec{\boldsymbol{\mu}}_\partial \in \mathbb{R}^{N_\partial}$.

Then the spatially semidiscrete variational formulation of (9.11.1) is: seek $u_h :]0,T[\to u(t) \in \mathcal{S}^0_p(\mathcal{M})$, $u_h(t)|_{\partial\Omega} = g(t)\mathbf{1}$ for all $0 < t < T$, such that

$$\int_\Omega \frac{\partial u_h}{\partial t}(\boldsymbol{x},t)\, v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u_h(\boldsymbol{x},t)\cdot\mathbf{grad}\, v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \qquad \forall v_h \in \mathcal{S}^0_{p,0}(\mathcal{M})\,, \tag{9.11.3a}$$

$$u_h(0) = 0 \qquad \text{in} \quad \Omega\,, \tag{9.11.3b}$$

(9-11.b) ⚄ (10 min.) The method-of-lines ODE satisfied by the basis expansion coefficient vector

$\vec{\mu}(t) = \begin{bmatrix} \vec{\mu}_0(t) \\ \vec{\mu}_\partial(t) \end{bmatrix}$ of the solution $u_h(t)$ of (9.11.3) is of the form

$$\begin{bmatrix} \mathbf{M}_{00} & \mathbf{M}_{0,\partial} \\ \mathbf{O} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \dot{\vec{\mu}}_0(t) \\ \dot{\vec{\mu}}_\partial(t) \end{bmatrix} + \begin{bmatrix} \mathbf{A}_{00} & \mathbf{A}_{0,\partial} \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\mu}_0(t) \\ \vec{\mu}_\partial(t) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ g(t)\vec{\beta}_\partial \end{bmatrix} . \tag{9.11.4}$$

Give formulas for the entries of the matrices $\mathbf{M}_{00}$, $\mathbf{M}_{0,\partial}$, $\mathbf{A}_{00}$, $\mathbf{A}_{0,\partial}$, and of the vector $\vec{\beta}_\partial$.

SOLUTION for (9-11.b) → 9-11-2-0:solmol.pdf ▲

In the sequel we exclusively focus on the lowest-degree Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$, that is, the case $p = 1$.

(9-11.c) ⚄ (16 min.) [depends on Sub-problem (9-11.b)]

The method-of-lines ODE as introduced in Sub-problem (9-11.b)

$$\begin{bmatrix} \mathbf{M}_{00} & \mathbf{M}_{0,\partial} \\ \mathbf{O} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \dot{\vec{\mu}}_0(t) \\ \dot{\vec{\mu}}_\partial(t) \end{bmatrix} + \begin{bmatrix} \mathbf{A}_{00} & \mathbf{A}_{0,\partial} \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\mu}_0(t) \\ \vec{\mu}_\partial(t) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ g(t)\vec{\beta}_\partial \end{bmatrix} \tag{9.11.4}$$

can compactly be written as

$$\widetilde{\mathbf{M}}\dot{\vec{\mu}}(t) + \widetilde{\mathbf{A}}\vec{\mu}(t) = \vec{\varphi}(t) , \quad \vec{\mu}(t) := \begin{bmatrix} \vec{\mu}_0 \\ \vec{\mu}_\partial \end{bmatrix} , \tag{9.11.5}$$

with matrices $\widetilde{\mathbf{M}}, \widetilde{\mathbf{A}} \in \mathbb{R}^{N,N}$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$.

In the file `gauslobattoparabolic.cc` complete the implementation of a LEHRFEM++-based C++ function

```
lf::assemble::COOMatrix<double>
  initMbig(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space);
```

that computes the matrix $\widetilde{\mathbf{M}}$ based on the information about the mesh and the numbering of local/global shape function passed in `fe_space`. The function is supposed to return the matrix $\widetilde{\mathbf{M}}$ in triplet/COO format.

HIDDEN HINT 1 for (9-11.c) → 9-11-3-0:hord.pdf

HIDDEN HINT 2 for (9-11.c) → 9-11-3-1:hrdem.pdf

HIDDEN HINT 3 for (9-11.c) → 9-11-3-2:glphz.pdf

SOLUTION for (9-11.c) → 9-11-3-3:sol3.pdf ▲

(9-11.d) ⚅ (20 min.) [depends on Sub-problem (9-11.c)]

In the file `gauslobattoparabolic.cc` complete the implementation of a LEHRFEM++-based C++ function

```
lf::assemble::COOMatrix<double>
  initAbig(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space);
```

that computes the matrix $\widetilde{\mathbf{A}}$ from (9.11.5) based on the information about the mesh and the numbering of local/global shape function passed in `fe_space`. The function is supposed to return the matrix $\widetilde{\mathbf{A}}$ in triplet/COO format.

HIDDEN HINT 1 for (9-11.d) → 9-11-4-0:hrdem.pdf

SOLUTION for (9-11.d) → 9-11-4-1:sl4.pdf ▲

(9-11.e) ⚂ (10 min.) [depends on Sub-problem (9-11.b)]

Show that for the matrices introduced in (9.11.5) their weighted sum $\widetilde{\mathbf{M}} + \zeta\widetilde{\mathbf{A}}$ is invertible for any $\zeta > 0$.

SOLUTION for (9-11.e) → 9-11-5-0:s4a.pdf ▲

(9-11.f) ⚄ (12 min.) [depends on Sub-problem (9-11.b)]

In the files `gauslobattoparabolic.h` and `gauslobattoparabolic.cc` finish the implementation of the class

```
class RHSProvider {
public:
  // Disabled constructors
  RHSProvider() = delete;
  RHSProvider(const RHSProvider &) = delete;
  RHSProvider(RHSProvider &&) = delete;
  RHSProvider &operator=(const RHSProvider &) = delete;
  RHSProvider &operator=(const RHSProvider &&) = delete;
  // Main constructor; precomputations to be done here
  RHSProvider(const lf::assemble::DofHandler &dofh,
              std::function<double(double)> g);
  // Destructor
  virtual ~RHSProvider() = default;
  /* Class member functions */
  // Evaluation operator for right-hand-side vector
  Eigen::VectorXd operator()(double t) const;
private:
  std::function<double(double)> g_;
  //====================
  // Other data members to be introduced by you
  //====================
};
```

The evaluation operator **operator** `()`, when supplied with a single time-argument `t` should return $\vec{\varphi}(t)$ as defined in (9.11.5). The mesh information and the local-to-global index mapping for the finite-element space $\mathcal{S}_1^0(\mathcal{M})$ is passed to the constructor through `dofh`. The constructor also gets the function g as argument `g`.

Your implementation should be efficient in the sense that invoking **operator** `()` should be as cheap as possible.

SOLUTION for (9-11.f) → 9-11-6-0:sol5.pdf ▲

In order to discretize (9.11.5) in time we rely on the **Gauss-Lobatto IIIC** implicit 2-stage Runge-Kutta single step method described by the Butcher scheme [Lecture → (7.3.3.3)].

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|cc} 0 & \frac{1}{2} & -\frac{1}{2} \\ 1 & \frac{1}{2} & \frac{1}{2} \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} . \tag{9.11.10}$$

(9-11.g) ⚂ (15 min.)

According to [Lecture → Def. 7.3.3.1], the application of s-stage one-step Runge-Kutta methods to

(9.11.5) entails the computation of s increment vectors $\vec{\kappa}_1, \ldots, \vec{\kappa}_s \in \mathbb{R}^N$ in each timestep, which can be obtained simultaneously by solving a linear system of equations. Describe this linear system of equations (in block form) for the Gauss-Lobatto IIIC implicit 2-stage Runge-Kutta single step method (9.11.10) when applied to (9.11.5) with uniform timestep $\tau > 0$.

HIDDEN HINT 1 for (9-11.g) → 9-11-7-0:h6x.pdf

SOLUTION for (9-11.g) → 9-11-7-1:sxs.pdf ▲

(9-11.h) ⚄ (30 min.) [depends on Sub-problem (9-11.c), Sub-problem (9-11.d), Sub-problem (9-11.f), and Sub-problem (9-11.g)]

In the file `gauslobattoparabolic.h` supplement the missing parts of the C++ function

```
template <typename GFUNCTION>
Eigen::VectorXd evolveIBVPGaussLobatto(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
      fe_space,
    double T, unsigned int M,
    GFUNCTION &&g);
```

that carries out $M \in \mathbb{N}$ (passed in `M`) equidistant timesteps of the Gauss-Lobatto IIIC implicit Runge-Kutta single step method (9.11.10) in order to solve the semi-discrete evolution (9.11.3) on the time interval $[0, T]$. Details of the lowest-order Lagrangian finite-element space $\mathcal{S}_1^0(\mathcal{M})$ are given in `fe_space`. The functor argument `g` describes the function $g : [0, T] \to \mathbb{R}$. The function should return the basis expansion coefficient vector for $u_h(T)$.

Your implementation should be as efficient as possible by reducing the amount of computation inside the main timestepping loop to a minimum.

HIDDEN HINT 1 for (9-11.h) → 9-11-8-0:h61x.pdf

HIDDEN HINT 2 for (9-11.h) → 9-11-8-1:h62.pdf

SOLUTION for (9-11.h) → 9-11-8-2:sp6sol.pdf ▲

(9-11.i) ⚄ (10 min.) Available is a sequence of meshes $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \ldots$ created by uniform regular refinement. Write $u_k^{(j)} \in \mathcal{S}_1^0(\mathcal{M}_k)$, $k = 0, 1, 2, \ldots$, $j = 0, \ldots, M_k$, M_k the number of uniform timesteps on mesh $\mathcal{M}_k$, for the fully discrete solution as obtained by invoking `evolveIBVPGaussLobatto()` from Sub-problem (9-11.h).

Our main interest is in the following norm of the spatio-temporal discretization error

$$\left(\tau \sum_{j=1}^{M_k} |u(\tau j) - u_k(\tau j)|^2_{H^1(\Omega)}\right)^{\frac{1}{2}}.$$

The Gauss-Lobatto IIIC implicit Runge-Kutta single step method is of *second order*. How does one have to tie the timestep size $\tau_k > 0$ on $\mathcal{M}_k$ to the meshwidth h_k of $\mathcal{M}_k$ in order to balance spatial and temporal discretization error. Smoothness of the solution $t \mapsto u(t)$ in space and time can be assumed.

SOLUTION for (9-11.i) → 9-11-9-0:s9.pdf ▲

End Problem 9-11, 133 min.

Problem 9-12: The Sobolev Evolution Initial-Boundary Value Problem

This problem discusses a class of second-order scalar linear evolution problems that arise in quasi-stationary models describing the transient behavior of slowly varying electromagnetic fields. In mathematics these special evolution problems have been named after S. Sobolev, the same Soviet mathematician, who also lent his name to the Sobolev spaces.

This problem centers around the analysis and LEHRFEM++-based implementation of method-of-lines timestepping for Sobolev evolution problems.
▷ problem name/problem code folder: SobolevEvolutionProblem

Let $\Omega \subset \mathbb{R}^2$ a bounded domain, and $\alpha, \beta \in C^0(\overline{\Omega})$ uniformly positive [Lecture → Def. 1.2.2.9], real-valued, and bounded coefficient functions. The spatial variational form of a **Sobolev evolution problem** on the time interval $[0,T]$ and with homogeneous Dirichlet boundary conditions and no excitation seeks $u : [0,T] \to H^1_0(\Omega)$ such that

$$\begin{aligned} \mathsf{b}(\dot{u}(t), v) + \mathsf{a}(u(t), v) &= 0 \quad \forall v \in H^1_0(\Omega) \,, \\ u(0) &= u_0 \,, \end{aligned} \tag{9.12.1}$$

where $u_0 \in H^1_0(\Omega)$ is a given initial value and the two bilinear forms are defined as

$$\begin{aligned} \mathsf{b}(w,v) &:= \int_\Omega \beta(\boldsymbol{x})\, \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,, \\ \mathsf{a}(w,v) &:= \int_\Omega \alpha(\boldsymbol{x})\, \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,, \end{aligned} \qquad w, v \in H^1(\Omega) \,. \tag{9.12.2}$$

(9-12.a) ⚂ (15 min.) Determine the partial differential equation occurring in the strong form ("PDE form") of the evolution problem (9.12.1).

SOLUTION for (9-12.a) → 9-12-1-0:s1sob.pdf ▲

(9-12.b) ⚄ (20 min.) Give an explicit analytic solution of (9.12.1) if $\alpha(\boldsymbol{x}) = \lambda\beta(\boldsymbol{x})$ for all $\boldsymbol{x} \in \Omega$ and for some $\lambda > 0$?

$$u(\boldsymbol{x},t) = \boxed{} \,, \quad (\boldsymbol{x},t) \in \Omega \times]0,T[\,.$$

SOLUTION for (9-12.b) → 9-12-2-0:s2sob.pdf ▲

(9-12.c) ⚄ (20 min.) The s.p.d. bilinear forms $\mathsf{b}, \mathsf{a} : H^1_0(\Omega) \times H^1_0(\Omega) \to \mathbb{R}$ induce norms $\|\cdot\|_b$ and $\|\cdot\|_a$ respectively. Give a sharp and explicit time-dependent bound for the time-dependent quantities $\|u(t)\|_b$ and $\|u(t)\|_a$.

HIDDEN HINT 1 for (9-12.c) → 9-12-3-0:sob3h1.pdf

SOLUTION for (9-12.c) → 9-12-3-1:.pdf ▲

We perform a method-of-lines spatial semdiscretization of (9.12.1) relying on the N-dimensional, $N \in \mathbb{N}$, trial and test space $V_{0,h} \subset H^1_0(\Omega)$ and using the basis $\{b^1_h, \ldots, b^N_h\}$ of $V_{0,h}$. This leads to the following initial-value problem for the time-dependent basis expansion coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t)$:

$$\mathbf{B}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = 0 \quad , \quad \vec{\boldsymbol{\mu}}(0) = \vec{\boldsymbol{\mu}}_0 \,, \tag{9.12.6}$$

where $\vec{\boldsymbol{\mu}}_0$ is related to a suitable projection of u_0 onto the finite element space.

(9-12.d) ⚄ (20 min.) As in Sub-problem (9-12.b) we assume $a(\boldsymbol{x}) = \lambda\beta(\boldsymbol{x})$ for all $\boldsymbol{x} \in \Omega$ and some $\lambda > 0$. To discretize (9.12.6) in time we apply a Runge-Kutta single-step method with constant timestep $\tau := T/M$, $M \in \mathbb{N}$ and, thus, obtain a sequence $\left(\vec{\boldsymbol{\mu}}^{(j)}\right)_{j=0}^{M}$ of approximations for $\vec{\boldsymbol{\mu}}(j\tau)$. Give an explicit formula for $\vec{\boldsymbol{\mu}}^{(j)}$ relying on the stability function $z \mapsto S(z)$ of the Runge-Kutta single-step method according to [Lecture → Thm. 7.3.4.4]. Which condition does τ have to satisfy in order to avoid a blow-up of $\vec{\boldsymbol{\mu}}^{(j)}$ for $j \to \infty$?

SOLUTION for (9-12.d) → 9-12-4-0:sobs4.pdf ▲

(9-12.e) ⚄ (60 min.) In the file `sobolevevolutionproblem.cc` complete the implementation of a LEHRFEM++-based C++ function

```
template <typename MESHFUNCTION_BETA, typename MESHFUNCTION_ALPHA>
Eigen::VectorXd solveRKSobEvl(
    std::shared_ptr<const lf::fe::ScalarFESpace<double>>
      fe_space_p,
    const MESHFUNCTION_BETA &beta, const MESHFUNCTION_ALPHA &alpha,
    const Eigen::VectorXd &mu0, double T,
    const Eigen::MatrixXd &RK_Mat, const Eigen::VectorXd &b,
    unsigned int M);
```

which uses an *explicit* Runge-Kutta single-step method (RK-SSM) and M uniform timesteps to solve the method-of-lines initial-value problem (9.12.6) over $[0, T]$. Its arguments have the following meaning:

1. `fe_space_p` supplies a pointer to an **lf::fe::ScalarFESpace** object describing the Lagrangian finite element space used for spatial discretization.
2. `beta`, `alpha` are **MeshFunction**-compatible objects with information about the coefficients β, α in (9.12.2).
3. `mu0` contains the coefficient vector $\vec{\boldsymbol{\mu}}_0$.
4. `T` is the final time.
5. `RK_Mat` and `b` are the Butcher matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the weight vector $\mathbf{b} \in \mathbb{R}^s$ in the **Butcher scheme** [Lecture → (6.4.0.11)] characterizing the explicit s-stage RK-SSM.
6. `M` is the number of uniform timesteps of size $\tau := T/M$.

The function is to return the basis expansion coefficient vector $\vec{\boldsymbol{\mu}}^{(M)} \approx \vec{\boldsymbol{\mu}}(T)$.

HIDDEN HINT 1 for (9-12.e) → 9-12-5-0:sob5h1.pdf

SOLUTION for (9-12.e) → 9-12-5-1:sobs5.pdf ▲

End Problem 9-12, 135 min.

Problem 9-13: 1D Wave Equation with Perfectly Matched Layers

Ideal absorbing boundary conditions for the linear wave equations are meant to let pass outgoing waves unaffected, which makes possible the artificial truncation of otherwise unbounded computational domains. In this problem we will learn about a widely used way to realize absorbing boundary conditions in 1D, the **perfectly matched layer (PML)** technique. It is hugely popular in computational engineering, because it can easily be extended to higher dimensions.

This problem is linked to [Lecture → Section 9.3.2] and assumes knowledge about the method of lines approach [Lecture → Section 9.3.3] and timestepping schemes for wave equations [Lecture → Section 9.3.4]. It involves EIGEN-based implementation in C++.
▷ problem name/problem code folder: PML1D

As in [Lecture → § 9.3.2.1] we consider the Cauchy problem for the 1D wave equation (on the unbounded domain $\Omega = \mathbb{R}$ the time interval $]0,T[$, $T > 0$):

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} - \frac{\partial}{\partial x}\left(\gamma(x)\frac{\partial u}{\partial x}\right) &= 0 \qquad \text{on } \Omega\times]0,T[, \\ u(x,0) = u_0(x), \quad \frac{\partial u}{\partial t}(x,0) &= v_0(x) \quad \text{on } \Omega\,, \end{aligned} \tag{9.13.1}$$

where $u_0, v_0 : \mathbb{R} \to \mathbb{R}$ are given initial data, $\gamma : \mathbb{R} \to \mathbb{R}$ is a *uniformly positive* continuous function and

$$\gamma(x) = 1 \quad \text{for } x \geq 1,\ x \leq -1, \quad \text{and} \quad \operatorname{supp}(u_0), \operatorname{supp}(v_0) \subset]-1,1[\,.$$

(9-13.a) ⚄ (5 min.) State the standard (spatial) variational formulation of (9.13.1) on $]-1,1[$, if homogeneous Neumann boundary conditions,

$$\frac{\partial u}{\partial x}(-1,t) = 0 \quad , \quad \frac{\partial u}{\partial x}(1,t) = 0\,, \quad t \in]0,T[\,, \tag{9.13.2}$$

are imposed at $x = \pm 1$.

SOLUTION for (9-13.a) → 9-13-1-0:pmlAs.pdf ▲

Concerning (9.13.1), we are only interested in that part of the solutions of (9.13.1) that lies inside $[-1,1]$. Nevertheless we can not simply restrict problem (9.13.1) to $[-1,1]$, for instance by imposing Dirichlet boundary conditions, since reflected waves would completely supersede the solution of the Cauchy problem after a short time, see [Lecture → § 9.3.2.9].

Instead we have to use absorbing boundary conditions that let waves pass undisturbed. One option are so-called **perfectly matched layers** (PML). This technique is based on introducing an artificial material in a zone outside the region of interest that absorbes waves. In the present setting these zones are $[-1-L,-1]$ and $[1,1+L]$ with width parameter $L > 0$.

The variational formulation of the PML augmented 1D wave equation then reads as follows.

Seek

$$u :]0,T[\to \in H^1(]-1-L,1+L[) \quad \text{and} \quad v :]0,T[\to \in L^2(]-1-L,1+L[)$$

such that for all $w \in H^1(]-1-L,1+L[)$ and $q \in L^2(]-1-L,1+L[)$

$$\begin{aligned} \int_{-1-L}^{1+L} \frac{\partial u}{\partial t}(x,t)\, w(x)\, \mathrm{d}x + \int_{-1-L}^{1+L} \sigma(x) u(x,t)\, w(x)\, \mathrm{d}x + \int_{-1-L}^{1+L} v(x,t)\, \frac{\partial w}{\partial x}(x)\, \mathrm{d}x &= \int_{-1-L}^{1+L} v_0(x)\, w(x)\, \mathrm{d}x\,, \\ \int_{-1-L}^{1+L} \frac{\partial v}{\partial t}(x,t)\, q(x)\, \mathrm{d}x + \int_{-1-L}^{1+L} \sigma(x) v(x,t)\, q(x)\, \mathrm{d}x - \int_{-1-L}^{1+L} \gamma(x) \frac{\partial u}{\partial x}(x,t)\, q(x)\, \mathrm{d}x &= 0\,, \end{aligned} \tag{9.13.6}$$

with a *non-negative* continuous function $\sigma : [-1-L,1+L] \to \mathbb{R}$ that vanishes on $[-1,1]$: $\sigma|_{[-1,1]} = 0$.

(9-13.b) ⚀ (5 min.) State the variational problem (9.13.6) for $L = 0$, which means that the absorbing layer is ignored.

SOLUTION for (9-13.b) → 9-13-2-0:pmlsa.pdf ▲

(9-13.c) ⚁ (10 min.) Show that the two variational formulations obtained in Sub-problem (9-13.a) and Sub-problem (9-13.b) are equivalent, when one makes the substitution

$$v(x,t) = \gamma(x) \int_0^t \frac{\partial u}{\partial x}(x,\tau)\, \mathrm{d}\tau \quad \Leftrightarrow \quad \frac{\partial v}{\partial t} = \gamma(x) \frac{\partial u}{\partial x}\,. \tag{9.13.10}$$

HIDDEN HINT 1 for (9-13.c) → 9-13-3-0:pmlbh1.pdf

SOLUTION for (9-13.c) → 9-13-3-1:pmlsb.pdf ▲

(9-13.d) ⚂ (15 min.) From (9.13.6) derive an *explicit expression* for the temporal derivative of the time-dependent total energy

$$E(t) := \frac{1}{2} \int_{-1-L}^{1+L} \left| \frac{\partial u}{\partial t}(x,t) \right|^2 + \frac{1}{\gamma(x)} \left| \frac{\partial v}{\partial t}(x,t) \right|^2 \mathrm{d}x\,, \quad 0 \le t \le T\,. \tag{9.13.13}$$

$$\frac{dE}{dt}(t) = \qquad .$$

HIDDEN HINT 1 for (9-13.d) → 9-13-4-0:pml0h1.pdf

SOLUTION for (9-13.d) → 9-13-4-1:.pdf ▲

Now we tackle the full discretization of (9.13.6) and we pursue the method-of-lines policy. In particular we resort to a Galerkin finite element discretization in space based on a on a mesh $\mathcal{M}$ with $N+1$, $N \in \mathbb{N}$, equidistant nodes $x_0 := -1-L < x_1 \le \cdots < x_{N-1} < x_N := 1+L$.

- Then, in (9.13.6), we replace the space $H^1(]-1-L,1+L[)$ with the space $\mathcal{S}_1^0(\mathcal{M})$ of piecewise linear continuous functions, see [Lecture → § 2.3.1.4].
- As trial and test space for $L^2(]-1-L,1+L[)$ we choose the space $\mathcal{S}_0^{-1}(\mathcal{M})$ of piecewise constant discontinuous functions on $\mathcal{M}$.

For both spaces we opt for the canonical *locally supported nodal basis functions*:

- "tent functions" $b_h^0,\ldots,b_n^N$ according to [Lecture → (2.3.2.1)] for $\mathcal{S}_1^0(\mathcal{M})$, and
- the characteristic functions $\chi_h^1,\ldots\chi_h^N$ of the mesh cells $]x_0,x_1[,\ldots,]x_{N-1},x_N[$ for $\mathcal{S}_0^{-1}(\mathcal{M})$.

The resulting ODE system is discretized via a special variant of Crank-Nicolson timestepping [Lecture → § 9.3.4.7] with uniform timestep $\tau>0$. In the $k+1$-th timestep it leads to the following discrete variational equation:

Seek $u_h^{(k+1)}\in\mathcal{S}_1^0(\mathcal{M})$ and $v_h^{(k+1)}\in\mathcal{S}_0^{-1}(\mathcal{M})$ such that

$$\int_{-1-L}^{L+1}\frac{u_h^{(k+1)}-u_h^{(k)}}{\tau}w_h\,\mathrm{d}x+\int_{-1-L}^{L+1}\sigma\frac{u_h^{(k+1)}+u_h^{(k)}}{2}w_h\,\mathrm{d}x+\int_{-1-L}^{L+1}\frac{v_h^{(k+1)}+v_h^{(k)}}{2}\frac{\partial w_h}{\partial x}\,\mathrm{d}x = \int_{-1-L}^{L+1}v_0\,w_h\,\mathrm{d}x\,,$$

$$\int_{-1-L}^{L+1}\frac{v_h^{(k+1)}-v_h^{(k)}}{\tau}q_h\,\mathrm{d}x+\int_{-1-L}^{L+1}\sigma\frac{v_h^{(k+1)}+v_h^{(k)}}{2}q_h\,\mathrm{d}x-\int_{-1-L}^{L+1}\gamma(x)\frac{\partial\frac{1}{2}(u_h^{(k+1)}+u_h^{(k)})}{\partial x}q_h\,\mathrm{d}x=0$$

for all $w_h\in\mathcal{S}_1^0(\mathcal{M})$ and $q_h\in\mathcal{S}_0^{-1}(\mathcal{M})$.

This scheme will underly the implementation requested in this problem.

(9-13.e) ⚄ (18 min.) Write $\vec{\mu}^{(k)}$ and $\vec{v}^{(k)}$ for the basis expansion coefficient vectors for $u_h^{(k)}$ and $v_h^{(k)}$, respectively. Furthermore, define the following Galerkin matrices:

- The $\mathcal{S}_1^0(\mathcal{M})\times\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix (mass matrix) for the bilinear form $(u,w)\mapsto\int_{-1-L}^{1+L}\mu(x)u(x)w(x)\,\mathrm{d}x$:

$$\mathbf{M}_\mu^1:=\left[\int_{-1-L}^{1+L}\mu(x)b_h^j(x)b_h^i(x)\,\mathrm{d}x\right]_{i,j=0}^{N}\in\mathbb{R}^{N+1,N+1}\,. \tag{9.13.16}$$

- The $\mathcal{S}_0^{-1}(\mathcal{M})\times\mathcal{S}_0^{-1}(\mathcal{M})$ Galerkin matrix (another mass matrix) for the bilinear form $(v,q)\mapsto\int_{-1-L}^{1+L}\mu(x)v(x)q(x)\,\mathrm{d}x$:

$$\mathbf{M}_\mu^0:=\left[\int_{-1-L}^{1+L}\mu(x)\chi_h^j(x)\chi_h^i(x)\,\mathrm{d}x\right]_{i,j=1}^{N}\in\mathbb{R}^{N,N}\,. \tag{9.13.17}$$

- The $\mathcal{S}_0^{-1}(\mathcal{M})\times\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrices (another mass matrix) for the bilinear form $(v,w)\mapsto\int_{-1-L}^{1+L}\mu(x)v(x)\frac{dw}{dx}(x)\,\mathrm{d}x$:

$$\mathbf{B}_\mu:=\left[\int_{-1-L}^{1+L}\mu(x)\chi_h^j(x)\frac{db_h^i}{dx}(x)\,\mathrm{d}x\right]_{\substack{i=0,\ldots,N\\ j=1,\ldots,N}}\in\mathbb{R}^{N+1,N}\,. \tag{9.13.18}$$

They depend on a generic function $\mu \in C^0([-1-L, 1+L])$ and this dependence will be indicated by a subscript. We also define the vector

$$\vec{\varphi}_0 := \left[\int_{-1-L}^{1+L} v_0(x) b_h^i(x)\right]_{i=0}^{N} \in \mathbb{R}^{N+1} .$$

Using the notations introduced above, state the linear systems of equations that have to be solved to obtain $\vec{\mu}^{(k+1)}$ and $\vec{v}^{(k+1)}$ from $\vec{\mu}^{(k)}$ and $\vec{v}^{(k)}$.

$$\begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix} \begin{bmatrix} \vec{\mu}^{(k+1)} \\ \vec{v}^{(k+1)} \end{bmatrix} = \begin{bmatrix} \square \\ \square \end{bmatrix} .$$

SOLUTION for (9-13.e) → 9-13-5-0:pml1s1.pdf ▲

(9-13.f) ⚂ (15 min.) Based on the local trapezoidal quadrature rule [Lecture → (2.3.3.8)]

$$\int_{x_{j-1}}^{x_j} \phi(\xi)\,\mathrm{d}\xi \approx \tfrac{1}{2}|x_j - x_{j-1}|\big(\phi(x_j) + \phi(x_{j-1})\big) \tag{9.13.21}$$

compute the element matrices for a mesh cell $]x_{j-1}, x_j[$, and the bilinear forms

(i) $(u_h, w_h) \mapsto \int_{-1-L}^{1+L} \mu(x) u_h(x) w_h(x)\,\mathrm{d}x$, $u_h, w_h \in \mathcal{S}_1^0(\mathcal{M})$,

► element matrix = ,

(ii) $(u_h, w_h) \mapsto \int_{-1-L}^{1+L} \mu(x) u_h(x) w_h(x)\,\mathrm{d}x$, $u_h, w_h \in \mathcal{S}_0^{-1}(\mathcal{M})$,

► element matrix = ,

(iii) $(v_h, w_h) \mapsto \int_{-1-L}^{1+L} \mu(x) v_h(x) \dfrac{\partial w_h}{\partial x}(x)\,\mathrm{d}x$, $v_h \in \mathcal{S}_0^{-1}(\mathcal{M})$, $w_h \in \mathcal{S}_1^0(\mathcal{M})$,

► element matrix = ,

SOLUTION for (9-13.f) → 9-13-6-0:pml2s.pdf ▲

(9-13.g) ⚄ (45 min.) In the file `pml1d.h` complete the implementation of the EIGEN-based C++ function

```
template <typename RECORDER =
          std::function<void(const Eigen::VectorXd &)>>
Eigen::VectorXd solve1DWavePML(
  const Eigen::VectorXd &zeta_0, const Eigen::VectorXd &gamma,
  const Eigen::VectorXd &sigma, unsigned int M, double T,
  RECORDER &&rec = [](Eigen::VectorXd &) -> void {});
```

The vector argument `zeta_0` is meant to supply the vector $\left[\vec{\mu}^{(0)},\vec{v}^{(0)}\right]^\top \in \mathbb{R}^{2N+1}$. The arguments `gamma` and `sigma` give the values of the coefficient functions γ and σ sampled at the nodes of the mesh $\mathcal{M}$. Calling this function should carry out `M` uniform timesteps of size $\tau := \frac{T}{M}$ as defined above. The final vector $\vec{\zeta}^{(M)} = \left[\vec{\mu}^{(M)},\vec{v}^{(M)}\right]^\top$ is returned.

The argument `rec` can be used to provide a recorder object, which will be given the vectors $\vec{\zeta}^{(k)}$, $k = 0,\ldots,M$, during the execution of the main timestepping loop.

The following listing shows which parts of the codes you have to supplement in order to complete the implementation of the function. Those are marked with **TO BE SUPPLEMENTED**.

C++ code 9.13.22: Incomplete implementation of `solve1DWavePML()`

```
template <typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
Eigen::VectorXd solve1DWavePML(
const Eigen::VectorXd &zeta_0, const Eigen::VectorXd &gamma,
const Eigen::VectorXd &sigma, unsigned int M, double T,
RECORDER &&rec = [](Eigen::VectorXd &/*zeta*/) -> void {}) {
  const unsigned int N = gamma.size() - 1;
  Eigen::VectorXd zeta{zeta_0}; // Vector of basis expansion coefficients
  // Obtain initial velocities at grid nodes by linear interpolation
  Eigen::VectorXd v0 =
  (Eigen::VectorXd(N + 1) << 0.0,
  0.5 * (zeta.segment(N + 1, N - 1) + zeta.segment(N + 2, N - 1)), 0.0)
  .finished();
  // Discretization parameters
  const double L = L_default;          // default width of PML layer
  const double h = (2.0 + 2 * L) / N;  // meshwidth
  const double tau = T / M;            // size of timestep
  // Initialize matrices
  Eigen::SparseMatrix<double> A(2 * N + 1, 2 * N + 1);
  Eigen::SparseMatrix<double> R(2 * N + 1, 2 * N + 1);
  // We know that the matrix A has at most 3 non-zero entries per
     row/column
  A.reserve(Eigen::VectorXi::Constant(2 * N + 1, 3));
  R.reserve(Eigen::VectorXi::Constant(2 * N + 1, 3));
  A.insert(0, 0) = TO BE SUPPLEMENTED;
  R.insert(0, 0) = TO BE SUPPLEMENTED;
  for (unsigned int i = 1; i < N; ++i) {
    A.insert(i, i) = TO BE SUPPLEMENTED;
    R.insert(i, i) = TO BE SUPPLEMENTED;
  }
  A.insert(N, N) = TO BE SUPPLEMENTED;
  R.insert(N, N) = TO BE SUPPLEMENTED;
  for (unsigned int i = 0; i < N; ++i) {
    A.insert(i + N + 1, i + N + 1) = TO BE SUPPLEMENTED;
    R.insert(i + N + 1, i + N + 1) = TO BE SUPPLEMENTED;
```

```
    A.insert(i, i + N + 1) = TO BE SUPPLEMENTED;
    A.insert(i + 1, i + N + 1) = TO BE SUPPLEMENTED;
    R.insert(i, i + N + 1) = TO BE SUPPLEMENTED;
    R.insert(i + 1, i + N + 1) = TO BE SUPPLEMENTED;
  }
  for (unsigned int j = 0; j < N; ++j) {
    A.insert(j + N + 1, j) = TO BE SUPPLEMENTED;
    A.insert(j + N + 1, j + 1) = TO BE SUPPLEMENTED;
    R.insert(j + N + 1, j) = TO BE SUPPLEMENTED;
    R.insert(j + N + 1, j + 1) = TO BE SUPPLEMENTED;
  }
  // Precompute LU factorization of A for the sake of efficiency
  Eigen::SparseLU<Eigen::SparseMatrix<double> > solver;
  solver.compute(A);
  if (solver.info() != Eigen::Success) {
    throw std::runtime_error("LU decomposition of A failed");
  }
  // Main timestepping loop
  rec(zeta);
  for (unsigned int k = 0; k < M; ++k) {
    Eigen::VectorXd beta = -R * zeta;
    beta[0] += 0.5 * h * v0[0];
    for (unsigned int i = 1; i < N; ++i) beta[i] += h * v0[i];
    beta[N] += 0.5 * h * v0[N];
    zeta = solver.solve(beta);
    rec(zeta);
  }
  return zeta;
}
```

SOLUTION for (9-13.g) → 9-13-7-0:pml4s.pdf ▲

(9-13.h) ⚄ (10 min.) The code was run with $\gamma \equiv 1$, $M = N = 200$, $T = 4.0$, $v_0 \equiv 0$, $L = 1$, $\sigma_0 = 10$, and

$$u_0(x) := \begin{cases} \cos^2(\pi x) & \text{for } |x| < \frac{1}{2}\,, \\ 0 & \text{for } |x| \geq \frac{1}{2}\,, \end{cases} \quad \sigma(x) := \sigma_0 \cdot \begin{cases} (x-1)^2 & \text{for } x > 1\,, \\ 0 & \text{for } -1 \leq x \leq 1\,, \\ (1+x)^2 & \text{for } x < -1\,. \end{cases}$$

The following plot shows the full discrete solution for $(x,t) \mapsto u(x,t)$ as a function of spaces and time (visualized by means of bilinear interpolation of $u_h(x,t)$-values at nodes of the space-time grid).

Discuss the quality of the solution.

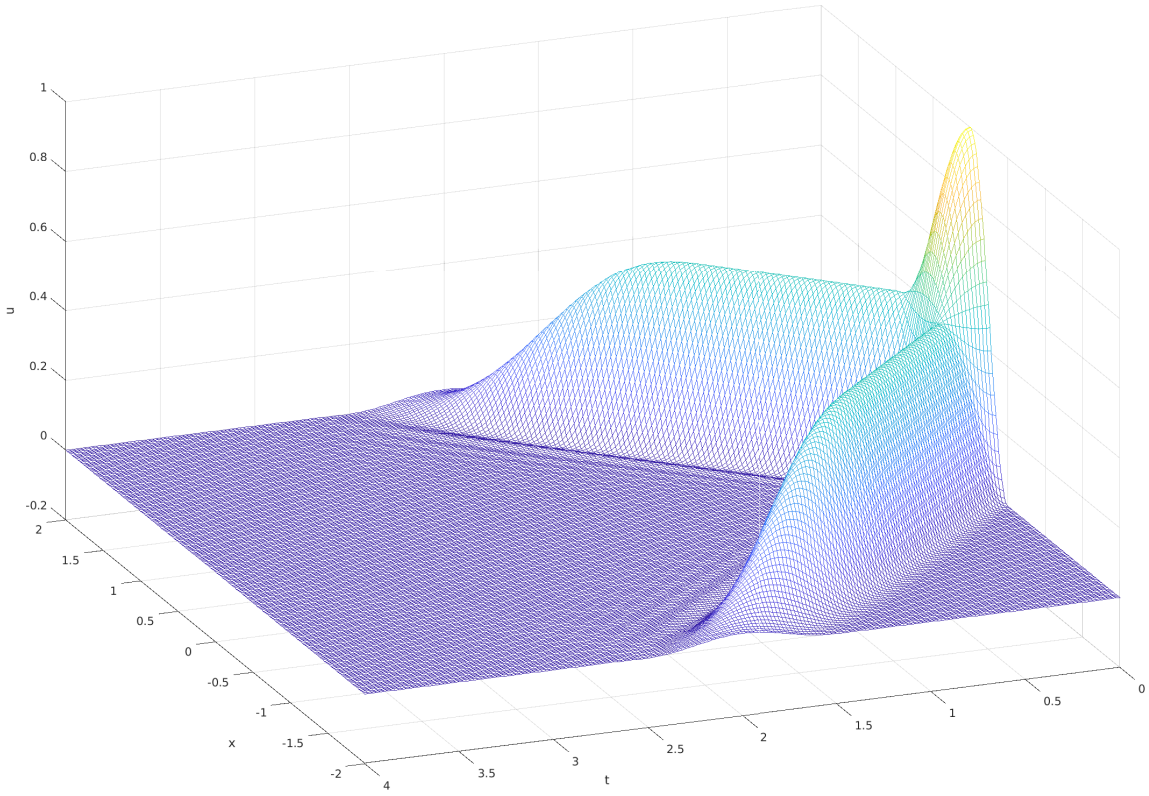


Fig. 145

SOLUTION for (9-13.h) → 9-13-8-0:.pdf ▲

(9-13.i) ⚀ (30 min.) In the file `pml1d.cc` write the missing parts of the C++ function

```
std::vector<double> trackEnergy(
const Eigen::VectorXd &zeta_0, const Eigen::VectorXd &gamma,
const Eigen::VectorXd &sigma, unsigned int M, double T);
```

The parameters have the same meaning as those of `solve1DWavePML()` with the same name. This function is supposed to call `solve1DWavePML()` with a suitable object passed as `rec` argument.

Based on the sequences $\vec{\mu}^{(k)}$, $\vec{v}^{(k)}$ of vectors produces by timestepping, function should return another sequence $(E_0, E_1, \ldots, E_{M-1})$ of values

$$E_k := \tfrac{1}{2}h\left(\delta\vec{\mu}^{(k+\frac{1}{2})}\right)^\top \mathbf{M}^1\delta\vec{\mu}^{(k+\frac{1}{2})} + \tfrac{1}{2}h\left(\delta\vec{v}^{(k+\frac{1}{2})}\right)^\top \delta\vec{v}^{(k+\frac{1}{2})}\,, \tag{9.13.28}$$

$$\delta\vec{\xi}^{(k+\frac{1}{2})} := \frac{\vec{\xi}^{(k+1)} - \vec{\xi}^{(k)}}{\tau}\,, \quad \xi \in \{\mu, v\}\,, \quad \mathbf{M}^1 := \operatorname{diag}(\tfrac{1}{2}, 1, \ldots, 1, \tfrac{1}{2})\,, . \tag{9.13.29}$$

HIDDEN HINT 1 for (9-13.i) → 9-13-9-0:pmlys.pdf

SOLUTION for (9-13.i) → 9-13-9-1:pmlys.pdf ▲

(9-13.j) ⚀ (30 min.) Using your notations from Sub-problem (9-13.e) and for the case $\gamma \equiv 1$ find an expression for the difference of two consecutive "energies" as introduced in .

$E_k - E_{k-1} =$

for $k = 1, \ldots, M-1$.

SOLUTION for (9-13.j) → 9-13-10-0:pmlws.pdf ▲

End Problem 9-13 , 183 min.

Problem 9-14: Two-Step Radau Runge-Kutta Single-Step Method for the Heat Equation

The method of lines for second-order linear evolution problem first employs Galerkin discretization in space and then applies a suitable timestepping scheme to the resulting ordinary differential equation. Since that ODE is stiff, implicit single-step method are the preferred choice. This problem recalls a particular variant from the class of Radau Runge-Kutta single-step methods.

This problem relies on [Lecture → Section 9.2.4] and [Lecture → Section 9.2.7.1].

For a bounded domain $\Omega \subset \mathbb{R}^2$ and final time $T > 0$ consider the parabolic evolution problem

$$\frac{\partial}{\partial t}(\rho(\boldsymbol{x})u) - \Delta u = 0 \quad \text{in} \quad \widetilde{\Omega} := \Omega \times]0,T[\,, \tag{9.14.1a}$$

$$\mathbf{grad}\, u(\boldsymbol{x},t) \cdot \boldsymbol{n}(\boldsymbol{x}) = 0 \quad \text{for} \quad (\boldsymbol{x},t) \in \partial\Omega \times]0,T[\,, \tag{9.14.1b}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \text{for all} \quad \boldsymbol{x} \in \Omega\,, \tag{9.14.1c}$$

with (uniformly positive) coefficient function $\rho : \Omega \to \mathbb{R}^+$ and given initial data $u_0 \in H^1(\Omega)$.

(9-14.a) ⚀ (20 min.) For the spatial Galerkin semidiscretization of (9.14.1) we rely on a finite-dimensional subspace V_h of $H^1(\Omega)$ equipped with an ordered basis $\mathfrak{B} := \{b_h^1, \ldots, b_h^N\}$, $N := \dim V_h$. We obtain the method-of-lines ordinary differential equation (MOL ODE) for the time-dependent basis expansion coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t)$ of an approximate solution of (9.14.1)

$$\mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{\varphi}}(t)\,. \tag{9.14.2}$$

Give concrete formulas for the matrices $\mathbf{M}, \mathbf{A} \in \mathbb{R}^{N,N}$ and the time-dependent vector $\vec{\boldsymbol{\varphi}}(t) \in \mathbb{R}^N$

$$\mathbf{M} = \left[\quad\right]_{i,j=1}^{N}\,,$$

$$\mathbf{A} = \left[\quad\right]_{i,j=1}^{N}\,,$$

$$\vec{\boldsymbol{\varphi}}(t) = \left[\quad\right]_{i=1}^{N}\,.$$

SOLUTION for (9-14.a) → 9-14-1-0:hgcrads1.pdf ▲

(9-14.b) ⚂ (30 min.) The implicit 2-stage Gauss-Radau Runge-Kutta single-step method (RK-SSM) is described by the Butcher scheme

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array}\,. \tag{9.14.4}$$

Fill in the details of the formulas describing a single step $t_k \to t_{k+1}$, $\vec{\boldsymbol{\mu}}^{(k)} \to \vec{\boldsymbol{\mu}}^{(k+1)}$, with step size $\tau := t_{k+1} - t_k > 0$ of that RK-SSM applied to the abstract method-of-lines ODE (9.14.2):

$$\begin{bmatrix} & \\ & \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\kappa}}_1 \\ \vec{\boldsymbol{\kappa}}_2 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix}$$

$$\vec{\mu}^{(k+1)} := \vec{\mu}^{(k)} + \boxed{}\,\vec{\kappa}_1 + \boxed{}\,\vec{\kappa}_2\,,$$

with auxiliary vectors $\vec{\kappa}_1, \vec{\kappa}_2 \in \mathbb{R}^N$.

No inverses of matrices must occur in the formulas!

HIDDEN HINT 1 for (9-14.b) → 9-14-2-0:hcradh11.pdf

SOLUTION for (9-14.b) → 9-14-2-1:hcrads2.pdf ▲

End Problem 9-14 , 50 min.

Problem 9-15: Implicit-Explicit Runge-Kutta Single-Step Methods

This problem studies a special class of timestepping schemes for parabolic evolution problems.

This problem is based on [Lecture → Section 9.2.1] and also requires familiarity with LEHRFEM++.
▷ problem name/problem code folder: IMEXRKSSM

The article [ARS97, Sect. 2] introduces s-stage ($s \in \mathbb{N}$) so-called **implicit-explicit (IMEX)** partitioned Runge-Kutta single-step methods (RK-SSMs). Their definition relies on two elementary RK-SSMs given through their respective Butcher schemes, see [Lecture → § 9.2.6.30]/[NumCSE course → (12.3.3.3)]:

$$\text{(A)} \quad \begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \;\widehat{=}\; \begin{array}{c|ccccc} c_1 & a_{11} & 0 & \dots & \dots & 0 \\ c_2 & a_{21} & a_{22} & & & 0 \\ \vdots & \vdots & & \ddots & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ c_s & a_{s1} & \dots & & & a_{ss} \\ \hline & b_1 & b_2 & \dots & \dots & b_s \end{array} , \quad \begin{array}{l} \mathbf{c}, \mathbf{b} \in \mathbb{R}^s, \\ \mathfrak{A} \in \mathbb{R}^{s,s} , \end{array} \tag{9.15.1}$$

$$\text{(B)} \quad \begin{array}{c|c} \widehat{\mathbf{c}} & \widehat{\mathfrak{A}} \\ \hline & \widehat{\mathbf{b}}^T \end{array} \;\widehat{=}\; \begin{array}{c|ccccc} 0 & 0 & & & & 0 \\ \widehat{c}_2 & \widehat{a}_{21} & 0 & \dots & \dots & 0 \\ \widehat{c}_3 & \widehat{a}_{31} & \widehat{a}_{32} & 0 & & 0 \\ \vdots & \vdots & & \ddots & \ddots & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ \widehat{c}_{s+1} & \widehat{a}_{s+1,1} & \dots & & \dots & \widehat{a}_{s+1,s} \quad 0 \\ \hline & \widehat{b}_1 & \widehat{b}_2 & \dots & & \dots \quad \widehat{b}_{s+1} \end{array} , \quad \begin{array}{l} \widehat{\mathbf{c}}, \widehat{\mathbf{b}} \in \mathbb{R}^{s+1}, \\ \widehat{\mathfrak{A}} \in \mathbb{R}^{s+1,s+1} . \end{array} \tag{9.15.2}$$

(9-15.a) ⊡ (5 min.) Which of the RK-SSMs (A) or (B) is explicit, which implicit?

SOLUTION for (9-15.a) → 9-15-1-0:imexm0s.pdf ▲

The IMEX RK-SSM is applied for the numerical integration of an autonomous ordinary differential equation of the form

$$\dot{\mathbf{u}} = \mathbf{f}(\mathbf{u}) + \mathbf{g}(\mathbf{u}) , \quad \mathbf{f}, \mathbf{g} : \mathbb{R}^N \to \mathbb{R}^N , \quad N \in \mathbb{N} . \tag{9.15.3}$$

This results in a discrete evolution given by

$$\mathbf{u}_1 := \Psi^{t,t+\tau}\mathbf{u}_0 : \begin{cases} \widehat{\mathbf{k}}_1 := \mathbf{f}(\mathbf{u}_0) \\ \textbf{for} \quad i = 1, \dots, s \quad \textbf{do} \\ \left[\begin{array}{l} \mathbf{v}_i := \mathbf{u}_0 + \tau \sum_{j=1}^{i} a_{i,j}\mathbf{k}_j + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j}\widehat{\mathbf{k}}_j , \\ \mathbf{k}_i := \mathbf{g}(\mathbf{v}_i) , \\ \widehat{\mathbf{k}}_{i+1} := \mathbf{f}(\mathbf{v}_i) , \end{array} \right. \\ \mathbf{u}_1 := \mathbf{u}_0 + \tau \sum_{j=1}^{s} b_j\mathbf{k}_j + \tau \sum_{j=1}^{s+1} \widehat{b}_j\widehat{\mathbf{k}}_j . \end{cases} \tag{9.15.4}$$

(9-15.b) ⚂ (15 min.) Algorithm (9.15.4) entails solving potentially non-linear equations to determine the increments $\mathbf{k}_i$, $i = 1, \dots, s$, which depend upon $\mathbf{k}_i = \mathbf{g}(\mathbf{v}_i)$. For the scalar case $N = 1$ formulate the Newton iteration that can be employed to approximately find $\mathbf{v}_i$, $i = 1, \dots, s$. The function $\mathbf{g}$ can be assumed to be continuously differentiable.

SOLUTION for (9-15.b) → 9-15-2-0:imex1.pdf ▲

(9-15.c) ⚄ (20 min.) State the definition of the IMEX RK-SSM from (9.15.4) in the "explicit form"

$$\mathbf{u}_1 := \Psi^{t,t+\tau}\mathbf{u}_0 : \left\{ \begin{array}{l} \widehat{\mathbf{k}}_1 := \mathbf{f}(\mathbf{u}_0) \\ \textbf{for} \quad i = 1,\ldots,s \quad \textbf{do} \\ \left[\begin{array}{l} \mathbf{v}_i := \boxed{} \\ \mathbf{k}_i := \mathbf{g}(\mathbf{v}_i)\,, \\ \widehat{\mathbf{k}}_{i+1} := \mathbf{f}(\mathbf{v}_i)\,, \end{array} \right. \\ \mathbf{u}_1 := \mathbf{u}_0 + \tau \sum_{j=1}^{s} b_j \mathbf{k}_j + \tau \sum_{j=1}^{s+1} \widehat{b}_j \widehat{\mathbf{k}}_j\,, \end{array} \right. \tag{9.15.5}$$

when it is applied to the ODE

$$\dot{\mathbf{u}} = \mathbf{f}(\mathbf{u}) + \mathbf{g}(\mathbf{u}) \quad \text{with} \quad \mathbf{g}(\mathbf{u}) := -\mathbf{A}\mathbf{u}\,, \quad \mathbf{A} \in \mathbb{R}^{N,N} \text{ s.p.d.}\,, \tag{9.15.6}$$

and a generic continuous function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$.

You need only fill out the box, but neither $\mathbf{k}_i$ nor $\mathbf{v}_i$ must occur in your expression.

SOLUTION for (9-15.c) → 9-15-3-0:imex2.pdf ▲

Now we focus on the concrete 2-stage IMEX RK-SSM defined by

$$\text{(A)} \quad \begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \widehat{=} \quad \begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}\,, \quad \gamma = \frac{3+\sqrt{3}}{6}\,, \tag{9.15.8}$$

$$\text{(B)} \quad \begin{array}{c|c} \widehat{\mathbf{c}} & \widehat{\mathfrak{A}} \\ \hline & \widehat{\mathbf{b}}^T \end{array} \quad \widehat{=} \quad \begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \gamma & \gamma & 0 & 0 \\ 1-\gamma & \gamma-1 & 2(1-\gamma) & 0 \\ \hline & 0 & \frac{1}{2} & \frac{1}{2} \end{array}\,. \tag{9.15.9}$$

(9-15.d) ⚄ (30 min.) [depends on Sub-problem (9-15.c)]

Perform an empirical study of the order of the IMEX RK-SSM given by (9.15.8) and (9.15.9) by solving the scalar initial value problem (IVP)

$$\dot{y} = f(y) + g(y)\,, \quad f(y) := -y^2\,, \quad g(y) = y\,, \quad y(0) = \tfrac{1}{4}\,, \tag{9.15.10}$$

with exact solution

$$y(t) = \frac{\exp(t)}{3+\exp(t)}\,,$$

on a sequence of equidistant meshes of $[0,1]$ with meshwidths $\tau := 2^{-3}, 2^{-4}, \ldots, 2^{-10}$ and monitoring the maximal error in the meshpoints:

$$\texttt{err}(\tau) := \max_{j=1,\ldots,M} |y(j\tau) - y^{(j)}|\,, \quad \tau := M^{-1}\,.$$

To that end, in the file `imexrkssm.cc` write a C++ function

```
double IMEXError(int M)
```

that computes $\texttt{err}(M)$ for the IVP given in (9.15.10). Then tabulate the error values for $M = 2^3, 2^4, \ldots, 2^{10}$ and describe the convergence qualitatively and quantitatively.

SOLUTION for (9-15.d) → 9-15-4-0:imex1x.pdf ▲

For a bounded connected polygonal $\Omega \subset \mathbb{R}^2$ we consider the following *non-linear* 2nd-order parabolic evolution problem in weak form: seek $u : [0,T] \to H^1(\Omega)$ such that

$$\begin{aligned} \int_\Omega \frac{\partial u}{\partial t} v + \mathbf{grad}\, u \cdot \mathbf{grad}\, v + \sinh(u)\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u\, v \,\mathrm{d}S = \int_{\partial\Omega} v \,\mathrm{d}S \quad \forall v \in H^1(\Omega)\,, \\ u(0) = u_0 \in H^1(\Omega)\,, \end{aligned} \tag{9.15.12}$$

where u_0 is given.

(9-15.e) ⚀ (20 min.) State the initial-boundary value problem in strong (PDE) form for which (9.15.12) is the associated spatial variational formulation.

SOLUTION for (9-15.e) → 9-15-5-0:imex3.pdf ▲

In the spirit of the **method of lines (MOL)** [Lecture → Section 9.2.4] we perform the spatial semidiscretization of (9.15.12) by means of a Galerkin finite element discretization based on the Lagrange finite element space $\mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh of Ω, and end up with an initial value problem for an ordinary differential equation of the form

$$\begin{aligned} \mathbf{M}\dot{\vec{\mu}} + \mathbf{A}\vec{\mu} + \mathbf{r}(\vec{\mu}) = \vec{\varphi}\,, \\ \vec{\mu}(0) = \vec{\mu}_0\,, \end{aligned} \tag{9.15.13}$$

with suitable matrices $\mathbf{M}, \mathbf{A} \in \mathbb{R}^{N,N}$, a non-linear function $\mathbf{r} : \mathbb{R}^N \to \mathbb{R}^N$, and $\vec{\varphi} \in \mathbb{R}^N$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$. Here $t \to \vec{\mu}(t)$ is the time-dependent vector of basis expansion coefficients of the semidiscrete solution $t \to u_N(t) \in \mathcal{S}_1^0(\mathcal{M})$. The use of standard nodal basis functions $\{b_N^\ell\}_{\ell=1}^N$ is assumed.

(9-15.f) ⚁ (15 min.) Give formulas for the entries of the matrices $\mathbf{M}$ and $\mathbf{A}$ and of the column vector $\vec{\varphi} \in \mathbb{R}^N$ in terms of suitable integrals of expressions involving the finite-element global shape functions b_h^ℓ.

SOLUTION for (9-15.f) → 9-15-6-0:imex4.pdf ▲

In the file `imexrkssm.h` we have at our disposal a LEHRFEM++-based C++ function

```
template <typename FUNC_ALPHA,
          typename FUNC_GAMMA,
          typename FUNC_BETA>
  Eigen::SparseMatrix<double> compGalerkinMatrix(
      const lf::assemble::DofHandler &dofh,
      FUNC_ALPHA &&alpha, FUNC_GAMMA &&gamma, FUNC_BETA &&beta);
```

that, given a mesh $\mathcal{M}$, (approximately, using "sufficiently high-order" local numerical quadrature) computes the $\mathcal{S}_1^0(\mathcal{M})$-based finite element Galerkin matrix for the left-hand side of the variational boundary value problem

$$\begin{aligned} u \in H^1(\Omega)\colon \int_\Omega \alpha(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \gamma(\boldsymbol{x})\, u(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \beta(\boldsymbol{x})\, u(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \\ = \int_\Omega f(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \end{aligned} \tag{9.15.14}$$

where $\alpha, \gamma : \Omega \to \mathbb{R}$, $\beta : \partial\Omega \to \mathbb{R}$ are bounded coefficient functions, and $f \in L^2(\Omega)$. The use of the standard nodal basis consisting of "tent functions" is assumed.

The argument `dofh` passes the local-to-global index mapping for $\mathcal{S}_1^0(\mathcal{M})$ and the mesh $\mathcal{M}$, while `alpha`, `gamma`, and `beta` are functors matching **std**::`function<`**double**`(Vector2D)` realizing the coefficient functions $\boldsymbol{x} \mapsto \alpha(\boldsymbol{x})$, $\boldsymbol{x} \mapsto \gamma(\boldsymbol{x})$, and $\boldsymbol{x} \mapsto \beta(\boldsymbol{x})$. For further information about `compGalerkinMatrix()` see also Problem 3-1.

(9-15.g) ⚄ (30 min.) In the file `imexrkssm.cc` complete the implementation of the LEHRFEM++-based C++ function

```
Eigen::VectorXd compNonlinearTerm(
  std::shared_ptr<const lf::uscalefe::FeSpaceLagrangeO1<double>>
     fe_test,
  const Eigen::VectorXd& u)
```

that globally assembles $\mathbf{r}(\vec{\mu})$, where r is the non-linear mapping (9.15.13). The vector of coefficients $\vec{\mu}$ is passed as argument `u`.

To that end you have to complete the `eval` method of the C++ `struct` **AssemblerLocalFunc**,

```
template<typename BUILDER_DATA, typename ELEMENT>
inline static result_t eval( const BUILDER_DATA& data, const
   ELEMENT& el );
```

which is called by `compNonlinearTerm` and locally assembles $\mathbf{r}(\vec{\mu})$. For the evaluation of the integral involved in the definition of $\mathbf{r}$, rely on the *composite trapezoidal rule* over the mesh $\mathcal{M}$, see [Lecture → (2.4.6.10)]. The member vector `u_loc` of the `data` argument stores the coefficients of the local FE basis functions.

SOLUTION for (9-15.g) → `9-15-7-0:imex6.pdf` ▲

(9-15.h) ⚄ (60 min.) [depends on Sub-problem (9-15.g)]

The method-of-lines ODE (9.15.13) can be rewritten in the form

$$\dot{\vec{\mu}} = \mathbf{f}(\vec{\mu}) + \mathbf{g}(\vec{\mu}) \,,$$

where $\mathbf{g} : \mathbb{R}^N \to \mathbb{R}^N$ is affine linear and collects all linear terms in (9.15.13), whereas $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$ represents the non-linear terms. This splitting forms the foundation for applying an IMEX RK-SSM.

Based on the functions `compGalerkinMatrix()` and `compNonlinearTerm()` from Sub-problem (9-15.g), in the file `imexrkssm.h` complete the implementation of a C++ class

```
class IMEXTimestep {
  public:
  IMEXTimestep(
    std::shared_ptr<const lf::uscalefe::FeSpaceLagrangeO1<double>>
       fe_test);
  void compTimestep(
    std::shared_ptr<const lf::uscalefe::FeSpaceLagrangeO1<double>>
       fe_test,
    double tau, Eigen::VectorXd& y) const;
  private:
    Eigen::SparseMatrix<double> M_;
    Eigen::SparseMatrix<double> A_;
    Eigen::VectorXd phi_;
    // Feel free to add more data members
};
```

The data `M_`, `A_`, and `phi_` store the time-independent Galerkin matrices and vectors denoted by $\mathbf{M}$, $\mathbf{A}$, and $\vec{\varphi}$ in (9.15.13). They should be initialized by the constructor `IMEXTimestep`. Argument `fe_test` passes an object describing the piecewise linear Lagrangian finite element space.

The method `compTimestep` realizes a single step of the IMEX RK-SSM (9.15.8)+(9.15.9) for (9.15.13). Argument `y` passes the vector of coefficients $\vec{\mu}_0$ of the previous timestep and the method should update `y` with $\vec{\mu}_1$ after one step of the IMEX RK-SSM.

HIDDEN HINT 1 for (9-15.h) → `9-15-8-0:imex7h1.pdf`

SOLUTION for (9-15.h) → `9-15-8-1:.pdf` ▲

References

[ARS97] Uri M. Ascher, Steven J. Ruuth, and Raymond J. Spiteri. "Implicit-explicit Runge-Kutta methods for time-dependent partial differential equations". In: *Appl. Numer. Math.* 25.2-3 (1997), pp. 151–167. DOI: 10.1016/S0168-9274(97)00056-1 (cit. on p. 339).

End Problem 9-15 , 195 min.

Problem 9-16: Finite-Element Discretization of the Guyer-Krumhansl Heat Equation

In one spatial dimension we study a heat conduction model with a non-local empirical flux law that was introduced in [GK66]. We perform a simple low-order finite-element discretization in space plus an L-stable Runge-Kutta timestepping.

C++ coding for this problem just relies on the EIGEN library.
▷ problem name/problem code folder: GuyerKrumhansl

We fix a final time $T > 0$. The one-dimensional heat evolution equation with the Guyer-Krumhansl flux law in non-dimensional variational form reads [Ram+23]:

Seek $u :]0,T[\to L^2(]0,1[)$, $j :]0,T[\to H^1_0(]0,1[)$, such that

$$\begin{aligned} \mathsf{m}(\frac{\partial u}{\partial t}, v) \quad + \qquad\qquad \mathsf{b}(j,v) \quad &= 0 \quad \forall v \in L^2(]0,1[)\,, \\ \mathsf{n}(\frac{\partial j}{\partial t}, q) \quad - \quad \mathsf{c}(u,q) \quad + \quad \mathsf{d}(j,q) \quad &= 0 \quad \forall q \in H^1_0(]0,1[)\,, \end{aligned} \tag{9.16.1}$$

with

$$\begin{aligned} \mathsf{m}(w,v) &:= \int_0^1 \rho w(x)v(x)\,\mathrm{d}x\,, && w,v \in L^2(]0,1[)\,, \\ \mathsf{n}(w,q) &:= \int_0^1 \sigma w(x)q(x)\,\mathrm{d}x\,, && w,q \in H^1(]0,1[)\,, \\ \mathsf{b}(q,v) &:= \int_0^1 \frac{dq}{dx}(x)v(x)\,\mathrm{d}x\,, && q \in H^1(]0,1[)\,,\quad v \in L^2(]0,1[)\,, \\ \mathsf{d}(w,q) &:= \int_0^1 w(x)q(x) + \mu\frac{dw}{dx}(x)\frac{dq}{dx}(x)\,\mathrm{d}x\,, && w,q \in H^1(]0,1[)\,, \\ \mathsf{c}(v,q) &:= \int_0^1 \kappa v(x)\frac{dq}{dx}(x)\,\mathrm{d}x\,, && v \in L^2(]0,1[)\,,\quad q \in H^1(]0,1[)\,. \end{aligned}$$

The coefficients are assumed to be constant, that is, ρ, σ, μ, and κ can be regarded as positive numbers.

(9-16.a) ⚀ (12 min.) Complete the strong (PDE) form of the two coupled evolution equations whose weak (variational) form is given in (9.16.1).

$$\rho\frac{\partial u}{\partial t} + \boxed{} = 0 \quad \text{in} \quad]0,1[\times]0,T[\,,$$

$$\sigma\frac{\partial j}{\partial t} + \boxed{} = 0 \quad \text{in} \quad]0,1[\times]0,T[\,.$$

HIDDEN HINT 1 for (9-16.a) → 9-16-1-0:gukha.pdf

SOLUTION for (9-16.a) → 9-16-1-1:guksa.pdf ▲

(9-16.b) ⚅ (16 min.) A crucial time-dependent quantity in the model is the energy

$$\boxed{E(t) := \frac{1}{2}\int_0^1 \rho\, u(x,t)^2 + \frac{\sigma}{\kappa} j(x,t)^2\,\mathrm{d}x\,, \quad t \in [0,T]}\,, \tag{9.16.6}$$

where $(x,t) \mapsto u(x,t)$ and $(x,t) \mapsto j(x,t)$ solve (9.16.1).

Find an expression in terms of u and j for the change of the energy over time, $t \in [0,T]$:

$$\frac{dE}{dt}(t) = \int_0^1 \boxed{} \,\mathrm{d}x\,.$$

The integrand must not contain temporal derivatives of u or j!

SOLUTION for (9-16.b) → 9-16-2-0:gukbs.pdf ▲

We pursue a spatial finite-element Galerkin discretization of (9.16.1) relying on an equidistant mesh $\mathcal{M} := \{]x_{j-1}, x_j[,\, j = 1,\dots M,\, t_j := j/M\}$ of $]0,1[$, $M \in \mathbb{N}$. In particular

- we replace $L^2(]0,1[)$ with the finite-element subspace $\mathcal{S}_0^{-1}(\mathcal{M})$ of *piecewise constant* functions, and
- $H_0^1(]0,1[)$ with the space $\mathcal{S}_{1,0}^0(\mathcal{M})$ of continuous *piecewise linear functions* that vanish in $x = 0$ and $x = 1$.

In the sequel the following **ordered bases** will be used throughout:

- For $\mathcal{S}_0^{-1}(\mathcal{M})$: the set of characteristic functions of the mesh cells, numbered lexicographically, that is, from left to right.
- For $\mathcal{S}_{1,0}^0(\mathcal{M})$: the set of tent basis functions [Lecture → (2.3.2.1)] associated with *interior* nodes of $\mathcal{M}$, also numbered lexikographically from left to right.

(9-16.c) ⚀ (30 min.) Compute the **element matrices** for a ("generic") mesh cell $K :=]0,h[$, $h := M^{-1}$, and for *all* the bilinear forms m, n, b, c, and d, introduced above.

bilinear form	element matrix
m on $\mathcal{S}_0^{-1}(\mathcal{M}) \times \mathcal{S}_0^{-1}(\mathcal{M})$	$\mathbf{M}_K = \begin{bmatrix} \boxed{} \end{bmatrix}$,
n on $\mathcal{S}_1^0(\mathcal{M}) \times \mathcal{S}_1^0(\mathcal{M})$	$\mathbf{N}_K = \begin{bmatrix} \boxed{} \end{bmatrix}$,
b on $\mathcal{S}_1^0(\mathcal{M}) \times \mathcal{S}_0^{-1}(\mathcal{M})$	$\mathbf{B}_K = \begin{bmatrix} \boxed{} \end{bmatrix}$,
d on $\mathcal{S}_1^0(\mathcal{M}) \times \mathcal{S}_1^0(\mathcal{M})$	$\mathbf{D}_K = \begin{bmatrix} \boxed{} \end{bmatrix}$,
c on $\mathcal{S}_0^{-1}(\mathcal{M}) \times \mathcal{S}_1^0(\mathcal{M})$	$\mathbf{C}_K = \begin{bmatrix} \boxed{} \end{bmatrix}$.

SOLUTION for (9-16.c) → 9-16-3-0:guksc.pdf ▲

(9-16.d) ⚂ (30 min.) [depends on Sub-problem (9-16.c)]

Write down the $\mathcal{S}^0_{1,0}(\mathcal{M}) \times \mathcal{S}_0^{-1}(\mathcal{M})$ Galerkin matrix $\mathbf{B}$ for the bilinear form $\mathfrak{b}$, and the $\mathcal{S}^0_{1,0}(\mathcal{M}) \times \mathcal{S}^0_{1,0}(\mathcal{M})$ Galerkin matrix $\mathbf{N}$ for the bilinear form $\mathfrak{n}$:

$$\mathbf{B} = \begin{bmatrix} \\ \\ \\ \end{bmatrix} \in \mathbb{R}^{\square,\square} ,$$

$$\mathbf{N} = \begin{bmatrix} \\ \\ \\ \end{bmatrix} \in \mathbb{R}^{\square,\square} .$$

SOLUTION for (9-16.d) → 9-16-4-0:duksd.pdf ▲

Our spatial finite-element Galerkin semi-discretization of (9.16.1) yields an ordinary differential equation, known as **method-of-lines ODE**, of the form

$$\widehat{\mathbf{M}} \frac{d}{dt} \begin{bmatrix} \vec{\mu} \\ \vec{\zeta} \end{bmatrix} = \widehat{\mathbf{A}} \begin{bmatrix} \vec{\mu} \\ \vec{\zeta} \end{bmatrix} \quad , \quad \widehat{\mathbf{M}}, \widehat{\mathbf{A}} \in \mathbb{R}^{N,N} , \quad N := 2M - 1 , \tag{9.16.16}$$

where $\vec{\mu} = \vec{\mu}(t)$ and $\vec{\zeta} = \vec{\zeta}(t)$ are the time-dependent basis expansion coefficient vectors of the solutions $t \mapsto u_h(t) \in \mathcal{S}_0^{-1}(\mathcal{M})$ and $t \mapsto j_h(t) \in \mathcal{S}^0_{1,0}(\mathcal{M})$ of the spatially semi-discrete variational problem.

(9-16.e) ⚂ (24 min.) [depends on Sub-problem (9-16.d)]

In the file `guyerkrumhansl.cc` finish the implementation of the C++ function

```
std::pair<std::vector<Eigen::Triplet<double>>,
          std::vector<Eigen::Triplet<double>>>
generateMOLMatrices(unsigned int M,
  double rho, double sigma, double mu, double kappa);
```

That returns the pair of matrices $(\widehat{\mathbf{M}}, \widehat{\mathbf{A}})$ in triplet (COO) format. The arguments pass the number M of cells of the equidistant mesh and the paramters $\rho, \sigma, \mu, \kappa > 0$. Make sure that your code is efficient.

HIDDEN HINT 1 for (9-16.e) → 9-16-5-0:gukhe.pdf

SOLUTION for (9-16.e) → 9-16-5-1:gukse.pdf ▲

(9-16.f) ⚂ (24 min.) [depends on Sub-problem (9-16.e)]

In the file `guyerkrumhansl.h` write a function

```
template <typename RECORDER
          = std::function<void(const Eigen::VectorXd &)>>
std::pair<Eigen::VectorXd, Eigen::VectorXd> timestepping_GKHeat(
  double T, unsigned int L, unsigned int M,
  const Eigen::VectorXd &mu0, const Eigen::VectorXd &zeta0,
  double rho, double sigma, double mu, double kappa,
  RECORDER &&rec = [](const Eigen::VectorXd & /*nu*/) {});
```

which uses the method of lines to solve the Guyer-Krumhansl heat evolution problem based on the spatial finite element discretization outlined above and **implicit Euler** timestepping with $L \in \mathbb{N}$ equidistant timesteps.

The argument `T` passes the final time $T > 0$, M is the number of cells in the finite-element mesh, the vectors `mu0` and `zeta0` provide the *initial values* $\vec{\mu}(0)$ and $\vec{\zeta}(0)$. The other arguments `rho, sigma, mu, kappa` are meant for the model parameters. Finally, `rec` may contain a recorder object, which is given the column vector $\vec{v}^{(k)} := \left[\vec{\mu}^{(k)}, \vec{\zeta}^{(k)}\right]$, $k = 0, \ldots, L$, in every timestep.

HIDDEN HINT 1 for (9-16.f) → 9-16-6-0:gukhf.pdf

SOLUTION for (9-16.f) → 9-16-6-1:gukfs.pdf ▲

(9-16.g) ⚂ (16 min.) [depends on Sub-problem (9-16.f)]

In the file `guyerkrumhansl.cc` and within the function

```
std::vector<double>
track_GKHeatEnergy(double T, unsigned int L, unsigned int M,
                   const Eigen::VectorXd &mu0,
                   const Eigen::VectorXd &zeta0, double rho,
                   double sigma, double mu, double kappa);
```

implement a **lambda function**

```
auto rec = [/* Your code here*/](const Eigen::VectorXd &nu) ->
   void {/* Your code here*/};
```

such that the function returns the sequence $E^{(k)}$, $k = 0, \ldots, L$ of energies. Here $E^{(k)}$ is the energy (9.16.6) evaluated for $u_h^{(k)} \in \mathcal{S}_0^{-1}(\mathcal{M})$ and $j_h^{(k)} \in \mathcal{S}_{1,0}^0(\mathcal{M})$:

$$E^{(k)} := \frac{1}{2}\int_0^1 \rho u_h^{(k)}(x)^2 + \frac{\sigma}{\kappa} j_h^{(k)}(x)^2 \,\mathrm{d}x\,,$$

where $u_h^{(k)}$ and $j_h^{(k)}$ are the approximations of $u(\cdot, k\tau)$ and $j(\cdot, k\tau)$ computed by the fully discrete evolution implemented in `timestepping_GKHeat()`. The arguments for `track_GKHeatEnergy()` are the same as for `timestepping_GKHeat()`.

HIDDEN HINT 1 for (9-16.g) → 9-16-7-0:guklh.pdf

SOLUTION for (9-16.g) → 9-16-7-1:guksg.pdf ▲

References

[GK66] R. A. Guyer and J. A. Krumhansl. "Thermal Conductivity, Second Sound, and Phonon Hydrodynamic Phenomena in Nonmetallic Crystals". In: *Phys. Rev.* 148 (2 Aug. 1966), pp. 778–788. DOI: 10.1103/PhysRev.148.778 (cit. on p. 344).

[Ram+23] A. J. A. Ramos, R. Kovács, M. M. Freitas, and D. S. Almeida Júnior. "Mathematical analysis and numerical simulation of the Guyer-Krumhansl heat equation". In: *Appl. Math. Model.* 115 (2023), pp. 191–202. DOI: 10.1016/j.apm.2022.10.054 (cit. on p. 344).

End Problem 9-16 , 152 min.

Problem 9-17: Scalar Parabolic Evolution Problems: Continuous and Discrete

Parabolic evolution problems as introduced in [Lecture → Section 9.2] provide mathematical models for dissipative transient phenomena. The related initial-boundary value problems (IBVPs) can be discretized using the method-of-lines approach combined with implicit single-step methods. This problem examines a few aspects of a particular parabolic IBVP and its discretization.

This problem requires the material of [Lecture → Section 9.2.4] and [Lecture → Section 9.2.7.1].
▷ problem name/problem code folder: ParabolicEvolutionAspects

On a 2D polygonal domain $\Omega \subset \mathbb{R}^2$ we consider the following parabolic evolution problem in variational form: seek $u :]0,T[\to H^1(\Omega)$, $u = u(x,t)$, such that

$$\int_{\partial\Omega} \frac{\partial u}{\partial t}(x,t)v(x)\,\mathrm{d}S(x) + \int_{\Omega} \mathbf{grad}\, u(x,t) \cdot \mathbf{grad}\, v(x)\,\mathrm{d}x = 0 \quad \forall v \in H^1(\Omega)\,, \tag{9.17.1}$$

$$u(x,0) = u_0(x)\,, \quad x \in \Omega\,, \tag{9.17.2}$$

where the initial data $u_0 \in H^1(\Omega)$ are given.

We perform a spatial Galerkin finite element semidiscretization of (9.17.1) based on the finite element space $\mathcal{S}_2^0(\mathcal{M})$ of continuous $\mathcal{M}$-piecewise quadratic polynomial functions, where $\mathcal{M}$ is a triangular mesh of Ω. This leads to the **method-of-lines (MOL) ODE** for the basis expansions coefficient vector $t \mapsto \vec{\mu}(t) \in \mathbb{R}^N$, $N := \dim \mathcal{S}_2^0(\mathcal{M})$, of the semi-discrete finite-element solution $t \mapsto u_h(t) \in \mathcal{S}_2^0(\mathcal{M})$:

$$\text{MOL-ODE:} \qquad \mathbf{M}\frac{d\vec{\mu}}{dt}(t) + \mathbf{A}\vec{\mu}(t) = \mathbf{0}\,, \tag{9.17.3}$$

with matrices $\mathbf{A}, \mathbf{M} \in \mathbb{R}^{N,N}$.

(9-17.a) ⚂ (15 min.) Denote by $\{b_h^i\}_{i=1}^N$ the nodal basis of $\mathcal{S}_2^0(\mathcal{M})$ used in the context of finite-element Galerkin discretization. Give concrete expressions for the entries of the matrices $\mathbf{A}$ and $\mathbf{M}$ in (9.17.3).

$$\mathbf{A} = \left[\quad\right]_{i,j=1}^{N}\,, \qquad \mathbf{M} = \left[\quad\right]_{i,j=1}^{N}\,.$$

SOLUTION for (9-17.a) → 9-17-1-0:parass1.pdf ▲

(9-17.b) ⚂ (15 min.) Please answer the following questions for "generic" initial data u_0, that is, tick "YES", if "YES" is the answer for every u_0.

(A)	Is the matrix $\mathbf{M}$ is symmetric, positive definite?	YES ◯	NO ◯
(B)	Is the matrix $\mathbf{A}$ is symmetric, positive definite?	YES ◯	NO ◯
(C)	Does $\vec{\mu}(t)$ converge to $\mathbf{0}$ for $t \to \infty$?	YES ◯	NO ◯
(D)	Does $\frac{d\vec{\mu}}{dt}(t)$ converge to $\mathbf{0}$ for $t \to \infty$?	YES ◯	NO ◯
(E)	Does $(\vec{\mu}(t))^\top \mathbf{A}\vec{\mu}(t)$ converge to $\mathbf{0}$ for $t \to \infty$?	YES ◯	NO ◯

SOLUTION for (9-17.b) → 9-17-2-0:parass2.pdf ▲

For the temporal discretization of (9.17.1) we employ a 2-stage singly diagonally implicit Runge-Kutta single step method (SDIRK) whose Butcher scheme is

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array} \quad , \quad \gamma = \tfrac{1}{2}(2-\sqrt{2}) \, . \tag{9.17.20}$$

This is an *L-stable* Runge-Kutta single-step method (RK-SSM) of *order 2*.

(9-17.c) ⚄ (30 min.) The application of the SDIRK RK-SSM (9.17.20) to the method-of-lines ODE (9.17.3) with constant timestep $\tau > 0$ yields a sequence $\left(\vec{\mu}^{(k)}\right)_k$ of vectors, where $\vec{\mu}^{(k+1)}$ is obtained from $\vec{\mu}^{(k)}$ according to

$$\begin{bmatrix} & \\ & \end{bmatrix} \begin{bmatrix} \vec{\kappa}_1 \\ \vec{\kappa}_2 \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix} \, , \tag{9.17.21}$$

$$\vec{\mu}^{(k+1)} = \vec{\mu}^{(k)} + \tfrac{1}{2}\tau(\vec{\kappa}_1 + \vec{\kappa}_2) \, .$$

Fill in the missing parts of the system matrix and of the right-hand side vector in (9.17.21) using the notations from (9.17.3).

HIDDEN HINT 1 for (9-17.c) → 9-17-3-0:rkssmpara.pdf

SOLUTION for (9-17.c) → 9-17-3-1:pareass3.pdf ▲

(9-17.d) ⚃ (15 min.) Write $u_h^{(j)} \in \mathcal{S}_2^0(\mathcal{M})$ for the fully-discete approximation of $u(\cdot, \tau j)$, that is, the finite-element function with basis expansion coefficient vector $\vec{\mu}^{(j)}$, $j \in \mathbb{N}_0$. Theoretical considerations provide the asymptotic a-priori error estimate

$$\underbrace{\left(\tau \sum_{j=1}^{M} \left\| u(\tau j) - u_h^{(j)} \right\|_{L^2(\Omega)}^2\right)^{\frac{1}{2}}}_{\text{error norm}} \leq C\left(h_{\mathcal{M}}^3 + \tau^2\right) \quad , \quad M := {}^T\!/\tau \in \mathbb{N} \, , \tag{9.17.23}$$

with some constant $C > 0$ and $h_{\mathcal{M}}$ the meshwidth of $\mathcal{M}$.

Assuming that this estimate is sharp,

- how many steps of *uniform regular refinement* of $\mathcal{M}$ have to be carried out and
- by what factor does the timestep τ have to be reduced in order to achieve a reduction of the norm (as given in (9.17.23)) of the full discretization error *by a factor* 100?

$$\sharp(\text{required steps of regular refinement of } \mathcal{M}) = \boxed{} \, ,$$

$$\text{factor by which timestep } \tau \text{ has to be reduced} = \boxed{} \, .$$

SOLUTION for (9-17.d) → 9-17-4-0:.pdf ▲

End Problem 9-17, 75 min.

Problem 9-18: Leapfrog Timestepping for Dissipative Wave Equation

The leapfrog timestepping method, presented in [Lecture → § 9.3.4.16], is a very popular timestepping method for linear wave propagation problems, because it is almost energy-conserving and explicit, provided that a mild CFL-condition in satisfied, see [Lecture → Exp. 9.3.4.20] and [Lecture → § 9.3.5.2].

To tackle this problem successfully you must know the contents of [Lecture → Section 9.3]. The coding parts rely on LEHRFEM++, which means that you also must be familiar with [Lecture → Section 2.7] and [Lecture → Section 2.8].
▷ problem name/problem code folder: LeapfrogDissipativeWave

On a bounded polygonal domain $\Omega \subset \mathbb{R}^2$ and for fixed final time $T > 0$ we consider the following linear evolution initial-boundary value problem (IBVP) in variational form: seek $u : [0,T] \to H^1(\Omega)$ such that for all $v \in H^1(\Omega)$

$$\int_\Omega \frac{\partial^2 u}{\partial t^2}(x,t)v(x)\,\mathrm{d}x + \int_{\partial\Omega} \frac{\partial u}{\partial t}(x,t)v(x)\,\mathrm{d}S(x) + \int_\Omega \mathbf{grad}\, u(x,t)\cdot\mathbf{grad}\, v(x)\,\mathrm{d}x = 0\,, \tag{9.18.1a}$$

$$u(\cdot,0) = u_0 \quad , \quad \frac{\partial u}{\partial t}(\cdot,0) = v_0 \quad \text{in} \quad \Omega\,, \tag{9.18.1b}$$

where $u_0, v_0 \in H^1(\Omega)$ are given initial data.

(9-18.a) ⚂ (20 min.) What is the strong (PDE) form of the evolution IBVP (9.18.1)?

$$\boxed{} = 0 \quad \text{on} \quad \Omega\times]0,T[\,,$$

$$\boxed{} = 0 \quad \text{on} \quad \partial\Omega\times]0,T[\,,$$

$$\boxed{} \quad \text{in} \quad \Omega\,.$$

SOLUTION for (9-18.a) → 9-18-1-0:ldws1.pdf ▲

(9-18.b) ⚂ (15 min.) The **total energy** associated with (9.18.1) is the expression

$$\mathcal{E}(t) := \frac{1}{2}\int_\Omega \left|\frac{\partial u}{\partial t}(x,t)\right|^2 \mathrm{d}x + \frac{1}{2}\int_\Omega \|\mathbf{grad}\, u(x,t)\|^2\,\mathrm{d}x\,, \quad t \in [0,T]\,, \tag{9.18.8}$$

where $u : [0,T] \to H^1(\Omega)$ is the solution of the evolution problem (9.18.1).

Show that $\mathcal{E}(t) \le \mathcal{E}(0)$ for all $t \in]0,T[$.

SOLUTION for (9-18.b) → 9-18-2-0:ldws2.pdf ▲

We follow the **method-of-lines** approach: For the spatial Galerkin semi-discretization of (9.18.1) we rely on the quadratic Lagrangian finite element space $\mathcal{S}_2^0(\mathcal{M})$ according to [Lecture → Rem. 2.6.2.12], $\mathcal{M}$ a triangular mesh of Ω. We use the standard basis of $\mathcal{S}_2^0(\mathcal{M})$ and denote it by $\mathfrak{B}_h := \{b_h^1,\ldots,b_h^N\}$, $N := \dim \mathcal{S}_2^0(\mathcal{M})$.

(9-18.c) ⚂ (15 min.)

The finite-element Galerkin semi-discretization converts (9.18.1a) into the following **method-of-lines ODE** for the time-dependent basis expansion coefficient vector $\vec{\mu} = \vec{\mu}(t)$ of the semi-discrete Galerkin

solution $t \mapsto u_h(t) \in \mathcal{S}_2^0(\mathcal{M})$:

$$\mathbf{M}\frac{d^2\vec{\mu}}{dt^2}(t) + \mathbf{B}\frac{d\vec{\mu}}{dt}(t) + \mathbf{A}\vec{\mu}(t) = 0\,, \tag{9.18.9}$$

with suitable matrices $\mathbf{M}, \mathbf{B}, \mathbf{A} \in \mathbb{R}^{N,N}$. Give expressions for the entries of these matrices.

$$(\mathbf{M})_{i,j} = \qquad\qquad , \quad i,j \in \{1,\ldots,N\}\,,$$

$$(\mathbf{B})_{i,j} = \qquad\qquad , \quad i,j \in \{1,\ldots,N\}\,,$$

$$(\mathbf{A})_{i,j} = \qquad\qquad , \quad i,j \in \{1,\ldots,N\}\,.$$

SOLUTION for (9-18.c) → 9-18-3-0:ldws3.pdf ▲

(9-18.d) ⚄ (40 min.) In the file `leapfrogdissipativewave.cc` complete the code of the LEHRFEM++-based function

```
std::tuple<lf::assemble::COOMatrix<double>,
           lf::assemble::COOMatrix<double>,
           lf::assemble::COOMatrix<double>>
  computeGalerkinMatrices(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>>
      fe_space_p);
```

It is supposed to compute the matrices $\mathbf{M}$, $\mathbf{B}$, and $\mathbf{A}$ as introduced in (9.18.9) relying on the finite element space pointed at by `fe_space_p`. The matrices are returned as triple $(\mathbf{M}, \mathbf{B}, \mathbf{A})$.

HIDDEN HINT 1 for (9-18.d) → 9-18-4-0:ldwh3a.pdf

SOLUTION for (9-18.d) → 9-18-4-1:ldws3a.pdf ▲

For the discretization of (9.18.9) in time we employ the following leapfrog-type scheme with $M \in \mathbb{N}$ uniform timesteps of size $\tau := \frac{T}{M} > 0$:

$$\boxed{\begin{aligned} \frac{\vec{\mu}^{(j+1)} - \vec{\mu}^{(j)}}{\tau} &= \vec{v}^{(j+\frac{1}{2})}\,, \\ \mathbf{M}\,\frac{\vec{v}^{(j+\frac{1}{2})} - \vec{v}^{(j-\frac{1}{2})}}{\tau} &= -\mathbf{A}\vec{\mu}^{(j)} - \tfrac{1}{2}\mathbf{B}\left(\vec{v}^{(j+\frac{1}{2})} + \vec{v}^{(j-\frac{1}{2})}\right)\,, \end{aligned} \quad j = 0,1,\ldots M}\,, \tag{9.18.12}$$

supplemented with the special initial step

$$\vec{\mu}^{(0)} := \vec{\mu}_0 \quad , \quad \tfrac{1}{2}(\vec{v}^{(\frac{1}{2})} + \vec{v}^{(-\frac{1}{2})}) = \vec{v}_0\,. \tag{9.18.13}$$

Here, $\vec{\mu}_0 \in \mathbb{R}^N$ and $\vec{v}_0 \in \mathbb{R}^N$ are basis expansion coefficient vectors of the nodal interpolants into $\mathcal{S}_2^0(\mathcal{M})$ of u_0 and v_0, respectively.

(9-18.e) ⚄ (20 min.) To study the timestepping scheme we temporarily consider the "scalar" case $N = 1$, $\mathbf{M} = \mathbf{A} = \mathbf{B} = [1] \in \mathbb{R}^{1,1}$, that is, (9.18.12), (9.18.13) applied to solve the following initial-value problem:

$$\ddot{y} + \dot{y} + y = 0 \quad , \quad y(0) = u_0 \in \mathbb{R}\,, \quad \dot{y}(0) = v_0 \in \mathbb{R}\,. \tag{9.18.14}$$

We choose $u_0 = 1$, $v_0 = -\frac{1}{2}$, which yields the exact solution $y(t) = e^{-\frac{1}{2}t}\cos(\frac{1}{2}\sqrt{3}t)$.

We approximate $y(1)$ by y_M obtain by taking M equidistant steps of (9.18.12) in $[0,1]$.

The table beside shows the measured errors for different values of M

Formulate a conjecture about the method (9.18.12), (9.18.13) based on the tabulated error values. Explain your answer.

M	$\lvert y(1) - y_M\rvert$
10	0.0011660874580030
20	0.0002911403222123
30	0.0001293643595861
40	0.0000727612849977
50	0.0000465653957716
60	0.0000323363913955
80	0.0000181888348186
100	0.0000116407401418
160	0.0000045471158128
200	0.0000029101469922
320	0.0000011367731537
500	0.0000004656218255
640	0.0000002841929364

SOLUTION for (9-18.e) → 9-18-5-0:ldws4.pdf ▲

(9-18.f) ⚂ (30 min.) Rewrite (9.18.12), (9.18.13) in a more "implementation-friendly" way:

$$\vec{v}^{(\frac{1}{2})} = \boxed{}\,, \qquad \text{(9.18.15a)}$$

$$\vec{\mu}^{(j)} = \boxed{}\,, \quad j = 1,\ldots,M\,, \qquad \text{(9.18.15b)}$$

$$\vec{v}^{(j+\frac{1}{2})} = \boxed{}\,, \quad j = 1,\ldots,M\,. \qquad \text{(9.18.15c)}$$

The term on the right of (9.18.15a) may only involve $\vec{\mu}_0$ and $\vec{v}_0$. In (9.18.15b) and (9.18.15c) the expressions on the right may only contain vectors with a strictly lower upper index than that of the vector on the left.

SOLUTION for (9-18.f) → 9-18-6-0:ldw5s.pdf ▲

(9-18.g) ⚄ (40 min.) [depends on Sub-problem (9-18.f)]

In the file `leapfrogdissipativewave.cc` finish the implementation of the LEHRFEM++-based function

```
Eigen::VectorXd timestepDissipativeWaveEquation(
  std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>>
     fe_space_p,
  double T, unsigned int M,
  Eigen::VectorXd mu0, Eigen::VectorXd nu0);
```

Here the argument `fe_space_p` specifies the finite-element space along with the mesh and the local-to-global index mapping, `T` passes the (positive) final time T, `M` the number of timesteps, and the vectors `mu0` and `nu0` give the basis expansion coefficients of approximations of u_0 and v_0 in $\mathcal{S}_2^0(\mathcal{M})$.

The function then carries out M timesteps of the dissipative leapfrog scheme (9.18.12), (9.18.13) for the method-of-lines ODE (9.18.9). It returns $\vec{\mu}^{(M)}$.

HIDDEN HINT 1 for (9-18.g) → 9-18-7-0:ldws6h1.pdf

SOLUTION for (9-18.g) → 9-18-7-1:ldws6.pdf ▲

(9-18.h) ⚄ (30 min.) [depends on Sub-problem (9-18.e)]

Let $(\mathcal{M}_\ell)_{\ell\in\mathbb{N}_0}$ be a sequence of triangular meshes of Ω created by successive uniform regular refinement as explained in [Lecture → Ex. 3.1.4.3]. Denote by $u_\ell^{(M)} \in \mathcal{S}_2^0(\mathcal{M}_\ell)$ the fully discrete finite-element Galerkin approximation of $u(\cdot,T)$, where u solves (9.18.1) and $M \in \mathbb{N}$ is the number of timesteps of the leapfrog-type scheme (9.18.12), (9.18.13). This number is assumed to be sufficient large to make the CFL-condition [Lecture → Section 9.3.5] hold. The exact solution $u = u(\boldsymbol{x},t)$ is known to be smooth, $u \in C^\infty(\overline{\Omega}\times[0,T])$.

If $M_\ell = 2^\ell M_0$, $M_0 \in \mathbb{N}$, what asymptotic behavior of $\left|u(\cdot,T) - u_\ell^{(M_\ell)}\right|_{H^1(\Omega)}$ do you expect for $\ell \to \infty$? Explain your answer.

SOLUTION for (9-18.h) → 9-18-8-0:ldws7.pdf ▲

End Problem 9-18 , 210 min.

Problem 9-19: Implicit Runge-Kutta Timestepping for a Degenerate Parabolic Evolution Problem

We consider a specimen of a 2nd-order linear degenerate parabolic evolution problem, degenerate in the sense that the operator acting on the time derivative has a kernel. For such evolution problems implicit timestepping is not only advisable in the interest of efficiency, but becomes mandatory, in order to obtain a well-defined fully discrete evolution.

For this project you need to be familiar with [Lecture → Section 9.2.4], [Lecture → Section 9.2.7.1] and LEHRFEM++.
▷ problem name/problem code folder: IRKDegenerateEvl

On a 2D polygonal domain $\Omega \subset \mathbb{R}^2$ we consider the following parabolic evolution problem in variational form: seek $u :]0,T[\to H^1(\Omega)$, $u = u(\boldsymbol{x},t)$, such that

$$\int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) + \int_{\Omega} \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H^1(\Omega)\,, \tag{9.19.1}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x})\,, \quad \boldsymbol{x}\in\Omega\,, \tag{9.19.2}$$

where the initial data $u_0 \in H^1(\Omega)$ are given.

(9-19.a) ⚄ (15 min.) Derive the strong (PDE) form of the evolution problem (9.19.1).

$$\square = \square \quad \text{in} \quad \Omega\,,$$
$$\square = \square \quad \text{on} \quad \partial\Omega\,.$$

SOLUTION for (9-19.a) → 9-19-1-0:sds0.pdf

▲

We perform a spatial Galerkin finite element semidiscretization of (9.19.1) based on the finite element space $\mathcal{S}_2^0(\mathcal{M})$ of continuous $\mathcal{M}$-piececwise quadratic polynomial functions, where $\mathcal{M}$ is a triangular mesh of Ω. We employ the standard global shape functions ("nodal basis") of $\mathcal{S}_2^0(\mathcal{M})$ as introduced in [Lecture → Ex. 2.6.1.2] and also used by LEHRFEM++. These global shape functions are denoted as b_h^i, $i = 1,\ldots,N$, $N := \dim \mathcal{S}_2^0(\mathcal{M})$.

We arrive at the **method-of-lines (MOL) ODE** for the basis expansions coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t) \in \mathbb{R}^N$ of the semi-discrete finite-element solution $t \mapsto u_h(t) \in \mathcal{S}_2^0(\mathcal{M})$:

$$\text{MOL-ODE:} \quad \mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \mathbf{0}\,, \tag{9.19.5}$$

with matrices $\mathbf{A},\mathbf{M} \in \mathbb{R}^{N,N}$.

(9-19.b) ⚄ (15 min.) Denote by $\{b_h^i\}_{i=1}^N$ the nodal basis of $\mathcal{S}_2^0(\mathcal{M})$ used in the context of finite-element Galerkin discretization. Give concrete expressions for the entries of the matrices $\mathbf{A}$ and $\mathbf{M}$ in (9.19.5).

$$\mathbf{A} = \left[\square\right]_{i,j=1}^{N}\,, \quad \mathbf{M} = \left[\square\right]_{i,j=1}^{N}\,.$$

SOLUTION for (9-19.b) → 9-19-2-0:parass1.pdf

▲

(9-19.c) ⚄ (20 min.) [depends on Sub-problem (9-19.b)]

Characterize the nullspace/kernel and the range/image of the Galerkin matrices $\mathbf{M}$ and $\mathbf{A}$:

$$\mathcal{N}(\mathbf{M}) := \{\vec{v} \in \mathbb{R}^N : \mathbf{M}\vec{v} = \mathbf{0}\} = \quad ,$$

$$\mathcal{R}(\mathbf{M}) := \{\mathbf{M}\vec{v} : \vec{v} \in \mathbb{R}^N\} = \quad ,$$

$$\mathcal{N}(\mathbf{A}) := \{\vec{v} \in \mathbb{R}^N : \mathbf{A}\vec{v} = \mathbf{0}\} = \quad ,$$

$$\mathcal{R}(\mathbf{A}) := \{\mathbf{A}\vec{v} : \vec{v} \in \mathbb{R}^N\} = \quad .$$

SOLUTION for (9-19.c) → 9-19-3-0:sdsk.pdf

▲

(9-19.d) ⚄ (10 min.) [depends on Sub-problem (9-19.c)]

Let $t \mapsto \vec{\mu}(t)$ be any solution of the MOL ODE (9.19.5). Find a non-trivial ($\hat{=}$ non-zero) *linear functional* $L : \mathbb{R}^N \to \mathbb{R}$ such that

$$L(\vec{\mu}(t)) = L(\vec{\mu}(0)) \quad \forall t\,, \quad \forall \vec{\mu}(0) \in \mathbb{R}^N\,. \tag{9.19.8}$$

$$L(\vec{v}) = \quad , \quad \vec{v} \in \mathbb{R}^N\,,$$

SOLUTION for (9-19.d) → 9-19-4-0:.pdf

▲

(9-19.e) ⚄ (25 min.) [depends on Sub-problem (9-19.b)]

In the file `irkdegenerateevl.cc` complete the code of the LEHRFEM++-based C++ function

```
[[nodiscard]] lf ::assemble::COOMatrix<double> buildM(
std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>>
   fes_p);
```

which is supposed to return the matrix $\mathbf{M}$ of (9.19.5) in triplet format. The argument `fes_p` supplies the underlying trial/test finite element space.

HIDDEN HINT 1 for (9-19.e) → 9-19-5-0:sdsh52.pdf

HIDDEN HINT 2 for (9-19.e) → 9-19-5-1:sdsh51.pdf

SOLUTION for (9-19.e) → 9-19-5-2:sds5.pdf

▲

For the temporal discretization of the method-of-lines ODE (9.19.5) we rely on the 2-stage Gauss-Radau implicit Runge-Kutta single-step method, which is characterized through the **Butcher scheme**

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \qquad \text{[Lecture → (9.2.7.50)]}$$

(9-19.f) ⚀ (35 min.) In the file `irkdegenerateevl.h` you find the C++ function

```
template <typename MATRIX,
typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
Eigen::VectorXd timesteppingIRKMOLODE(
const lf::assemble::COOMatrix<double> &M,
const lf::assemble::COOMatrix<double> &A, const MATRIX &Ark,
const Eigen::VectorXd &mu0, unsigned int no_ts, double T,
RECORDER &&rec);
```

which is supposed to implement an **implicit Runge-Kutta single-step method** with *uniform timestep size* for the method-of-lines ODE (9.19.5). Its arguments have the following meaning:

1. `A` and `M` are the matrices defining (9.19.5).
2. `Ark` is the Butcher matrix $\mathfrak{A}$ of an **L-stable** [Lecture → Def. 9.2.7.46] implicit s-stage Runge-Kutta single step method characterized by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^\top \end{array} \;\widehat{=}\; \begin{array}{c|ccccc} c_1 & a_{11} & a_{12} & \dots & \dots & a_{1s} \\ c_2 & a_{21} & \ddots & & & a_{2s} \\ \vdots & \vdots & & \ddots & & \vdots \\ c_s & a_{s1} & \dots & & & a_{ss} \\ \hline & b_1 & b_2 & \dots & \dots & b_s \end{array}, \qquad \mathbf{c},\mathbf{b} \in \mathbb{R}^s,\ \mathfrak{A} \in \mathbb{R}^{s,s}, \quad \boxed{\mathbf{b} = (\mathfrak{A})^\top_{s,:}}, \tag{9.19.14}$$

3. `mu0` is the coefficient vector providing the initial condition,
4. `no_ts` is the total number of equidistant timesteps to be carried out,
5. `T` is the (positive))final time, and
6. `rec` is a recorder object that gets all the coefficient vectors produced during timestepping.

The function should return the basis expansion coefficient vector approximating $\vec{\mu}(T)$.

The current implementation of `timesteppingIRKMOLODE()` `irkdegenerateevl.h` is incomplete; parts of the code are missing and those are indicated by **REPLACE**, see the following listing. Your task is to replace the **REPLACE** placeholders, and only those, with *valid C++ code* so that the resulting functions complies with the specification.

C++ code 9.19.15: Incomplete code for `timesteppingIRKMOLODE()`

```
2  template <typename MATRIX,
3            typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
4  Eigen::VectorXd timesteppingIRKMOLODE(
5      const lf::assemble::COOMatrix<double> &M,
6      const lf::assemble::COOMatrix<double> &A, const MATRIX &Ark,
7      const Eigen::VectorXd &mu0, unsigned int no_ts, double T,
8      RECORDER &&rec = [](const Eigen::VectorXd & /*mu_vec*/) -> void {}) {
9    const Eigen::Index s = Ark.cols();
10   const Eigen::Index N = A.cols();
11   LF_ASSERT_MSG(s == Ark.rows(), "Butcher matrix must be square");
12   LF_ASSERT_MSG(N == A.rows(), "Matrix A must be square");
13   LF_ASSERT_MSG((N == M.rows()) and (N == M.cols()),
14                 "Matrix M same size as A!");
15   LF_ASSERT_MSG(mu0.size() == N, "Illegal size of mu0!");
16   const double tau = T / no_ts;
17   // Initialization of system matrix for combined increment equations
18   lf::assemble::COOMatrix<double> ISM(REPLACE, REPLACE);
19   const lf::assemble::COOMatrix<double>::TripletVec Mtv{M.triplets()};
```

```
  const lf::assemble::COOMatrix<double>::TripletVec Atv{A.triplets()};
  for (int i = 0; i < REPLACE; ++i) {
    for (int j = 0; j < REPLACE; ++j) {
      if (i == j) {
        for (const Eigen::Triplet<double> &M_trp : Mtv) {
          ISM.AddToEntry(REPLACE, REPLACE, REPLACE);
        }
      }
      for (const Eigen::Triplet<double> &A_trp : Atv) {
        ISM.AddToEntry(REPLACE, REPLACE, REPLACE);
      }
    }
  }

  // Advance LU decomposition of system matrix for increment equations
  Eigen::SparseMatrix<double> ISM_crs{ISM.makeSparse()};
  Eigen::SparseLU<Eigen::SparseMatrix<double>> ISLU;
  ISLU.compute(ISM_crs);

  // Main timestepping loop
  Eigen::VectorXd mu{mu0};
  rec(mu);
  Eigen::VectorXd incs(REPLACE);
  Eigen::VectorXd Amu(N);
  for (int k = 0; k < no_ts; ++k) {
    // Solve increment system
    Amu = -A.MatVecMult(REPLACE, REPLACE);
    incs = ISLU.solve(Amu.replicate(REPLACE, REPLACE));
    LF_VERIFY_MSG(ISLU.info() == Eigen::Success,
            "Solving increment system failed");
    mu += REPLACE *
      Eigen::Map<Eigen::MatrixXd>(incs.data(), REPLACE, REPLACE) * REPLACE;
    rec(mu);
  }
  return mu;
}
```

SOLUTION for (9-19.f) → 9-19-6-0:sds6.pdf

▲

(9-19.g) ⚂ (15 min.) We pass to `timesteppingIRKMOLODE()` the "Butcher matrix" $\mathfrak{A} \in \mathbb{R}^{2,2}$ of the 2-stage *third-order* Radau IRK-SSM [Lecture → Ex. 7.3.4.21]. The recursion in the timesteppping loop then produces the coefficient vectors $\vec{\mu}^{(j)}$, $j \in \mathbb{N}_0$.

Write $u_h^{(j)} \in \mathcal{S}_2^0(\mathcal{M})$ for the fully-discete approximation of $u(\cdot, \tau j)$, that is, the finite-element function with basis expansion coefficient vector $\vec{\mu}^{(j)}$, $j \in \mathbb{N}_0$.

We consider the discretization error

$$\operatorname{err}(h,\tau) := \left(\tau \sum_{j=1}^{M} \left| u(\tau j) - u_h^{(j)} \right|^2_{H^1(\Omega)}\right)^{\frac{1}{2}} \tag{9.19.17}$$

Assuming that the solution $u = u(x,t)$ of the parabolic evolution problem is smooth and that the asymptotic estimates of [Lecture → Thm. 9.2.8.5] apply and are sharp, answer the following questions.

- How many steps of *uniform regular refinement* of $\mathcal{M}$ have to be carried out and
- by what factor does the timestep τ have to be reduced in order to achieve a reduction of the norm

(as given in (9.19.17)) of the full discretization error *by a factor* 100?

$$\sharp(\text{required steps of regular refinement of } \mathcal{M}) = \boxed{}\,,$$

$$\text{factor by which timestep } \tau \text{ has to be reduced} = \boxed{}\,.$$

SOLUTION for (9-19.g) → 9-19-7-0:irkd4xs.pdf

▲

(9-19.h) ⚄ (25 min.) In the file `irkdegenerateevl.cc` finish the implementation of the LEHRFEM++-based C++ function

```
template <typename U0FUNCTOR>
std::vector<std::pair<double, double>> tabulateSolNorms(
    std::shared_ptr<const
       lf::uscalfe::UniformScalarFESpace<double>> fes_p,
    U0FUNCTOR &&u0, unsigned int no_ts, double T, std::ostream &o);
```

which runs `timesteppingIRKMOLODE()` with the Butcher matrix of the 2-stage Radau IRK-SSM, `no_ts` timesteps until final time `T`. Spatial Galerkin finite element discretization is based on the Lagrangian finite element space encoded in `*fes_p`.

The function should write a table

$$\begin{array}{ccc} & \vdots & \\ j \qquad \left|u_h^{(j)}\right|_{H^1(\Omega)} & & \left\|u_h^{(j)}\right\|_{L^2(\partial\Omega)} \\ & \vdots & \end{array}$$

to the output stream `o`. Here, as in Sub-problem (9-19.g), $u_h^{(j)} \in H^1(\Omega)$ is the finite-element function described by the basis expansion coefficient vector $\vec{\mu}^{(j)}$ generated in the j-th timestep.

All you need to write is a suitable λ-function to be passed to `timesteppingIRKMOLODE` as `rec` argument.

SOLUTION for (9-19.h) → 9-19-8-0:.pdf

▲

End Problem 9-19 , 160 min.

Problem 9-20: Method-of-Lines with SDIRK timestepping

For the sake of efficiency in order to avoid a prohibitively severe CFL-condition L-stable implicit timestepping methods are recommended for the solution of the finite-dimensional initial value problems arising from the method-of-lines applied to parabolic initial boundary value problems, see [Lecture → Rem. 9.2.8.10]. In this problem we review the method-of-lines approach with focus on a particular class of implicit Runge-Kutta single-step methods known as **SDIRK schemes** (**s**ingly **d**iagonally **i**mplicit **R**unge-**K**utta methods)

Requires familiarity with [Lecture → Section 9.2.2], [Lecture → Section 9.2.4], [Lecture → Section 9.2.6.5], [Lecture → Section 9.2.7.1], and [Lecture → Section 9.2.8].
▷ problem name/problem code folder: MOLSDIRK

On a 2D polygonal domain $\Omega \subset \mathbb{R}^2$ we consider the following parabolic evolution problem in variational form: seek $u :]0,T[\to H^1(\Omega)$, $u = u(\boldsymbol{x},t)$, such that

$$\int_\Omega \sigma(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H^1(\Omega)\,, \tag{9.20.1}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Omega\,, \tag{9.20.2}$$

where the initial data $u_0 \in H^1(\Omega)$ are given and $\sigma \in C^0(\Omega)$ is a uniformly positive coefficient function that does not depend on time.

(9-20.a) ⚂ (15 min.) Assuming sufficient smoothness of u, write down the strong (PDE) form of the evolution problem (9.20.1).

$$\square = \square \quad \text{in} \quad \Omega \times]0,T[\,,$$
$$\square = \square \quad \text{on} \quad \partial\Omega \times]0,T[\,,$$
$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x})\,,\ quad \boldsymbol{x} \in \Omega\,,$$

SOLUTION for (9-20.a) → 9-20-1-0:molss1.pdf ▲

We perform a spatial Galerkin finite element semidiscretization of (9.20.1) based on the finite element space $\mathcal{S}_2^0(\mathcal{M})$ of continuous $\mathcal{M}$-piececwise quadratic polynomial functions, where $\mathcal{M}$ is a triangular mesh of Ω. We employ the standard global shape functions ("nodal basis") of $\mathcal{S}_2^0(\mathcal{M})$ as introduced in [Lecture → Ex. 2.6.1.2]. These global shape functions are denoted as b_h^i, $i = 1,\ldots,N$, $N := \dim \mathcal{S}_2^0(\mathcal{M})$.

We arrive at the **method-of-lines (MOL) ODE** for the basis expansions coefficient vector $t \mapsto \vec{\boldsymbol{\mu}}(t) \in \mathbb{R}^N$ of the semi-discrete finite-element solution $t \mapsto u_h(t) \in \mathcal{S}_2^0(\mathcal{M})$:

$$\text{MOL-ODE:} \qquad \mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \mathbf{0}\,, \tag{9.20.5}$$

with matrices $\mathbf{A}, \mathbf{M} \in \mathbb{R}^{N,N}$.

(9-20.b) ⚂ (10 min.) Denote by $\{b_h^i\}_{i=1}^N$ the nodal basis of $\mathcal{S}_2^0(\mathcal{M})$ used in the context of finite-element Galerkin discretization. Give concrete expressions for the entries of the matrices $\mathbf{A}$ and $\mathbf{M}$ in (9.20.5).

$$\mathbf{A} = \left[\ \square\ \right]_{i,j=1}^{N}\,, \qquad \mathbf{M} = \left[\ \square\ \right]_{i,j=1}^{N}\,.$$

SOLUTION for (9-20.b) → 9-20-2-0:parass1.pdf ▲

For timestepping, the second step of the method-of-lines approach, we employ the L-stable SDIRK-2 implicit 2-stage Runge-Kutta method of [Lecture → Ex. 9.2.7.49] defined by the **Butcher scheme**

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \;\hat{=}\; \begin{array}{c|cc} \zeta & \zeta & 0 \\ 1 & 1-\zeta & \zeta \\ \hline & 1-\zeta & \zeta \end{array} \quad , \quad \zeta := 1 - \tfrac{1}{2}\sqrt{2} > 0 \, . \tag{9.20.6}$$

From literature we know that this single-step method is of *second order*.

(9-20.c) ⚄ (20 min.) The single SDIRK-2 RK-SSM applied to the MOL-ODE (9.20.5) with uniform timstep $\tau > 0$ over $M \in \mathbb{N}$ steps yields a sequence of basis expansion coefficient vectors $\left(\vec{\mu}^{(k)}\right)_{k=0}^{M} \subset \mathbb{R}^N$, where $\vec{\mu}^{(0)}$ is obtained by an interpolation of $\mathbf{u}_0$.

Determine the $N \times N$ matrix blocks in the following $2N \times 2N$ linear system of equations, such it can be used to generate the sequence $\left(\vec{\mu}^{(k)}\right)_{k=0}^{M}$.

$$\begin{bmatrix} & \\ & \end{bmatrix} \begin{bmatrix} \vec{\gamma}_1 \\ \vec{\mu}^{(k)} \end{bmatrix} = \begin{bmatrix} & \\ & \end{bmatrix} \vec{\mu}^{(k-1)} \, ,$$

$k = 1, \ldots, M$. Here $\vec{\gamma}_1 \in \mathbb{R}^N$ is an auxiliary vector.

HIDDEN HINT 1 for (9-20.c) → 9-20-3-0:molsh1.pdf

SOLUTION for (9-20.c) → 9-20-3-1:molss4.pdf ▲

(9-20.d) ⚄ (15 min.) We write $u_h^{(k)} \in \mathcal{S}_2^0(\mathcal{M})$, $k = 0, \ldots, M$ for the method-of-lines solution of (9.20.1) obtained by the approach described above and using $M = \frac{1}{\tau}$ uniform timesteps $\tau := \frac{T}{M}$, $T > 0$ some final time for the simulation.

Assuming that the exact solution $u : [0, T] \to H^1(\Omega)$ of (9.20.1) is sufficiently smooth in space and time, complete the following estimate for the space-time MOL discretization error:

$$\left(\tau \sum_{j=1}^{M} \left| u(\tau j) - u_h^{(j)} \right|_{H^1(\Omega)}^2 \right)^{\frac{1}{2}} \leq C \Big(\quad \Big) \, .$$

where $C > 0$ must not depend on the meshwidth $h_{\mathcal{M}}$ and τ.

SOLUTION for (9-20.d) → 9-20-4-0:molys.pdf ▲

End Problem 9-20 , 60 min.

Problem 9-21: Magnetic Diffusion in a Wire

We consider a boundary value problem from 2D magneto-quasistatic electrodynamics, which boils down to a degenerate scalar parabolic evolution problem. In the framework of the method of lines (MOL) finite element discretization in space is combined with an implicit Runge-Kutta single step method for timestepping.

Requires familiarity with [Lecture → Section 9.2.2], [Lecture → Section 9.2.4], [Lecture → Section 9.2.6.5], [Lecture → Section 9.2.7.1], and [Lecture → Section 9.2.8].
▷ problem name/problem code folder: MagDiffWire

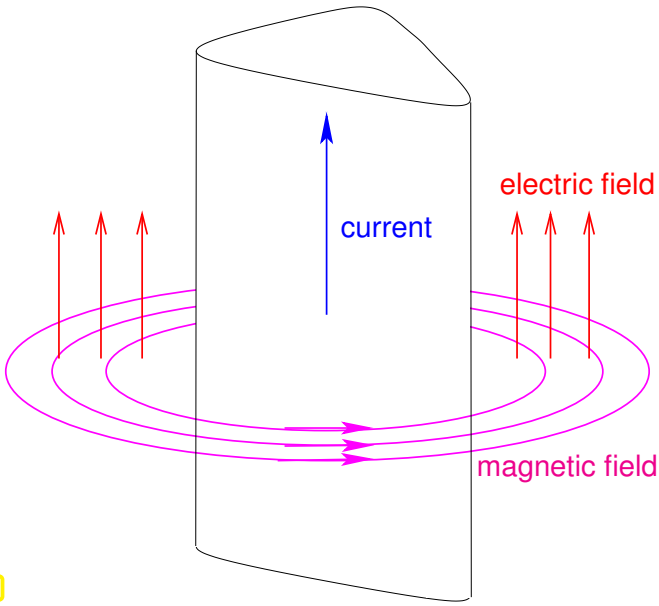


Fig. 149

A given time-dependent current $t \mapsto I(t) \in \mathbb{R}$ is sent through a long straight aluminum bar with cross section $\Omega_C \subset \mathbb{R}^2$.
Concerning its electromagnetic properties aluminum can be modeled as homogeneous, isotropic, and linear medium with constant conductivity $\sigma(\boldsymbol{x}) = \sigma_c > 0$ and permeability $\mu(\boldsymbol{x}) = \mu_c > 0$.

We are interested in the spatio-temporal behavior of the electromagnetic fields over a finite time interval $[0,T]$, final time $T > 0$ given, when the current is switched on at $t_0 = 0$.

A suitable mathematical model is a parabolic evolution problem for the component $u = u(\boldsymbol{x},t) : \mathbb{R}^2 \times]0,T[\to \mathbb{R}$ of the electric field parallel to the bar. Since the electric field decays away from Ω_C we resort to artificial truncation, switch to a sufficiently large bounded domain $\Omega \subset \mathbb{R}^2$ with $\overline{\Omega_C} \subset \Omega$, and impose zero Dirichlet boundary conditions on $\partial\Omega$. After scaling, we arrive at the following non-dimensional parabolic evolution problem in variational form.

Seek $u : [0,T] \mapsto H^1_0(\Omega)$, $U : [0,T] \to \mathbb{R}$, such that

$$\begin{aligned} \int_\Omega \sigma(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x}) + \frac{1}{\mu(\boldsymbol{x})} \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \;+\; U(t) \int_\Omega \sigma(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} &= 0\,, \\ \int_\Omega \sigma(\boldsymbol{x}) u(\boldsymbol{x},t)\, \mathrm{d}\boldsymbol{x} &= I(t) \end{aligned}$$

for all $v \in H^1_0(\Omega)$. (9.21.1)

Here we have set

$$\sigma(\boldsymbol{x}) = \begin{cases} \sigma_c & \text{for } \boldsymbol{x} \in \Omega_C\,, \\ 0 & \text{for } \boldsymbol{x} \in \Omega \setminus \overline{\Omega_C} \end{cases}\,, \quad \mu(\boldsymbol{x}) = \begin{cases} \mu_c & \text{for } \boldsymbol{x} \in \Omega_C\,, \\ 1 & \text{for } \boldsymbol{x} \in \Omega \setminus \overline{\Omega_C}\,. \end{cases} \tag{9.21.2}$$

At initial time $t_0 = 0$ no fields have been excited yet, which amounts to the initial condition

$$u(\boldsymbol{x},0) = 0 \quad \forall \boldsymbol{x} \in \Omega\,. \tag{9.21.3}$$

(9-21.a) ⚀ (4 min.) Assuming sufficient smoothness of u and $\boldsymbol{x} \mapsto \mu(\boldsymbol{x})$, write down the strong

(PDE) form of the evolution problem (9.21.1),

$$\square = \square \quad \text{in} \quad \Omega\times]0,T[\,,$$
$$\square = \square \quad \text{on} \quad \partial\Omega\times]0,T[\,,$$
$$\square = \square \quad \text{for all } 0 \le t \le T\,,$$
$$u(\boldsymbol{x},0) = 0\,, \quad \text{for all } \boldsymbol{x} \in \Omega\,.$$

SOLUTION for (9-21.a) → 9-21-1-0:mdws1.pdf ▲

We assume that both Ω and Ω_C have a polygonal boundary. We perform a spatial Galerkin finite element semidiscretization of (9.21.1) based on the finite element space $\mathcal{S}^0_{2,0}(\mathcal{M})$ of continuous $\mathcal{M}$-piecewise quadratic polynomial functions vanishing on $\partial\Omega$, where $\mathcal{M}$ is a triangular mesh of Ω. The mesh is supposed to resolve Ω_C in the sense that $\overline{\Omega}_C$ is the union of a subset of (closed) cells of $\mathcal{M}$. We employ the standard global shape functions ("nodal basis") of $\mathcal{S}^0_2(\mathcal{M})$ as introduced in [Lecture → Ex. 2.6.1.2]. These global shape functions are denoted as b^i_h, $i = 1,\ldots,N$, $N := \dim \mathcal{S}^0_{2,0}(\mathcal{M})$.

We write $t \mapsto \vec{\boldsymbol{\nu}}(t) \in \mathbb{R}^N$ for the time-dependent basis expansion coefficient vector for the semi-discrete finite element solution $t \mapsto u_h(t) \in \mathcal{S}^0_{2,0}(\mathcal{M})$. This vector-valued function of time satisfies the **method-of-lines (MOL) ODE**

$$\text{MOL-ODE:} \quad \begin{bmatrix} \mathbf{M} & \mathbf{d} \\ \mathbf{d}^\top & \alpha \end{bmatrix} \frac{d}{dt} \begin{bmatrix} \vec{\boldsymbol{\nu}}(t) \\ U(t) \end{bmatrix} + \begin{bmatrix} \mathbf{A} & \mathbf{q} \\ \mathbf{q}^\top & \beta \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\nu}}(t) \\ U(t) \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}(t) \\ \psi(t) \end{bmatrix}\,, \tag{9.21.8}$$

with matrices $\mathbf{A},\mathbf{M} \in \mathbb{R}^{N,N}$, vectors $\mathbf{d},\mathbf{q} \in \mathbb{R}^N$, scalars $\alpha,\beta \in \mathbb{R}$ and functions $\vec{\boldsymbol{\varphi}} : [0,T] \to \mathbb{R}^N$ and $\psi : [0,T] \to \mathbb{R}$.

(9-21.b) ⚀ (6 min.) As above, denote by $\{b^i_h\}_{i=1}^N$ the nodal basis of $\mathcal{S}^0_{2,0}(\mathcal{M})$ used in the context of finite-element Galerkin discretization. Give concrete expressions for the entries of the matrices and vectors and the values of the scalars in (9.21.8).

$$\mathbf{A} = \Big[\square\Big]_{i,j=1}^N\,, \quad \mathbf{M} = \Big[\square\Big]_{i,j=1}^N\,,$$
$$\mathbf{d} = \Big[\square\Big]_{i,j=1}^N\,, \quad \mathbf{q} = \Big[\square\Big]_{i,j=1}^N\,,$$
$$\vec{\boldsymbol{\varphi}}(t) = \Big[\square\Big]_{i=1}^N\,,$$
$$\alpha = \square\,, \quad \beta = \square\,, \quad \psi(t) = \square\,.$$

SOLUTION for (9-21.b) → 9-21-2-0:mdws2.pdf ▲

We rewrite (9.21.8) in more compact notation

$$(9.21.8) \quad \Leftrightarrow \quad \widetilde{\mathbf{M}}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \widetilde{\mathbf{A}}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{\rho}}(t)\,, \tag{9.21.12}$$

$$\text{with} \quad \vec{\mu}(t) := \begin{bmatrix} \vec{v}(t) \\ U(t) \end{bmatrix} : [0,T] \to \mathbb{R}^{N+1} \quad , \quad \vec{\rho}(t) := \begin{bmatrix} \vec{\varphi}(t) \\ \psi(t) \end{bmatrix} : [0,T] \mapsto \mathbb{R}^{N+1} \, , \tag{9.21.13}$$
$$\widetilde{\mathbf{M}} := \begin{bmatrix} \mathbf{M} & \mathbf{d} \\ \mathbf{d}^\top & \alpha \end{bmatrix} \in \mathbb{R}^{N+1,N+1} \quad , \quad \widetilde{\mathbf{A}} := \begin{bmatrix} \mathbf{A} & \mathbf{q} \\ \mathbf{q}^\top & \beta \end{bmatrix} \in \mathbb{R}^{N+1,N+1} \, .$$

(9-21.c) ⚄ (12 min.) [depends on Sub-problem (9-21.b)]

The file `magdiffwire.h` contains an unfinished implementation of the function

```
std::pair<lf::assemble::COOMatrix<double>,
          lf::assemble::COOMatrix<double>>
buildExtMOLMatrices(
  std::shared_ptr<const lf::fe::ScalarFESpace<double>> fes_p,
  lf::mesh::utils::CodimMeshDataSet<bool> Oc_flags,
  double sigma_c, double mu_c);
```

which provides the pair of matrices $(\widetilde{\mathbf{M}}, \widetilde{\mathbf{A}})$ as defined in (9.21.13). Its arguments are

- `fes_p`, which points to an object completely characterizing the Lagrangian finite element space,
- `Oc_flags` a cell-indexed array of flags, which mark a cell as located in Ω_C when `true`,
- `sigma_c`, `mu_c`, passing the constant values σ_c and μ_c of the coefficients $\sigma(\boldsymbol{x})$ and $\mu(\boldsymbol{x})$ inside Ω_C, see (9.21.2).

The sparse matrices $\widetilde{\mathbf{M}}$ and $\widetilde{\mathbf{A}}$ are returned in LehrFEM++'s triplet matrix format as **lf::assemble::COOMatrix** objects.

The given code does not yet set the bottom row and rightmost column of $\widetilde{\mathbf{A}}$. Finish the implementation.

Hidden hint 1 for (9-21.c) → `9-21-3-0:mdws2a.pdf`

Hidden hint 2 for (9-21.c) → `9-21-3-1:mdwh2a1.pdf`

Hidden hint 3 for (9-21.c) → `9-21-3-2:mdw2ah2.pdf`

Solution for (9-21.c) → `9-21-3-3:mdw2as.pdf` ▲

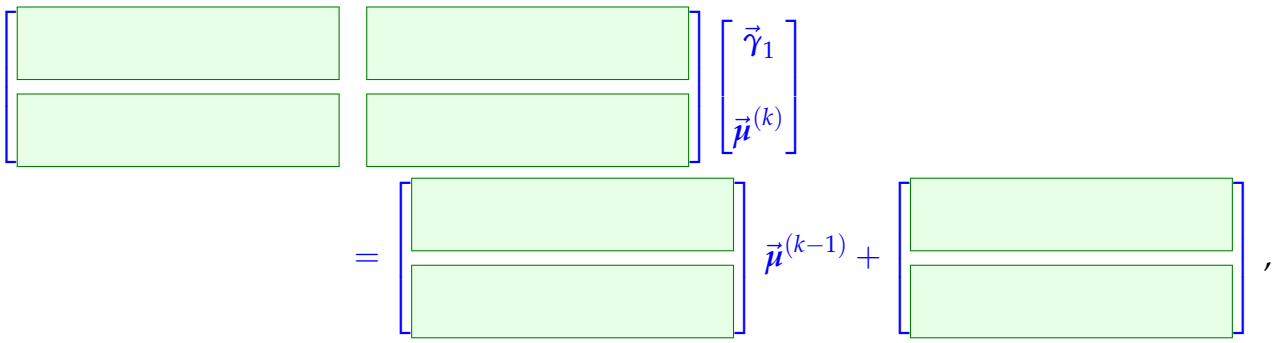
For timestepping, the second step of the method-of-lines approach, we employ the L-stable SDIRK-2 implicit 2-stage Runge-Kutta method of [Lecture → Ex. 9.2.7.49] defined by the **Butcher scheme**

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \hat{=} \quad \begin{array}{c|cc} \zeta & \zeta & 0 \\ 1 & 1-\zeta & \zeta \\ \hline & 1-\zeta & \zeta \end{array} \quad , \quad \zeta := 1 - \tfrac{1}{2}\sqrt{2} > 0 \, . \tag{9.21.16}$$

From literature we know that this single-step method is of *second order*.

(9-21.d) ⚄ (6 min.) The SDIRK-2 RK-SSM applied to the MOL-ODE (9.21.8) with uniform timstep $\tau > 0$ over $M \in \mathbb{N}$ steps yields a sequence of basis expansion coefficient vectors $\left(\vec{\mu}^{(k)}\right)_{k=0}^{M} \subset \mathbb{R}^{N+1}$, where $\vec{\mu}^{(0)}$ is obtained by an interpolation of $\mathbf{u}_0$.

Determine the $(N+1) \times (N+1)$ matrix blocks (as expressions of $\widetilde{\mathbf{M}}$ and $\widetilde{\mathbf{A}}$) in the following $2(N+1) \times 2(N+1)$ linear system of equations, such it can be used to generate the sequence $\left(\vec{\mu}^{(k)}\right)_{k=0}^{M}$.



$k = 1, \ldots, M$. Here $\vec{\gamma}_1 \in \mathbb{R}^{N+1}$ is an auxiliary vector.

HIDDEN HINT 1 for (9-21.d) → 9-21-4-0:mdwh31.pdf

HIDDEN HINT 2 for (9-21.d) → 9-21-4-1:mdwh32.pdf

SOLUTION for (9-21.d) → 9-21-4-2:mdws3.pdf ▲

(9-21.e) ⚄ (12 min.) [depends on Sub-problem (9-21.d)]

In the file `magdiffwire.h` you find the incomplete code of the C++ function

```
template <typename SCALARFUNCTOR,
          typename RECORDER =
            std::function<void(double, const Eigen::VectorXd &)>>
void sdirkMagDiffWire(
    std::shared_ptr<const lf::fe::ScalarFESpace<double>> fes_p,
    lf::mesh::utils::CodimMeshDataSet<bool> Oc_flags,
    double sigma_c, double mu_c, SCALARFUNCTOR I_source,
    unsigned int M, double T_final, Eigen::VectorXd &mu_vec,
    RECORDER rec = [](double, const Eigen::VectorXd &) -> void {})
```

It is supposed to carry out M equidistant steps of the SDIRK-2 RK-SSM for (9.21.12). Its parameters have the following purposes:

- `fes_p` passes information on the underlying Lagrangian finite element space, see [Lecture → § 2.8.3.28].
- `OC_flags` is a cell-based flag array, which sets a `true` flag for all cells $\subset \Omega_C$.
- `sigma_c` and `mu_c` contain the coefficients σ_c and μ_c as used in (9.21.2).
- `I_source` is a functor object of type `std::function<double(double)>` encoding the source current $t \mapsto I(t)$.
- `M` is the total number of timesteps.
- `T_final` is the final time $T > 0$.
- `mu_vec` provides the initial state vector $\vec{\mu}^{(0)}$ and is used to return the final state $\vec{\mu}^{(M)}$.
- `rec` is an object that received all the states $\vec{\mu}^{(k)}$ during timestepping.

Finish implementing this function.

HIDDEN HINT 1 for (9-21.e) → 9-21-5-0:mdws4s.pdf

SOLUTION for (9-21.e) → 9-21-5-1:mdws4.pdf ▲

End Problem 9-21, 40 min.

Chapter 10

Convection-Diffusion Problems

Problem 10-1: Exponentially fitted upwind scheme

In this problem we deal with a special convection-diffusion boundary value problem arising in semiconductor modeling and known as (stationary) drift-diffusion model in that field [Mar86]. From a fundamental assumption we derive the resulting linear system of equations and verify stability in the form of a discrete maximum principle, see [Lecture → Section 3.7] and [Lecture → Section 10.2.2]. This problem was inspired by [MZ88; XZ99].

This problem is related to [Lecture → Section 10.2] and it involves C++ coding tasks based on LEHRFEM++.
▷ problem name/problem code folder: ExpFittedUpwind

Let $\Omega \subset \mathbb{R}^2$ denote the polygonal cross-section of a semiconductor and $u : \Omega \to \mathbb{R}$, units $[u] = \mathrm{m}^{-2}$, stand for the number density of a species of charge carriers, either holes (positive elementary charge) or electrons (negative elementary charge). We focus on electrons.

The derivation of the drift-diffusion model for a stationary electron distribution parallels that of the heat conduction model in a moving fluid from [Lecture → Section 10.1.2]. The fundamental conservation law is the conservation of charge, in local form written as

$$\operatorname{div} \mathbf{j}(\boldsymbol{x}) = f(\boldsymbol{x}) , \quad \boldsymbol{x} \in \Omega , \tag{10.1.1}$$

where

- $\mathbf{j} : \Omega \to \mathbb{R}^2$ is the electron current, $[\mathbf{j}] = \frac{\mathrm{A}}{\mathrm{m}^2}$, and
- the source term $f : \Omega \to \mathbb{R}$ is a so-called generation-recombination term. In the actual model it will depend on u, which we ignore in this problem.

We also need a counterpart of the extended Fourier's law [Lecture → (10.1.2.4)]. A net movement of electrons is caused by two effects:

(I) diffusion due to spatial variation of charge carrier density

$$\text{diffusive flux:} \quad \mathbf{j}_{\mathrm{diff}}(\boldsymbol{x}) = -\alpha \operatorname{\mathbf{grad}} u(\boldsymbol{x}) , \quad \boldsymbol{x} \in \Omega , \tag{10.1.2}$$

with a positive diffusion coefficient α, $[\alpha] = \mathrm{Am}$,

(II) electron drift triggered by the force due to an electric field $\mathbf{E} : \Omega \to \mathbb{R}^2$. In a stationary setting this electric field will be the gradient of a potential $\Psi : \Omega \to \mathbb{R}$, $[\Psi] = \mathrm{V}$: $\mathbf{E} = -\operatorname{\mathbf{grad}} \Psi$, see [Lecture → (1.2.2.2)].

$$\text{convective flux:} \quad \mathbf{j}_{\mathrm{drift}}(\boldsymbol{x}) = \mu u(\boldsymbol{x}) \operatorname{\mathbf{grad}} \Psi(\boldsymbol{x}) , \quad \boldsymbol{x} \in \Omega , \tag{10.1.3}$$

with a positive mobility μ, $[\mu] = \frac{\mathrm{m}^2\mathrm{A}}{\mathrm{V}}$.

Adding both contributions yields the total electron flux:

$$\mathbf{j}(\boldsymbol{x}) := \mathbf{j}_{\mathrm{diff}}(\boldsymbol{x}) + \mathbf{j}_{\mathrm{drift}}(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,. \tag{10.1.4}$$

Then, inserting (10.1.2), (10.1.3) into (10.1.1), we end up with

$$\operatorname{div}(\mu\, u(\boldsymbol{x})\, \mathbf{grad}\, \Psi(\boldsymbol{x}) - \alpha\, \mathbf{grad}\, u(\boldsymbol{x})) = f(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,. \tag{10.1.5}$$

After scaling (→ [Lecture → Rem. 1.2.1.25]) and retaining the notations also for the rescaled quantities, we arrive at the non-dimensional equation

$$\operatorname{div}(-\mathbf{grad}\, u(\boldsymbol{x}) + u(\boldsymbol{x})\, \mathbf{grad}\, \Psi(\boldsymbol{x})) = f(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,. \tag{10.1.6}$$

This has to be supplemented with boundary conditions of the usual kind [Lecture → (1.7.0.2)], [Lecture → (1.7.0.3)]:

$$u = g \quad \text{on} \quad \Gamma_D \qquad \text{(on contacts)} \tag{10.1.7a}$$

$$\mathbf{j} \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \Gamma_I \qquad \text{(on insulated boundary parts)} \,. \tag{10.1.7b}$$

for a partition $\partial\Omega = \Gamma_D \cup \Gamma_I$.

Throughout we assume that $\Psi \in H^1(\Omega) \cap C^0(\overline{\Omega})$ with bounded gradient. Of course, the Dirichlet data g must be continuous: $g \in C^0(\overline{\Gamma_D})$.

(10-1.a) ⚄ (15 min.) Performing a single integration by parts by means of Green's formula from [Lecture → Thm. 1.5.2.7], cast the boundary value problem (10.1.6)+(10.1.7) into variational form.

SOLUTION for (10-1.a) → 10-1-1-0:sp1.pdf ▲

(10-1.b) ⚄ (10 min.) Show that, in the non-dimensionalized model (10.1.6), the **flux**

$$\mathbf{j}(\boldsymbol{x}) := u(\boldsymbol{x})\, \mathbf{grad}\, \Psi(\boldsymbol{x}) - \mathbf{grad}\, u(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,, \tag{10.1.10}$$

can be expressed as

$$\mathbf{j}(\boldsymbol{x}) = -\exp(\Psi(\boldsymbol{x}))\, \mathbf{grad}[\exp(-\Psi(\boldsymbol{x}))\, u(\boldsymbol{x})] \,, \quad \boldsymbol{x} \in \Omega \,. \tag{10.1.11}$$

Sufficient smoothness of u and Ψ can be taken for granted.

HIDDEN HINT 1 for (10-1.b) → 10-1-2-0:sp2h1.pdf

SOLUTION for (10-1.b) → 10-1-2-1:sp2.pdf ▲

(10-1.c) ⚄ (10 min.) [depends on Sub-problem (10-1.a)]

State the variational form of the boundary value problem (10.1.6)+(10.1.7) in terms of the new unknown function $w(\boldsymbol{x}) := \exp(-\Psi(\boldsymbol{x}))u(\boldsymbol{x})$.

HIDDEN HINT 1 for (10-1.c) → 10-1-3-0:sp2ah1.pdf

SOLUTION for (10-1.c) → 10-1-3-1:sp2a.pdf ▲

(10-1.d) ⚄ (10 min.) [depends on Sub-problem (10-1.c)]

Which result from the course ensures existence and uniqueness of weak solutions of the BVP (10.1.6)+(10.1.7)?

HIDDEN HINT 1 for (10-1.d) → 10-1-4-0:sp2bh1.pdf

SOLUTION for (10-1.d) → 10-1-4-1:sp2b.pdf ▲

The **exponentially fitted upwind method** for the BVP (10.1.6)+(10.1.7) approximates u by a piecewise linear $u_h \in \mathcal{S}_1^0(\mathcal{M})$ built on a triangular mesh $\mathcal{M}$ of Ω.

The method takes for granted that the flux $\mathbf{j}$ is smooth.

It assumes $\mathbf{j}$ to be a $\mathcal{M}$-piecewise constant vectorfield

Step by step we will derive the equations arising from this premise and the variational form of the charge conservation law (10.1.1).

(10-1.e) ⚄ (20 min.) Show that for every triangle $K \in \mathcal{M}$ with vertices $\boldsymbol{a}_K^1$, $\boldsymbol{a}_K^2$, and $\boldsymbol{a}_K^3$

$$\int_K \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \,\mathrm{d}\boldsymbol{x} = -\sum_{1\le i<j\le 3} (\mathbf{A}_K)_{i,j} \Big(u_h(\boldsymbol{a}_K^i) - u_h(\boldsymbol{a}_K^j)\Big)\Big(v_h(\boldsymbol{a}_K^i) - v_h(\boldsymbol{a}_K^j)\Big) \tag{10.1.13}$$

for all $u_h, v_h \in \mathcal{S}_1^0(\mathcal{M})$, where $\mathbf{A}_K \in \mathbb{R}^{3,3}$ is the element matrix on K, for the bilinear form of the negative Laplacian $-\Delta$,

$$(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}\,,$$

and the barycentric coordinate functions as local shape functions, see [Lecture → Section 2.4.5].

HIDDEN HINT 1 for (10-1.e) → 10-1-5-0:sp3h1.pdf

SOLUTION for (10-1.e) → 10-1-5-1:sp3.pdf ▲

(10-1.f) ⚄ (10 min.) [depends on Sub-problem (10-1.e)]

We use the notations of Sub-problem (10-1.e). Show that for any constant vector $\boldsymbol{j} \in \mathbb{R}^2$

$$\int_K \boldsymbol{j} \cdot \mathbf{grad}\, v_h \,\mathrm{d}\boldsymbol{x} = -\sum_{1\le i<j\le 3} (\mathbf{A}_K)_{i,j}\, \boldsymbol{j}\cdot(\boldsymbol{a}_K^i - \boldsymbol{a}_K^j)\Big(v_h(\boldsymbol{a}_K^i) - v_h(\boldsymbol{a}_K^j)\Big) \tag{10.1.16}$$

for all $v_h \in \mathcal{S}_1^0(\mathcal{M})$.

HIDDEN HINT 1 for (10-1.f) → 10-1-6-0:sp4h1.pdf

SOLUTION for (10-1.f) → 10-1-6-1:sp4.pdf ▲

(10-1.g) ⚄ (30 min.) [depends on Sub-problem (10-1.b)]

For a triangle $K \in \mathcal{M}$ with vertices $\boldsymbol{a}_K^1$, $\boldsymbol{a}_K^2$, and $\boldsymbol{a}_K^3$, write $\boldsymbol{d}_{j,i} := \boldsymbol{a}_K^i - \boldsymbol{a}_K^j$, $1 \le j < i \le 3$, for the edge vectors, and introduce the parameterization

$$\gamma_{j,i} : [0,1] \to \mathbb{R}^2 \quad , \quad \gamma_{j,i}(\xi) := \boldsymbol{a}_K^j + \xi \boldsymbol{d}_{j,i}\,,$$

of the line segment $[\boldsymbol{a}_K^j, \boldsymbol{a}_K^i]$.

For the flux $\mathbf{j}$ as defined in (10.1.10) we *assume* that it is constant along all edges:

$$\xi \mapsto \boldsymbol{c}_{j,i} := \mathbf{j}(u(\gamma_{j,i}(\xi))) \overset{!}{\equiv} \text{const} \in \mathbb{R}^2\,. \tag{10.1.17}$$

Under this assumption derive an expression for $\boldsymbol{c}_{j,i} \cdot \boldsymbol{d}_{j,i}$ that merely depends on the potential $\Psi|_{[\boldsymbol{a}_K^j, \boldsymbol{a}_K^i]}$ and $u(\boldsymbol{a}_K^j)$, $u(\boldsymbol{a}_K^i)$.

HIDDEN HINT 1 for (10-1.g) → 10-1-7-0:sp5h1.pdf

SOLUTION for (10-1.g) → 10-1-7-1:sigmap5.pdf ▲

(10-1.h) ⚁ (30 min.) [depends on Sub-problem (10-1.g)]

Under the fundamental premise (10.1.17) from Sub-problem (10-1.g) we established a concrete expression for $\boldsymbol{j}\cdot(\boldsymbol{a}_K^i-\boldsymbol{a}_K^j)$ as a function of $u(\boldsymbol{a}_K^j)$, $u(\boldsymbol{a}_K^i)$, and Ψ. Replacing $u\to u_h\in\mathcal{S}_1^0(\mathcal{M})$, this expression is used in (10.1.16) to *define* the exponentially fitted upwind method (letting K run through all cells of the mesh).

Consider the (local) bilinear form of (10.1.16). Fixing a triangle $K\in\mathcal{M}$ and adopting the short notation

$$\beta_{j,i}:=\left(\int_0^1\exp(-\Psi(\gamma_{j,i}(\tau)))\,\mathrm{d}\tau\right)^{-1},\quad \gamma_{j,i}(\tau):=\boldsymbol{a}_K^j+\tau(\boldsymbol{a}_K^i-\boldsymbol{a}_K^j)\,,\quad 0\le\tau\le 1\,, \tag{10.1.19}$$

write the formulas for the entries of the corresponding element matrices (→ [Lecture → Def. 2.7.4.5]) $\mathbf{A}_K^{\exp}\in\mathbb{R}^{3,3}$ of the exponentially fitted upwind method, when standard "tent function", basis functions of $\mathcal{S}_1^0(\mathcal{M})$, are used. The formulas may involve the entries $(\mathbf{A}_K)_{i,j}$ of the element matrix of the bilinear form $(u,v)\mapsto\int_\Omega\mathbf{grad}\,u\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}$ with respect to the finite element space $\mathcal{S}_1^0(\mathcal{M})$ and its standard nodal basis functions.

HIDDEN HINT 1 for (10-1.h) → `10-1-8-0:sp5h1.pdf`

HIDDEN HINT 2 for (10-1.h) → `10-1-8-1:cp5h2.pdf`

SOLUTION for (10-1.h) → `10-1-8-2:sigmap5.pdf` ▲

(10-1.i) ⚂ (20 min.) Now assume that $\Psi\in\mathcal{S}_1^0(\mathcal{M})$, which is realistic, because the potential will usually be available as an approximate finite element solution of another boundary value problem.

Then, based on the result of Sub-problem (10-1.h) write the entries of the element matrix $\mathbf{A}_K^{\exp}$ as functions of the values $\Psi(\mathbf{a}_K^i)$, $i=1,2,3$. To do so, use the Bernoulli function

$$B(\tau):=\begin{cases}\dfrac{\tau}{e^\tau-1} & \text{for } \tau\neq 0\,,\\ 1 & \text{for } \tau=0\,.\end{cases} \tag{10.1.21}$$

HIDDEN HINT 1 for (10-1.i) → `10-1-9-0:sp6h1.pdf`

SOLUTION for (10-1.i) → `10-1-9-1:sp7.pdf` ▲

(10-1.j) ⚂ (30 min.) Write a C++ function

```
double Bernoulli(double tau);
```

that provides a stable, that is, cancellation free, implementation of the Bernoulli function $B(\tau)$ for all arguments $\tau\in\mathbb{R}$.

HIDDEN HINT 1 for (10-1.j) → `10-1-10-0:sp8h1.pdf`

SOLUTION for (10-1.j) → `10-1-10-1:sp8.pdf` ▲

(10-1.k) ⚂ (30 min.) Assuming again $\Psi\in\mathcal{S}_1^0(\mathcal{M})$, in the file `expfittedupwind.cc` implement an LEHRFEM++-based C++ function

```
std::shared_ptr<CodimMeshDataSet<double>>
CompBeta(const lf::uscalfe::FeSpaceLagrangeO1<double> &fe_space,
const Eigen::VectorXd& mu);
```

that computes the quantities $\beta_{j,i}$ defined in (10.1.19) for all the edges of a triangular mesh passed through `fe_space`, when the "tent function" basis expansion coefficients of $\Psi\in\mathcal{S}_1^0(\mathcal{M})$ are supplied through the vector `mu` based on the numbering of basis functions as implicitly

given by the **lf::uscalfe::FeSpaceLagrangeO1** argument. The values $\beta_{j,i}$ are returned in an **lf::mesh::utils::CodimMeshDataSet** object assigning a real number to every edge of the mesh.

SOLUTION for (10-1.k) → 10-1-11-0:sp9.pdf ▲

(10-1.l) ⚄ (30 min.) [depends on Sub-problem (10-1.h) and Sub-problem (10-1.j)]

In the file `expfittedupwind.cc` complete the implementation of the class **ExpFittedEMP** complying with the concept of **ENTITY_MATRIX_PROVIDER** ⇝ (documentation), see [Lecture → Ex. 2.7.4.22]. Its `eval()` method is to return the element matrix $\mathbf{A}_K^{\exp}$ for each cell K of a triangular mesh (quadrilaterals are not supported). The constructor of this class should take two arguments, an **lf::uscalfe::FeSpaceLagrangeO1** object `fe_space` and a basis expansion coefficient vector `mu` which pass the local "tent function" basis expansion coefficients of the potential $\Psi \in \mathcal{S}_1^0(\mathcal{M})$. This information can be used to compute the values $\beta_{j,i}$ introduced in Sub-problem (10-1.h).

SOLUTION for (10-1.l) → 10-1-12-0:sp10.pdf ▲

(10-1.m) ⚄ (30 min.) [depends on Sub-problem (10-1.l)]

In the file `expfittedupwind.h` implement a C++ function

```
template<typename FUNC_F, typename FUNC_G>
Eigen::VectorXd solveDriftDiffusionDirBVP(
const lf::uscalfe::FeSpaceLagrangeO1<double> &fe_space,
const Eigen::VectorXd& mu,
FUNC_F &&func_f,FUNC_G &&func_g);
```

that uses the exponentially fitted upwind method to compute the basis expansion coefficient of an approximate solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ of a pure Dirichlet boundary value problem for (10.1.6).

The argument `gfe_space` contains information about the mesh and the numbering of basis functions, `mu` the basis expansion coefficients of $\Psi \in \mathcal{S}_1^0(\mathcal{M})$, and `func_f` and `func_f` are functors of type `std::function<double(Eigen::Vector2d)>` providing the source function f and the Dirichlet data g.

Of course, assembly of the linear system of equations resulting from the exponentially fitted upwind method should rely on the class **ExpFittedEMP** from Sub-problem (10-1.l).

HIDDEN HINT 1 for (10-1.m) → 10-1-13-0:h111.pdf

SOLUTION for (10-1.m) → 10-1-13-1:sp11.pdf ▲

(10-1.n) ⚄ (25 min.) [depends on Sub-problem (10-1.m)]

Given a potential Ψ and compatible Dirichlet data, the pure Dirichlet problem for (10.1.6) with $f \equiv 0$ has the solution $u(\boldsymbol{x}) = \exp(\Psi(\boldsymbol{x}))$, provided that the Dirichlet data are chosen as $g := \exp \circ \Psi|_{\Gamma_D}$.

On the unit square $\Omega :=]0,1[^2$ use this solution for $\Psi(\boldsymbol{x}) = \boldsymbol{q} \cdot \boldsymbol{x}$, $\boldsymbol{q} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, to test your implementation.

To that end compute the $L^2(\Omega)$-norm of the discretization error on a sequence of regularly refined meshes that are generated by the code. Rely on the function `solveDriftDiffusionDirBVP()`, call it from `main()` (file expfittedupwind_main.cc), compute the error norms as in [Lecture → Code 3.2.3.5] and print them to the console.

SOLUTION for (10-1.n) → 10-1-14-0:sp12.pdf ▲

References

[MZ88] P. Markovich and M. Zlamal. "Inverse average type discretizations of second order ellitpic problems". In: *Math. Comp.* 51 (1988), pp. 431–449 (cit. on p. 366).

[Mar86] P.A. Markowich. *The Stationary Semiconductor Device Equations*. Wien, New York: Springer-Verlag, 1986 (cit. on p. 366).

[XZ99] J. Xu and L. Zikatanov. "A monotone finite element scheme for convection diffusion equations". In: *Math. Comp.* 68.228 (May 1999), pp. 1429–1446 (cit. on p. 366).

End Problem 10-1 , 300 min.

Problem 10-2: Upwind Quadrature Method

In [Lecture → Section 10.2.2.1] we have seen that the strategy of *upwind quadrature* makes it possible to generalize the idea of 1D upwinding, that is, the use of backward difference quotients to discretize convective terms, to higher dimensions.

This problem requires familiarity with [Lecture → Section 10.2.2.1] and its contents should be refreshed beforehand. C++ implementation based on LEHRFEM++ is requested.
▷ problem name/problem code folder: UpwindQuadrature

In this problem we discuss the implementation of upwind quadrature for the stationary convection-diffusion Dirichlet boundary value problem

$$-\epsilon\Delta u(\boldsymbol{x}) + \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\, u(\boldsymbol{x}) = 0 \quad \text{in} \quad \Omega \quad , \quad u = g \quad \text{on} \quad \partial\Omega \, , \tag{10.2.1}$$

$g \in C^0(\partial\Omega)$. Concretely, Ω is unit square $]0,1[^2$, and

$$g(\boldsymbol{x}) := \begin{cases} \frac{1}{2} - \left|x_1 - \frac{1}{2}\right| & \text{for} \quad x_2 = 0, \\ 0 & \text{otherwise,} \end{cases} \quad , \qquad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} . \tag{10.2.2}$$
$$\mathbf{v}(\mathbf{x}) := \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix} ,$$

This boundary value problem models the temperature distribution $u = u(\boldsymbol{x})$ in a fluid streaming with stationary velocity $\mathbf{v}$.

(10-2.a) ⚀ (10 min.) Show that $\mathbf{v}$ describes the flow of an incompressible fluid according to [Lecture → Def. 10.1.3.1].

HIDDEN HINT 1 for (10-2.a) → 10-2-1-0:2D-upwind-quadr11.pdf

SOLUTION for (10-2.a) → 10-2-1-1:2D-upwind-quadr1.pdf ▲

(10-2.b) ⚁ (30 min.) Give an explicit formula for the flow map induced by the velocity field $\mathbf{v}$ from, (10.2.2).

HIDDEN HINT 1 for (10-2.b) → 10-2-2-0:2D-upwind-quadr21.pdf

SOLUTION for (10-2.b) → 10-2-2-1:2D-upwind-quadr2.pdf ▲

(10-2.c) ⚂ (30 min.) [depends on Sub-problem (10-2.b)]

We consider (10.2.1) with data (10.2.2). As $\epsilon \to 0$, the convection term will dominate the diffusion term (singular perturbation, [Lecture → Notion 10.2.1.9]).

What do you expect will happen in the limit? Will there be a boundary layer somewhere in the domain? Where?

HIDDEN HINT 1 for (10-2.c) → 10-2-3-0:2D-upwind-quadr31.pdf

SOLUTION for (10-2.c) → 10-2-3-1:2D-upwind-quadr3.pdf ▲

(10-2.d) ⚁ (15 min.) The executable generated from `upwindquadrature_main.cc` solves (10.2.1) with $\epsilon = 10^{-4}$ using the standard Galerkin approach, see [Lecture → § 10.2.0.2], and linear Lagrangian finite elements. It writes the solution data to the file `upwind_quadrature_solution_unstable.vtk` , from which we can create a plot using PARAVIEW. Examine the plot and describe your observations.

SOLUTION for (10-2.d) → 10-2-4-0:2D-upwind-quadr5.pdf ▲

We will now attempt to cure the instability of the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin FEM for small ϵ ($\mathcal{M}$ a triangular mesh of Ω). To this end, we will implement upwind quadrature as introduced in [Lecture → Section 10.2.2.1], and also known as *Tabata scheme*, for the convective term, whereas for the diffusive part we just employ the standard Galerkin method.

For the treatment of the convective term the upwind quadrature scheme employs a special non-local trapezoidal rule: The convective contribution to the element matrix for the triangle K is obtained using the quadrature formula

$$\int_K (\mathbf{v}(\mathbf{x}) \cdot \mathbf{grad}\, u_h(\mathbf{x}))w_h(\mathbf{x})\,\mathrm{d}\mathbf{x} \approx \frac{|K|}{3}\sum_{l=1}^{3}(\mathbf{v}(\mathbf{a}^l) \cdot (\mathbf{grad}\, u_h)^+(\mathbf{a}^l))w_h(\mathbf{a}^l)\,, \tag{10.2.4}$$

where $\mathbf{a}^l$ are the vertices of the triangle K, and where $(\mathbf{grad}\, u_h)^+(\mathbf{a}^l)$ is *not* the gradient of $u_h \in \mathcal{S}_1^0(\mathcal{M})$ in the triangle K, but rather the gradient of u_h in the triangle *upstream* from K (the one located in the direction of $-\mathbf{v}(\mathbf{a}^l)$, see [Lecture → Fig. 432]). Note that this is unambigously defined, as the gradient of a piecewise linear function is constant on each triangle. More details are given in [Lecture → Section 10.2.2.1].

(10-2.e) ⊡ (15 min.) Study [Lecture → § 10.2.2.16] carefully again. ▲

(10-2.f) ⚂ (20 min.) We set $N := \dim \mathcal{S}_1^0(\mathcal{M})$ and use the nodal tent function basis $\mathfrak{B} = \{b_h^1, \ldots, b_h^N\}$ of $\mathcal{S}_1^0(\mathcal{M})$ as introduced in [Lecture → Section 2.4.3] throughout, where b_h^i is supposed to be associated with the mesh node $\boldsymbol{a}^i$. Write $\mathbf{B} \in \mathbb{R}^{N,N}$ for the contribution of the convective term to the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix for the upwind quadrature scheme and show that

$$(\mathbf{B})_{j,i} = m(\mathbf{a}^j)\Big(\mathbf{v}(\mathbf{a}^j) \cdot (\mathbf{grad}\, b_h^i)^+(\mathbf{a}^j)\Big), \tag{10.2.5}$$

where $m(\mathbf{a}^j)$ is defined as the *mass* of vertex j, i.e.

$$m(\mathbf{a}^j) = \frac{1}{3}\sum_{\mathbf{a}^j \in K} |K|\,, \tag{10.2.6}$$

where the sum runs over all triangles containing $\mathbf{a}^j$, see also [Lecture → (10.2.2.18)]

SOLUTION for (10-2.f) → 10-2-6-0:2D-upwind-quadr6.pdf ▲

The upwind quadrature method does not completely fit the concept of local assembly as introduced in [Lecture → Section 2.7.4] because $(\mathbf{B})_{j,i}$ cannot be written as the sum of *purely local* contributions of individual cells. Nevertheless, after the vertex masses have been precomputed, there is a smart way to use local assembly to initialize the convective part $\mathbf{B}$ of the system matrix. To that end we introduce the following C++ class

```
template <typename FUNCTOR>
class UpwindConvectionElementMatrixProvider {
  public:
  explicit UpwindConvectionElementMatrixProvider(
  FUNCTOR v, lf::mesh::utils::CodimMeshDataSet<double> masses)
  : v_(v), masses_(masses) {}
  Eigen::Matrix3d Eval(const lf::mesh::Entity &entity);
  bool isActive(const lf::mesh::Entity & /*entity*/) const { return
     true; }
  private:
  FUNCTOR v_;  // velocity field
  // masses of all vertices of the mesh.
  lf::mesh::utils::CodimMeshDataSet<double> masses_;
};
```

It matches the concept of **ENTITY_MATRIX_PROVIDER** ↪ (documentation), see [Lecture → Ex. 2.7.4.22]. The argument `masses` of the constructor passes the "mass" $m(\boldsymbol{a}^j)$ of a mesh node according to (10.2.6) for every node. The other argument `v` supplies a functor object for the velocity field $\mathbf{v} : \Omega \to \mathbb{R}^2$.

(10-2.g) ⚄ (60 min.) Implement the `eval()` method of the C++ class **UpwindConvectionLocalAssembler**

```
Eigen::Matrix3d Eval(const lf::mesh::Entity &entity)
```

that computes the "local" contribution of a triangle K which is passed through the `entity` argument.

HIDDEN HINT 1 for (10-2.g) → `10-2-7-0:2D-upwind-quadr71.pdf`

SOLUTION for (10-2.g) → `10-2-7-1:2D-upwind-quadr7.pdf` ▲

(10-2.h) (60 min.) [depends on Sub-problem (10-2.g)]

Complete the `main()` function in the file `upwindquadrature_main` so that it saves the solution computed with the upwind quadrature scheme to the file `upwind_quadrature_solution_stable.vtk`.

HIDDEN HINT 1 for (10-2.h) → `10-2-8-0:uq7h1.pdf`

HIDDEN HINT 2 for (10-2.h) → `10-2-8-1:uq7h2.pdf`

SOLUTION for (10-2.h) → `10-2-8-2:2D-upwind-quadr9.pdf` ▲

End Problem 10-2, 240 min.

Problem 10-3: Upwind Finite Volume Method

In this problem we will study another stable method for approximately solving the convection-diffusion equation with a general velocity field, not necessarily divergence-free. We employ a finite-volume method as the one introduced in [Lecture → Section 4.2] for scalar 2nd-order pure diffusion problems. We will see that the design principle of finite-volume methods is well suited for dealing with convection-diffusion boundary-value problems.

This problem assumes knowledge about finite-volume methods as covered in [Lecture → Section 4.2]. You should also be familiar with the contents of [Lecture → Section 10.1.2], where we derived convection-diffusion PDEs. This problem requests implementation in LEHRFEM++.
▷ problem name/problem code folder: UpwindFiniteVolume

On a bounded computational domain $\Omega \subset \mathbb{R}^2$ and for a given velocity field $\mathbf{v} \in (C^0(\overline{\Omega}))^2$, non necessarily divergence-free, we consider the convection-diffusion Dirichlet boundary value problem

$$\operatorname{div}(\epsilon \operatorname{\mathbf{grad}} u + \mathbf{v}u) = f \quad , \quad u = g \quad \text{on} \quad \partial\Omega \, , \tag{10.3.1}$$

where $f \in C^0(\overline{\Omega})$ and $g \in C^0(\partial\Omega)$ are given functions, possibly only available in procedural form [Lecture → Rem. 2.1.2.5].

The discretization scheme adheres to the spirit of finite volume methods, see [Lecture → Section 4.2], and proceeds as follows: given a control volume C_i (→ [Lecture → Section 4.2.1], [Lecture → (4.2.1.7)]), we state the balance law encoded in (10.3.1) for it:

$$\int_{\partial C_i} \mathbf{j}(u_N) \cdot \boldsymbol{n} \, \mathrm{d}S \approx \sum_{k \in U_i} \Psi_{ik}(\mu_i, \mu_k) = \int_{C_i} f \, \mathrm{d}\boldsymbol{x} \, , \tag{10.3.2}$$

where U_i is the set of control volumes adjacent to C_i, and $\Psi_{ik}(\mu_i, \mu_k)$ is the **numerical flux** from C_i to C_k. Flux balance requires

$$\Psi_{ik}(\mu_i, \mu_k) = -\Psi_{ki}(\mu_k, \mu_i) \quad \forall \mu_i, \mu_k \in \mathbb{R} \, . \tag{10.3.3}$$

The coefficient μ_i agrees with the value $u_N(\boldsymbol{p}_i)$ of the approximate solution $u_N \in \mathcal{S}_1^0(\mathcal{M})$ in the vertex $\boldsymbol{p}_i$ of the primal mesh. In this problem triangular (primal) meshes and **orthogonal dual meshes** (Voronoi cells) will be used, see [Lecture → § 4.2.2.1].

Finding a suitable expression for the numerical flux is key to constructing a good finite volume method. Here the idea is to "project" the PDE onto the line connecting the two vertices at the center of adjacent control volumes. This projection yields a scalar linear ODE, which can be solved analytically. From this analytic solution we recover an approximate flux.

In detail, we start from the expression

$$\Psi_{ik}(\mu_i, \mu_k) := |\boldsymbol{f}_{ik}| J_{ik}(\mu_i, \mu_k) \, , \tag{10.3.4}$$

where $|\boldsymbol{f}_{ik}|$ is the length of the common interface Γ_{ik} between the dual cells C_i and C_k, and

$$J_{ik}(\mu_i, \mu_k) := \epsilon w'(0) + \widehat{v}(\boldsymbol{m}_{ik}, i, k) w(0), \tag{10.3.5}$$

where

- $\widehat{v}(\boldsymbol{x}, i, k)$ is the length of the projection of $\boldsymbol{v}(\boldsymbol{x})$ onto the vector $\boldsymbol{p}_k - \boldsymbol{p}_i$, i.e. it is the component of the velocity $\mathbf{v}$ directed in the $i \to k$ direction (note that it is a scalar),

$$\widehat{v}(\boldsymbol{m}_{ik}, i, k) := \mathbf{v}(\boldsymbol{m}_{ik}) \cdot \frac{\boldsymbol{p}_k - \boldsymbol{p}_i}{\|\boldsymbol{p}_k - \boldsymbol{p}_i\|} \, , \quad \boldsymbol{m}_{ik} := \tfrac{1}{2}(\boldsymbol{p}_i + \boldsymbol{p}_k) \, , \quad i \neq k \, , \tag{10.3.6}$$

- and $w = w(\xi) : [0, d_{ik}] \mapsto \mathbb{R}$ solves a linear scalar 2-point boundary value problem "along the vector $\boldsymbol{p}_k - \boldsymbol{p}_i$",

$$\left(\epsilon w'(\xi) + \widehat{v}(\boldsymbol{m}_{ik}, i, k) w(\xi)\right)' = 0, \tag{10.3.7}$$

with Dirichlet boundary conditions

$$w(0) = \mu_i \in \mathbb{R} \quad , \quad w(d_{ik}) = \mu_k \in \mathbb{R} \, , \tag{10.3.8}$$

where $d_{ik} := \|\boldsymbol{p}_k - \boldsymbol{p}_i\|$ is the length of the edge $[\boldsymbol{p}_i, \boldsymbol{p}_k]$.

(10-3.a) ⚀ (15 min.) Treating $\widehat{v} = \widehat{v}(\boldsymbol{m}_{ik}, i, k)$, μ_i and μ_k as parameters $\in \mathbb{R}$, show that the analytical solution to (10.3.7) is:

- if $\widehat{v} \neq 0$:

$$w(\xi) = \frac{1}{\widehat{v}}(C_1 - C_2 e^{-\frac{\widehat{v}}{\epsilon}\xi}) \, , \quad \xi \in [0, d_{ik}] \, , \tag{10.3.9}$$

with $C_1 = \widehat{v}\mu_i + C_2$ and $C_2 = \frac{\widehat{v}(\mu_k - \mu_i)}{1 - e^{-\frac{\widehat{v}}{\epsilon} d_{ik}}}$.

- if $\widehat{v} = 0$:

$$w(\xi) = \frac{\mu_k - \mu_i}{d_{ik}} \xi + \mu_i \, , \quad \xi \in [0, d_{ik}] \, . \tag{10.3.10}$$

SOLUTION for (10-3.a) → 10-3-1-0:upwind-fin-vol1.pdf ▲

(10-3.b) ⚁ (15 min.) Based on (10.3.9)/(10.3.10) give a formula for $J(\mu_i, \mu_k)$ as defined in (10.3.5).

SOLUTION for (10-3.b) → 10-3-2-0:upwind-fin-vol2.pdf ▲

(10-3.c) ⚂ (20 min.) [depends on Sub-problem (10-3.b)]

In the file upwindfinitevolume.cc complete the body of the function

```
double computeUpwindFlux(double mui, double muk, double vhat,
   double dik, double epsilon);
```

that returns the value $J_{ik}(\mu_i, \mu_k)$ when supplied with μ_i, μ_k, $\widehat{v}$, d_{ik}, and ϵ (in that order).

Make sure that your code provides a *stable* implementation.

HIDDEN HINT 1 for (10-3.c) → 10-3-3-0:fvuwh2a.pdf

SOLUTION for (10-3.c) → 10-3-3-1:ufw2as.pdf ▲

(10-3.d) We will use LEHRFEM++ to handle a triangular finite element mesh for this problem, and then build a Voronoi dual mesh for finite volumes, as described in [Lecture → Section 4.2.2]. Implement a C++ function

```
Eigen::Vector2d computeCircumcenters(
  const Eigen::Vector2d a1,
  const Eigen::Vector2d a2,
  const Eigen::Vector2d a3);
```

that takes the corners of a triangle of mesh whose vertices have the coordinates a1, a2, and a3, and that computes the circumcenter of that element, *cf.* [Lecture → Rem. 4.2.2.3].

Throw an exception of type **std::runtime_error**, in case you detect an obtuse triangle for which the circumcenter falls outside the triangle *cf.* [Lecture → Rem. 4.2.2.3]. The position of the circumcenter relative to the triangle can be checked with barycentric coordinates, see [Lecture → Code 2.4.5.11].

HIDDEN HINT 1 for (10-3.d) → `10-3-4-0:upwind-fin-vol31.pdf`

SOLUTION for (10-3.d) → `10-3-4-1:upwind-fin-vol3.pdf` ▲

(10-3.e) ⚄ (30 min.) Let $\mathbf{A} \in \mathbb{R}^{N,N}$ denote the system matrix of the linear system of equations arising from the upwind finite-volume discretization of (10.3.1) using a triangular mesh $\mathcal{M}$ and before any elimination of d.o.f.s on $\partial\Omega$. Thus N will be the number of nodes of $\mathcal{M}$.

In [Lecture → Section 2.7.4.3], [Lecture → Code 2.7.4.20] we learned about cell-oriented assembly, an algorithm that builds the system matrix of a linear system of equations arising from a Galerkin discretization by summing up local contributions in the form of element matrices. If

$$J_{ik}(\mu_i, \mu_k) = \alpha_{ik}^i \mu_i + \alpha_{ik}^k \mu_k \quad \forall \mu_i, \mu_k \in \mathbb{R} \quad \text{for some} \quad \alpha_{ik}^i, \alpha_{ik}^k \in \mathbb{R}\,, \tag{10.3.23}$$

express the entries of the 3×3 element matrix $\mathbf{A}_K \in \mathbb{R}^{3,3}$ of a triangle K with vertices $\boldsymbol{p}_i$, $\boldsymbol{p}_k$, $\boldsymbol{p}_j$ (in this order) in terms of

- the weights α_{ik}^i, α_{ik}^k, α_{ij}^i, α_{ij}^j, α_{jk}^j, α_{jk}^k,
- and of the circumcenter $\boldsymbol{c}_K$ of K and of the midpoints $\boldsymbol{m}_{ik}$, $\boldsymbol{m}_{ij}$, $\boldsymbol{m}_{kj}$ of its edges.

This means when we use these element matrices $\mathbf{A}_K$ in the algorithm of [Lecture → Code 2.7.4.20], then we obtain $\mathbf{A}$ as the result.

HIDDEN HINT 1 for (10-3.e) → `10-3-5-0:ufvh12b.pdf`

HIDDEN HINT 2 for (10-3.e) → `10-3-5-1:fvuhint2b.pdf`

SOLUTION for (10-3.e) → `10-3-5-2:fvuws2b.pdf` ▲

(10-3.f) ⚀ (15 min.) [depends on Sub-problem (10-3.b)]

Since $(\mu_i, \mu_k) \mapsto J_{ik}(\mu_i, \mu_k)$ is an affine linear function, it will have the representation (10.3.23). Compute the coefficients α_{ik}^i and α_{ik}^k for the edge $[\boldsymbol{p}_i, \boldsymbol{p}_k]$, $i \neq k$.

SOLUTION for (10-3.f) → `10-3-6-0:s2cufv.pdf` ▲

(10-3.g) (30 min.) [depends on Sub-problem (10-3.d), Sub-problem (10-3.e), Sub-problem (10-3.f)]

In the file `upwindfinitevolume.h` implement the `Eval()` method of the class **ElementMatrixProvider** compatible with the LEHRFEM++ concept of **ENTITY_MATRIX_PROVIDER** [Lecture → Ex. 2.7.4.22]. That method is to compute and return a 3×3 "element" matrix $\mathbf{A}_K$ for triangle K. Use the function `computeCircumcenters()` of Sub-problem (10-3.d) to get the circumcenter of the current element. The velocity field $\mathbf{v}$ and the coefficient ϵ have to be stored as class data members.

Make sure that your implementation avoids cancellation!

SOLUTION for (10-3.g) → `10-3-7-0:upwind-fin-vol4.pdf` ▲

(10-3.h) Implement the `Eval()` method of the class **ElementVectorProvider**, which should be compatible with the concept of an **ENTITY_VECTOR_PROVIDER** [Lecture → Ex. 2.7.4.27]. It is to compute and return a 3×1 "element" vector for the triangle passed as argument.

Following [Lecture → (4.2.1.8)] use the one-point numerical quadrature

$$\int_{C_i} f \,\mathrm{d}\boldsymbol{x} \approx |C_i| f(\boldsymbol{p}_i)\,,$$

and the function `computeCircumcenters()` of Sub-problem (10-3.d) to get the circumcenter of the current cell.

HIDDEN HINT 1 for (10-3.h) → 10-3-8-0:upwind-fin-vol51.pdf

SOLUTION for (10-3.h) → 10-3-8-1:upwind-fin-vol5.pdf ▲

(10-3.i) To test our method we will use a test problem where $\boldsymbol{v}(\boldsymbol{x}) = (1, 1/2 - x_2)$ and $u(\boldsymbol{x}) = \sin(4\pi x_2)e^{-x_1}$. Compute the corresponding f. For the Dirichlet data g, simply use the restriction of u to the boundary.

SOLUTION for (10-3.i) → 10-3-9-0:upwind-fin-vol6.pdf ▲

(10-3.j) Complete the C++ `main` file that computes the finite volume solution on a sequence of refined meshes. Mimic the procedure used in other subproblems.

SOLUTION for (10-3.j) → 10-3-10-0:upwind-fin-vol6a.pdf ▲

End Problem 10-3 , 125 min.

Problem 10-4: Method of Characteristics and Semi-Lagrangian Discretization

This problem studies the implementation of the basic Semi-Lagrangian method of [Lecture → Section 10.3] for a convection-diffusion problem.

This problem employs structured meshes and asks for implementation in C++ based on EIGEN.
▷ problem name/problem code folder: SemiLagrangian

On the square $\Omega :=]0,1[^2$ and for the final time $T > 0$ we consider the transient convection-diffusion initial boundary value problem (IBVP) for the scalar unknown $u = u(\boldsymbol{x},t)$

$$\begin{aligned} \frac{\partial u}{\partial t} - \epsilon \Delta u + \mathbf{v} \cdot \mathbf{grad}\, u = 0 \quad &\text{on} \quad \Omega \times]0,T[\,, \\ u = 0 \quad &\text{on} \quad \partial\Omega \times]0,T[\,, \\ u(0) = u_0 \quad &\text{in} \quad \Omega\,, \end{aligned} \tag{10.4.1}$$

with $\epsilon > 0$, $u_0 \in H^1_0(\Omega)$ given, and $\mathbf{v} : \overline{\Omega} \to \mathbb{R}^2$ a continuous velocity field. Both Δ and $\mathbf{grad}$ act only on the spatial variable $\boldsymbol{x}$.

(10-4.a) ⚀ (15 min.) Give the spatial variational formulation of (10.4.1).

HIDDEN HINT 1 for (10-4.a) → 10-4-1-0:sl1h1.pdf

SOLUTION for (10-4.a) → 10-4-1-1:sl1.pdf ▲

(10-4.b) ⚄ (30 min.) [depends on Sub-problem (10-4.a)]

Under which assumptions on $\operatorname{div}\mathbf{v}$ can we predict an exponential decay of $\|u(\cdot,t)\|_{L^2(\Omega)}$ as $t \to \infty$? Justify your answer.

HIDDEN HINT 1 for (10-4.b) → 10-4-2-0:sl2h1.pdf

SOLUTION for (10-4.b) → 10-4-2-1:sl2.pdf ▲

Next, we consider the singular perturbation limit $\epsilon = 0$, for which (10.4.1) becomes a pure transport problem. In this case the spatial boundary conditions $u = 0$ can be imposed only on the **inflow boundary**

$$\Gamma_{\text{in}} := \{\boldsymbol{x} \in \partial\Omega : \quad \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\}\,,$$

$\boldsymbol{n}$ the exterior unit normal on $\partial\Omega$. Then the solution of (10.4.1) is given by the formula, see [Lecture → (10.3.2.7)],

$$u(\boldsymbol{x},t) = \begin{cases} u_0(\boldsymbol{y}(\boldsymbol{x},t;0)) & \text{if} \quad \boldsymbol{y}(\boldsymbol{x},t;s) \in \Omega \quad \forall\, 0 \le s \le t\,, \\ 0 & \text{otherwise,} \end{cases} \qquad \boldsymbol{x} \in \Omega\,,\ t > 0\,, \tag{10.4.5}$$

where we have written $s \to \boldsymbol{y}(\boldsymbol{x},t;s)$ for the solution of the initial-value problem

$$\frac{d}{ds}\boldsymbol{y}(\boldsymbol{x},t;s) = \mathbf{v}(\boldsymbol{y}(\boldsymbol{x},t;s)) \quad , \quad \boldsymbol{y}(\boldsymbol{x},t;t) = \boldsymbol{x}\,. \tag{10.4.6}$$

(10-4.c) ⚂ (45 min.) In the file `semilagrangian.h` write a C++ function

```
template<typename FUNCTOR_V, typename FUNCTOR_U0>
double solveTransport(const Eigen::Vector2d& x,
int K, double t,
FUNCTOR_V&& v, FUNCTOR_U0&& u0)
```

that computes $u(\boldsymbol{x},t)$, $\boldsymbol{x}\in]0,1[^2$, $t>0$, using the formula (10.4.5). For the approximate computation of $y(\boldsymbol{x},t;s)$, use (at most) K steps of **Heun's method**, a 2-stage explicit Runge-Kutta single-step method described by the Butcher scheme

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{2}{3} & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array} . \tag{10.4.7}$$

The functor objects passed in `v` and `u0` have the signature **std**::function<Eigen::Vector2d(Eigen::Vector2d)> and **std**::function<**double**(Eigen::Vector2d)>, respectively.

HIDDEN HINT 1 for (10-4.c) → 10-4-3-0:sl3h1.pdf

SOLUTION for (10-4.c) → 10-4-3-1:sl3.pdf ▲

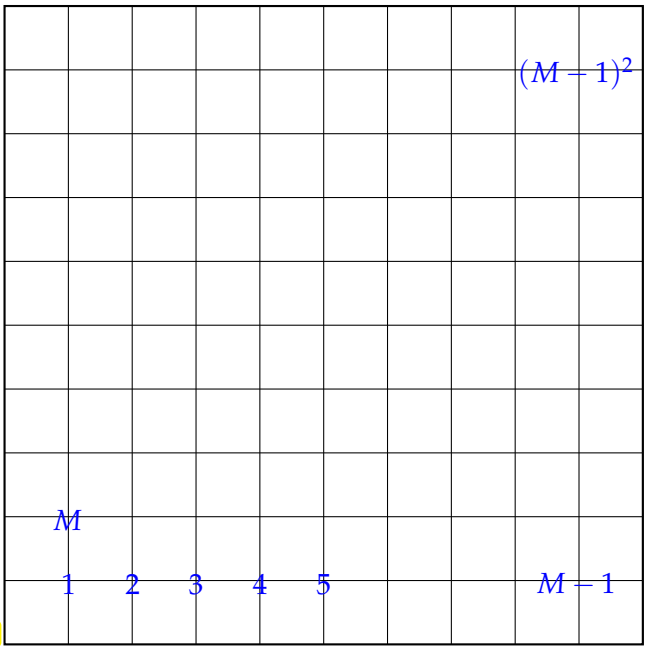


Fig. 156

◁ We equip Ω with an equispaced quadrilateral tensor product mesh $\mathcal{M}_M$ with $M\in\mathbb{N}$ cells in each coordinate direction.

Its cells are all squares with edge length $h:=M^{-1}$.

Row-first lexikographic ordering of the interior gridpoints from bottom-left to top-right is assumed.

We write $\mathcal{S}^0_{1,0}(\mathcal{M}_M)\subset H^1_0(\Omega)$ for the space of piecewise bilinear finite element functions on $\mathcal{M}_M$, see [Lecture → Section 2.6.2]. The nodal basis functions are numbered along with the interior gridpoints.

According to [Lecture → § 10.3.4.13] a single timestep of size $\tau>0$ of the **semi-Lagrangian finite element scheme** for the discretization of (10.4.1) reads: find $u_h^{(j)}\in\mathcal{S}^0_{1,0}(\mathcal{M}_M)$

$$\int_\Omega \left[u_h^{(j)}(\boldsymbol{x}) - \mathsf{I}_1(u_h^{(j-1)}(\cdot-\tau\mathbf{v}(\cdot)))(\boldsymbol{x})\right] w_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \tau\epsilon\int_\Omega \mathbf{grad}\, u_h^{(j)}\cdot\mathbf{grad}\, w_h\,\mathrm{d}\boldsymbol{x} = 0 \tag{10.4.9}$$

for all $w_h\in\mathcal{S}^0_{1,0}(\mathcal{M}_M)$. Here $\mathsf{I}_1:C^0(\overline{\Omega})\to\mathcal{S}^0_{1,0}(\mathcal{M}_M)$ is the nodal interpolation operator onto the finite element space, defined by

$$(\mathsf{I}_1 f)(\boldsymbol{p}) = f(\boldsymbol{p}) \quad \forall \text{ interior vertices } \boldsymbol{p} \text{ of the mesh } \mathcal{M}_M .$$

For the approximate evaluation of *all* integrals over Ω, we employ the composite trapezoidal rule [Lecture → (2.4.6.10)] on $\mathcal{M}_M$, that is, vertex-based numerical quadrature: for $f\in C^0_0(\overline{\Omega})$

$$\int_\Omega f\,\mathrm{d}\boldsymbol{x} \approx h^2\sum_{i=1}^{M-1}\sum_{j=1}^{M-1} f\left(\begin{bmatrix} ih \\ jh\end{bmatrix}\right) \tag{10.4.10}$$

(10-4.d) ⚂ (20 min.) Writing $\vec{\boldsymbol{\mu}}^{(j)}$ for the vector of basis expansion coefficients of $u_h^{(j)}$ with respect to the standard nodal "tent function basis" [Lecture → Ex. 2.6.2.1] of $\mathcal{S}^0_{1,0}(\mathcal{M}_M)$ ordered lexikographically,

we can recast (10.4.9) in algebraic form

$$(\mathbf{M}+\tau\mathbf{A})\vec{\boldsymbol{\mu}}^{(j)} = \mathbf{f}(\vec{\boldsymbol{\mu}}^{(j-1)})\,, \quad \mathbf{M},\mathbf{A}\in\mathbb{R}^{N,N},\ N\in\dim\mathcal{S}^0_{1,0}(\mathcal{M}_M)\,. \tag{10.4.11}$$

Give concrete formulas for the entries of the matrix $\mathbf{M}$ using the 2D trapezoidal rule (10.4.10). What is the number of its nonzero entries in terms of M?

SOLUTION for (10-4.d) → 10-4-4-0:sl4.pdf ▲

(10-4.e) ⚂ (30 min.) For the square cell $K=[0,h]^2$, $h>0$, compute the *element matrix* for the bilinear form $(u,v)\mapsto\int_\Omega \mathbf{grad}\,u\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}$ for bilinear Lagrangian finite elements.

As done above, use the composite trapezoidal rule to evaluate all the integrals.

HIDDEN HINT 1 for (10-4.e) → 10-4-5-0:sl4xh1.pdf

HIDDEN HINT 2 for (10-4.e) → 10-4-5-1:sl4xh2.pdf

SOLUTION for (10-4.e) → 10-4-5-2:sl4x.pdf ▲

(10-4.f) ⚂ (30 min.) Complete the code of the function (in the file `semilagrangian.cc`)

```
double evalFEfunction(const coord_t&  x,
const vector_t& u)
```

that returns the value $u_h(\boldsymbol{x})$ for a function $u_h\in\mathcal{S}^0_{1,0}(\mathcal{M}_M)$ and a point $\boldsymbol{x}\in\Omega$. The point coordinates are passed in `x`, the FE basis expansion coefficients of u_h in `u`.

SOLUTION for (10-4.f) → 10-4-6-0:sls4z.pdf ▲

(10-4.g) ⚄ (60 min.) [depends on Sub-problem (10-4.f)]

In the file `semilagrangian.h` complete the implementation of the C++ function

```
template<typename FUNCTOR>
Eigen::VectorXd semiLagrangeSource(const Eigen::VectorXd& u_old,
double tau, FUNCTOR&& velocity)
```

that realizes the function $\mathbf{f}:\mathbb{R}^N\to\mathbb{R}^N$ on the right-hand side of (10.4.11). The vector `u_old` contains the FE coefficients of $u_h^{(j-1)}$ on an equispaced grid of *internal* points on the square $]0,1[^2$. The velocity is passed by means of a functor providing point evaluation through

```
Eigen::Vector2d operator() (const Eigen::Vector2d& x) const;
```

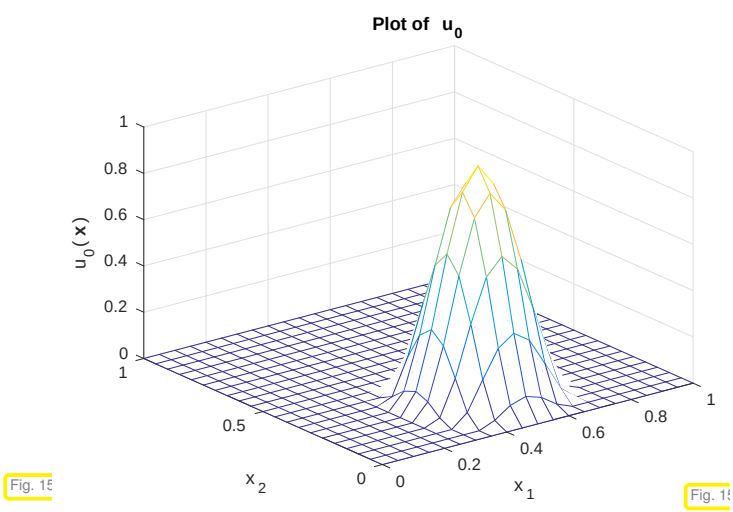
The argument `tau` specifies the step size $\tau>0$. The function `evalFEfunction()` from Sub-problem (10-4.f) comes handy.

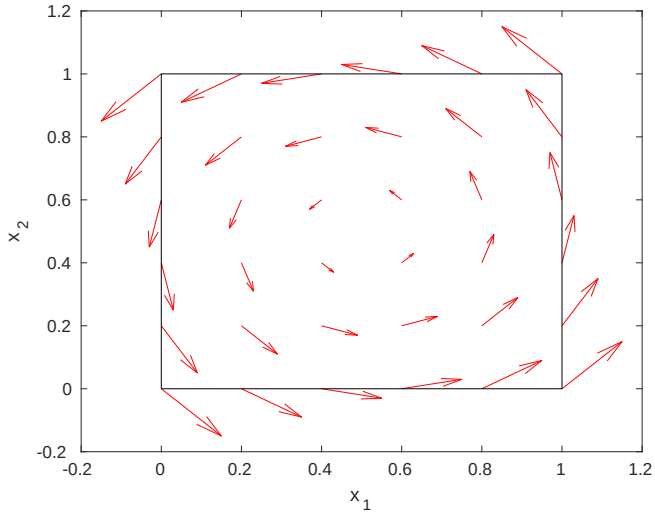
SOLUTION for (10-4.g) → 10-4-7-0:sl5.pdf ▲

(10-4.h) ⚄ (45 min.) [depends on Sub-problem (10-4.d), Sub-problem (10-4.g)]

Let the transport velocity $\mathbf{v}$ and the initial value u_0 be given by

$$\mathbf{v}(\boldsymbol{x})=\begin{bmatrix}-\left(x_2-\frac{1}{2}\right)\\ x_1-\frac{1}{2}\end{bmatrix}\,,\quad t\in\mathbb{R}\quad,\quad u_0(\boldsymbol{x})=\begin{cases}\cos^2\left(2\pi\left\|\boldsymbol{x}-\begin{bmatrix}\frac{1}{4}\\ \frac{1}{2}\end{bmatrix}\right\|\right) & \text{for } \left\|\boldsymbol{x}-\begin{bmatrix}\frac{1}{4}\\ \frac{1}{2}\end{bmatrix}\right\|<\frac{1}{4}\,,\\ 0 & \text{elsewhere in } \Omega\,.\end{cases} \tag{10.4.16}$$





Relying on EIGEN, in the file `semilagrangian.h` write a C++ function

```
template <typename RECORDER = std::function<void(const
   Eigen::VectorXd &)>>
Eigen::VectorXd semiLagrangePureTransport(int M, int K, double T,
  RECORDER &&rec = [] (const Eigen::VectorXd &) -> void {});
```

that employs the semi-Lagrangian method according to (10.4.9) plus the composite trapezoidal rule (10.4.10) to solve the pure transport problem (10.4.1) for $\epsilon = 0$. You may use the function `semiLagrangeSource` in your code.

The function arguments have the following meaning:

- `M` is the number of mesh cells in one direction (the mesh is as in Fig. 156),
- `K` is the total number of equidistant timesteps, and
- `T` is the final time: the simulation is over the period $[0, T]$.
- `rec` passes an object to which the basis expansion coefficient vector $\vec{\mu}^{(j)}$ should be passed for every timestep $j = 10, \ldots, K$.

Obtain the initial approximation $u_h^{(0)}$ by sampling the values of u_0 at all vertices of the mesh $\mathcal{M}_M$.

HIDDEN HINT 1 for (10-4.h) → `10-4-8-0:sp5slh.pdf`

SOLUTION for (10-4.h) → `10-4-8-1:sl6.pdf` ▲

(10-4.i) ⚄ (45 min.) [depends on Sub-problem (10-4.h)]

In the file `semilagrangian.cc` and based on `semiLagrangePureTransport()` from Sub-problem (10-4.h) implement a function

```
void SemiLagrangeVis(int M, int K, double T);
```

that creates 3D plots of the graphs of 10 snapshots (equidistant in time) of the numerical solution of the pure transport problem (10.4.1) for $\epsilon = 0$ with initial data (10.4.16).

The arguments `M`, `K`, and `T` have the same meaning as in Sub-problem (10-4.h).

To create the plots, write the corresponding basis expansion coefficient vectors to file and, from the C++ code, call a PYTHON program that reads that file, for instance, via `numpy.loadtxt()`, and visualizes the data using, for instance, `matplotlib.plot_surface()`.

SOLUTION for (10-4.i) → `10-4-9-0:slxs.pdf` ▲

(10-4.j) ⚄ (15 min.) The following table gives the error

$$\mathrm{err}(M,K) := \max_{\boldsymbol{p}\in\mathcal{V}(\mathcal{M}_M)} |u_h^{(K)}(\boldsymbol{p}) - u(\boldsymbol{p},T)| \quad , \quad \mathcal{V}(\mathcal{M}_M) := \{\text{vertices of } \mathcal{M}_M\} \, , \tag{10.4.18}$$

for the semi-Lagrangian method used to solve the *pure transport problem* with velocity $\mathbf{v}$ and initial distribution u_0 given by (10.4.16) and final time $T = \pi/2$. Here, M is the number of mesh cells in one direction and K the total number of equidistant timesteps. For the exact solution $u(\boldsymbol{p},T)$ we use the solution of Sub-problem (10-4.c).

$M\backslash K$	10	20	40	80	160	320	640
10	0.627466	0.646622	0.65448	0.658128	0.659892	0.660759	0.66119
20	0.341762	0.462079	0.507055	0.525879	0.534579	0.538771	0.540829
40	0.291969	0.187201	0.292734	0.334311	0.352605	0.361219	0.365401
80	0.290045	0.153201	0.0976401	0.163794	0.193055	0.206928	0.213644
160	0.289268	0.148721	0.0781957	0.0497284	0.0868246	0.104278	0.112746
320	0.289629	0.147799	0.0749274	0.0394266	0.0251081	0.0446948	0.0541931
640	0.289624	0.147583	0.0743156	0.0375747	0.0197919	0.0126224	0.0226748

Describe the asymptotic behavior of the error in terms of timestep $\tau \to 0$ and meshwidth $h_{\mathcal{M}} \to 0$.

SOLUTION for (10-4.j) → `10-4-10-0:sl7.pdf` ▲

End Problem 10-4 , 335 min.

Problem 10-5: Semi-Lagrangian Discretization of Transport Problem

[Lecture → Section 10.3.4] introduced the semi-Lagrangian discretization of linear scalar convection-diffusion problems as mesh-based finite element Galerkin discretization with streamline/characteristics)-based approach to the material derivative [Lecture → Def. 10.3.4.2].

The project is closely linked to [Lecture → Section 10.3.4]. It involves substantial implementation in C++ based on LEHRFEM++.
▷ problem name/problem code folder: TranspSemiLagr

As in [Lecture → Section 10.3.4] we start from the simple transient convection-diffusion problem

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u + \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u = f(\boldsymbol{x},t) \quad & \text{in } \Omega \times (0,T), \\ u = 0 \quad & \text{on } \partial\Omega \times (0,T), \\ u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad & \text{on } \Omega\,. \end{aligned} \tag{10.5.1}$$

Here $\Omega \subset \mathbb{R}^2$ is a polygonal computational domain described by a triangular mesh $\mathcal{M}$, $\mathbf{v} : \Omega \to \mathbb{R}^2$ a given continuous velocity field, and $f \in C^0(\overline{\Omega})$ a given source function.

The simplest **semi-Lagrangian discretization** of (10.5.1) combines an implicit-Euler discretization of the material derivative [Lecture → § 10.3.4.7] with a finite element Galerkin method (based on $\mathcal{S}^0_{1,0}(\mathcal{M})$) for the diffusive term. Thus, we arrive at the update formula [Lecture → (10.3.4.12)]

$$\begin{aligned} u_h^{(j)} \in \mathcal{S}^0_{1,0}(\mathcal{M}): \quad \int_\Omega \frac{u_h^{(j)}(\boldsymbol{x}) - u_h^{(j-1)}(\boldsymbol{\Phi}^{t_j,t_{j-\tau}}\boldsymbol{x})}{\tau} v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \epsilon \int_\Omega \mathbf{grad}\, u_h^{(j)} \cdot \mathbf{grad}\, v_h\,\mathrm{d}\boldsymbol{x} \\ = \int_\Omega f(\boldsymbol{x},t_j) v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v_h \in \mathcal{S}^0_{1,0}(\mathcal{M})\,. \end{aligned} \tag{10.5.2}$$

for a single timestep $u_h^{(j-1)} \to u_h^{(j)}$ of size $\tau := t_j - t_{j-1} > 0$. Point interpolation yielded the fully discrete version in [Lecture → (10.3.4.14)]:

$$\begin{aligned} u_h^{(j)} \in \mathcal{S}^0_{1,0}(\mathcal{M}): \quad \tfrac{1}{3}|U_{\boldsymbol{p}}|(\mu_{\boldsymbol{p}}^{(j)} - u_h^{(j-1)}(\boldsymbol{p} - \tau\mathbf{v}(\boldsymbol{p})) + \tau \int_\Omega \mathbf{grad}\, u_h^{(j)} \cdot \mathbf{grad}\, b_h^{\boldsymbol{p}}\,\mathrm{d}\boldsymbol{x} \\ = \tfrac{1}{3}\tau|U_{\boldsymbol{p}}| f(\boldsymbol{p})\,, \quad \forall \boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega\,, \end{aligned} \tag{10.5.3}$$

where $\mu_{\boldsymbol{p}}^{(j)}$ are the nodal values of $u_h^{(j)} \in \mathcal{S}^0_{1,0}(\mathcal{M})$ associated with the interior nodes of the mesh $\mathcal{M}$, $b_h^{\boldsymbol{p}}$ is the "tent function" belonging to node $\boldsymbol{p}$, $|U_{\boldsymbol{p}}|$ is the sum of the areas of all triangles adjacent to $\boldsymbol{p}$.

(10-5.a) ⚀ (15 min.) Study [Lecture → Section 10.3.4] again to refresh yourself on the motivation behind and the derivation of (10.5.2) and (10.5.3). ▲

(10-5.b) ⚄ (15 min.) The equations (10.5.3) give rise to a linear system of equations for the basis expansion coefficient vector $\vec{\boldsymbol{\mu}}^{(j)}$ of $u_h^{(j)}$ with respect to the standard nodal basis of $\mathcal{S}^0_{1,0}(\mathcal{M})$. This linear system has the form

$$(\mathbf{D}(\tau) + \mathbf{A})\vec{\boldsymbol{\mu}}^{(j)} = \vec{\boldsymbol{\varphi}}(\tau, u_h^{(j-1)})\,, \tag{10.5.4}$$

where $\mathbf{D}(\tau)$ is a τ-dependent *diagonal* $N \times N$ matrix, $N := \dim \mathcal{S}^0_{1,0}(\mathcal{M})$, and $\mathbf{A} \in \mathbb{R}^{N,N}$ does not depend on τ.

Determine the matrices $\mathbf{D}(\tau)$, $\mathbf{A}$, and the right-hand side vector $\vec{\boldsymbol{\varphi}}(\tau, u_h^{(j-1)})$.

SOLUTION for (10-5.b) → 10-5-2-0:sls0x.pdf ▲

The first part of this project is devoted to developing the class

```
class SemiLagrStep {
  public:
  template <typename FUNCTOR>
  SemiLagrStep(std::shared_ptr<const
     lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
                FUNCTOR &&v);
  Eigen:VectorXd step(const Eigen::VectorXd &u0_vector, double tau)
     const;
  private:
  // suitable data members to store precomputed matrices and vectors
};
```

which you can find in the file `transpsemilagr.h`.

This class is supposed to implement the update formula (10.5.3) in its `step()` member function, which yields $u_h^{(j)} = u_h^{(j)}(u_h^{(j-1)}) \in \mathcal{S}^0_{1,0}(\mathcal{M})$ by solving (10.5.4). The returned vector contains the basis expansion coefficients ("nodal values") of $u_h^{(j)}$, while the argument `u0_vector` passes those of $u_h^{(j-1)}$ and `tau` is the timestep. The functor argument `v` for the constructor gives the velocity field $\boldsymbol{x} \mapsto \mathbf{v}(\boldsymbol{x})$.

(10-5.c) ⚀ (15 min.) [depends on Sub-problem (10-5.b)]

Which data can be precomputed in the constructor of **SemiLagrStep** and stored in class variables, and which data have to be regenerated in every step.

SOLUTION for (10-5.c) → `10-5-3-0:sl0ys.pdf` ▲

(10-5.d) ⚁ (15 min.) [depends on Sub-problem (10-5.c)]

In light of Sub-problem (10-5.c) propose suitable private data members for the class **SemiLagrStep**.

SOLUTION for (10-5.d) → `10-5-4-0:lss0z.pdf` ▲

(10-5.e) ⚁ (60 min.) [depends on Sub-problem (10-5.d)]

In the file `transpsemilagr.h` implement the constructor of **SemiLagrStep** using LEHRFEM++'s assembly utility functions and the helper class **lf::uscalfe::LinearFELaplaceElementMatrix**.

Note that zero Dirichlet boundary conditions have to be taken into account, see [Lecture → Code 2.7.6.20].

SOLUTION for (10-5.e) → `10-5-5-0:lss0x.pdf` ▲

(10-5.f) ⚁ (60 min.) [depends on Sub-problem (10-5.c) and Sub-problem (10-5.d)]

In the file `transpsemilagr.cc` implement the `step()` member function of **SemiLagrStep**. `local_assembly.h`.

HIDDEN HINT 1 for (10-5.f) → `10-5-6-0:tslh3a.pdf`

HIDDEN HINT 2 for (10-5.f) → `10-5-6-1:locpsl.pdf`

SOLUTION for (10-5.f) → `10-5-6-2:.pdf` ▲

(10-5.g) ⚀ (30 min.) [depends on Sub-problem (10-5.f)]

As specified above, the member function `step()` of the class **SemiLagrStep** carries out a single timestep according to (10.5.3) with timestep size $\tau > 0$. The nodal values of the approximate solution $u_h^{(j)}$ after time τ has passed are returned.

Thus, you can use this function `SemiLagrStep::step()` to solve (10.5.1) on the unit disc $\Omega := \{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\| \leq 1\}$ with

$$\mathbf{v}(\boldsymbol{x}) = \begin{pmatrix} -x_2 + 3x_1^2 \\ x_1 \end{pmatrix}, \quad \boldsymbol{x} \in \Omega. \tag{10.5.8}$$

and final time $T = 1$. To that end, in the file `transpsemilagr.cc` write a C++ function

```
Eigen::VectorXd solverot(
  std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
    fe_space,
  Eigen::VectorXd u0_vector, int N, double T);
```

which relies on N uniform timesteps of the semi-Lagrangian method to solve the convection-diffusion problem (10.5.1) approximately over the period $[0, T]$.

The argument `u0_vector` is a vector of the basis expansion coefficients for a function $u_{0,h} \in \mathcal{S}^0_{1,0}(\mathcal{M})$ that approximates u_0. The function is to return the nodal values of the finite element solution at $t = T$ and at interior vertices.

SOLUTION for (10-5.g) → `10-5-7-0:transp-semi-lagr1.pdf` ▲

(10-5.h) ⚄ (30 min.) [depends on Sub-problem (10-5.g)]

We consider the setting of Sub-problem (10-5.g) and use the velocity field from (10.5.8). As initial value we choose (in polar coordinates)

$$u_0(r, \varphi) = (1 - r^2) r \cos(\varphi), \quad 0 \leq r \leq 1, \quad 0 \leq \varphi \leq 2\pi. \tag{10.5.10}$$

In the file `transpsemilagr.cc` complete the function

```
void visSLSolution();
```

that uses the semi-Lagrangian method with $N = 5$ timesteps and mesh $\mathcal{M}$ provided in the file `circle.msh` to solve the concrete convection-diffusion problem approximately over the period $[0, 0.2]$.

With PARAVIEW create a surface plot of the finite-element solution at final time $T = 1$, see [Lecture → Suppl. 2.7.1.14] for instructions how to do with with LEHRFEM++'s infrastructure.

SOLUTION for (10-5.h) → `10-5-8-0:cpp:vistsl.pdf` ▲

(10-5.i) For a stationary velocity field $\mathbf{v} \in (H^1(\Omega))^2$ we consider the evolution problem

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u + \operatorname{div}(\mathbf{v}(\boldsymbol{x}) u) &= 0 && \text{in } \Omega \times (0, T), \\ u &= 0 && \text{on } \partial\Omega \times (0, T), \\ u(\boldsymbol{x}, 0) &= u_0 && \text{on } \Omega \times \{0\}. \end{aligned} \tag{10.5.12}$$

Convert it into an initial-boundary value problem for the PDE

$$\frac{\partial u}{\partial t} - \Delta u + \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u - c(\boldsymbol{x}) u = 0, \tag{10.5.13}$$

with a suitable scalar coefficient function $c : \Omega \mapsto \mathbb{R}$, which may depend on $\mathbf{v}$.

HIDDEN HINT 1 for (10-5.i) → `10-5-9-0:transp-semi-lagr21.pdf`

SOLUTION for (10-5.i) → `10-5-9-1:transp-semi-lagr2.pdf` ▲

(10-5.j) (30 min.) Which evolution problems (over small time intervals) have to be solved in each step, when split-step timestepping based on the **Strang splitting**, see [Lecture → Section 10.3.3.1] or [NumCSE course → **??**], is applied to the evolution problem

$$\begin{aligned} \frac{\partial u}{\partial t} &= \underbrace{\Delta u - \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u}_{=:g(u)} + \underbrace{c(\boldsymbol{x})u}_{=:h(u)} \quad \text{in} \quad \Omega \times]0,T[\,, \\ u &= 0 \quad \text{on} \quad \partial\Omega \times]0,T[\,, \\ u(\boldsymbol{x},0) &= u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Omega\,. \end{aligned} \tag{10.5.14}$$

Here the decomposition of the right-hand side of the PDE indicated by the underbraces defines the splitting to be used. The timestep size should be denoted by $\tau > 0$.

SOLUTION for (10-5.j) → 10-5-10-0:transp-semi-lagr3.pdf ▲

(10-5.k) (30 min.) In the file `transpsemilagr.h` complete the implementation of the C++ class

```
template <typename FUNCTOR>
class ReactionStep {
  public:
  ReactionStep(std::shared_ptr<const
     lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
                FUNCTOR &&c);
  Eigen:VectorXd step(const Eigen::VectorXd &u0_vector, double
     tau) const;
  private:
  // suitable data members to store precomputed matrices and
     vectors
};
```

whose member function `step()` solves the discrete variational evolution problem: seek $t \mapsto u_h(t) \in \mathcal{S}^0_{1,0}(\mathcal{M})$

$$\int_\Omega \frac{\partial u_h}{\partial t}(\boldsymbol{x},t)\, v_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega c(\boldsymbol{x}) u_h(\boldsymbol{x})\, v_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v_h \in \mathcal{S}^0_{1,0}(\mathcal{M})\,, \tag{10.5.19}$$

over one timestep of length $\tau > 0$ using the **explicit midpoint rule** [NumCSE course → Ex. 11.4.0.17], a 2-stage explicit Runge-Kutta method described by the Butcher scheme [Lecture → § 9.2.6.30], [NumCSE course → (11.4.0.13)],

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}\,. \tag{10.5.20}$$

The integrals in (10.5.19) should be approximated by means of local vertex based quadrature rule ("2D trapezoidal rule" [Lecture → (2.4.6.10)], "mass lumping", see [Lecture → Rem. 9.3.4.18]).

The argument `u0_vector` contains the nodal values for u_h before the timestep, and `c` should be a functor providing the scalar coefficient $\boldsymbol{x} \mapsto \mathbf{c}(\boldsymbol{x})$ to the constructor.

SOLUTION for (10-5.k) → 10-5-11-0:transp-semi-lagr3a.pdf ▲

(10-5.l) ⚄ (30 min.) [depends on Sub-problem (10-5.k) and Sub-problem (10-5.i)]

In the file `transpsemilagr.cc` write a C++ function

```
Eigen::VectorXd solvetrp(
  std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
    fe_space,
  Eigen::VectorXd u0_vector, int N, double T);
```

which relies on N uniform steps of the Strang-splitting split-step method discussed in Sub-problem (10-5.j) to solve (10.5.14) approximately over the period $[0, T]$.

The argument `u0_vector` is the basis expansion coefficient vector of $u_{0,h} \in \mathcal{S}^0_{1,0}(\mathcal{M})$ that provides an approximation for u_0. As velocity field $\boldsymbol{x} \mapsto \mathbf{v}(\boldsymbol{x})$ use the vectorfield given by (10.5.8). The coefficient function $\boldsymbol{x} \mapsto c(\boldsymbol{x})$ can be derived from it as in Sub-problem (10-5.i).

HIDDEN HINT 1 for (10-5.l) → 10-5-12-0:transp-semi-lagr41.pdf

SOLUTION for (10-5.l) → 10-5-12-1:transp-semi-lagr4.pdf ▲

(10-5.m) ⚄ (30 min.) [depends on Sub-problem (10-5.l)]

Using the scheme implemented in `solvetrp()` from Sub-problem (10-5.l) compute *and plot* (surface plot) an approximate solution finite-element solution of (10.5.14) with final time $T = 0.2$, $N = 5$ timesteps, the mesh from `circle.msh`, the initial data from (10.5.10), and the vectorfield $\mathbf{v}$ from (10.5.8). Do this within a function

```
void vistrp(void);
```

For plotting rely on PARAVIEW as in Sub-problem (10-5.h).

SOLUTION for (10-5.m) → 10-5-13-0:.pdf ▲

End Problem 10-5, 360 min.

Problem 10-6: Streamline Upwind Method for Pure Advection Problem

The standard Galerkin finite-element method for pure transport problems suffers from instability as we saw [Lecture → Exp. 10.2.2.4] and [Lecture → Exp. 10.2.2.12]. There are remedies: in [Lecture → Section 10.2.2.2] we learned about the consistent streamline upwinding, which relies on a stabilized variational formulation of the convection-diffusion boundary value problem. In this task we implement the method for pure transport boundary value problems.

This problem builds on [Lecture → Section 10.2.2.2]. The coding part relies on LEHRFEM++, which makes familiarity with [Lecture → Section 2.7] and also [Lecture → Section 2.8] indispensable.
▷ problem name/problem code folder: AdvectionSUPG

On a polygonal domain $\Omega \subset \mathbb{R}^2$ and for a given velocity field $\mathbf{v} \in (C^0(\overline{\Omega}))^2$ and right-hand-side source function $f \in C^0(\overline{\Omega})$, we consider the pure transport boundary value problem

$$\mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x}) = f(\boldsymbol{x}) \quad \text{in} \quad \Omega \quad , \quad u = g \quad \text{on} \quad \Gamma_{\text{in}} \, , \tag{10.6.1}$$

where the inflow boundary part Γ_{in} is defined as

$$\Gamma_{\text{in}} := \{\boldsymbol{x} \in \partial\Omega \colon \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\} \, , \qquad [\text{Lecture} \rightarrow (10.2.1.11)]$$

and the Dirichlet data $g \in C^0(\overline{\Gamma_{\text{in}}})$ are given.

(10-6.a) ⚄ (15 min.) As a boundary value problem for testing we consider the special setting:

- unit square domain $\Omega \in]0,1[^2$,
- given velocity field

$$\mathbf{v}(\boldsymbol{x}) := \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix} \, , \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \, , \tag{10.6.2}$$

- Dirichlet data $g := \widetilde{g}|_{\Gamma_{\text{in}}}$ with

$$\widetilde{g}(\boldsymbol{x}) := \sin^2(\pi x_1) \, , \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \, , \tag{10.6.3}$$

- vanishing source function $f \equiv 0$.

Visualization of the given velocity field ▷

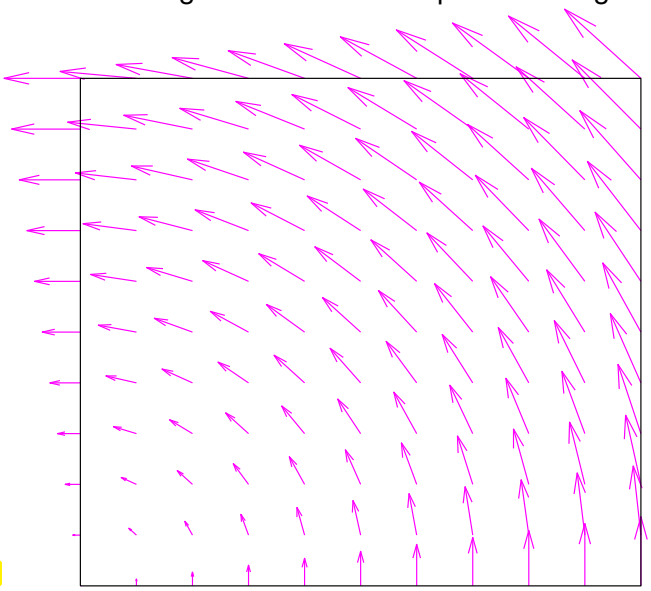


Fig. 161

Write down the solution of (10.6.1) in this case.

$$u(\boldsymbol{x}) = \boxed{} \, , \quad \boldsymbol{x} \in \Omega \, .$$

HIDDEN HINT 1 for (10-6.a) → 10-6-1-0:adsuah1.pdf

SOLUTION for (10-6.a) → 10-6-1-1:supgs0.pdf ▲

The **streamline upwind** method for (10.6.1) employs a finite-element Galerkin discretion of the variational problem:

$$\begin{matrix} u \in H^1(\Omega) \\ u|_{\Gamma_{\text{in}}} = g \end{matrix} \colon \quad \int_\Omega \delta(\boldsymbol{x}) \mathbf{v}(\boldsymbol{x}) \mathbf{v}^\top(\boldsymbol{x}) \, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, w(\boldsymbol{x}) + \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x})\, w(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} =$$

$$\int_\Omega f(x)(w(x)+\delta(x)\mathbf{v}(x)\cdot\mathbf{grad}\,w(x))\,\mathrm{d}x \quad \forall w\in H^1_{\Gamma_{\rm in}}(\Omega)\,. \tag{10.6.5}$$

Here $H^1_{\Gamma_{\rm in}}(\Omega) := \{w\in H^1(\Omega):\ w|_{\Gamma_{\rm in}}=0\}$ is the space of functions $\S H^1(\Omega)$ vanishing on the inflow boundary part.

If $\mathcal{M}$ denotes the finite-element mesh used, then the function $\delta:\Omega\to\mathbb{R}$ is usually chosen $\mathcal{M}$-piecewise constant according to [Bur10]

$$\delta|_K = \frac{\operatorname{diam} K}{2\max\{\|\mathbf{v}(a_K^1)\|,\ldots,\|\mathbf{v}(a_K^p)\|\}}\,,\quad a_K^\ell \hat{=} \text{ corners of } K\,. \tag{10.6.6}$$

Instructions for implementation:

- Throughout in this problem we rely on **quadratic Lagrangian finite elements** on triangular meshes, that is, subspaces of $\mathcal{S}_2^0(\mathcal{M})$, $\mathcal{M}$ comprising only triangles.
- We use the standard local and global shape functions for $\mathcal{S}_2^0(\mathcal{M})$ as introduced in [Lecture → Ex. 2.6.1.2].
- For the approximation of all integrals over cells of the mesh use the quadrature rule supplied by the LehrFEM++ function

```
lf::quad::QuadRule make_TriaQR_P6O4();
```

(10-6.b) ⚂ (10 min.)

We consider the Galerkin finite element discretization of the advection boundary value problem described in Sub-problem (10-6.a) using the uniform "tensor-product" triangular mesh $\mathcal{M}$ as sketched beside. ▷

What is the number of *unknown* expansion coefficients for the Galerkin solution, if the mesh has $N\in\mathbb{N}$ cells in each direction, that is, $h=N^{-1}$?

No. of FE unknowns = ☐ .

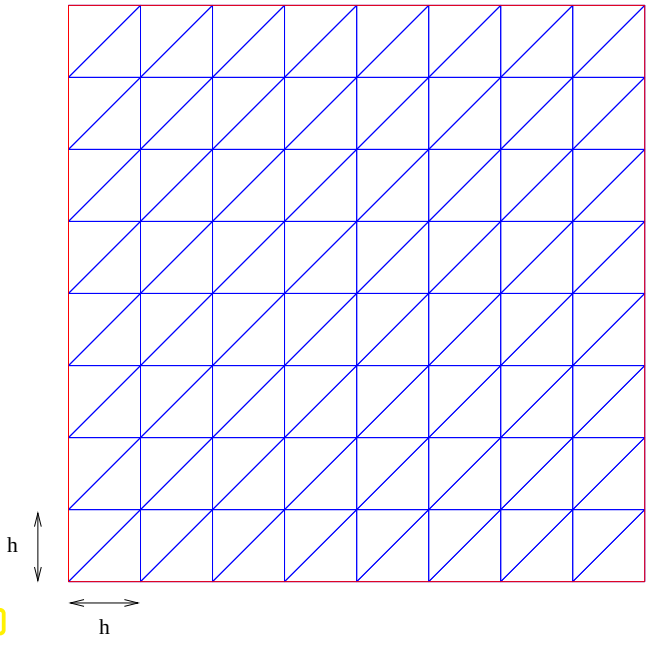


Fig. 162

SOLUTION for (10-6.b) → 10-6-2-0:asusa.pdf ▲

(10-6.c) ⚂ (10 min.) We write $(\omega_\ell,\widehat{\zeta}_\ell)$, $\ell=1,\ldots,6$, for the weight-node pairs characterizing the six-point quadrature rule on the reference triangle $\widehat{K}$ provided by the LehrFEM++ function `make_TriaQR_P6O4()`, and $\widehat{b}^1,\ldots,\widehat{b}^6$ for the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ on $\widehat{K}$. Moreover, for a triangle $K\in\mathcal{M}$ let $\mathbf{\Phi}_K:\widehat{K}\to K$ denote the unique affine mapping taking $\widehat{K}$ to K.

Using only these notations, complete the following explicit formula for the entries of the $\mathcal{S}_2^0(\mathcal{M})$ element matrix $\mathbf{A}_K\in\mathbb{R}^{6,6}$ induced by the bilinear form

$$(u,w)\mapsto\int_\Omega \mathbf{v}(x)\mathbf{v}(x)^\top\,\mathbf{grad}\,u(x)\cdot\mathbf{grad}\,w(x)\,\mathrm{d}x\,,\quad u,w\in H^1(\Omega)\,. \tag{10.6.7}$$

$$(\mathbf{A}_K)_{i,j} = \sum_{\ell=1}^{6} \boxed{} \, |\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)| \,,$$

$i,j \in \{1,\ldots,6\}$.

SOLUTION for (10-6.c) → 10-6-3-0:asubs.pdf ▲

(10-6.d) ⚂ (10 min.) Using the notations $(\omega_\ell, \widehat{\boldsymbol{\zeta}}_\ell, \widehat{b}^i, \boldsymbol{\Phi}_K)$ from Sub-problem (10-6.c) give an explicit formula for the $\mathcal{S}_2^0(\mathcal{M})$ element matrix $\mathbf{B}_K \in \mathbb{R}^{6,6}$ belonging to the bilinear form

$$(u,w) \mapsto \int_\Omega \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x})\, w(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,, \quad u,w \in H^1(\Omega) \,, \tag{10.6.9}$$

and the triangle $K \in \mathcal{M}$.

$$(\mathbf{B}_K)_{i,j} = \sum_{\ell=1}^{6} \boxed{} \, |\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)| \,,$$

$i,j \in \{1,\ldots,6\}$.

HIDDEN HINT 1 for (10-6.d) → 10-6-4-0:asupch1.pdf

SOLUTION for (10-6.d) → 10-6-4-1:asubb.pdf ▲

(10-6.e) ⚂ (45 min.) [depends on Sub-problem (10-6.c), Sub-problem (10-6.d)]

In the file `advectionsupg.h` finish the implementation of the `Eval()` method of the class

```
template <class MESHFUNCTION_V>
class SUAdvectionElemMatrixProvider {
 public:
  using ElemMat = Eigen::Matrix<double, Eigen::Dynamic,
     Eigen::Dynamic>;
  SUAdvectionElemMatrixProvider(MESHFUNCTION_V &v, bool use_delta
     = true);
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) {
     return true; }
  ElemMat Eval(const lf::mesh::Entity &cell);
 private:
  // six-point quadrature rule
  const lf::quad::QuadRule qr_{lf::quad::make_TriaQR_P6O4()};
  // A mesh function object providing the velocity field v
  MESHFUNCTION_V &v_;
  // Values of reference shape functions at quadrature points
  Eigen::MatrixXd val_ref_lsf_;
  // Gradients of reference shape functions at quadrature points
  Eigen::MatrixXd grad_ref_lsf_;
  // Flag for controlling use of delta scaling (for testing)
  bool use_delta_;
};
```

Here **MESHFUNCTION_V** is the type of an LEHRFEM++ **MeshFunction** that provides the function $\boldsymbol{x} \mapsto \mathbf{v}(\boldsymbol{x})$, returning a vector of `Eigen::Vector2d`s. The corresponding velocity **MeshFunction** object is given to the constructor as argument `v`.

This class fits the concept of a LEHRFEM++ **ENTITY_MATRIX_PROVIDER**, see [Lecture → Ex. 2.7.4.22]. For a *triangular* cell it is supposed to compute the $\mathcal{S}_2^0(\mathcal{M})$ element matrix (approximately by numerical quadrature) for the bilinear form

$$(u,w) \mapsto \int_\Omega \delta(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})\mathbf{v}^\top(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,w(\boldsymbol{x}) + \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\,u(\boldsymbol{x})\,w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \tag{10.6.11}$$

$u, w \in H^1(\Omega)$, where δ is defined in (10.6.6).

The code for the constructor is a follows

C++ code 10.6.12: Implementation of the constructor of SUAdvectionElemMatrixProvider
➙ GitLab

```
template <class MESHFUNCTION_V>
SUAdvectionElemMatrixProvider<MESHFUNCTION_V>::SUAdvectionElemMatrixProvider(
    MESHFUNCTION_V &v, bool use_delta)
    : v_(v), use_delta_(use_delta) {
  const lf::uscalfe::FeLagrangeO2Tria<double> ref_fe;
  LF_ASSERT_MSG(ref_fe.RefEl() == lf::base::RefEl::kTria(),
                "Only implemented for triangles");
  LF_ASSERT_MSG(ref_fe.NumRefShapeFunctions() == 6,
                "Quadratic Lagrange FE must have six local shape functions");
  LF_ASSERT_MSG(qr_.RefEl() == lf::base::RefEl::kTria(),
                "Quadrature rule must be for triangles");
  LF_ASSERT_MSG(qr_.NumPoints() == 6, "Six-point quadrature rule required");
  // Obtain values and gradients of reference shape functions at
  // quadrature nodes, see [Lecture → § 2.8.3.25]
  val_ref_lsf_ = ref_fe.EvalReferenceShapeFunctions(qr_.Points());
  grad_ref_lsf_ = ref_fe.GradientsReferenceShapeFunctions(qr_.Points());
}
```

HIDDEN HINT 1 for (10-6.e) → `10-6-5-0:sauh2.pdf`

HIDDEN HINT 2 for (10-6.e) → `10-6-5-1:sauh3.pdf`

HIDDEN HINT 3 for (10-6.e) → `10-6-5-2:sauh4.pdf`

SOLUTION for (10-6.e) → `10-6-5-3:supgs2.pdf` ▲

(10-6.f) ⚄ (60 min.) [depends on Sub-problem (10-6.e)]

In the file `advectionsupg.cc` complete the implementation of the C++ function

```
Eigen::VectorXd solveRotSUPG(std::shared_ptr<const
   lf::uscalfe::FeSpaceLagrangeO2<double>> fes_ptr);
```

that relies on the streamline upwind $\mathcal{S}_2^0(\mathcal{M})$ finite element method to solve the pure advection problem (10.6.1)

- on the unit square $\Omega =]0,1[^2$,
- with the velocity field $\boldsymbol{x} \mapsto \mathbf{v}(\boldsymbol{x})$ from (10.6.2), see also Fig. 166, and $f \equiv 0$,
- and non-zero Dirichlet boundary conditions (10.6.3) on the inflow boundary Γ_{in}.

The argument `fes_ptr` provides the *triangular* mesh $\mathcal{M}$ and the local-to-global index mappings. The

function should return the vector of expansion coefficients of the finite-element solution with respect to the standard nodal basis of $\mathcal{S}_2^0(\mathcal{M})$.

HIDDEN HINT 1 for (10-6.f) → `10-6-6-0:asu9h1.pdf`

SOLUTION for (10-6.f) → `10-6-6-1:asu9hs.pdf` ▲

(10-6.g) ⚄ (45 min.) [depends on Sub-problem (10-6.a) and Sub-problem (10-6.f)]

In Sub-problem (10-6.a) we found the exact solution of the boundary value problem solved by `solveRotSUPG()`. Use it to write a C++ function

```
void cvgL2SUPG(void);
```

that *tabulates* the error norms $\|u - u_\ell\|_{L^2(\Omega)}$ (as a function of $N := \dim \mathcal{S}_2^0(\mathcal{M}_\ell)$) for the setting of Sub-problem (10-6.f), where $\boldsymbol{x} \mapsto u(\boldsymbol{x})$ is the exact solution and $u_\ell \in \mathcal{S}_2^0(\mathcal{M}_\ell)$ is the streamline upwind finite-element solution on a triangular tensor-product mesh $\mathcal{M}_\ell$ as displayed in Fig. 162 with meshwidth $h_\ell := 0.1 \cdot 2^{-\ell}$, $\ell = 0, 1, 2, \ldots, 5$.

Note. A triangular tensor-product mesh in LEHRFEM++ can be built by means of the helper class **lf::mesh::hybrid2d::TPTriagMeshBuilder**.

Describe qualitatively and quantitatively the observed convergence of $\|u - u_h\|_{L^2(\Omega)}$ as $N \to \infty$, where $N \in \mathbb{N}$ is the dimension of the finite-element space.

HIDDEN HINT 1 for (10-6.g) → `10-6-7-0:supxh1.pdf`

SOLUTION for (10-6.g) → `10-6-7-1:asupxs.pdf` ▲

References

[Bur10] Erik Burman. "Consistent SUPG-method for transient transport problems: stability and convergence". In: *Comput. Methods Appl. Mech. Engrg.* 199.17-20 (2010), pp. 1114–1123. DOI: 10.1016/j.cma.2009.11.023 (cit. on p. 390).

End Problem 10-6 , 195 min.

Problem 10-7: Streamline-Upwind Stabilized Finite Element Method

As we have seen in [Lecture → Exp. 10.2.2.12] applying a Lagrangian finite element Galerkin method to a stationary scalar transport problem likely results in useless solutions haunted by spurious oscillations. The use of so-called stabilized Galerkin finite element methods is mandatory and one of the most popular is the streamline upwind method presented in [Lecture → Section 10.2.2.2].

This problem asks for C++ implementation based on the LEHRFEM++ library.
▷ problem name/problem code folder: SUFEM

On a polygonal domain $\Omega \subset \mathbb{R}^2$ and for a given velocity field $\mathbf{v} \in (C^1(\overline{\Omega}))^2$ and right-hand-side source function $f \in C^0(\overline{\Omega})$, we consider the pure transport boundary value problem

$$\mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x}) = f(\boldsymbol{x}) \quad \text{in} \quad \Omega \quad , \quad u = g \quad \text{on} \quad \Gamma_{\text{in}} \, , \tag{10.7.1}$$

where the inflow boundary part Γ_{in} is defined as

$$\Gamma_{\text{in}} := \{\boldsymbol{x} \in \partial\Omega \colon \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\} \, , \qquad [\text{Lecture} \rightarrow (10.2.1.11)]$$

and the Dirichlet data $g \in C^0(\overline{\Gamma_{\text{in}}})$ are given.

Assumption 10.7.2. Properties of advection velocity field

We assume that the velocity field $\mathbf{v} \in (C^1(\overline{\Omega}))^2$ and its associated flow map $\boldsymbol{\Phi}$ [Lecture → § 10.1.1.3] satisfy

(i) $\mathbf{v}(\boldsymbol{x}) \neq \mathbf{0}$ for all $\boldsymbol{x} \in \overline{\Omega}$,
(ii) For all $\boldsymbol{x} \in \overline{\Omega}$ there is an $s \in \mathbb{R}$, $\boldsymbol{x}_0 \in \Gamma_{\text{in}}$ such that $\boldsymbol{\Phi}^s \boldsymbol{x}_0 = \boldsymbol{x}$, and
(iii) $\operatorname{div} \mathbf{v}(\boldsymbol{x}) \leq 0$ for all $\boldsymbol{x} \in \overline{\Omega}$.

(10-7.a) ⚄ (15 min.) As a boundary value problem for testing we consider the special setting:

- unit square domain $\Omega \in]0,1[^2$,
- given velocity field

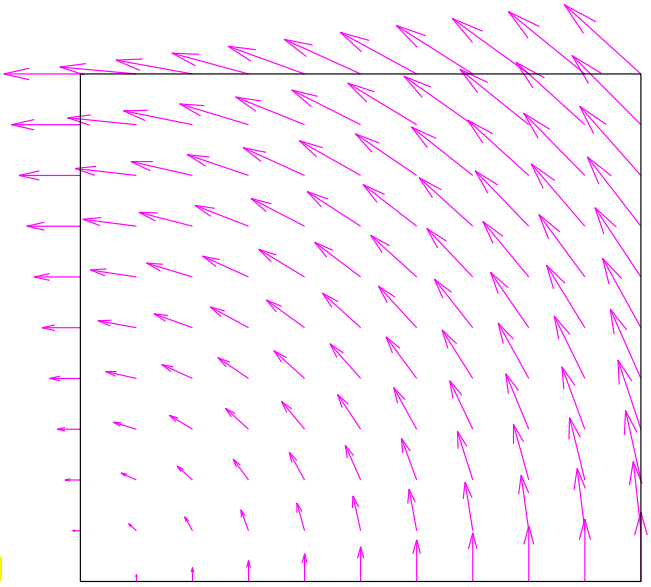
$$\mathbf{v}(\boldsymbol{x}) := \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix} \, , \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \, , \tag{10.7.3}$$

- Dirichlet data $g := \widetilde{g}|_{\Gamma_{\text{in}}}$ with

$$\widetilde{g}(\boldsymbol{x}) := x_1(1 - x_1) \, , \quad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \, , \tag{10.7.4}$$

- vanishing source function $f \equiv 0$.

Visualization of the given velocity field ▷ Fig. 165



Write down the solution of (10.7.1) in this case.

$$u(\boldsymbol{x}) = \qquad , \quad \boldsymbol{x} \in \Omega \, .$$

HIDDEN HINT 1 for (10-7.a) → 10-7-1-0:adsuah1.pdf

SOLUTION for (10-7.a) → 10-7-1-1:supgs0.pdf ▲

The **streamline-upwind** variational formulation of (10.7.1) is

$$u \in H^1(\Omega)\,, \quad u|_{\Gamma_{\text{in}}} = g\colon \quad a_{\text{SU}}(u,w) = \ell_{\text{SU}}(w) \quad \forall w \in H^1_{\Gamma_{\text{in}}}(\Omega)\,, \tag{10.7.6}$$

with the test space ("do not test where the solution is known" [Lecture → § 1.8.0.1])

$$H^1_{\Gamma_{\text{in}}}(\Omega) := \{w \in H^1(\Omega) : \ w|_{\Gamma_{\text{in}}} = 0\}\,, \tag{10.7.7}$$

the bilinear form

$$a_{\text{SU}}(u,w) := \int_\Omega \delta(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})\mathbf{v}^\top(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,w(\boldsymbol{x}) + \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\,u(\boldsymbol{x})\,w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \tag{10.7.8}$$

and the right-hand side linear form

$$\ell_{\text{SU}}(w) := \int_\Omega f(\boldsymbol{x})(w(\boldsymbol{x}) + \delta(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\,w(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}\,. \tag{10.7.9}$$

Further, $\delta : \Omega \to \mathbb{R}^+$ is a strictly positive, bounded function. In the context of a finite-element Galerkin discretization of (10.7.6) relying on the mesh $\mathcal{M}$ the function $\delta : \Omega \to \mathbb{R}$ is usually chosen $\mathcal{M}$-piecewise constant according to [Bur10]:

$$\delta|_K = \min\left\{1, \frac{\sqrt{|K|}}{\max\{\|\mathbf{v}(\boldsymbol{a}_K^1)\|,\ldots,\|\mathbf{v}(\boldsymbol{a}_K^p)\|\}}\right\}, \quad \begin{array}{l}\boldsymbol{a}_K^\ell \hat{=} \text{ corners of } K\,,\\ |K| \hat{=} \text{ area of } K\,.\end{array} \tag{10.7.10}$$

(10-7.b) ⚄ (30 min.) In the file `sufem.h` complete the implementation of the `Eval()` method of the class **AdvectionElementMatrixProvider**, which fits LehrFEM++'s concept of an **ENTITY_MATRIX_PROVIDER**. It is supposed to compute the element matrices for parametric Lagrangian finite element spaces and the "advection bilinar form"

$$b(u,w) := \int_\Omega \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\,u(\boldsymbol{x})\,w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u,w \in H^1(\Omega)\,, \tag{10.7.11}$$

where $\mathbf{v} : \overline{\Omega} \to \mathbb{R}^2$ is a given velocity vector field.

A skeletal definition of the class is given in Code 10.7.12.

C++ code 10.7.12: Skeletal definition of AdvectionElementMatrixProvider

```
template <typename VELOCITY>
class AdvectionElementMatrixProvider {
  using ElemMat = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>;
  AdvectionElementMatrixProvider(
      std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>> fe_space,
      VELOCITY velo);
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) { return true; }
  ElemMat Eval(const lf::mesh::Entity &cell);
 private:
  VELOCITY velo_;
  std::array<lf::uscalfe::PrecomputedScalarReferenceFiniteElement<double>, 5>
      fe_precomp_;
};
```

The template parameter **VELOCITY** must be a **MeshFunction** type whose evaluation operator returns a vector of type **Eigen::Vector2d**. The constructor expects the description of the parametric Lagrangian finite element space as arguement `fe_space` and the velocity **MeshFunction** in `velo`. For instance, an object of type **AdvectionElementMatrixProvider** can be created as follows:

```
SUFEM::AdvectionElementMatrixProvider elmat_provider(
    fe_space, lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> Eigen::Vector2d {
        return Eigen::Vector2d(-x[1], x[0]);
      }));
```

HIDDEN HINT 1 for (10-7.b) → 10-7-2-0:sufemh1.pdf

SOLUTION for (10-7.b) → 10-7-2-1:sufemsd.pdf ▲

(10-7.c) ⚄ (20 min.) [depends on Sub-problem (10-7.b)]

The function `buildSUGalerkinMatrix()` shown in Code 10.7.18 creates the *full* Galerkin matrix for the bilinear form a_{SU} defined in (10.7.8) and a Lagrangian finite element space, whose description is passed in the argument `fe_space`. A **MeshFunction** providing the velocity vector field $\mathbf{v} : \overline{\Omega} \to \mathbb{R}^2$ is supplied through `velo`.

C++ code 10.7.18: Function building Galerkin matrix for a_{SU}

```
template <typename VELOCITY>
lf::assemble::COOMatrix<double> buildSUGalerkinMatrix(
    std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>> fe_space,
    VELOCITY velo) {
  static_assert(lf::mesh::utils::MeshFunction<VELOCITY>);
  // Create object of helper class providing diffusion tensor for SU bilinear
  // form
  MeshFunctionDiffTensor mf_diff(velo);
  // Local computations for advection bilinear form
  AdvectionElementMatrixProvider adv_elmat(fe_space, velo);
  // Local computations for diffusive part
  lf::fe::DiffusionElementMatrixProvider diff_elmat(fe_space, mf_diff);
  // The local-to-global index map for the finite element space
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  const lf::base::size_type N_dofs = dofh.NumDofs();
  // Galerkin matrix in triplet format
  lf::assemble::COOMatrix<double> A_COO(N_dofs, N_dofs);
  // Invoke cell-oriented assembly in turns for both parts of the bilinear form
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, adv_elmat, A_COO);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, diff_elmat, A_COO);
  return A_COO;
}
```

The auxiliary class **MeshFunctionDiffTensor**, which fits the LEHRFEM++ concept of a **MeshFunction** is defined as follows:

C++ code 10.7.19: Skeletal definition of MeshFunctionDiffTensor

```
template <typename VELOCITY>
class MeshFunctionDiffTensor {
 public:
  explicit MeshFunctionDiffTensor(VELOCITY velo) : velo_(std::move(velo)) {}
  [[nodiscard]] std::vector<Eigen::Matrix2d> operator()(
      const lf::mesh::Entity &e, const Eigen::MatrixXd &local) const;
 private:
  VELOCITY velo_;
};
```

In the file `sufem.h` complete the implementation of `MeshFunctionDiffTensor::operator()` so that objects of the class **MeshFunctionDiffTensor** perform properly in the function `buildSUGalerkinMatrix()`.

SOLUTION for (10-7.c) → `10-7-3-0:sufemse.pdf` ▲

(10-7.d) ⚄ (20 min.) [depends on Sub-problem (10-7.c)]

The the file `sufem.h` contains the code of the function

```
template <typename VELOCITY, typename DIRICHLET_DATA>
  Eigen::VectorXd solveAdvectionDirichlet(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>>
       fe_space,
    VELOCITY velo, DIRICHLET_DATA mf_g);
```

This function is supposed to solve the Dirichlet boundary value problem (10.7.1) with $f \equiv 0$ approximately by means of a Galerkin discretization based on a triangular mesh $\mathcal{M}$ of Ω and $\mathcal{M}$-piecewise linear Lagrangian finite elements. the space $\mathcal{S}_1^0(\mathcal{M})$.

The parameter `fe_space` provides information about the mesh and the local→global index mapping, `velo` and `mf_g` are **MeshFunction** objects, which describe the velocity vector field $\mathbf{v}$ and the boundary data. Note that, nevertheless, `mf_g` is a mesh function on $\mathcal{M}$. Only its values on Γ_{in} will be used.

The treatment of Dirichlet boundary conditions is still missing. Supplement the code of `solveAdvectionDirichlet()` so that it fully complies with the specification.

HIDDEN HINT 1 for (10-7.d) → `10-7-4-0:sufemhf.pdf`

SOLUTION for (10-7.d) → `10-7-4-1:sufemsf.pdf` ▲

(10-7.e) ⚅ [depends on Sub-problem (10-7.d)]

In the file `sufem.cc` implement a LEHRFEM++-based C++ function

```
void solveBVPSUFEM(const char *meshfile, const char *vtkfile);
```

that uses your implementation of `solveAdvectionDirichlet()` to solve the boundary value problem as described in Sub-problem (10-7.a). `meshfile` must be the name (without suffix) of a GMSH-compatible `.msh` file providing a triangulation of the unit square, while `vtkfile` specifies (without suffix) the name of the output `.vtk` file, which then can be used to visualize the solution with PARAVIEW.

SOLUTION for (10-7.e) → `10-7-5-0:.pdf` ▲

References

[Bur10] Erik Burman. "Consistent SUPG-method for transient transport problems: stability and convergence". In: *Comput. Methods Appl. Mech. Engrg.* 199.17-20 (2010), pp. 1114–1123. DOI: 10.1016/j.cma.2009.11.023 (cit. on p. 395).

End Problem 10-7 , 85 min.

Problem 10-8: Upwind Quadrature Scheme for Transport

The upwind quadrature scheme [Lecture → Section 10.2.2.1] offers a stable discretization of scalar transport problems by taking into account the flow of information. In this problem we examine the upwind quadrature scheme for pure transport.

The contents of [Lecture → Section 10.2.1] and [Lecture → Section 10.2.2.1] are required for this problem.
▷ problem name/problem code folder: TransportUpwindQuadrature

On the unit square domain we consider the stationary convection-diffusion boundary value problem ($\epsilon > 0$)

$$-\epsilon\Delta u + \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u = 0 \quad \text{in} \quad \Omega :=]0,1[^2 \quad , \quad u = g \quad \text{on} \quad \partial\Omega \, , \tag{10.8.1}$$

The velocity field is that of a rigid rotation around $\mathbf{0}$,

$$\mathbf{v}(\boldsymbol{x}) = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix} , \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} . \tag{10.8.2}$$

Vector plot of velocity field $\mathbf{v}$ ▷

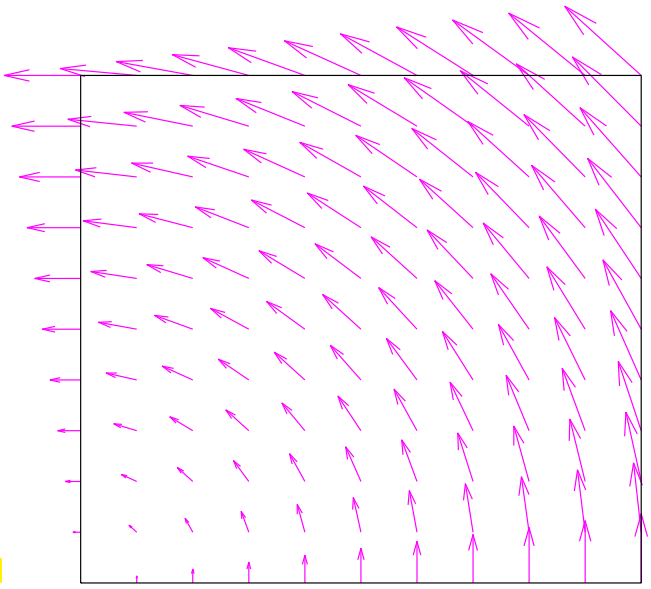


Fig. 166

The boundary data $g : \partial\Omega \to \mathbb{R}$ are nonzero only on the bottom side,

$$g(\boldsymbol{x}) = \begin{cases} \sin^2(\pi x_1) & \text{for} \quad x_2 = 0 \, , \\ 0 & \text{elsewhere on } \partial\Omega \, , \end{cases} \quad \boldsymbol{x} \in \partial\Omega \, . \tag{10.8.3}$$

(10-8.a) ⚄ (10 min.)

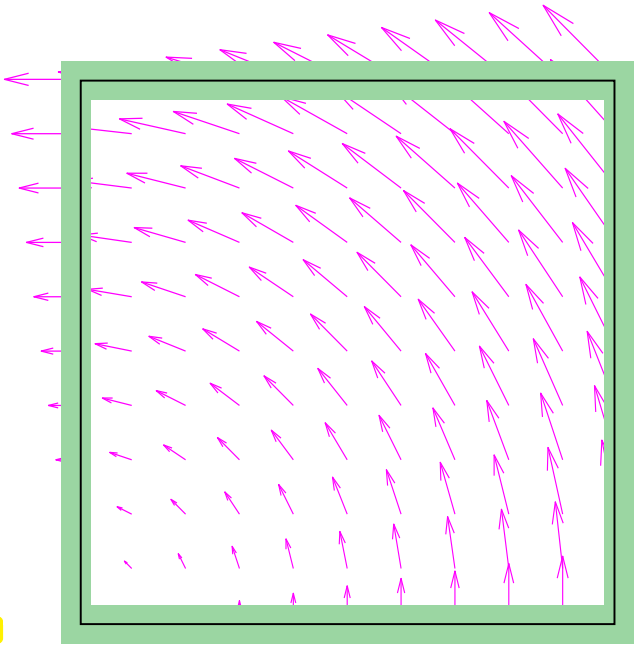


Fig. 167

In the figure beside mark those parts of the boundary $\partial\Omega$, where you expect **boundary layers** to emerge in the solution of (10.8.1) for $\epsilon \to 0$.

Your marking should look like this:

~~xxxxxxxxxxxxxxxxxx~~

SOLUTION for (10-8.a) → 10-8-1-0:.pdf ▲

In the remainder of this problem we focus on the case of pure advection $\epsilon = 0$!

In this case, as explained in [Lecture → Section 10.2.2.1], the **upwind quadrature** discretization of (10.8.1) is a stabilized Galerkin method based on the lowest-order Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$ and leading to the discrete variational problem:

$$u_h \in \mathcal{S}_1^0(\mathcal{M}) , \quad u_h|_{\Gamma_{\rm in}} = g_h : \quad \mathsf{a}_{h,\rm uw}(u_h, w_h) = 0 \quad \forall w_h \in \mathcal{S}^0_{1,\Gamma_{\rm in}}(\mathcal{M}) , \tag{10.8.4}$$

$$\mathsf{a}_{h,\rm uw}(u_h, w_h) := \sum_{\boldsymbol{p}\in\mathcal{V}(\mathcal{M})} \left(\frac{1}{3}\sum_{K\in\mathcal{U}_{\boldsymbol{p}}} |K|\right) \cdot \mathbf{v}(\boldsymbol{p}) \cdot \mathbf{grad}\, u_h|_{K\in\mathcal{K}_{\rm uw}(\boldsymbol{p})}(\boldsymbol{p})\, w_h(\boldsymbol{p}) , \tag{10.8.5}$$

where

- the inflow boundary part $\Gamma_{\rm in}$ is defined as ($\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ the exterior unit normal vector field)

$$\Gamma_{\rm in} := \{\boldsymbol{x} \in \partial\Omega : \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\} , \qquad \text{[Lecture → (10.2.1.11)]}$$

- g_h is a piecewise linear interpolant of the boundary data g
- $\mathcal{S}^0_{1,\Gamma_{\rm in}}(\mathcal{M}) := \left\{ w_h \in \mathcal{S}_1^0(\mathcal{M}) : \ w_h|_{\Gamma_{\rm in}} = 0 \right\}$
- $\mathcal{V}(\mathcal{M}) = \{\boldsymbol{p}^1, \ldots, \boldsymbol{p}^N\}$ is the set of nodes of the mesh $\mathcal{M}$,
- $\mathcal{U}_{\boldsymbol{p}}$ is the collection of mesh cells having $\boldsymbol{p}$ as a vertex,
- $\mathcal{K}_{\rm uw}(\boldsymbol{p})$ is the set of cells adjacent to $\boldsymbol{p} \in \mathcal{V}(\mathcal{M})$ and located in *upwind direction*

$$\mathcal{K}_{\rm uw}(\boldsymbol{p}) := \{K \in \mathcal{M} : \ \boldsymbol{p} \in \overline{K},\ \exists s > 0 : \ \boldsymbol{p} - s\mathbf{v}(\boldsymbol{p}) \in \overline{K}\} , \quad \overline{K} \text{ the closure of } K , \tag{10.8.6}$$

- and $\mathbf{grad}\, u_h|_{K\in\mathcal{K}_{\rm uw}(\boldsymbol{p})}$ stands for the gradient of u_h on some arbitrarily selected cell $K \in \mathcal{K}_{\rm uw}(\boldsymbol{p})$, $\boldsymbol{p} \in \mathcal{V}(\mathcal{M})$.

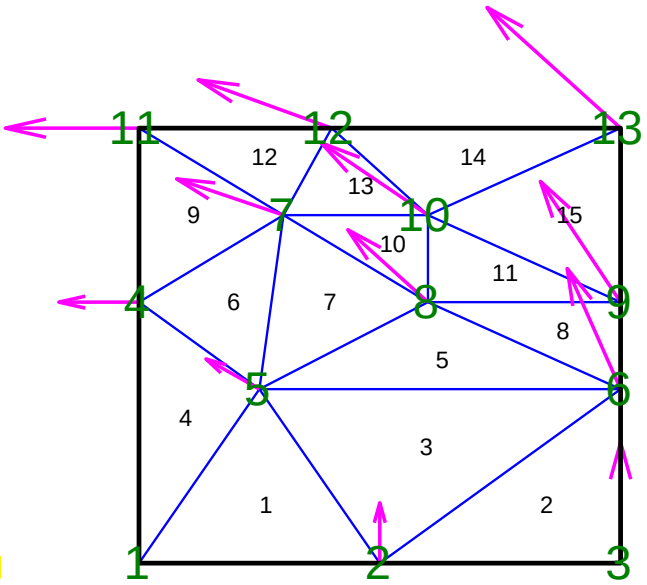


Fig. 169

For the remainder of this problem we consider the triangular mesh $\mathcal{M}$ as depicted in Fig. 169 beside.

The numbering of the 13 nodes $\{\boldsymbol{p}^1, \ldots, \boldsymbol{p}^{13}\}$ of the mesh is given through large green numbers on top of them.

The numbers of the 15 cells $\{K_1 \ldots, K_{15}\}$ of the mesh are printed in black inside the cells.

$\longrightarrow \hat{=} \mathbf{v}(\boldsymbol{p})$ for $\boldsymbol{p} \in \mathcal{V}(\mathcal{M})$

(10-8.b) ⚀ (10 min.) Using the numbering given in Fig. 169 determine the nodes of the mesh located on the (closure of) the inflow boundary part $\Gamma_{\rm in}$:

$$\left\{ i \in \{1, \ldots, 13\} : \ \boldsymbol{p}^i \in \overline{\Gamma}_{\rm in} \right\} = \left\{ \qquad \right\} .$$

SOLUTION for (10-8.b) → 10-8-2-0:upwqs2.pdf ▲

(10-8.c) ⚄ (10 min.) Find the cell(s) located in the upwind direction with respect to node ♯7.

$$\left\{ i : \ K_i \in \mathcal{K}_{\mathrm{uw}}(\boldsymbol{p}^7) \right\} = \left\{ \qquad \right\} .$$

SOLUTION for (10-8.c) → 10-8-3-0:upwqs3.pdf ▲

(10-8.d) ⚄ (10 min.) [depends on Sub-problem (10-8.c)]

Let $\mathbf{B}$ be the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix for $\mathsf{a}_{h,\mathrm{uw}}$ as defined in (10.8.5) and based on the standard nodal basis functions (tent functions) of $\mathcal{S}_1^0(\mathcal{M})$. The rows and columns of $\mathbf{B}$ are indexed according to the numbering of the mesh nodes as specified in Fig. 169. Where are the non-zero entries in its 7-th row?

$$\left\{ i \in \{1,\ldots,13\} : \ (\mathbf{B})_{7,i} \neq 0 \right\} = \left\{ \qquad \right\} .$$

SOLUTION for (10-8.d) → 10-8-4-0:upwqs4.pdf ▲

(10-8.e) ⚅ (35 min.) [depends on Sub-problem (10-8.d)]

Let $\mathcal{V}_+(\mathcal{M})$ be the mesh nodes not located on $\overline{\Gamma}_{\mathrm{in}}$. Write $\mathbf{B}_+$ for that square block of the $\mathcal{S}_1^0(\mathcal{M})$-Teleworking matrix of $\mathsf{a}_{h,\mathrm{uw}}$ associated with those nodes.

Give an ordering of the nodes in $\mathcal{V}_+(\mathcal{M})$ such that $\mathbf{B}^+$ is an *upper triangular* matrix.

$$\text{ordering} = \left(\ , \ , \ , \ , \ , \ , \ \right) .$$

(This ordering should be given as a sequence of node numbers according to the numbering scheme of Fig. 169.)

SOLUTION for (10-8.e) → 10-8-5-0:upwqs5.pdf

▲

End Problem 10-8, 75 min.

Problem 10-9: Potential Flow

The vortex-free laminar flow of an inviscid fluid can be modeled by means of a Dirichlet boundary value problem for the Laplace operator. In problem we study its finite-element solution on a tensor-product mesh and the computation of streamlines.

This problem adopts the finite-difference perspective of [Lecture → Section 4.1] and also requires knowledge about explicit adaptive time integration discussed in [Lecture → Section 6.5.3]. This problem requires familiarity with EIGEN.
▷ problem name/problem code folder: PotentialFlow

The irrotational laminar flow of an inviscid fluid in a bounded region $\Omega \subset \mathbb{R}^2$ of space[1] can be modeled by the following Dirichlet boundary value problem:

$$\Delta u = 0 \quad \text{in} \quad \Omega \quad , \quad u = g \quad \text{on} \quad \partial\Omega \, , \tag{10.9.1}$$

with given Dirichlet data $g \in C^0(\partial\Omega)$. Then the **fluid velocity** $\mathbf{v} : \Omega \to \mathbb{R}^2$ can be recovered as

$$\mathbf{v}(\boldsymbol{x}) := \mathbf{grad}\, u(\boldsymbol{x}) \, , \quad \boldsymbol{x} \in \Omega \, . \tag{10.9.2}$$

In the sequel we consider only the unit-square domain $\Omega =]0,1[^2$ and approximately solve (10.9.1) by means of a Galerkin finite-element discretization based on **bilinear Lagrangian finite elements** [Lecture → Ex. 2.6.2.1] on a uniform (quadrilateral) tensor-product mesh $\mathcal{M}$ with $M+1$ cells in each direction, $M \in \mathbb{N}$, and meshwidth $h := (M+1)^{-1}$.

We employ a lexikographic numbering of the *interior* nodes/vertices of the mesh $\mathcal{M}$ as depicted in the figure (numbering from 1!). ▷

This means that we number the interior nodes first from left to right and then from bottom to top, assigning them numbers $\in \{1, 2, \ldots, M^2\}$. So the bottom row of interior nodes has the numbers $1, 2, \ldots, M$, the top row $M(M-1)+1, \ldots, M^2$.

The same numbering is used for the standard global shape functions of the bilinear Lagrangian finite-element space $\mathcal{S}^0_{1,0}(\mathcal{M}) \subset H^1_0(\Omega)$. Those are associated with the nodes of the mesh.

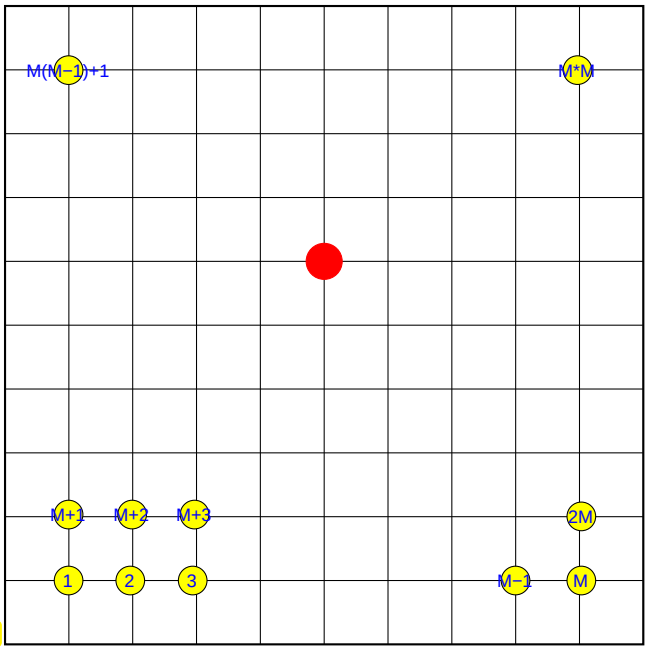


Fig. 171

Eventually, the finite-element Galerkin discretization yields a linear system of equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$ for the vector $\vec{\mu}$ of *unknown* basis expansion coefficients for the finite-element solution.

Hint. Throughout this problem, without verification, you may use that the element matrix [Lecture → Def. 2.7.4.5] for bilinear Lagrangian finite elements, the bilinear form $(u,v) \mapsto \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$, and *any* square cell K is given by

[1] Translation invariance allows the reduction of the fluid flow problem to two spatial dimensions.

$$\mathbf{A}_K = \frac{1}{6}\begin{bmatrix} 4 & -1 & -2 & -1 \\ -1 & 4 & -1 & -2 \\ -2 & -1 & 4 & -1 \\ -1 & -2 & -1 & 4 \end{bmatrix}, \qquad (10.9.3)$$

when using the numbering of the vertices, and, consequently, of the local shape functions, as indicated beside, see also [Lecture → (2.6.2.3)].

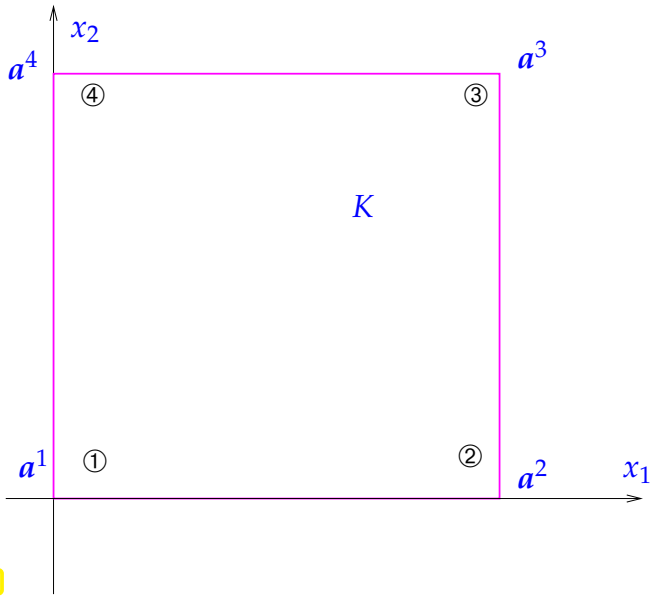


Fig. 172

(10-9.a) ⚀ (6 min.) What is the size of the matrix $\mathbf{A}$ and of the right-hand side vector $\vec{\varphi}$?

$$\mathbf{A} \in \mathbb{R}^{\square,\square}, \quad \vec{\varphi} \in \mathbb{R}^{\square}.$$

SOLUTION for (10-9.a) → 10-9-1-0:.pdf ▲

(10-9.b) ⚁ (10 min.) What is the minimal and maximal number of non-zero entries in a row of $\mathbf{A}$?

$$\min_j \sharp\{i : (\mathbf{A})_{i,j} \neq 0\} = \square,$$
$$\max_j \sharp\{i : (\mathbf{A})_{i,j} \neq 0\} = \square,$$

SOLUTION for (10-9.b) → 10-9-2-0:msts2.pdf ▲

(10-9.c) ⚁ (15 min.) Describe the row of the matrix $\mathbf{A}$ belonging to the node marked with • in Fig. 171 in **stencil notation** as introduced in [Lecture → § 4.1.2.10].

$$\text{Stencil} = \begin{bmatrix} \square & \square & \square \\ \square & \square & \square \\ \square & \square & \square \end{bmatrix}_h .$$

SOLUTION for (10-9.c) → 10-9-3-0:pfa3.pdf ▲

(10-9.d) ⚂ (24 min.) [depends on Sub-problem (10-9.c)]

The following code implements a C++ function

```
Eigen::SparseMatrix<double> initializeA(unsigned int M);
```

that is supposed to return an EIGEN **SparseMatrix** object that contains the matrix $\mathbf{A}$. The parameter `M` passes the resolution of the underlying tensor-product mesh as described above.

Fill in the blanks with valid C++ code so that the function matches its specification.

C++ code 10.9.5: Initialization of sparse Galerkin matrix $\mathbf{A}$

```
1  Eigen::SparseMatrix<double> initializeA(unsigned int M) {
2    Eigen::SparseMatrix<double> A(          ,          );
3    // We already know that the matrix has a small maximal number of
4    // non-zero entries per row and column.  This information is
5    // passed to Eigen via the reserve() member function.
6    A.reserve(Eigen::VectorXi::Constant(          ,          ));
7    for (int i = 0; i < M; ++i) {
8      for (int j = 0; j < M; ++j) {
9        const int k = i * M + j;
10       A.insert(          ,          ) =          ;
11       if (i > 0 && j > 0) A.insert(          ,          ) =          ;
12       if (i > 0) A.insert(          ,          ) =          ;
13       if (i > 0 && j < M - 1) A.insert(          ,          ) =          ;
14       if (j > 0) A.insert(          ,          ) =          ;
15       if (j < M - 1) A.insert(          ,          ) =          ;
16       if (i < M - 1 && j > 0) A.insert(          ,          ) =          ;
17       if (i < M - 1) A.insert(          ,          ) =          ;
18       if (i < M - 1 && j < M - 1) A.insert(          ,          ) =
             ;
19     }
20   }
21   A.makeCompressed();
22   return A;
23 }
```

HIDDEN HINT 1 for (10-9.d) → 10-9-4-0:pfhc1.pdf

SOLUTION for (10-9.d) → 10-9-4-1:pfsp4s.pdf ▲

(10-9.e) ⚄ (30 min.) [depends on Sub-problem (10-9.c)]

The following code shows an implementation of the C++ function

```
Eigen::VectorXd initializeRHSVector(const
   std::function<double(double,double)> &g, unsigned int M);
```

that provides the right-hand-side vector $\vec{\varphi}$. The argument `g` passes a functor with signature `std::function<double(double, double)>` that takes the two components x_1, x_2 of a position vector $\boldsymbol{x} = [x_1, x_2]^\top \in \partial\Omega$ as arguments and returns $g(\boldsymbol{x}) \in \mathbb{R}$. The second argument `M` is the mesh parameter.

C++ code 10.9.7: Initialization of right hand side vector $\vec{\varphi}$

```
Eigen::VectorXd initializeRHSVector(const std::function<double(double,double)> &g,
    unsigned int M) {
  const double h = 1.0 / (M + 1);   // Mesh width
  Eigen::VectorXd phi = Eigen::VectorXd::Zero(          );
  // Four corner points
  phi[0] =                                                ;
  phi[M - 1] =                                            ;
  phi[M * (M - 1)] =                                      ;
  phi[M * M - 1] =                                        ;
  for (unsigned int l = 1; l < M - 1; ++l) {
    phi[l] =                                              ;
    phi[M * l] =                                          ;
    phi[M * (l + 1) - 1] =                                ;
    phi[M * (M - 1) + l] =                                ;
  }
  return phi;
}
```

SOLUTION for (10-9.e) → 10-9-5-0:.pdf ▲

A **streamline** is a curve $t \in I \mapsto \Omega$, $I \subset \mathbb{R}$ an interval, which is the solution of the ODE

$$\dot{\mathbf{y}} = \mathbf{v}(\mathbf{y}) = \mathbf{grad}\, u(\mathbf{y}) \,. \tag{10.9.10}$$

The state space of (10.9.10) is $\Omega \subset \mathbb{R}^2$.

(10-9.f) ⚂ (10 min.) Let $\mathbf{y} : I \to \Omega$ be a streamline and let the **potential** $u : \Omega \to \mathbb{R}$ be a solution of the boundary value problem (10.9.1). How does the potential behave along a streamline, that is, we are interested in properties of the function $t \in I \mapsto u(\mathbf{y}(t))$.

Along a streamline u is .

SOLUTION for (10-9.f) → 10-9-6-0:.pdf ▲

End Problem 10-9 , 95 min.

Problem 10-10: Least-squares Galerkin FEM for Scalar Advection-Reaction BVP

A simple stable discretization for stationary scalar advection-reaction boundary value problems is the L^2-least-squares Galerkin approach, which is based on the minimization of the $L^2(\Omega)$-norm of the equation residual [Wid09, Sect. 2.1]. In this project we examine the resulting linear variational problem and implement its finite element Galerkin discretization.

This project assumes familiarity with the theory of [Lecture → Section 1.3.1], [Lecture → Section 1.4.1], and LEHRFEM++.
▷ problem name/problem code folder: LeastSquaresAdvection

On a two-dimensional polygonal bounded domain $\Omega \subset \mathbb{R}^2$ we consider the stationary convection-diffusion-reaction boundary value problem ($\epsilon > 0$)

$$-\epsilon\Delta u + \mathbf{v}\cdot\mathbf{grad}\, u + \kappa(\boldsymbol{x})u = 0 \quad \text{in} \quad \Omega \quad , \quad u = g \quad \text{on} \quad \partial\Omega \, , \tag{10.10.1}$$

with given constant velocity $\mathbf{v} \in \mathbb{R}^2 \setminus \{\mathbf{0}\}$, non-negative reaction coefficient $\kappa \in C^0(\overline{\Omega})$, and Dirichlet data $g \in C^0(\partial\Omega)$.

(10-10.a) ⚀ (20 min.) We look at the special setting

- $\Omega =]0,1[^2$ (unit square domain),
- $\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\kappa \equiv 0$, and
- the boundary data $g : \partial\Omega \to \mathbb{R}$ are nonzero only on the bottom side,

$$g(\boldsymbol{x}) = \begin{cases} \sin^2(\pi x_1) & \text{for} \quad x_2 = 0 \, , \\ 0 & \text{elsewhere on } \partial\Omega \, , \end{cases} \quad \boldsymbol{x} \in \partial\Omega \, . \tag{10.10.2}$$

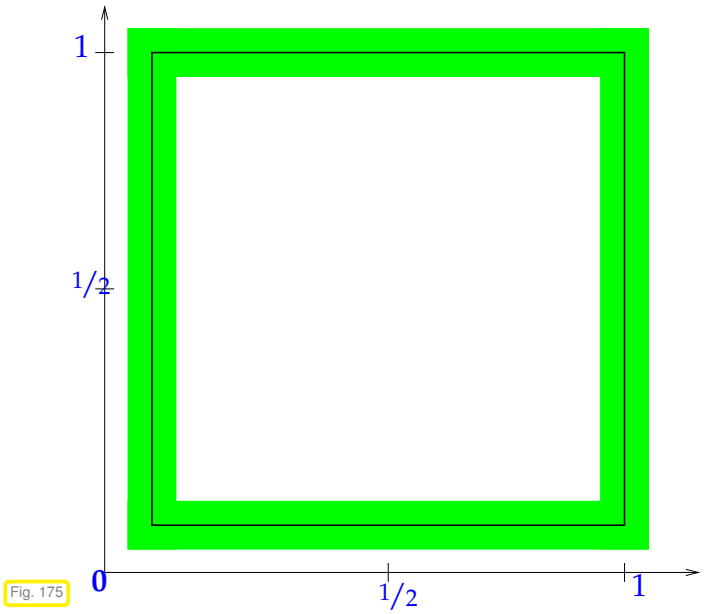
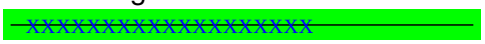


Fig. 175

In the figure beside mark those parts of the boundary $\partial\Omega$, where you expect **boundary layers** to emerge in the solution of (10.10.1) for $\epsilon \to 0$.

Your marking should look like this:



SOLUTION for (10-10.a) → 10-10-1-0:lsas1.pdf

▲

In the remainder of this problem we confine ourselves to the case $\epsilon = 0$ of pure advection-reaction!

As we have seen in [Lecture → § 10.2.1.10], in the case $\epsilon = 0$, Dirichlet boundary conditions can be imposed only on the inflow boundary part

$$\Gamma_{\text{in}} := \{\boldsymbol{x} \in \partial\Omega : \ \mathbf{v}\cdot\boldsymbol{n}(\boldsymbol{x}) < 0\} \, , \tag{10.10.3}$$

which yields the well-posed boundary value problem:

$$\mathbf{v}\cdot\mathbf{grad}\,u+\kappa(\boldsymbol{x})u=0 \quad \text{in} \quad \Omega \quad , \quad u=g \quad \text{on} \quad \Gamma_{\text{in}}\,. \tag{10.10.4}$$

(10-10.b) ⚄ (20 min.) Using the **method of characteristics** give a closed-form formula for the solution of (10.10.4).

$$u(\boldsymbol{x}) = \boxed{} \quad , \quad \boldsymbol{x}\in\Omega$$

HIDDEN HINT 1 for (10-10.b) → 10-10-2-0:lsah2b.pdf

SOLUTION for (10-10.b) → 10-10-2-1:lsas2.pdf

▲

Let $\widetilde{g}\in H^1(\Omega)$ denote an extension of the Dirichlet data g into the interior of Ω, that is, $\widetilde{g}|_{\Gamma_{\text{in}}}=g$ on Γ_{in}. Then the **least-squares variational formulation** of (10.10.4) reads:

$$u\in\widetilde{g}+V_0\text{:}$$
$$\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\,u(\boldsymbol{x})+\kappa(\boldsymbol{x})u(\boldsymbol{x}))(\mathbf{v}\cdot\mathbf{grad}\,w(\boldsymbol{x})+\kappa(\boldsymbol{x})w(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}=0 \quad \forall w\in V_0\,, \tag{10.10.9}$$

with a suitable function space V_0.

(10-10.c) ⚀ (10 min.) Every solution of the linear variational problem (10.10.9) is the minimizer of a quadratic functional J. What is that quadratic functional?

$$u \text{ solves (10.10.9)} \quad \Leftrightarrow \quad u=\operatorname*{argmin}_{w\in\widetilde{g}+V_0} J(w)\,,$$

with $J(w) := \boxed{}$.

SOLUTION for (10-10.c) → 10-10-3-0:.pdf

▲

(10-10.d) ⚂ (15 min.) What is the largest suitable function space V_0 such that the variational problem (10.10.9) is well-defined and consistent with (10.10.4) for any $\kappa\in C^0(\overline{\Omega})$.

$$V_0 := \left\{ w\in L^2(\Omega) : \boxed{} \right\}.$$

The function space V_0 may depend on $\mathbf{v}$, but must not depend on κ.

HIDDEN HINT 1 for (10-10.d) → 10-10-4-0:lsah3.pdf

SOLUTION for (10-10.d) → 10-10-4-1:lsas3.pdf

▲

(10-10.e) ⚄ (20 min.) Fill the blanks in

$$\int_\Omega \mathbf{v}\cdot\mathbf{grad}\,u(\boldsymbol{x})\,\kappa(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} =$$

$$\int_\Omega u^2(\boldsymbol{x})\,\boxed{}\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u^2(\boldsymbol{x})\,\boxed{}\,\mathrm{d}S(\boldsymbol{x})$$

so that the resulting formula holds true for every $u \in C^1(\overline{\Omega})$, $\kappa \in C^1(\overline{\Omega})$, $\mathbf{v} \in \mathbb{R}^2$.

This identity is important for the analysis of the bilinear form a from (**??**).

HIDDEN HINT 1 for (10-10.e) → 10-10-5-0:las3ah.pdf

SOLUTION for (10-10.e) → 10-10-5-1:.pdf

▲

The goal is to implement a Galerkin finite element discretization of the advection-reaction least-squares variational problem (10.10.9) in LEHRFEM++. As trial and test space suitable subspaces of the Lagrangian finite element space $\mathcal{S}_p^0(\mathcal{M})$ on a hybrid 2D mesh $\mathcal{M}$ should be used. So the task boils down to solving the discrete variational problem: seek $u_h \in \widetilde{g}_h + V_{0,h}$ such that

$$\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\,u_h(\boldsymbol{x}) + \kappa(\boldsymbol{x})u_h(\boldsymbol{x}))(\mathbf{v}\cdot\mathbf{grad}\,w_h(\boldsymbol{x}) + \kappa(\boldsymbol{x})w_h(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall w \in V_{0,h} := \mathcal{S}_p^0(\mathcal{M}) \cap V_0\,. \tag{10.10.14}$$

Here, $\widetilde{g}_h$ denotes a function $\in \mathcal{S}_p^0(\mathcal{M})$ such that

$$\widetilde{g}_h|_{\Gamma_{\mathrm{in}}} = \mathrm{I}_p\widetilde{g}|_{\Gamma_{\mathrm{in}}}\,,$$

where I_p is the standard nodal interpolation operator for the Lagrangian finite element space $\mathcal{S}_p^0(\mathcal{M})$.

In LEHRFEM++ the functionality of parametric Lagrangian finite element spaces can be accessed through the abstract base class **lf::uscalfe::UniformScalarFESpace** and its derived classes, refer to [Lecture → § 2.8.3.28] for details.

A key ingredient of the implementation in LEHRFEM++ will be a suitable **ENTITY_MATRIX_PROVIDER** type, whose definition is listed next and can also be found in `leastsquaresadvection.h`:

```
template <lf::mesh::utils::MeshFunction REACTION_COEFF>
class LSQAdvectionMatrixProvider {
 public:
  using Scalar = double;
  using ElemMat = Eigen::Matrix<Scalar, Eigen::Dynamic, Eigen::Dynamic>;
  // Standard constructors
  LSQAdvectionMatrixProvider(const LSQAdvectionMatrixProvider &) = delete;
  LSQAdvectionMatrixProvider(LSQAdvectionMatrixProvider &&) noexcept =
      default;
  LSQAdvectionMatrixProvider &operator=(const LSQAdvectionMatrixProvider
      &) =
       delete;
  LSQAdvectionMatrixProvider &operator=(LSQAdvectionMatrixProvider &&) =
      delete;
  virtual ~LSQAdvectionMatrixProvider() = default;
  // Constructor, initializes data members and cell-indepedent quantities
  LSQAdvectionMatrixProvider(
      std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>>
          fe_space,
      Eigen::Vector2d velocity, REACTION_COEFF kappa);
```

```
  // Standard interface for ENTITY MATRIX PROVIDERS
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) { return true; }
  ElemMat Eval(const lf::mesh::Entity &cell);

 private:
  REACTION_COEFF kappa_;       // Reaction coefficient κ = κ(x)
  Eigen::Vector2d velocity_;   // Velocity vector v
  std::array<
    lf::uscalfe::PrecomputedScalarReferenceFiniteElement<double>, 5>
      fe_precomp_;
};
```

The non-trivial constructor (implemented in `leastsquaresadvection.h`) initializes the `fe_precomp_` data member, which selects a local quadrature rule and supplies the values and gradients of the local shape functions in the quadrature nodes on the reference element, *cf.* [Lecture → § 2.8.3.15].

(10-10.f) ⚀ (60 min.) Implement the `Eval()` member function of the class **LSQAdvectionMatrixProvider**.

HIDDEN HINT 1 for (10-10.f) → `10-10-6-0:lsah4.pdf`

SOLUTION for (10-10.f) → `10-10-6-1:.pdf`

▲

(10-10.g) ⚀ (30 min.) In the file `leastsquaresadvection.h` complete the code for the C++ function

```
template <lf::mesh::utils::MeshFunction REACTION_COEFF,
          lf::mesh::utils::MeshFunction GFUNCTION>
Eigen::VectorXd solveAdvectionDirichletBVP(
    std::shared_ptr<const
      lf::uscalfe::UniformScalarFESpace<double>> fe_space,
    Eigen::Vector2d velocity, const REACTION_COEFF &kappa, const
      GFUNCTION &g);
```

It is supposed to return the basis expansion coefficient vector for the solution of (10.10.14) in the standard nodal basis of $\mathcal{S}_p^0(\mathcal{M})$ used by LEHRFEM++.

The parameter `fe_space` supplies complete information about the (Lagrangian) finite element space (mesh and local→global index mapping), `kappa` passes the coefficient κ as a **MeshFunction** object, and `g` does the same for the *extended* boundary data $\widetilde{g}$.

All that is missing in the implementation is the treatment of the Dirichlet boundary conditions.

HIDDEN HINT 1 for (10-10.g) → `10-10-7-0:lsah51.pdf`

HIDDEN HINT 2 for (10-10.g) → `10-10-7-1:lsah5.pdf`

SOLUTION for (10-10.g) → `10-10-7-2:lsasol5.pdf`

▲

(10-10.h) ⚀ (30 min.) [depends on Sub-problem (10-10.b)]

The following tables ($N := \dim \mathcal{S}_p^0(\mathcal{M})$, produced by `testCVGLSQAdvectionReaction()` in `leastsquaresadvection.h`) list the $L^2(\Omega)$-norms of the discretization errors for the least-squares finite element discretization (10.10.14) of the boundary value problem (10.10.4) for the unit

square domain $\Omega =]0,1[^2$, on a sequence of 2D triangular meshes created by *uniform regular refinement* and the following two settings:

(I) $\kappa \equiv 1$, $\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $g(\boldsymbol{x}) = \begin{cases} \sin^2(\pi x_1) & \text{for } x_2 = 0\,, \\ 0 & \text{elsewhere on } \partial\Omega\,, \end{cases}$ $\boldsymbol{x} \in \partial\Omega$,

$p=1$		$p=2$		$p=3$	
N	$\|u-u_h\|_{L^2(\Omega)}$	N	$\|u-u_h\|_{L^2(\Omega)}$	N	$\|u-u_h\|_{L^2(\Omega)}$
13	0.488	40	0.154	82	0.0252
40	0.309	139	0.024	298	0.00223
139	0.128	517	0.00473	1135	0.000103
517	0.0474	1993	0.000529	4429	5.39e-06
1993	0.0169	7825	5.95e-05	17497	3.26e-07
7825	0.00534	31009	7.51e-06	69553	2.05e-08
31009	0.00152	123457	1.01e-06	277345	1.31e-09

(II) $\kappa \equiv 0$, $\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, manufactured solution $u(\boldsymbol{x}) = \cos(\mathbf{v}^\perp \cdot \boldsymbol{x})$,

$p=1$		$p=2$		$p=3$	
N	$\|u-u_h\|_{L^2(\Omega)}$	N	$\|u-u_h\|_{L^2(\Omega)}$	N	$\|u-u_h\|_{L^2(\Omega)}$
13	0.185	40	0.0901	82	0.0686
40	0.124	139	0.0421	298	0.00732
139	0.075	517	0.00985	1135	0.00122
517	0.0381	1993	0.00193	4429	0.000348
1993	0.0164	7825	0.000494	17497	9.61e-05
7825	0.00603	31009	0.000154	69553	2.66e-05
31009	0.00202	123457	4.91e-05	277345	7.33e-06

In both cases,

(i) describe qualitatively and quantitatively the empiric asymptotic convergence of the $L^2(\Omega)$-norm of the finite element discretization error, and

(ii) explain your observations.

SOLUTION for (10-10.h) → `10-10-8-0:.pdf` ▲

References

[Wid09] G. Widmer. "Sparse Finite Elements for Radiative Transfer". ETH Diss. Nr. 18420. ETH Zürich, 2009 (cit. on p. 405).

End Problem 10-10 , 205 min.

Chapter 11

Numerical Methods for Conservation Laws

Problem 11-1: Burgers Equation

In [Lecture → § 11.1.3.3] a one-dimensional scalar conservation law [Lecture → (11.1.3.4)], known as **Burgers Equation** was derived as a simple model for the directed frictionless flow of gas. This is a famous model problem for testing numerical methods for conservation laws. In this problem we will recast Burgers equation and also discretize the related Cauchy problem by means of a conservative finite volume method [Lecture → Section 11.3] with Godunov flux [Lecture → Section 11.3.4.4].

This problem requires the contents of [Lecture → Section 11.2] and [Lecture → Section 11.3]. Implementation in C++ based on EIGEN will be requested.
▷ problem name/problem code folder: BurgersEquation

In [Lecture → Section 11.1.3] the Cauchy problem for the Burgers equation [Lecture → (11.1.3.4)] was presented as a simple model for inertia driven gas flow in a straight pipe. The Burgers equation written as a scalar conservation law in 1D reads

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right) = 0 \quad \text{on} \quad \mathbb{R}\times]0,T[\tag{11.1.1}$$

Assuming smoothness of u, an equivalent formulation for $w = u^2$ is

$$\frac{\partial w}{\partial t} + \frac{\partial}{\partial x}\left(\frac{2}{3}w^{3/2}\right) = 0 \quad \text{on } \mathbb{R}]0,T[\,. \tag{11.1.2}$$

(11-1.a) ⚀ (5 min.) Write down the flux functions for both (11.1.1) and (11.1.2).

SOLUTION for (11-1.a) → 11-1-1-0:s0.pdf ▲

(11-1.b) ⚁ (10 min.) Show that, if $u \in C^1(\mathbb{R}\times]0,1[)$ solves (11.1.1) with initial condition $u(x,0) = u_0(x) > 0$ for all $x \in \mathbb{R}$. Then $w(x,t) := u(x,t)^2$ solves (11.1.2).

HIDDEN HINT 1 for (11-1.b) → 11-1-2-0:ah1.pdf

SOLUTION for (11-1.b) → 11-1-2-1:sa.pdf ▲

(11-1.c) ⚂ (5 min.) For initial data $w_0 > 0$ (11.1.2) has the features of a 1D conservation law with convex flux function. Why?

HIDDEN HINT 1 for (11-1.c) → 11-1-3-0:2h1.pdf

SOLUTION for (11-1.c) → 11-1-3-1:s2.pdf ▲

(11-1.d) ⚄ (15 min.) Determine *entropy solutions* of (11.1.1) and (11.1.2) for the Riemann problems with the following initial data (→ [Lecture → Section 11.2.5])

$$u_0(x) = \begin{cases} 2, & x < 0, \\ 1, & x > 0, \end{cases} \quad , \quad w_0(x) = \begin{cases} 4, & x < 0, \\ 1, & x > 0. \end{cases}$$

HIDDEN HINT 1 for (11-1.d) → 11-1-4-0:3h1.pdf

SOLUTION for (11-1.d) → 11-1-4-1:s4.pdf ▲

(11-1.e) ⚂ (20 min.) Now find the entropy solutions of (11.1.1) and (11.1.2) for the Riemann problems with

$$u_0(x) = \begin{cases} 1, & x < 0, \\ 2, & x > 0 \end{cases} \quad , \quad w_0(x) = \begin{cases} 1, & x < 0 \\ 4, & x > 0 \, . \end{cases}$$

HIDDEN HINT 1 for (11-1.e) → 11-1-5-0:h41.pdf

SOLUTION for (11-1.e) → 11-1-5-1:s4.pdf ▲

(11-1.f) ⚄ (10 min.) [depends on Sub-problem (11-1.d), Sub-problem (11-1.e)]

Compare the solutions of the two Riemann problems studied in subproblems Sub-problem (11-1.d)–Sub-problem (11-1.e). Comment on your findings in light of Sub-problem (11-1.b) and try to find an explanation for the baffling mismatch.

SOLUTION for (11-1.f) → 11-1-6-0:s5.pdf ▲

In the sequel we study the discretization of the Cauchy problem for (11.1.2) by means of a conservative finite volume method.

(11-1.g) ⚂ (15 min.) Derive a spatial semi-discretization of (11.1.2) using conservative finite differences [Lecture → Section 11.3.2] with the Godunov numerical flux

$$F_{\mathrm{GD}}(v,w) = \begin{cases} \min\limits_{v \le u \le w} f(u) & \text{, if } \quad v < w \, , \\ \max\limits_{w \le u \le v} f(u) & \text{, if } \quad w \le v \, , \end{cases} \qquad \text{[Lecture → (11.3.4.33)]}$$

see also [Lecture → (11.3.4.31)], on an equidistant spatial mesh with meshwidth h.

SOLUTION for (11-1.g) → 11-1-7-0:Sh1.pdf ▲

(11-1.h) ⚂ (30 min.) [depends on Sub-problem (11-1.g)]

In the file `burgersequation.cc` implement a C++ function

```
Eigen::VectorXd solveBurgersGodunov(
    double T,unsigned int N);
```

that solves (11.1.2) based on the semi-discretization developed in Sub-problem (11-1.g) using `N` equispaced nodes x_j in space on the interval $[-1,4]$, that is $x_0 = -1$ and $x_N = 4$. For temporal discretization use explicit Euler timestepping with timestep $\tau = h$, where h is the mesh width in space.

The function is supposed to return the "nodal" values (= averages over dual cells, [Lecture → Section 11.3.2]) of the solution at time `T` (or the closest possible time given the timestep τ). For computing the flux you may assume that the solution is zero outside this domain. Use the initial condition

$$w_0(x) = \begin{cases} \sin^2(\pi x), & 0 \le x \le 1 \\ 0, & \text{otherwise} \, , \end{cases} \tag{11.1.5}$$

and find $\vec{\mu}(0)$ by simply sampling w_0 at the nodes of the mesh. Outside $[-1,4]$ the solution is set to zero.

SOLUTION for (11-1.h) → 11-1-8-0:S2.pdf ▲

(11-1.i) ⚄ (20 min.) Call the C++ function `solveBurgersGodunov()` from `main()` (in the file `burgersequation_main.cc`) with $N = 100$, $T = 0.3$ and $N = 100$, $T = 3$.

Plot (linear interpolants of the nodal values of) the approximate solutions at these two final times. To that end you may write the solution to a file that is read by a suitable PYTHON or MATLAB script, or you summon PYTHON-based visualization right from the C++ code.

SOLUTION for (11-1.i) → 11-1-9-0:s3.pdf ▲

(11-1.j) ⚄ (20 min.) What is the structure of the solution w of the Cauchy problem in Sub-problem (11-1.h)?

HIDDEN HINT 1 for (11-1.j) → 11-1-10-0:S4h1.pdf

SOLUTION for (11-1.j) → 11-1-10-1:S4.pdf ▲

(11-1.k) ⚄ (20 min.) [depends on Sub-problem (11-1.h)]

We consider the scheme implmented in `solveBurgersGodunov()`. Let $\mu_j^{(k)}$ denote the nodal value of the fully discrete solution at x_j and after timestep k. Use the discrete error norm

$$e(t_k,h) = h\sum_{j=0}^{N+1}\left|\mu_j^{(k)} - w(x_j,t_k)\right|, \qquad k = 0,1,\ldots, \quad t_k := k\tau\,,$$

as an approximation of the time-dependent L^1-norm.

In the file `burgersequation.cc` code a C++ function

```
Eigen::Matrix<double, 3, 4> numexpBurgersGodunov();
```

that tabulates $e(0.3,h)$ and $e(3,h)$ as a function of the spatial meshwidth h for $h = 0.1, 0.05, 0.025, 0.001xb25$. As in Sub-problem (11-1.h) the timestep τ is tied to the spatial meshwidth by demanding $\tau = h$. Since we cannot solve this problem analytically, we are forced to use a *reference solution* in place of w, computed with a very dense mesh, namely $N = 3200$.

Extend the code in `burgersequation_main.cc` to store the result of the C++ function `numexpBurgersGodunov()` just implemented to a `.csv` file and produce a plot using the provided python script.

SOLUTION for (11-1.k) → 11-1-11-0:S5.pdf ▲

(11-1.l) ⚄ (15 min.) [depends on Sub-problem (11-1.k)]

Discuss qualitatively and quantitatively the asymptotic convergence of both $e(0.3,h)$ and $e(3,h)$ as $h \to 0$.

SOLUTION for (11-1.l) → 11-1-12-0:6as.pdf ▲

End Problem 11-1, 185 min.

Problem 11-2: Conservative finite-volume discretization based on Engquist-Osher numerical flux

In [Lecture → Section 11.3.2] the concept of a conservative finite-volume semi-discretization was introduced for 1D scalar conservation laws $\partial_t u + \partial_x f(u) = 0$, see [Lecture → (11.3.2.7)]. The only ingredient is the **numerical flux** (function). In this problem we will practise the construction of a conservative finite-volume scheme based on a special numerical flux proposed in [EO81].

This problem relies on [Lecture → Section 11.3.4], the beginning of [Lecture → Section 11.4], and [Lecture → Section 11.4.2]. A few sub-problems will involve implemenation in C++.
▷ problem name/problem code folder: EngquistOsherNumericalFlux

We consider the following Cauchy problem for a scalar conservation law with solution $u = u(x,t)$

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= 0 && \text{on } \mathbb{R}\times]0,T[\,. \\ u(x,0) &= u_0(x) && \text{in } \mathbb{R}, \end{aligned} \tag{11.2.1}$$

where $f : \mathbb{R} \mapsto \mathbb{R}$ is the flux function, which is supposed to be continuously differentiable.

The Cauchy problem (11.2.1) should be solved by a fully discrete conservative finite volume method on an equidistant mesh $\mathcal{M} = \{]x_{j-1}, x_j[: x_j = jh, j \in \mathbb{Z}\}$ with meshwidth $h > 0$ combined with explicit Euler timestepping, see [Lecture → (11.4.1.10)]. We rely on the **2-point Engquist-Osher numerical flux** defined as

$$F_{\mathrm{EO}}(v,w) := \tfrac{1}{2}(f(v) + f(w)) - \tfrac{1}{2}\int_v^w |f'(\xi)|\,\mathrm{d}\xi\ . \tag{11.2.2}$$

(11-2.a) ⚀ (10 min.) Appealing to [Lecture → Def. 11.3.3.5] show that the Engquist-Osher flux is consistent with the flux function f.

SOLUTION for (11-2.a) → 11-2-1-0:s0.pdf ▲

(11-2.b) ⚁ (10 min.) Show that the Engquist-Osher numerical flux F_{EO} from (11.2.2) is a monotone numerical flux in the sense of [Lecture → Def. 11.3.5.5].

HIDDEN HINT 1 for (11-2.b) → 11-2-2-0:hA.pdf

SOLUTION for (11-2.b) → 11-2-2-1:sA.pdf ▲

For the remainder of this problem consider the special choice

$$f(u) := \cosh(u) = \frac{1}{2}(e^u + e^{-u})\,, \quad u \in \mathbb{R}\,.$$

(11-2.c) ⚀ (10 min.) A crucial feature of the evolutions described by 1D scalar conservation laws like (11.2.1) is the finite speed of propagation that leads to so-called regions of influence and dependence, as discussed in [Lecture → Section 11.2.7].

Please, make yourself familiar with these notions and, in particular, try to understand why they are consequences of [Lecture → Thm. 11.2.7.3].

▲

(11-2.d) ⚁ (10 min.) If u_0 is supported in $[0,1]$ and $-1 \le u_0(x) \le 1$ for all $x \in \mathbb{R}$, find the maximal possible support of $u(\cdot,t)$ for time $t > 0$, where $u(x,t)$ solves (11.2.1).

HIDDEN HINT 1 for (11-2.d) → 11-2-4-0:1ah1.pdf

SOLUTION for (11-2.d) → 11-2-4-1:s1a.pdf ▲

(11-2.e) ⚁ (15 min.) In the file `engquistoshernumericalflux.cc` implement a C++ function

```
double EngquistOsherNumFlux(double v, double w);
```

that implements the 2-point Engquist-Osher numerical flux function F_{EO} according to (11.2.2) for the flux $f(u) := \cosh(u)$.

SOLUTION for (11-2.e) → 11-2-5-0:s3.pdf ▲

(11-2.f) ⚂ (15 min.) Try to understand the argument elaborated in [Lecture → Section 11.4.2] that the so-called CFL-condition as specified in [Lecture → Def. 11.4.2.11] is necessary for the convergence of a fully discrete finite volume scheme for a 1D scalar conservation law when we keep the timestep $\tau > 0$ proportional to the spatial meshwidth $h > 0$ and let both tend to zero. ▲

(11-2.g) ⚁ (15 min.) [depends on Sub-problem (11-2.d)]

Determine the CFL-condition introduced in [Lecture → Def. 11.4.2.11] for the conservative finite volume method for (11.2.1) introduced above, if it is known that $A \le u_0(x) \le B$ for all $x \in \mathbb{R}$, with $A, B \in \mathbb{R}$, $A < B$.

SOLUTION for (11-2.g) → 11-2-7-0:s3.pdf ▲

(11-2.h) ⚂ (15 min.) [depends on Sub-problem (11-2.g)]

Write a C++ function (in the file `engquistoshernumericalflux.cc`)

```
Eigen::VectorXd solveCP(double a, double b,
                        Eigen::VectorXd u0, double T);
```

that solves the Cauchy problem (11.2.1) using the conservative finite volume method described above and *explicit Euler* timestepping to final time $T > 0$. The computations should be done on the spatial interval $[a,b]$ on a grid with nodes $x_j = a + \frac{b-a}{N}(j - \frac{1}{2})$, $j = 1,\dots,N$. The vector `u0` passes the cell averages of u_0 and the scalar `T` the final time. Constant continuation of u_0 outside $[a,b]$ is assumed. The returned vector contains the approximate cell averages of $u(\cdot,T)$. Use the maximal timestep possible according to the CFL-condition, see Sub-problem (11-2.g).

SOLUTION for (11-2.h) → 11-2-8-0:s3a.pdf ▲

(11-2.i) ⚁ (5 min.) Use the function `solveCP()` to solve (11.2.1) for initial data

$$u_0(x) = \begin{cases} 1 & \text{for } 0 < x \le 1, \\ -1 & \text{elsewhere,} \end{cases}$$

and final time $T = 1$. Use $[-1.2, 2.2]$ as computational interval and $N = 100$. The cell avarages of u_0 can be approximated by its values at the cell centers. To that end, extend the `main()` function so that it outputs the approximate dual cell averages at final time to a file `ufinal.csv`. Then use the PYTHON script `plot_solution.py` to plot the numerical approximation of $u(x,T)$ on $[-1.2, 2.2]$. This will create from `ufinal.csv` a plot called `ufinal.eps`. Alternatively, you may write your own MATLAB or PYTHON script to create the plot.

SOLUTION for (11-2.i) → 11-2-9-0:s5.pdf ▲

References

[EO81] Björn Engquist and Stanley Osher. "One-sided difference approximations for nonlinear conservation laws". In: *Math. Comp.* 36.154 (1981), pp. 321–351. DOI: 10.2307/2007646 (cit. on p. 413).

End Problem 11-2 , 105 min.

Problem 11-3: Conservative finite-volume discretization based on Godunov numerical flux

This problem studies the full discretization of the Cauchy problem for a 1D scalar conservation law by means of a conservative finite volume method on an equidistant spatial mesh.

You need to know the material of [Lecture → Section 11.3.4], the beginning of [Lecture → Section 11.4], and [Lecture → Section 11.4.2] to tackle this problem. Implementation in C++ based on EIGEN will be requested.
▷ problem name/problem code folder: FiniteVolumeSineConsLaw

We consider the Cauchy problem on $\mathbb{R}\times]0,T[$ for the scalar conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\sin(\pi u) = 0 \quad \text{on} \quad \mathbb{R}\times[0,T]\,, \tag{11.3.1}$$

with initial condition $u(x,0) = u_0(x)$, $x \in \mathbb{R}$, satisfying

$$0 \le u_0(x) \le 1, \quad x \in \mathbb{R}. \tag{11.3.2}$$

Be aware that in this example the flux function $f(u) := \sin(\pi u)$ is *concave* on the range of possible states.

(11-3.a) ⚀ (15 min.) Determine the entropy solutions of the Riemann problems [Lecture → Section 11.2.5] with

$$\text{(i)}\quad u_0(x) = \begin{cases} 0 & \text{for } x \le 0, \\ 1 & \text{for } x > 0, \end{cases}, \qquad \text{(ii)}\quad u_0(x) = \begin{cases} 1 & \text{for } x \le 0, \\ 0 & \text{for } x > 0, \end{cases}$$

SOLUTION for (11-3.a) → 11-3-1-0:s1.pdf ▲

(11-3.b) ⚁ (15 min.) Assume $\operatorname{supp}(u_0) \subset [0,1]$ and (11.3.2). Describe the maximal support of the solution u of (11.3.1) in the x-t-plane.

SOLUTION for (11-3.b) → 11-3-2-0:s2.pdf ▲

(11-3.c) ⚂ (20 min.) In the file `finitevolumesineconslaw.cc` implement a C++ function

```
double sineGodFlux(double v, double w);
```

that implements the Godunov numerical flux $F_{\mathrm{GD}}(u,v) = f(u^{\downarrow}(v,w))$ as introduced in [Lecture → Section 11.3.4.4] for (11.3.1) under the assumption $0 \le v,w \le 1$. The arguments `v` and `w` pass the left and right state, respectively.

HIDDEN HINT 1 for (11-3.c) → 11-3-3-0:s3h1.pdf

SOLUTION for (11-3.c) → 11-3-3-1:s3.pdf ▲

For the spatial semi-discretization of (11.3.1) we use a conservative finite volume method on an equidistant spatial mesh covering the interval $[-6,6]$. We rely on the Godunov flux and assume that $u = 0$ outside this interval for all relevant times.

(11-3.d) ⚁ (20 min.) [depends on Sub-problem (11-3.c)]

Using EIGEN write a C++ function (in the file `finitevolumesineconslaw.cc`)

```
Eigen::VectorXd sineClawRhs(const Eigen::VectorXd &mu)
```

that realizes the right hand side of the method-of-lines ODE for a conservative finite volume discretization of (11.3.1), namely a truncated version of the expression

$$\mathcal{L}_h(\vec{\mu}) := \left[-\frac{1}{h}\big(F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big)\right]_{j\in\mathbb{Z}} ,$$

arising from the finite volume semi-discretization of (11.3.1) on a mesh of $[-6,6]$ with N equal cells, taking an N-vector `mu` with cell averages. Here, h is the mesh size and F is the Godunov numerical flux function. The zero state $u = 0$ outside $[-6,6]$ is assumed.

SOLUTION for (11-3.d) → `11-3-4-0:s3a.pdf` ▲

For timestepping we use the explicit trapezoidal rule, a 2-stage second-order explicit Runge-Kutta single step method characterized by the Butcher tableau [Lecture → § 9.2.6.30]

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array} . \tag{11.3.5}$$

(11-3.e) ⚃ (20 min.) [depends on Sub-problem (11-3.d)]

In the file `finitevolumesineconslaw.h` code a C++ function

```
template <typename RHSFUNCTOR>
  Eigen::VectorXd explTrpzTimestepping(
    RHSFUNCTOR &&g,const Eigen::VectorXd &y0,
    double T, unsigned int M);
```

that integrates the abstract ODE $\dot{\mathbf{y}} = \mathbf{g}(\mathbf{y})$ with state space $\mathbb{R}^N$ over $[0,T]$ with initial state vector $\mathbf{y}_0$ using M equidistant timesteps of the explicit trapezoidal rule (11.3.5). The `g`-argument passes a functor with signature compatible with `std::function<Eigen::VectorXd(const Eigen::VectorXd &)>` that supplies the right-hand-side function $\mathbf{g} : \mathbb{R}^N \to \mathbb{R}^N$.

SOLUTION for (11-3.e) → `11-3-5-0:s3b.pdf` ▲

(11-3.f) ⚂ (20 min.) [depends on Sub-problem (11-3.e)]

Relying on `sineClawRhs()` and `explTrpzTimestepping()` write a C++ function (in the file `finitevolumesineconslaw.h`)

```
template <typename RHSFUNCTOR>
Eigen::VectorXd solveSineConsLaw(
   RHSFUNCTOR &&g, unsigned int N,
   unsigned int M);
```

that solves the Cauchy problem for (11.3.1) numerically over the time interval $[0,1]$ with initial data

$$u_0(x) = \begin{cases} 1 & \text{for } 0 \le x < 1 , \\ 0 & \text{elsewhere,} \end{cases}$$

using an equidistant spatial mesh with N cells and M uniform timesteps. Choose the spatial interval $[-6,6]$ as in Sub-problem (11-3.d) which ensures that the spatial support of $u(x,t)$ will be contained in it for all $0 \le t \le 1$. Sample the initial data u_0 at midpoints of mesh cells.

SOLUTION for (11-3.f) → `11-3-6-0:s4.pdf` ▲

(11-3.g) ⚄ (25 min.) [depends on Sub-problem (11-3.f)]

Add code lines to the `main()` function in the file `finitevolumesineconslaw_main.cc` such that the approximate cell average values of the numerical solution at $T = 1$ of the Cauchy problem for (11.3.1) are written to the file `ufinal.csv`. Of course for the computation you should invoke `solveSineConsLaw()` from Sub-problem (11-3.f). Use a spatial mesh with $N = 600$ cells and $M = 200$ uniform timesteps.

Then create a MATLAB or PYTHON script to visualize the solution.

SOLUTION for (11-3.g) → 11-3-7-0:s4a.pdf ▲

(11-3.h) ⚀ (10 min.) Assuming a uniform spatial meshwidth $h > 0$ determine the constraint on the timestep such that the numerical domain of dependence is guranteed to cover the exact domain of dependence (CFL-conditon) for the method used in the previous sub-problem.

SOLUTION for (11-3.h) → 11-3-8-0:s5.pdf ▲

(11-3.i) ⚁ (25 min.) [depends on Sub-problem (11-3.f)]

Realize a C++ function (in the file `finitevolumesineconslaw.cc`)

```
unsigned int findTimesteps();
```

that determines the maximum possible timestep size for the numerical experiment of Sub-problem (11-3.f). It returns the minimal number M of timesteps that barely avoids blow-up. Use a bisection algorithm using the supplied values `ML` and `MR` supplied in the template.

HIDDEN HINT 1 for (11-3.i) → 11-3-9-0:6h1.pdf

SOLUTION for (11-3.i) → 11-3-9-1:s6.pdf ▲

(11-3.j) ⚁ (15 min.) [depends on Sub-problem (11-3.d)]

In the file `finitevolumesineconslaw.cc` create a modification

```
Eigen::VectorXd sineClawReactionRhs(
   const Eigen::VectorXd &mu, double c)
```

of the function `sineClawRhs()` from Sub-problem (11-3.d) so it still serves the same purpose but now for the augmented conservation law with reaction term

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\sin(\pi u) = -cu\,, \quad c \geq 0\,. \tag{11.3.12}$$

HIDDEN HINT 1 for (11-3.j) → 11-3-10-0:7h1.pdf

SOLUTION for (11-3.j) → 11-3-10-1:s7.pdf ▲

(11-3.k) ⚂ (15 min.) [depends on Sub-problem (11-3.f), Sub-problem (11-3.g) and Sub-problem (11-3.j)]

Conduct the numerical experiment of Sub-problem (11-3.g) including the instructions to visualize the solution at final time for (11.3.12) with $c = 1$. To that end, modify your implementation of `solveSineConsLaw()` from Sub-problem (11-3.f) so that the new version calls `sineClawReactionRhs()`.

SOLUTION for (11-3.k) → 11-3-11-0:s8.pdf ▲

End Problem 11-3, 200 min.

Problem 11-4: Lax-Wendroff Scheme

Throughout [Lecture → Chapter 11], we pursued the numerical treatment of scalar conservation laws from a method of lines perspective, that is, the first perform discretization in space based on a conservative finite volume scheme, and then apply some explicit single step method for timestepping.
However, many all-in-one schemes have been developed that combine both discretization in space and time in one formula. In this problem we will meet a famous representative of this class.

The problem requires familiarity with the thoery of 1D scalar conservation laws as elaborated in [Lecture → Section 11.2] and linear stability analysis [Lecture → Section 11.4.3]. Implementation in C++ based on EIGEN is requested.
▷ problem name/problem code folder: LaxWendroffScheme

The Cauchy problem for a generic scalar conservation law with flux function $f : \mathbb{R} \to \mathbb{R}$ seeks $u : \mathbb{R} \times [0,T] \mapsto \mathbb{R}$, which solves

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \qquad & \text{in } \mathbb{R} \times [0,T], \\ u(x,0) = u_0(x) \qquad & \text{in } \mathbb{R}. \end{aligned} \tag{11.4.1}$$

We compute approximations $\mu_j^{(k)} \approx u(jh, k\tau)$ on an equidistant space-time mesh with spatial meshwidth $h > 0$ and uniform timestep $\tau > 0$ by means of the so-called **Lax-Wendroff scheme** [LeV02, Sect. 6.1]

$$\mu_j^{(k)} = \mu_j^{(k-1)} - \tfrac{1}{2}\gamma\Big(f(\mu_{j+1}^{(k-1)}) - f(\mu_{j-1}^{(k-1)})\Big) + \tfrac{1}{2}\gamma^2\Big(f'(\tfrac{1}{2}(\mu_{j+1}^{(k-1)} + \mu_j^{(k-1)}))^2(\mu_{j+1}^{(k-1)} - \mu_j^{(k-1)}) - f'(\tfrac{1}{2}(\mu_j^{(k-1)} + \mu_{j-1}^{(k-1)}))^2(\mu_j^{(k-1)} - \mu_{j-1}^{(k-1)})\Big)\,, \quad \gamma := \frac{\tau}{h}\,, \tag{11.4.2}$$

with initial values $\mu_j^{(0)} := u_0(jh)$, $j \in \mathbb{Z}$.

(11-4.a) ⚀ (10 min.) What geometric CFL-condition according to [Lecture → Def. 11.4.2.11] applies in the case of the Lax-Wendroff fully discrete evolution (11.4.1)?

SOLUTION for (11-4.a) → 11-4-1-0:s0.pdf ▲

(11-4.b) ⚁ (10 min.) Formulate the Lax-Wendroff scheme for the special case of the Cauchy problem for linear advection with constant velocity v, see [Lecture → (11.1.1.11)].

SOLUTION for (11-4.b) → 11-4-2-0:sa.pdf ▲

(11-4.c) ⚄ (30 min.) [depends on Sub-problem (11-4.b)]

Carry out a linear stability analysis (→ [Lecture → Section 11.4.3]) of the Lax-Wendroff scheme for linear advection, as stated in the previous sub-problem. To that end consider the fully discrete evolution with intial data

$$\left(\vec{\boldsymbol{\psi}}_\xi\right)_j := \exp(\imath\xi j)\,, \quad j \in \mathbb{Z}\,, \quad -\pi < \xi \le \pi\,, \tag{11.4.4}$$

which are the bounded "eigenvectors" of scalar, linear translation invariant operators on $\mathbb{R}^{\mathbb{Z}}$, see [Lecture → (11.4.3.5)], and find out what condition the timestep τ has to satisfy that the solution does not experience exponential blow-up. This will give you a CFL-type stability condition.

SOLUTION for (11-4.c) → 11-4-3-0:sb.pdf ▲

For the remainder of this problem we again consider the non-linear conservation law (11.4.1) with the special flux function

$$f(u) := \exp(u)\,, \quad u \in \mathbb{R}\,.$$

(11-4.d) ⚀ Assume that there are $A, B \in \mathbb{R}$, $A < B$, such that

$$u_0(x) = A \quad \text{for } x \le 0, \quad A \le u_0(x) \le B \quad \text{for } 0 < x < 1, \quad u_0(x) = B \quad \text{for } x \ge 1. \tag{11.4.6}$$

Sketch the maximal possible set $\{(x,t) \in \mathbb{R} \times [0,1] : A < u(x,t) < B\}$ in the $x-t$-plane.

SOLUTION for (11-4.d) → 11-4-4-0:s1.pdf ▲

(11-4.e) ⚁ (20 min.) In the file `laxwendroffscheme.cc` implement a C++ function

```
Eigen::VectorXd solveLaxWendroff(
  const Eigen::VectorXd &u0,
  double T, unsigned int M);
```

that uses the Lax-Wendroff scheme (11.4.2) to solve (11.4.1) for $f(u) = \exp(u)$ over the time period $[0,T]$ under the assumption (11.4.6) with $0 \le A < B \le 1$. Use an equidistant spatial mesh with meshwidth $h = e\tau$ that covers a sufficiently wide spatial interval $[a,b]$ with left boundary $a = 0$.

The arguments have the following meaning:

`u0` : (finite) vector of initial cell averages
`T` : final time $T > 0$,
`M` : number of timesteps, hence $\tau := \frac{T}{M}$.

In `muend` the function is to return the approximations $\mu_j^{(M)}$ for $j = \lfloor \frac{-3T}{h} \rfloor, \ldots, \lceil \frac{3T+1}{h} \rceil$. Here $\lfloor \cdot \rfloor$ and $\lceil \cdot \rceil$ give the integer predecessor and successor, respectively, of a real number. The sizes of `muend` and `u0` have to be the same.

SOLUTION for (11-4.e) → 11-4-5-0:1as.pdf ▲

(11-4.f) ⚁ (20 min.) Find the entropy solution of the Riemann problem

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\exp(u) = 0 \quad \text{in } \mathbb{R} \times [0,1], \quad u(x,0) = \begin{cases} 0 & \text{for } x < 0, \\ 1 & \text{for } x \ge 0. \end{cases} \tag{11.4.8}$$

HIDDEN HINT 1 for (11-4.f) → 11-4-6-0:h2a1.pdf

SOLUTION for (11-4.f) → 11-4-6-1:s2a.pdf ▲

(11-4.g) ⚂ (15 min.) [depends on Sub-problem (11-4.e) and Sub-problem (11-4.f)]

Use your implementation of `solveLaxWendroff()` to solve the Riemann problem (11.4.8). Compute the error norm

$$\mathrm{err}(M) = h \sum_{j \in \mathbb{Z}} |\mu_j^{(M)} - u(jh,1)| \tag{11.4.10}$$

for $M = 20, 40, 80, 160, 320, 640$ uniform timesteps, tabulate it, create a doubly logarithmic plot of $\mathrm{err}(M)$ versus M, and determine the rate of convergence.

All this should be done by extending the `main()` function in `laxwendroffscheme_main.cc` and, from there, calling the function (in the file `laxwendroffscheme.cc`)

```
Eigen::VectorXd
  numexpLaxWendroffRP(const Eigen::VectorXi &M)
```

which, in turn, calls `solveLaxWendroff()` from Sub-problem (11-4.e) and computes the error $\mathrm{err}(M)$ for the numbers M of timestepps specified in the argument array `M`. These errors are returned in a vector. Finally, from `main()` write the table of error norms to a file `convergence.csv` for visualization by means of a PYTHON script.

SOLUTION for (11-4.g) → `11-4-7-0:s2.pdf` ▲

(11-4.h) ⚄ (15 min.) [depends on Sub-problem (11-4.g)]

Repeat the numerical experiment of Sub-problem (11-4.g) for the C^1 initial data

$$u_0(x) = \begin{cases} \sin^2(\frac{\pi}{2}x) & \text{for } 0 \le x \le 1, \\ 0 & x < 0, \\ 1 & x > 1. \end{cases} \tag{11.4.13}$$

To that end, in the file `laxwendroffscheme.cc` , create a C++ function

```
Eigen::VectorXd
   numexpLaxWendroffSmoothU0(const Eigen::VectorXi &M);
```

with the same purpose and signature as `numexpLaxWendroffRP()`.

The "exact" solution $x \mapsto u(x,1)$ is provided by the C++ function

```
Eigen::VectorXd referenceSolutiuon(const Eigen::VectorXd &x)
```

that expects a vector `x` of locations and returns very accurate approximations to the point values of the exact solution there.

SOLUTION for (11-4.h) → `11-4-8-0:s3.pdf` ▲

(11-4.i) ⚄ (10 min.) Derive the Godunov numerical flux for (11.4.1) and the flux function $f(u) = \exp(u)$.

SOLUTION for (11-4.i) → `11-4-9-0:s4.pdf` ▲

(11-4.j) ⚄ (15 min.) [depends on Sub-problem (11-4.i)]

In the file `laxwendroffscheme.cc` implement a C++ function

```
Eigen::VectorXd solveGodunov(
  const Eigen::VectorXd &u0,
  double T, unsigned int M);
```

that conducts the simulation requested in Sub-problem (11-4.d) based on a conservative finite volume method and Godunov numerical flux with explicit Euler timestepping. The arguments are the same as for `solveLaxWendroff()`.

SOLUTION for (11-4.j) → `11-4-10-0:s5.pdf` ▲

(11-4.k) ⚄ (10 min.) [depends on Sub-problem (11-4.j)]

Carry out the qualitative and quantitative empiric investigations of asymptotic convergence as requested in Sub-problem (11-4.h), but this time relying on `solveGodunov()` from Sub-problem (11-4.j).

The driver function should be realized in the file `laxwendroffscheme.cc` and should be defined as follows, analoguous to `numexpLaxWendroffSmoothU0()`:

```
Eigen::VectorXd
   numexpGodunovSmoothU0(const Eigen::VectorXi &M);
```

SOLUTION for (11-4.k) → 11-4-11-0:s6.pdf ▲

References

[LeV02] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge: Cambridge University Press, 2002, pp. xx+558. DOI: 10.1017/CBO9780511791253 (cit. on p. 419).

End Problem 11-4 , 155 min.

Problem 11-5: Discontinous Galerkin Discretization for 1D Conservation Laws

In [Lecture → Section 11.5] we studied linear reconstruction as a means to lift the order of convergence of conservative finite volume methods to 2. The so-called **discontinous Galerkin approach** for the spatial semi-discretization of conservation laws is another way to enhance orders of convergence. It also extends the lowest-order conservative finite volume methods discussedin [Lecture → Section 11.3].

This problem will ask for code development in C++ based on EIGEN. It requires the contents
▷ problem name/problem code folder: DiscontinuousGalerkin1D

We consider the Cauchy problem for the scalar non-liner conservation law

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \qquad & \text{in } \mathbb{R}]0,T[\,, \\ u(x,0) = u_0(x) \quad & \text{in } \mathbb{R}\,, \end{aligned} \tag{11.5.1}$$

with smooth flux function $f : \mathbb{R} \mapsto \mathbb{R}$ and initial data u_0 compactly supported in $[0,1]$.

Based on the infinite equidistant spatial mesh $\mathcal{M} := \{]x_{j-1}, x_j[: x_j = hj, j \in \mathbb{Z}\}$ with meshwidth $h > 0$ we define the finite-dimensional function space

$$V_h = \left\{ v \in L^2(\Omega) : v|_{[x_{j-1},x_j]} \in \mathcal{P}_1 \right\}, \tag{11.5.2}$$

of *discontinuous* piecewise linear functions on $\mathcal{M}$ (see Fig. 193 for an example, $\mathcal{P}_1$ is the space of polynomials of degree ≤ 1).

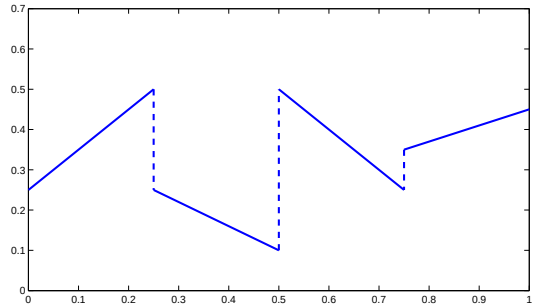


Fig. 193

◁ A sample discontinuous piecewise linear function $\in V_h$, visualized on a finite interval.

This is also the space of piecewise linear reconstructions as presented in [Lecture → Section 11.5.1]. Yet, in the case of discontinous Galerkin methods, the functions are piecewise linear on the cells of the mesh proper, not on dual cells.

(11-5.a) ⚀ (5 min.) What is the dimension of V_h?

SOLUTION for (11-5.a) → 11-5-1-0:s0.pdf ▲

(11-5.b) ⚅ (15 min.) Show that a continuous solution u of (11.5.1) satisfies

$$\int_{x_{j-1}}^{x_j} \frac{\partial u}{\partial t}(x,t)\, v_h(x)\mathrm{d}x - \int_{x_{j-1}}^{x_j} f(u(x,t)) \frac{\mathrm{d}v_h}{\mathrm{d}x}(x)\mathrm{d}x + f(u(x_j,t))v_h(x_j-) - f(u(x_{j-1},t))v_h(x_{j-1}+) = 0 \quad \forall v_h \in V_h, j \in \mathbb{Z}. \tag{11.5.3}$$

Here, we adopted the notation $v(x_j\pm) := \lim_{\delta\to 0^+} v(x_j \pm \delta)$ in order to resolve the ambiguity of $v(x_j)$.

HIDDEN HINT 1 for (11-5.b) → 11-5-2-0:h11.pdf

SOLUTION for (11-5.b) → 11-5-2-1:s1.pdf ▲

Next, we aim for a spatial Galerkin semi-discretization of (11.5.3) based on the space V_h. However, plugging $u_h \in V_h$ into (11.5.3) instead of u faces the problem of ambiguity of $f(u(x_j,t))$. Therefore

$f(u(x_j,t))$ in (11.5.3) is replaced with a numerical flux $F(u(x_j-),u(x_j+)))$, which leads to the semi-discrete **discontinuous Galerkin variational problem**: seek $u_h : [0,T] \mapsto V_h$ such that

$$\int_{-\infty}^{\infty} \frac{\partial u_h}{\partial t}(x,t)\, v_h(x)\mathrm{d}x - \sum_{j=\infty}^{\infty} \int_{x_{j-1}}^{x_j} f(u_h(x,t)) \frac{\mathrm{d}v_h}{\mathrm{d}x}(x)\mathrm{d}x + \sum_{j=-\infty}^{\infty} F(u_h(x_j-,t),u_h(x_j+,t))(v_h(x_j-)-v_h(x_j+)) = 0 \quad \forall v_h \in V_h. \tag{11.5.4}$$

(11-5.c) ⚂ (10 min.) We know *a priori* that u_0 is compactly supported in $[0,1]$, $0 \le u_0(x) \le 1$ and that $|f'(u)| \le c$ for all $0 \le u \le 1$. If $T = 1$, which is the smallest truncated spatial mesh

$$\widehat{\mathcal{M}} := \{]x_{j-1}, x_j[,\ x_j = hj,\ j \in \{-M_l, \ldots, M_r\} \} \tag{11.5.5}$$

that allows the spatial discretization of (11.5.1) without any impact of the truncation? Find the numbers $M_l, M_r \in \mathbb{N}_0$.

SOLUTION for (11-5.c) → 11-5-3-0:s2.pdf ▲

(11-5.d) ⚄ (30 min.) We denote by $\widehat{V}_h$ the space V_h restricted to the spatial interval covered by the truncated spatial mesh $\widehat{\mathcal{M}}$ from (11.5.5). As basis of $\widehat{V}_h$ we choose

$$\left\{ b_1^{-M_l}, b_2^{-M_l}, b_1^{-M_l+1}, b_2^{-M_l+1}, \ldots, b_1^{M_r-1}, b_2^{M_r-1}, b_2^{M_r}, b_2^{M_r} \right\}, \tag{11.5.6}$$

where

$$b_1^j(x) = \begin{cases} 1 & \text{for } x_{j-1} \le x < x_j, \\ 0 & \text{elsewhere,} \end{cases} \quad , \quad b_2^j(x) = \begin{cases} x - \frac{1}{2}(x_{j-1}+x_j) & \text{for } x_{j-1} \le x < x_j, \\ 0 & \text{elsewhere.} \end{cases} \tag{11.5.7}$$

Note that (11.5.6) also *fixes the ordering* of the basis functions and, thus, the basis expansion coefficient vector $\vec{\boldsymbol{\mu}} \in \mathbb{R}^N$, $N := 2(M_l + M_r + 1)$, associated with every function $u_h \in V_h$.

In (11.5.4) we expand u_h into these basis functions on the truncated mesh $\widehat{\mathcal{M}}$ with meshwidth h, testing with $v_h = b_1^j$ and $v_h = b_2^j$. This leads to an ordinary differential equation (ODE) for the time-dependent coefficient vector $\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\mu}}(t) \in \mathbb{R}^N$, $N := 2(M_l + M_r + 1)$, of the semi-discrete Galerkin solution, assuming the ordering of basis functions given in (11.5.6). This ODE takes the form

$$\boxed{\mathbf{B}\frac{\mathrm{d}\vec{\boldsymbol{\mu}}}{\mathrm{d}t}(t) + G(\vec{\boldsymbol{\mu}}(t)) = 0} \quad , \tag{11.5.8}$$

with a matrix $\mathbf{B} \in \mathbb{R}^{N\times N}$ and potentially non-linear function $G : \mathbb{R}^N \mapsto \mathbb{R}^N$.

What are the entries of the matrix $\mathbf{B}$?

SOLUTION for (11-5.d) → 11-5-4-0:s3.pdf ▲

(11-5.e) ⚂ (20 min.) [depends on Sub-problem (11-5.d)]

In the file `discontinuousgalerkin1d.cc` implement a C++ function (based on EIGEN)

```
Eigen::SparseMatrix<double> compBmat(int Ml, int Mr, double h);
```

that returns the sparse matrix $\mathbf{B}$ given $M_l \in \mathbb{Z}$, $M_r \in \mathbb{Z}$, and the meshwidth $h > 0$. Make sure that you handle the initialization of the sparse matrix efficiently.

SOLUTION for (11-5.e) → 11-5-5-0:s3a.pdf ▲

(11-5.f) ⚀ (15 min.) [depends on Sub-problem (11-5.d)]

Give a formula for the function G from (11.5.8) in terms of a general flux function f and two-point numerical flux $F = F(v,w)$. Simple point evaluations and integrals of f and F may be used.

HIDDEN HINT 1 for (11-5.f) → 11-5-6-0:h4.pdf

SOLUTION for (11-5.f) → 11-5-6-1:.pdf ▲

(11-5.g) ⚁ (15 min.) [depends on Sub-problem (11-5.f)]

Now use the two point Gaussian quadrature rule

$$\int_0^1 \varphi(\xi)\mathrm{d}\xi \approx \frac{1}{2}\Big(\varphi(\tfrac{1}{2}(1-\tfrac{1}{\sqrt{3}})) + \varphi(\tfrac{1}{2}(1+\tfrac{1}{\sqrt{3}}))\Big) \tag{11.5.10}$$

to approximate all integrals occurring in the expression for G obtained in the previous sub-problem. State the resulting formula.

HIDDEN HINT 1 for (11-5.g) → 11-5-7-0:h5.pdf

SOLUTION for (11-5.g) → 11-5-7-1:s5.pdf ▲

(11-5.h) ⚁ (25 min.) [depends on Sub-problem (11-5.f) and Sub-problem (11-5.g)]

In the file `discontinousgalerkin1d.h` write a C++/EIGEN-based function

```
template <typename FUNCTOR, typename NUMFLUX>
  Eigen::VectorXd G(
    const Eigen::VectorXd &muvec,
    FUNCTOR &&f, NUMFLUX &&F,
    int Ml, int Mr, double h);
```

that evaluates $G(\vec{\mu})$ for a coefficient vector $\vec{\mu}$ passed in `muvec`, on a finite equidistant mesh $\widehat{\mathcal{M}}$ described by M_l, M_r, and h. The arguments `f` and `F` contain function handles to the flux function $f = f(u)$ and the numerical flux $F = F(v,w)$, respectively. 2-point Gaussian quadrature as in the previous sub-problem is to be used. Outside the interval $[-M_l h, M_r h]$ the function u_h described by the basis expansion coefficient vector $vec\mu$ is supposed to vanish.

SOLUTION for (11-5.h) → 11-5-8-0:s6.pdf ▲

(11-5.i) ⚅ (25 min.) [depends on Sub-problem (11-5.h)]

We rely on the spatial discontinuous Galerkin semi-discretization on a truncated mesh $\widehat{\mathcal{M}}$ described above. In the file `discontinousgalerkin1d.h` implement an EIGEN-based C++ function

```
template <typename FUNCTOR, typename NUMFLUX>
  Eigen::VectorXd dgcl(
    Eigen::VectorXd mu0,
    FUNCTOR &&f, NUMFLUX &&F, double T,
    int Ml, int Mr, double h,
    unsigned int m);
```

that uses m uniform timesteps of the explicit 2-stage Runge-Kutta timestepping scheme described by the Butcher scheme

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 \\ \hline & 0 & 1 \end{array}\,, \tag{11.5.12}$$

to compute an approximation of $u(x,T)$ for final time $T > 0$. `mu0` passes the coefficient vector of an approximation of u_0. All other arguments play the same roles as in subproblems Sub-problem (11-5.g) and Sub-problem (11-5.h).

SOLUTION for (11-5.i) → `11-5-9-0:s7.pdf` ▲

From now on we consider the traffic flow problem with $f(u) = u(1-u)$ and use the so-called Engquist-Osher numerical flux

$$F_{\mathrm{EO}}(v,w) := \tfrac{1}{2}(f(v)+f(w)) - \tfrac{1}{2}\int_v^w |f'(\xi)|\mathrm{d}\xi. \tag{11.5.14}$$

(11-5.j) ⚄ (30 min.) In the file `discontinousgalerkin1d.cc` implement the engquist-Osher numerical flux for $f(u) = u(1-u)$ as a C++ function

```
double Feo(double v, double w);
```

The arguments `v` and `w` pass the left and right state.

SOLUTION for (11-5.j) → `11-5-10-0:s7a.pdf` ▲

(11-5.k) ⚄ (30 min.) Use the fully discrete discontinuous Galerkin solver implemented in `dgcl` from Sub-problem (11-5.i) with Engquist-Osher numerical flux and $f(u) = u(1-u)$ to solve the evolution problem over the time interval $[0,1]$ with initial data

$$u_0(x) = \begin{cases} 1 & \text{for } 0 \le x \le 1, \\ 0 & \text{elsewhere .} \end{cases} \tag{11.5.16}$$

As spatial meshwidth use $h = 0.05$, timestep $\tau = h/3$, and truncate the mesh to $[-2,2]$.

Plot the spatial cell averages of the obtained approximation of $u(x,1)$ versus the spatial variable x. To that end, in the file `discontinuousgalerkin1d.cc` realize a C++ function

```
void solveTrafficFlow();
```

that write the solution data to the file `trafficflowsolutuion.csv`, which can be read by another program (a PYTHON or MATLAB script) that creates the plot in the file `solution.eps`.

SOLUTION for (11-5.k) → `11-5-11-0:s8.pdf` ▲

End Problem 11-5, 220 min.

Problem 11-6: One-Dimensional Scalar Conservation Law with Empiric Flux

In some mathematical models of transport phenomena the flux function f is not given as an explicit expression, but in procedural form, because f itself might be the result of another computation or even be reconstructed from measurements. In this problem we encounter such a situation where f is modeled by a cubic spline.

This problem is linked to [Lecture → Chapter 11]. It is a balanced blend of theoretical questions and EIGEN-based coding tasks.
▷ problem name/problem code folder: CLEmpiricFlux

We consider a mathematical model given in the form of a Cauchy problem for a one-dimensional scalar conservation law for an unknown quantity $u = u(x,t)$:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{on } \mathbb{R}\times]0,T[\quad , \quad u(x,0) = u_0(x) \quad \text{for } x \in \mathbb{R} \, , \tag{11.6.1}$$

with initial data $u_0 : \mathbb{R} \to \mathbb{R}$ and a twice continuously differentiable flux function $f : \mathbb{R} \to \mathbb{R}$. Through suitable scaling we can ensure that

$$-1 \le u_0(x) \le 1 \quad \forall x \in \mathbb{R} \, , \tag{11.6.2}$$

which will be assumed throughout this problem. The flux function is provided in procedural form by the following data type (see file `uniformcubicspline.h`)

```
class UniformCubicSpline {
  public:
    UniformCubicSpline(double a, double b,
      Eigen::VectorXd f, Eigen::VectorXd M);
    // Point evaluation operator for cubic spline
    double operator ()(double u) const;
    // Point evaluation of derivative of cubic spline
    double derivative(double u) const;
  private:
    unsigned int _n;    // Number of knots - 1
    double _a, _b;      // Interval boundaries
    Eigen::VectorXd _f; // Values of flux function at knots
    Eigen::VectorXd _M; // Values of second derivatives of f at knots
};
```

In the interval $[a,b] \subset \mathbb{R}$ an object of type **UniformCubicSpline** represents f by a C^2-function that is a *piecewise cubic polynomial* on the equidistant grid

$$\mathcal{N} := \{]\xi_{j-1},\xi_j[: \ \xi_j = a + \frac{j}{n}(b-a), \ j = 0,\ldots,n\} \, . \tag{11.6.3}$$

Such a function is called a **cubic spline**. In particular, for $\xi_{j-1} \le u \le \xi_j$ we can write

$$\begin{aligned} f(u) = {} & f_j\tau + f_{j-1}(1-\tau) + \\ & M_j\frac{h^2}{6}(\tau^3-\tau) + M_{j-1}\frac{h^2}{6}(-\tau^3+3\tau^2-2\tau) \end{aligned} \, , \qquad \tau := \frac{u-\xi_{j-1}}{h} \, , \quad h := \frac{b-a}{n} \, . \tag{11.6.4}$$

with coefficients $f_j, M_j \in \mathbb{R}$. They have a special meaning, which can be verified by direct computation:

$$f(\xi_j) = f_j \quad , \quad f''(\xi_j) = M_j \, , \quad j = 0,\ldots,n \, . \tag{11.6.5}$$

The required continuity of f' will lead to n linear constraints that we assume to be satisfied. It is the coefficients f_j and M_j that are stored in the data members `_f` and `_M` of **UniformCubicSpline**. In the sequel, it will always be an immutable object of type **UniformCubicSpline** that describes the flux function f in (11.6.1).

(11-6.a) ⚀ (2 min.)

Which range $[a,b]$ has to be covered by the cubic spline model so that the evolution problem (11.6.1) will always be well-defined?

SOLUTION for (11-6.a) → 11-6-1-0:s1.pdf ▲

(11-6.b) ⚁ (5 min.)

Give a necessary and sufficient condition on the coefficients f_j and M_j, $j = 0,\dots,n$, that guarantees that the resulting cubic spline f according to (11.6.4) is **strictly convex**.

HIDDEN HINT 1 for (11-6.b) → 11-6-2-0:h21.pdf

SOLUTION for (11-6.b) → 11-6-2-1:s2.pdf ▲

For the remainder of the problem we take for granted that f is strictly convex.

(11-6.c) ⚂ (5 min.)

For the Cauchy problem (11.6.1) and assuming (11.6.2), in the file `solvecauchyproblem.cc` implement a C++ function

```
Eigen::Vector2d findSupport(const UniformCubicSpline &f,
  Eigen::Vector2d initsupp, double t);
```

that finds a minimal enclosing interval of the support of $\{x \mapsto u(x,t)\}$ and returns its bounds in a 2-vector. The argument `initsupp` gives the bounds of a (minimal) interval known to enclose the support of u_0, whereas `t` passes the time $t \geq 0$.

SOLUTION for (11-6.c) → 11-6-3-0:.pdf ▲

(11-6.d) ⚃ (7 min.) [depends on Sub-problem (11-6.c)]

We take for granted that $\operatorname{supp}(u_0) \subset [-1,1]$ and (11.6.2), and we write $u = u(x,t)$ for the solution of the Cauchy problem (11.6.1). In the file `solvecauchyproblem.h` supplement the missing parts of the function

```
template <typename FUNCTOR>
Eigen::VectorXd computeInitVec(
   const UniformCubicSpline &f,
   FUNCTOR &&u0, double h, double T);
```

The argument `f` gives the flux function, `u0` provides a functor for the (continuous) function u_0, `h` is a meshwidth $h > 0$, and `T` gives the final time $T > 0$.

The purpose of this function is the following:

(I) It should first determine two integers $m^-, m^+ \in \mathbb{Z}$ such that

$$\bigcup_{0\leq t\leq T} \operatorname{supp} u(\cdot,t) \subset [m^- h, m^+ h] .$$

(II) It should then return the vector

$$\vec{\mu}_0 := \left[u_0((j+m^- -1)h)\right]_{j=1}^{m^+-m^-+1} \in \mathbb{R}^{m^+-m^-+1} .$$

HIDDEN HINT 1 for (11-6.d) → 11-6-4-0:h5a1.pdf

SOLUTION for (11-6.d) → 11-6-4-1:s5a.pdf ▲

(11-6.e) ⚂ (3 min.)

What is the solution of the Riemann problem for (11.6.1) with

$$u_0(x) = \begin{cases} 1 & \text{for } x < 0\,, \\ -1 & \text{for } x \geq 0\,. \end{cases}$$

SOLUTION for (11-6.e) → 11-6-5-0:s4.pdf ▲

(11-6.f) ⚄ (10 min.)

In the file `clempiricflux.cc` complete the implementation of the class

```
class GodunovFlux {
  public:
    GodunovFlux(const UniformCubicSpline &f);
    double operator () (double v, double w) const;
  private:
    UniformCubicSpline _f;
};
```

whose evaluation operator supplies the 2-point Godunov numerical flux as introduced in [Lecture → Section 11.3.4.4] based on the cubic spline model for the flux function f. You may assume that the two states `v` and `w` lie in a range covered by the cubic spline model for the flux function.

HIDDEN HINT 1 for (11-6.f) → 11-6-6-0:h51.pdf

SOLUTION for (11-6.f) → 11-6-6-1:s5.pdf ▲

(11-6.g) ⚂ (5 min.)

The so-called **Ralston scheme** is a 2-stage explicit Runge-Kutta single step method described by the Butcher scheme

$$\begin{array}{c|cc} 0 & 0 & 0 \\ \frac{2}{3} & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & \frac{3}{4} \end{array}\,, \tag{11.6.14}$$

In the file `solvecauchyproblem.cc` implement the function

```
template <typename FUNCTOR>
Eigen::VectorXd RalstonODESolver(
  FUNCTOR &&rhs,
  Eigen::VectorXd mu0,
  double tau,
  int n);
```

that realizes `n` uniform timesteps of size $\tau > 0$ (passed in `tau`) of the Ralston scheme for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$, with initial state passed through the vector `mu0`$\in \mathbb{R}^N$. The argument `rhs` provides $\mathbf{f}$ through a functor of signature **std**::function<Eigen::VectorXd(Eigen::VectorXd)>.

SOLUTION for (11-6.g) → 11-6-7-0:s5b.pdf ▲

(11-6.h) ⚄ (10 min.) [depends on Sub-problem (11-6.f) and Sub-problem (11-6.g)]

In the file `solvecauchyproblem.h` complete the implementation of the function

```
Eigen::VectorXd solveCauchyProblem(const UniformCubicSpline &f,
                                   const Eigen::VectorXd &mu0, double h, double T);
```

that employs a conservative finite volume method with

- Godunov numerical flux on an equidistant spatial mesh with meshwidth $h > 0$ (passed in `h`),
- timestepping (with uniform timestep $\tau > 0$ as large as possible under the CFL-condition) based on Ralston's method (11.6.14), *cf.* Sub-problem (11-6.g),
- constant continuation of states beyond the interval covered by the vector of cell averages,

to solve (11.6.1) approximately on the time interval $[0, T]$. The argument `f` passes a cubic spline model for the (strictly convex) flux function f, `mu0` a sufficiently long vector containing approximate cell averages of u_0 on dual cells, and `T` the final time $T > 0$.

The function should return a vector of approximate cell averages of $\{x \mapsto u(x, T)\}$. This vector should have the same length as `mu0`.

SOLUTION for (11-6.h) → `11-6-8-0:s6.pdf` ▲

End Problem 11-6 , 47 min.

Problem 11-7: Higher-Order Finite Volume Method

In [Lecture → Section 11.5.3] we came across an extension of the classical first-order conservative finite volume policy that achieve enhanced convergence rates for smooth solution by means of piecewise linear reconstruction, the use of limiters, and second-order timestepping. In this problem we study this approach for a particular non-linear scalar conservation law.

Code development in C++ based on EIGEN.
▷ problem name/problem code folder: ExtendedMUSCL

We consider initial-value problems for the non-linear scalar conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{on} \quad \mathbb{R} \times \mathbb{R}^+ \quad , \quad f(u) := \begin{cases} u(\log u - 1) & \text{for } u > 0 \, , \\ 0 & \text{for } u = 0 \, . \end{cases} \tag{11.7.1}$$

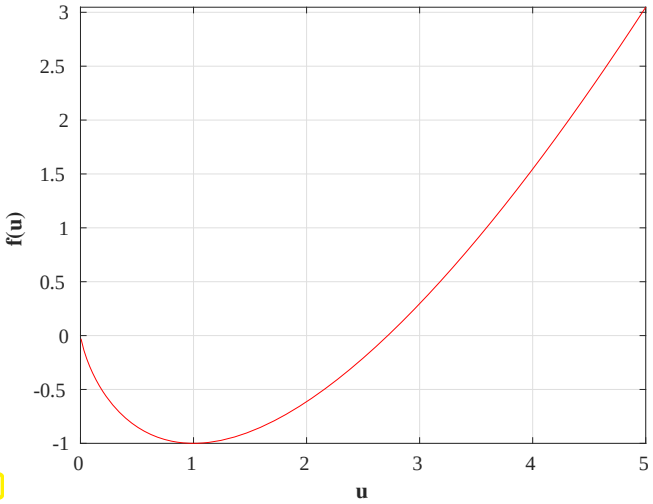


Fig. 195

◁ Graph of the flux function $u \mapsto f(u)$ on $]0,5[$.

(11-7.a) ⚀ (5 min.) Show that that the **flux function** $u \mapsto f(u)$ of (11.7.1) is *convex* on its maximal domain of definition.

SOLUTION for (11-7.a) → 11-7-1-0:s1.pdf ▲

(11-7.b) ⚀ (5 min.) Write down the entropy solution of the Riemann problem for (11.7.1) with initial data

$$u_0(x) = u(x,0) = \begin{cases} u_l := 2 & \text{for} \quad x < 0 \, , \\ u_r := 1 & \text{for} \quad x \geq 0 \, . \end{cases}$$

SOLUTION for (11-7.b) → 11-7-2-0:s2.pdf ▲

(11-7.c) ⚀ (10 min.) Now give the formula for the solution of the Riemann problem for (11.7.1) with initial data

$$u(x,0) = \begin{cases} u_l := 1 & \text{for} \quad x < 0 \, , \\ u_r := 2 & \text{for} \quad x \geq 0 \, . \end{cases}$$

SOLUTION for (11-7.c) → 11-7-3-0:.pdf ▲

(11-7.d) ⚀ (10 min.) In the file `extendedmuscl.cc` complete the implementation of the function

```
double logGodunovFlux(double v, double w);
```

that implements the **Godunov numerical 2-point flux** F_{GD} as introduced in [Lecture → Section 11.3.4.4] for the flux function $f(u) = u(\log u - 1)$.

SOLUTION for (11-7.d) → 11-7-4-0:s4.pdf ▲

We discretize the Cauchy problem for (11.7.1) with initial data $u_0 \in C^0(\mathbb{R})$. On an equidistant infinite mesh $\mathcal{M} := \{]x_{j-1}, x_j[: x_j := h(j+\frac{1}{2}),\ j \in \mathbb{Z}\}$ with meshwidth $h > 0$ we consider the standard conservative finite volume fully discrete evolution with timestep $\tau > 0$ and Godunov numerical flux:

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\left(F_{\mathrm{GD}}(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F_{\mathrm{GD}}(\mu_{j-1}^{(k)}, \mu_j^{(k)})\right), \quad j \in \mathbb{Z},\ k \in \mathbb{N}_0\,, \tag{11.7.7}$$

for the dual cell averages $\mu_j^{(k)} \approx h^{-1} \int_{x_{j-1/2}}^{x_{j+1/2}} u(x, k\tau)\,\mathrm{d}x$, $x_{j\pm\frac{1}{2}} := x_j \pm \frac{1}{2}h$, and for initial data

$$\mu_j^{(0)} = \frac{1}{h}\int_{x_{j-1/2}}^{x_{j+1/2}} u_0(x)\,\mathrm{d}x\,.$$

Remark. Our choice of $\mathcal{M}$ means that the j-th dual cell is $]hj, h(j+1)[$. Hence the dual cell average $\mu_j^{(k)}$ can be associated with the grid point $x_j = hj$, $j \in \mathbb{Z}$.

(11-7.e) ⚀ (8 min.) What timestep constraint is implied by the **CFL-condition**, if we know that the initial data u_0 satisfy

$$0 < \alpha < u_0(x) < \beta \quad \forall x \in \mathbb{R} \qquad \text{for known} \quad \alpha, \beta > 0\,.$$

SOLUTION for (11-7.e) → 11-7-5-0:s3.pdf ▲

(11-7.f) ⚁ (12 min.) [depends on Sub-problem (11-7.e)]

What kind of difficulty arises when applying the fully discrete timestepping for solving a Cauchy problem for (11.7.1) with *compactly supported* initial data u_0?

SOLUTION for (11-7.f) → 11-7-6-0:s6.pdf ▲

Now we consider a *slope-limited higher-order conservative finite volume method* in the spirit of [Lecture → Section 11.5.2] for (11.7.1). We restrict ourselves to equidistant meshes with uniform meshwidth $h > 0$.

The file `slopelimfluxdiff.h` contains the implementation of a C++ function

```
template <typename FunctionF, typename FunctionSlopes>
Eigen::VectorXd slopelimfluxdiff(
  const Eigen::VectorXd& mu,
  FunctionF &&F,
  FunctionSlopes &&slopes);
```

which realizes

$$\texttt{slopelimfluxdiff(mu,...)} \quad \sim \quad -h\cdot \mathcal{L}_h(\vec{\mu})\,.$$

Here $\mathcal{L}_h$ is the (cut-off) right-hand-side function $\mathcal{L}_h : \mathbb{R}^n \to \mathbb{R}^n$ of the method-of-lines ODE $\dot{\vec{\mu}}(t) = \mathcal{L}_h(\vec{\mu}(t))$ after spatial discretization of a generic scalar conservation law $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0$ by

means of a slope-limited higher-order conservative finite volume method on a (cut-off) equidistant spatial mesh.

The argument `mu` passes the vector $\vec{\mu}$, `F` contains a functor of signature **std::function<double(double,double)>** for the **numerical flux function** F, and `slopes` must be a functor of type **std::function<double(double,double,double)>** for the computation of $h\cdot$ the local slopes σ_j on every dual cell of the mesh.

C++11 code 11.7.9: Code for `slopelimfluxdiff()` ⇝ GitLab

```
// Function parameters:
// Vector textttmu:  (finite) vector μ⃗ of cell averages
// Functor textttF: 2-point numerical flux function F = F(v,w)
// Functor textttslope:  slope reconstruction function
// hσ_j = slopes(μ_{j-1}, μ_j, μ_{j+1})
//
// returns a vector with differences of numerical fluxes
//
// Function that realizes -h· the right hand side operator
// L_h for the mehtod-of-lines ODE arising from conservative finite
// volume semidiscretization of the Cauchy problem for a 1D scalar
    conservation
// law [Lecture → (11.2.2.1)].
template <typename FunctionF, typename FunctionSlopes>
Eigen::VectorXd slopelimfluxdiff(const Eigen::VectorXd &mu, FunctionF &&F,
                                 FunctionSlopes &&slopes) {
  int n = mu.size();  // Number of active dual grid cells
  Eigen::VectorXd sigma = Eigen::VectorXd::Zero(n);  // Vector of slopes
  Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);     // Flux differences

  // Computation of slopes σ_j, uses μ_0 = μ_1,
  // m_{N+1} = μ_N, which amounts to constant extension of states
  // beyond domain of influence [a,b] of non-constant intial data.  Same
  // technique has been applied in [Lecture → Code 11.3.2.9]
  sigma[0] = slopes(mu[0], mu[0], mu[1]);
  for (int j = 1; j < n - 1; ++j)
    sigma[j] = slopes(mu[j - 1], mu[j], mu[j + 1]);
  sigma[n - 1] = slopes(mu[n - 2], mu[n - 1], mu[n - 1]);

  // Compute linear reconstruction at endpoints of dual cells [Lecture →
      (11.5.1.2)]
  Eigen::VectorXd nup = mu + 0.5 * sigma;
  Eigen::VectorXd num = mu - 0.5 * sigma;

  // As in [Lecture → Code 11.3.2.10]:  constant continuation of data
     outside
  // [a,b]!
  fd[0] = F(nup[0], num[1]) - F(mu[0], num[0]);
  for (int j = 1; j < n - 1; ++j)
    // see [Lecture → (11.3.2.8)]
    fd[j] = F(nup[j], num[j + 1]) - F(nup[j - 1], num[j]);
  fd[n - 1] = F(nup[n - 1], mu[n - 1]) - F(nup[n - 2], num[n - 1]);
  return fd;
}
```

(11-7.g) ⚀ (10 min.) The so-called **MC slope limiter**, sets the slope associated with the j-th dual

cell to the value

$$\sigma_j := \begin{cases} \min\{\sigma_l,\sigma_c,\sigma_r\} & \text{, if } \sigma_l,\sigma_c,\sigma_r > 0\,, \\ \max\{\sigma_l,\sigma_c,\sigma_r\} & \text{, if } \sigma_l,\sigma_c,\sigma_r < 0\,, \\ 0 & \text{, otherwise,} \end{cases}$$

$$\text{with} \quad \sigma_c := \frac{\mu_{j+1}-\mu_{j-1}}{2h}\,, \quad \sigma_l := 2\frac{\mu_j-\mu_{j-1}}{h}\,, \quad \sigma_r := 2\frac{\mu_{j+1}-\mu_j}{h}\,. \tag{11.7.10}$$

In the file `extendedmuscl.cc` implement a C++ function

```
double limiterMC(double mu_left, double mu_center, double
   mu_right);
```

that is suitable as `slopes` argument for `slopelimfluxdiff()`, if one wants to use the MC slope limiter.

SOLUTION for (11-7.g) → 11-7-7-0:s6a.pdf ▲

(11-7.h) ⚄ (20 min.) Instead of a Cauchy problem we now study initial-value problems for (11.7.1) in a *1-periodic setting*, that is for the initial data we assume

$$u_0(x) > 0 \quad , \quad u_0(x) = u_0(x+1) \quad \forall x \in \mathbb{R}\,, \tag{11.7.12}$$

which leads to

$$u(x,t) = u(x+1,t) \quad \forall x \in \mathbb{R}\,, \quad \forall t \geq 0\,.$$

Hence, all computations can be confined to the interval $[0,1]$.

In the file `slopelimfluxdiff.h` you find a function

```
template <typename FunctionF, typename FunctionSlopes>
Eigen::VectorXd slopelimfluxdiffper(
  const Eigen::VectorXd& mu,
  FunctionF &&F,
  FunctionSlopes &&slopes);
```

which is supposed to provide $-h\cdot$ the function $\mathcal{L}_h^{\mathrm{per}} : \mathbb{R}^n \to \mathbb{R}^n$ for the method-of-lines ODE $\dot{\vec{\mu}}(t) = \mathcal{L}_h^{\mathrm{per}}(\vec{\mu}(t))$ for the slope-limited higher-order conservative finite volume method on a equidistant mesh with meshwidth $h := \frac{1}{n}$, $n \in \mathbb{N}$. Of course, n is the length of the argument vector `mu`.

However, currently the code for this function is just a copy of the code of `slopelimfluxdiff()`. Adapt the code to the periodic setting.

HIDDEN HINT 1 for (11-7.h) → 11-7-8-0:h78.pdf

SOLUTION for (11-7.h) → 11-7-8-1:s7.pdf ▲

For timestepping we want to rely on a so-called SSP Runge-Kutta method proposed in [GST01, Section 4.1]. For the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ its discrete evolution operator Ψ^τ for timestep $\tau > 0$ is defined according to:

$$\mathbf{k}_1 := \mathbf{y} + \tau\mathbf{f}(\mathbf{y})\,, \quad \mathbf{k}_2 := \tfrac{3}{4}\mathbf{y} + \tfrac{1}{4}\mathbf{k}_1 + \tfrac{1}{4}\tau\mathbf{f}(\mathbf{k}_1)\,, \quad \Psi^\tau\mathbf{y} := \tfrac{1}{3}\mathbf{y} + \tfrac{2}{3}\mathbf{k}_2 + \tfrac{2}{3}\tau\mathbf{f}(\mathbf{k}_2)\,. \tag{11.7.14}$$

(11-7.i) ⚂ (12 min.) In the file `extendedmuscl.h` complete the implementation of the function

```
template<typename FUNCTOR, typename State>
State sspEvolop(FUNCTOR &&f, State y, double tau);
```

that implements the discrete evolution operator Ψ^τ according to (11.7.14). The type **State** must support basic vector operations like addition (**operator** `+`) and (left) scalar multiplication (**operator** `*`) with a real number. The `f` must supply the right-hand-side function $\mathbf{f}$ of the autonomous ODE and is expected to have the signature **std::**`function<State(State)>`.

SOLUTION for (11-7.i) → `11-7-9-0:s9.pdf` ▲

(11-7.j) ⚄ (5 min.) [depends on Sub-problem (11-7.i)]

τ	error	$\log_2(\text{error})$
0.062	2.6e-05	-15
0.031	3.4e-06	-18
0.016	4.3e-07	-21
0.0078	5.4e-08	-24
0.0039	6.7e-09	-27
0.0020	8.4e-10	-30
0.00098	1.1e-10	-33

We rely on `sspEvolop()` for integrating the the scalar linear ODE $\dot{y} = y$ over $[0,1]$ with a constant timestep size $\tau > 0$. We do this for $y(0) = 1$, timesteps $\tau = 2^{-4}, 2^{-5}, \ldots, 2^{-10}$ and tabulate the modulus of the error at final time $T = 1$ versus τ. The results are printed beside.

Determine the order of the SSP Runge-Kutta timestepping method (11.7.14) from the listed errors.

SOLUTION for (11-7.j) → `11-7-10-0:s10.pdf` ▲

(11-7.k) ⚄ (15 min.) [depends on Sub-problem (11-7.i), Sub-problem (11-7.h), Sub-problem (11-7.g), Sub-problem (11-7.d)]

In the file `extendedmuscl.h` complete the implementation of the function

```
template <typename U0_FUNCTOR>
Eigen::VectorXd solveClaw(
  U0_FUNCTOR &&u0, double T, unsigned int n);
```

that approximately solves an initial-value problem for (11.7.1) in the *1-periodic setting* with initial data given by u_0 (passed through `u0`, a functor with signature **std::**`function<`**double**`(`**double**`)>`) and using an equidistant spatial mesh of $[0,1]$ with n (dual) cells. The slope-limited high-order conservative finite volume method implemented in `slopelimfluxdiffper()` with the MC slope limiter and timestepping according to (11.7.14) with uniform timestep $\tau > 0$ chosen to match the CFL-threshold found in Sub-problem (11-7.e) is to be used.

Of course, you can reuse your implementations of `slopelimfluxdiffper`, `sspEvolop()`, and `limiterMC()`.

The following plots display solutions produced by `solveClaw()` on grids with $n = 100$ cells and C^1 initial data u_0 of the form of a "raised bump":

```
static auto bump = [](double x) {
return ((x >= 0.25) && (x <= 0.75))
           ? (2.0 - std::pow(std::cos(M_PI * (2 * (x - 0.25))), 2))
           : 1.0;
```

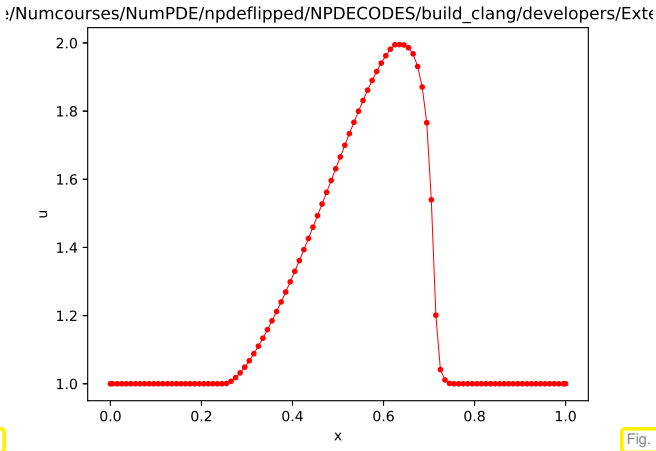


Fig. 197

Final time $T = 0.2$

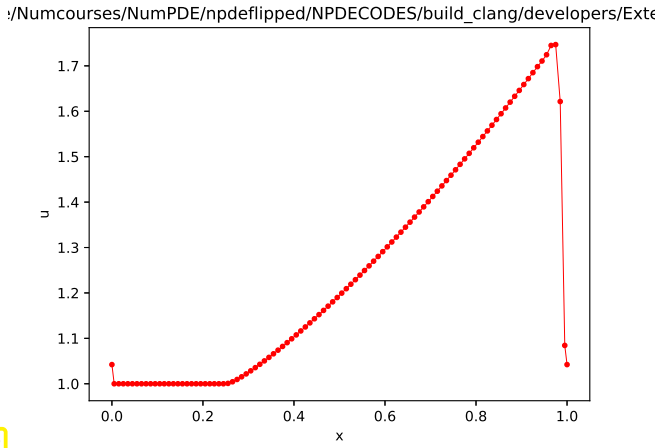


Fig. 198

Final time $T = 1.0$

We clearly see that the solution stays smooth over short times, but then develops the typical shock + rarefaction shape.

SOLUTION for (11-7.k) → 11-7-11-0:s11.pdf ▲

(11-7.l) ⚄ (30 min.) In preparation for a convergence study implement a C++ function

```
template <typename VECSOURCE, typename VECDEST>
void interpolate(const VECSOURCE &s, VECDEST &d);
```

that takes two vector arguments of types compatible with **std::vector<double>** or `Eigen::VectorXd`. These two argument vectors `s` and `d` have lengths $n \in \mathbb{N}$ (`n = s.size()`) and $N \in \mathbb{N}$ respectively and their entries should be read as cell averages of 1-periodic functions.

More concretely, let us view values `s[j]`, $0 \le j < n$, as the values in the nodes $x_j := H(j+\frac{1}{2})$, $H := 1/n$, $0 \le j < n$, of a continuous, *piecewise linear* 1-periodic function $f_h \in \mathcal{S}_1^0(\mathcal{M})$ on the equidistant infinite mesh $\mathcal{M} := \{]x_j, x_{j+1}[: j \in \mathbb{Z}\}$. Then, in the entries of `d`, the function should return the values

$$\mathtt{d[j]} := f_h(y_k)\,, \quad y_k := 1/N(k+\frac{1}{2})\,, \quad k = 0,\ldots,N-1\,. \tag{11.7.17}$$

SOLUTION for (11-7.l) → 11-7-12-0:s12.pdf ▲

(11-7.m) ⚄ (20 min.) [depends on Sub-problem (11-7.l) and Sub-problem (11-7.k)]

In the file `extendedmuscl.h` implement a C++ function

```
void studyCvgMUSCLSolution(U0_FUNCTOR &&u0, double T);
```

that lists approximations of the $L^1(]0,1[)$- and $L^\infty(]0,1[)$-norms of the discretization errors for the MUSCL solutions on meshes with $n = 2^m$, $m = 4,5,\ldots,10$, cells.

To obtain such approximations, we first use the MUSCL scheme to compute a reference solution on a mesh with 8192 cells. Onto that mesh we interpolate the solutions on the coarser meshes using the `interpolate()` function from Sub-problem (11-7.l). Then we take the difference of the solutions and compute their norms, assuming that we deal with piecewise constant functions on dual cells.

SOLUTION for (11-7.m) → 11-7-13-0:s13.pdf ▲

References

[GST01] S. Gottlieb, C-W. Shu, and E. Tadmor. "Strong Stability-Preserving High-Order Time Discretization Methods". In: *SIAM Review* 43.1 (2001), pp. 89–112 (cit. on p. 434).

End Problem 11-7 , 162 min.

Problem 11-8: 2D Finite Volume Method for Advection

Throughout [Lecture → Chapter 11] the development of numerical method was confined to evolution problems in one spatial dimension. Yet, most physical models that are phrased in terms of conservation laws (not necessarily scalar) are posed on higher spatial dimensions. In this problem we will discuss the numerical treatment of linear advection problems [Lecture → (11.1.1.5)] in two spatial dimensions. We will see how the idea of conservative finite volume discretization can be generalized to this case.

This problem is loosely related to [Lecture → Section 11.3], but also relies on concepts and techniques introduced in [Lecture → Section 2.7]. Code to be developed in this problem will be based on LEHRFEM++.
▷ problem name/problem code folder: AdvectionFV2D

Let $\Omega \subset \mathbb{R}^2$ be a bounded polygon and $\boldsymbol{\beta} : \overline{\Omega} \to \mathbb{R}^2$ be a continuous velocity field. We consider the following initial-boundary value problem for linear advection, see [Lecture → Section 11.1.1] or [Lecture → Section 10.3.2]:

$$\frac{\partial u}{\partial t} + \operatorname{div}_{\boldsymbol{x}}(\boldsymbol{\beta}(\boldsymbol{x})u) = 0 \quad \text{in} \quad \Omega\times]0,T[\,, \tag{11.8.1}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Omega\,, \quad u(\boldsymbol{x},t) = 0 \quad \forall 0 \le t \le T\,, \quad \boldsymbol{x} \in \Gamma_{\text{in}}\,, \tag{11.8.2}$$

with prescribed initial data $u_0 \in C^0(\overline{\Omega})$, and zero Dirichlet boundary conditions on the inflow boundary

$$\Gamma_{\text{in}} := \{\boldsymbol{x} \in \partial\Omega : \ \boldsymbol{n}(\boldsymbol{x}) \cdot \boldsymbol{\beta}(\boldsymbol{x}) < 0\}\,,$$

where $\boldsymbol{n}$ is the exterior unit normal vector field on $\partial\Omega$.

Throughout this problem we assume $\|\boldsymbol{\beta}\| \le 1$, where $\|\cdot\|$ designates the Euclidean norm of a vector.

For the sake of discretization Ω will be equipped with a hybrid 2D mesh $\mathcal{M} = \{K_1, K_2, \ldots, K_N\}$, whose set of edges will be denoted by $\mathcal{E}$. An approximation of the solution u is sought in the space $\mathcal{S}_0^{-1}(\mathcal{M})$ of *piecewise constant* functions on $\mathcal{M}$. Throughout, the characteristic functions of the mesh cells,

$$b_h^j(\boldsymbol{x}) = \begin{cases} 1 & \text{, if } \boldsymbol{x} \in K_j\,, \\ 0 & \text{elsewhere,} \end{cases} \tag{11.8.3}$$

will serve as global shape functions. In the formulas below their numbering will be induced by the numbering of the cells of the mesh, but this need not be the case in the LEHRFEM++-based codes.

(11-8.a) ⚀ (15 min.) Show that every continuous classical solution of (11.8.1) satisfies

$$\sum_{K\in\mathcal{M}} \left\{ \int_K \frac{\partial u}{\partial t}(\boldsymbol{x},t) w_h(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial K} \boldsymbol{\beta}(\boldsymbol{x}) \cdot \boldsymbol{n}_{\partial K}(\boldsymbol{x})\, u(\boldsymbol{x})\, w_{h\,|K}(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = 0 \right\} \tag{11.8.4}$$

for all $w_h \in \mathcal{S}_0^{-1}(\mathcal{M})$. Here, $w_h|_K$ means that the value of w_h on K has to be used. The symbol $\boldsymbol{n}_{\partial K}$ stands for the exterior unit normal vectorfield on ∂K.

HIDDEN HINT 1 for (11-8.a) → 11-8-1-0:h1.pdf

SOLUTION for (11-8.a) → 11-8-1-1:s1.pdf ▲

(11-8.b) ⚄ (15 min.) Temporarily, we consider the Cauchy problem [Lecture → § 9.3.2.1], [Lecture → Section 11.1], for the linear advection equation, that is $\Omega = \mathbb{R}^2$ in (11.8.1). For it, if $\operatorname{supp}(u_0) \subset B_1 := \{\boldsymbol{x} \in \mathbb{R}^2 : \ \|\boldsymbol{x}\| < 1\}$, and final time $T = 1$, find the minimal $R > 0$ such that the (spatial) support of $u(\boldsymbol{x},t)$ is contained in $]-R,R[^2$ for all $0 \le t \le T$.

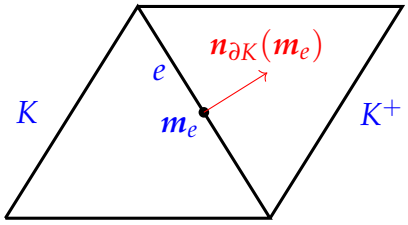
HIDDEN HINT 1 for (11-8.b) → 11-8-2-0:h21.pdf

HIDDEN HINT 2 for (11-8.b) → 11-8-2-1:h22.pdf

SOLUTION for (11-8.b) → 11-8-2-2:s2.pdf ▲

For every pair (K,e), where K is a triangle of $\mathcal{M}$, and $e \in \mathcal{E}$ is one of the edges of K, we define the **upwind numerical flux** according to

$$F_e^K(v,w) := \begin{cases} \boldsymbol{\beta}(\boldsymbol{m}_e)\cdot\boldsymbol{n}_{\partial K}(\boldsymbol{m}_e)v & \text{, if } \boldsymbol{\beta}(\boldsymbol{m}_e)\cdot\boldsymbol{n}_{\partial K}(\boldsymbol{m}_e) \geq 0\,, \\ \boldsymbol{\beta}(\boldsymbol{m}_e)\cdot\boldsymbol{n}_{\partial K}(\boldsymbol{m}_e)w & \text{, if } \boldsymbol{\beta}(\boldsymbol{m}_e)\cdot\boldsymbol{n}_{\partial K}(\boldsymbol{m}_e) < 0 \text{ and } e \not\subset \partial\Omega\,, \\ 0 & \text{, if } \boldsymbol{\beta}(\boldsymbol{m}_e)\cdot\boldsymbol{n}_{\partial K}(\boldsymbol{m}_e) < 0 \text{ and } e \subset \partial\Omega\,. \end{cases} \tag{11.8.8}$$



Here $\boldsymbol{m}_e$ is the midpoint of e and $\boldsymbol{n}_{\partial K}$ is the exterior unit normal for K, see figure.

Then the **upwind finite volume spatial semi-discretization** of (11.8.1) on $\mathcal{M}$ boils down to the variational problem: seek $u_h(t) \in \mathcal{S}_0^{-1}(\mathcal{M})$ such that for all $w_h \in \mathcal{S}_0^{-1}(\mathcal{M})$

$$\sum_{K\in\mathcal{M}} \left[\int_K \frac{\partial u_h}{\partial t}(\boldsymbol{x},t) w_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \sum_{e\in\mathcal{E}, e\subset\partial K} |e| F_e^K(u_h|_K, u_h|_{K^+})\, w_h|_K \right] = 0\,, \tag{11.8.9}$$

with $|e|$ the length of the edge, F_e^K from (11.8.8), and, given K and $e \subset \partial K$, K^+ a formal notation for the cell on the other side of e, see the above figure.

(11-8.c) ⚀ (5 min.) At first glance (11.8.8) seems to be plagued by a failure to give a meaning to K^+ in case $e \subset \partial\Omega$. Explain why this need not worry us.

SOLUTION for (11-8.c) → 11-8-3-0:s3.pdf ▲

(11-8.d) ⚂ (20 min.) We expand the function $u_h \in \mathcal{S}_0^{-1}(\mathcal{M})$ into basis functions b_h^j, $j = 1,\ldots,N$, $N := \sharp\mathcal{M}$:

$$u_h(\boldsymbol{x},t) = \sum_{j=1}^N \mu_j(t) b_j^N(\boldsymbol{x})\,, \quad \boldsymbol{x}\in\Omega\,, \quad t \geq 0\,.$$

Then (11.8.9) spawns the autonomous linear ordinary differential equation

$$\dot{\vec{\boldsymbol{\mu}}}(t) = \mathbf{B}\vec{\boldsymbol{\mu}}(t)\,, \quad \mathbf{B} \in \mathbb{R}^{N,N}\,, \tag{11.8.10}$$

for the N time-dependent expansion coefficients $\mu_j(t)$, $j = 1,\ldots,N$, collected into the time-dependent vector $\vec{\boldsymbol{\mu}}(t) = \left[\mu_j(t)\right]_{j=1}^N \in \mathbb{R}^N$. Write down the j-th equation as explicitly as possible, if $K_j \in \mathcal{M}$ is a triangle with adjacent cells K_ℓ, K_m, K_k, $\{\ell,m,k\} \subset \{1,\ldots,N\}$.

SOLUTION for (11-8.d) → 11-8-4-0:s3a.pdf ▲

(11-8.e) ⚄ (10 min.) [depends on Sub-problem (11-8.d)]

Give a sharp upper bound for the total number of non-zero entries of the matrix $\mathbf{B}$ in (11.8.10), and also for the maximal number of non-zero entries per row and column.

SOLUTION for (11-8.e) → 11-8-5-0:s3b.pdf ▲

(11-8.f) ⚄ (10 min.) To prepare the LEHRFEM++-based implementation of the upwind finite volume method describe how to initialize a **lf::assemble::UniformFEDofHandler** object suitable for the finite element space $\mathcal{S}_0^{-1}(\mathcal{M})$, see [Lecture → § 2.7.4.16]. Put the right numbers into the boxes.

```
lf::assemble::UniformFEDofHandler dof_handler(
      mesh_p, {{lf::base::RefEl::kPoint(), [ ]},
               {lf::base::RefEl::kSegment(), [ ]},
               {lf::base::RefEl::kTria(), [ ]},
               {lf::base::RefEl::kQuad(), [ ]}});
```

SOLUTION for (11-8.f) → 11-8-6-0:s4.pdf ▲

(11-8.g) ⚄ (30 min.) In the file `advectionfv2d.cc` implement a C++-functions

```
std::shared_ptr<
  lf::mesh::utils::CodimMeshDataSet<
     Eigen::Matrix<double,2,Eigen::Dynamic>>>
  computeCellNormals(std::shared_ptr<const lf::mesh::Mesh> mesh_p);
```

that computes the exterior edge normals for every cell K of the 2d hybrid mesh passed through `mesh_p` and stores them in an **lf::mesh::utils::CodimMeshDataSet** array of $2 \times \sharp\mathcal{V}(K)$-matrices.

HIDDEN HINT 1 for (11-8.g) → 11-8-7-0:h4a1.pdf

SOLUTION for (11-8.g) → 11-8-7-1:s4a.pdf ▲

(11-8.h) ⚄ (30 min.) In the file `advectionfv2d.cc` implement a C++ function

```
std::shared_ptr<
  lf::mesh::utils::CodimMeshDataSet<
    std::array<const lf::mesh::Entity*,4>>>
      getAdjacentCellPointers(
         std::shared_ptr<const lf::mesh::Mesh> mesh_p);
```

that creates an array of **lf::mesh::Entity** pointers to the neighbors of each cell of $\mathcal{M}$. The i-th pointer in the array for the cell K should contain a pointer to the other cell adjacent to the i-th edge of K, **nullptr** if there is none.

HIDDEN HINT 1 for (11-8.h) → 11-8-8-0:h4b1.pdf

SOLUTION for (11-8.h) → 11-8-8-1:s4b.pdf ▲

(11-8.i) ⚄ (25 min.) [depends on Sub-problem (11-8.g), Sub-problem (11-8.h), Sub-problem (11-8.c)]

Code an *efficient* C++ function (in the file `advectionfv2d.h`)

```
template<typename VECTORFIELD>
  Eigen::SparseMatrix<double>
    initializeMOLODEMatrix(
      const lf::assemble::DofHandler &dofh,
      VECTORFIELD &&beta);
```

that computes the matrix $\mathbf{B} \in \mathbb{R}^{N,N}$, $N := \sharp\mathcal{M}$, occurring in the linear method-of-lines ODE (11.8.10). The `dofh` argument passes a suitable **lf::assemble::DofHandler** object, see Sub-problem (11-8.f).

`beta` is a functor object providing the vectorfield $\boldsymbol{\beta}$. It must have an evaluation operator **operator** `()(const Eigen::Vector2d x);`

For the implementation you can use the function `getAdjacentCellPointers()` and `computeCellNormals()`.

HIDDEN HINT 1 for (11-8.i) → `11-8-9-0:h51.pdf`

SOLUTION for (11-8.i) → `11-8-9-1:s5.pdf` ▲

We obtain a fully discrete evolution by applying the explicit trapezoidal rule, a 2-stage second-order explicit Runge-Kutta single step method characterized by the Butcher tableau [Lecture → § 9.2.6.30]

$$\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}, \tag{11.8.18}$$

for the discretization of (11.8.10) in time.

(11-8.j) ⚀ (10 min.) State the recursion formula for the fully discrete evolution with uniform timestep $\tau > 0$.

SOLUTION for (11-8.j) → `11-8-10-0:s6.pdf` ▲

(11-8.k) ⚄ (15 min.) [depends on Sub-problem (11-8.j)]

Argue, what timestep constraint is imposed on the fully discrete evolution by the geometric CFL-condition as presented in [Lecture → Def. 11.4.2.11]. You may assume that the shape-regularity measure, see [Lecture → Def. 3.3.2.20] of $\mathcal{M}$, is below a small bound.

SOLUTION for (11-8.k) → `11-8-11-0:s7.pdf` ▲

(11-8.l) ⚅ (20 min.) [depends on Sub-problem (11-8.h)]

In the file `advectionfv2d.cc` implement a C++ function

```
double computeHmin(std::shared_ptr<constd lf::mesh::Mesh> mesh_p);
```

that computes h_{min} defined as the minimal distance of the barycenters of two cells of the mesh that share an edge.

HIDDEN HINT 1 for (11-8.l) → `11-8-12-0:h81.pdf`

SOLUTION for (11-8.l) → `11-8-12-1:s8.pdf` ▲

(11-8.m) ⚂ (45 min.) [depends on Sub-problem (11-8.i)]

In the file `advectionfv2d.h` implement a C++ function

```
template <typename VECTORFIELD>
Eigen::VectorXd solveAdvection2D(
    const lf::assemble::DofHandler &dofh, VECTORFIELD &&beta,
    const Eigen::VectorXd &u0_h,
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        std::array<const lf::mesh::Entity *, 4>>>
        adjacentCells,
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        Eigen::Matrix<double, 2, Eigen::Dynamic>>>
        normal_vectors,
    double T, unsigned int M);
```

that implements the fully discrete evolution as introduced above for the time interval $[0,T]$, $T > 0$. The argument

- `dofh` passes a **lf::assemble::DofHandler** object containing the mesh information and the local-to-global index mapping for the shape functions,
- `beta` give a functor object describing the vectorfield $\boldsymbol{\beta}$, see Sub-problem (11-8.i),
- `u0_h` supplies $\vec{\boldsymbol{\mu}}(0)$, that is, the basis expansion coefficients for $u_h(0)$,
- `T` specifies the final time $T > 0$,
- `M` gives the number $M \in \mathbb{N}$ of uniform timesteps, which implies $\tau = \frac{T}{M}$.

The function should throw an exception of type **std::overflow_error** in case $\max_j |\mu_j^{(k)}| \geq 10^3 \max_j |\mu_j^{(0)}|$ for some $k \in \{1,\ldots,M\}$.

SOLUTION for (11-8.m) → `11-8-13-0:s3.pdf` ▲

(11-8.n) ⚄ (20 min.) [depends on Sub-problem (11-8.m)]

Now we want to use the function `solveAdvection2D()` to simulate the advection IBVP (11.8.1)

- for the time interval $[0,1]$,
- on the square domain $\Omega =]0,1[^2$,
- with the rotational velocity field

$$\boldsymbol{\beta}(\boldsymbol{x}) = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix} \quad \blacktriangleright \quad \boldsymbol{\Phi}(\boldsymbol{x},t) = \begin{bmatrix} \cos t & -\sin t \\ \sin t & \cos t \end{bmatrix} \boldsymbol{x}\,, \tag{11.8.23}$$

- and the bump initial condition

$$u_0(\boldsymbol{x}) = \begin{cases} \cos^2\left(\frac{\pi}{2d}(\boldsymbol{x}-\boldsymbol{x}_0)\right) & \text{, if } \|\boldsymbol{x}-\boldsymbol{x}_0\| < d\,, \\ 0 & \text{elsewhere} \end{cases} \quad \text{, with } d = 0.2\,, \quad \boldsymbol{x}_0 = \begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}. \tag{11.8.24}$$

According to [Lecture → Thm. 11.1.1.8]I, the exact solution is

$$u(\boldsymbol{x},t) = u_0\left(\begin{bmatrix} \cos t & \sin t \\ -\sin t & \cos t \end{bmatrix} \boldsymbol{x}\right). \tag{11.8.25}$$

To conduct the simulations, in the file `advectionfv2d.h` , implement a function

```
template <typename FUNCTOR, typename VECTORFIELD>
Eigen::VectorXd simulateAdvection(
    const lf::assemble::DofHandler &dofh, VECTORFIELD &&beta,
      FUNCTOR &&u0,
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        std::array<const lf::mesh::Entity *, 4>>>
        adjacentCells,
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        Eigen::Matrix<double, 2, Eigen::Dynamic>>>
        normal_vectors,
    double T);
```

The argument `dofh` communicates all the information about the numbering of basis functions, and the mesh. The mesh is supposed to cover the unit square.

The initial values for the basis expansion coefficients can be obtained by sampling u_0 at the barycenters of the cells of the mesh.

SOLUTION for (11-8.n) → 11-8-14-0:sa.pdf ▲

(11-8.o) ⚂ (20 min.) [depends on Sub-problem (11-8.n)]

Extend the `main()`-function in the file `advectionfv2d_main.cc` so that you solve the advection problem presented in Sub-problem (11-8.n) on a sequence of meshes created by uniform regular refinement starting from the 2D hybrid planar mesh returned by the function call

```
auto mesh_p{
  lf::mesh::test_utils::generateHybri2DTestMesh(0,1./3.)};
```

and then using the call

```
auto mesh_seq_p{
  lf::refinement::GenerateMeshHierarchyByUniformRefinemnt(mesh_p,5)};
```

Compute the error norms $\|u_h(\cdot,1) - u(\cdot,1)\|_{L^2(\Omega)}$ for all the meshes by means of a simple local 1-point quadrature formula based on the barycenters of the mesh cells and write them to a `.csv` file together with the number of cells of the meshes. Plot relevant error curves by a suitable PYTHON or MATLAB script.

HIDDEN HINT 1 for (11-8.o) → 11-8-15-0:hb1.pdf

SOLUTION for (11-8.o) → 11-8-15-1:s3.pdf ▲

(11-8.p) ⚂ (20 min.) [depends on Sub-problem (11-8.m) and prbautorefsp:7]

In the file `advectionfv2d.h` implement a C++ function

```
template <typename VECTORFIELD>
int findCFLthreshold(const lf::assemble::DofHandler &dofh,
   VECTORFIELD &&beta,
                     double T);
```

that uses a bisection approach to determine the CFL-threshold on the timestep when using the current fully discrete evolution to solve (11.8.1) over the time interval $[0,T]$. The arguments have the same meaning as the arguments of the same name to `solveAdvection2D()` from Sub-problem (11-8.m).

Inside the bisection loop call `solveAdvection2D()` and catch possible **std::overflow_error** exceptions. As a "safe" number of timesteps to start with use 10 times the number as suggested by the formula found in Sub-problem (11-8.k).

Use a random initial coefficient vector created by `Eigen::VectorXd::Random()`.

SOLUTION for (11-8.p) → 11-8-16-0:.pdf ▲

(11-8.q) ⚂ (15 min.) [depends on Sub-problem (11-8.p) and Sub-problem (11-8.n)]

Further extend the `main()`-function in the file `advectionfv2d_main.cc` so that it also outputs the CFL-threshold for the timestep in the `.csv`-file requested in Sub-problem (11-8.n).

Create a plot of the found maximal timesteps versus the numbers N of cells of the meshes.

SOLUTION for (11-8.q) → 11-8-17-0:s11.pdf ▲

End Problem 11-8, 325 min.

Problem 11-9: Non-Linear Conservation Laws with Source Terms

In [Lecture → Section 11.2], [Lecture → (11.2.1.5)] we have seen a scalar conservation law with a right-hand-side source term. However, for most considerations and the derivation of numerical methods such source terms were ignored. In this problem we extend conservative finite volume methods to 1D scalar conservation laws with source terms.

This problem assumes familiarity with [Lecture → Section 11.3], in particular [Lecture → Section 11.3.2] and [Lecture → Section 11.3.4.4]. Parts rely on ideas from [Lecture → Section 11.3.5].
▷ problem name/problem code folder: ConsLawWithSource

For a given continuously differentiable flux function $f : \mathbb{R} \to \mathbb{R}$ and a continuous source function $s : \mathbb{R} \to \mathbb{R}$ we consider the following Cauchy problem for a 1D scalar conservation law:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= s(u) \quad \text{in} \quad \mathbb{R}\times]0,T[\,, \\ u(x,0) &= u_0(x) \quad \forall x \in \mathbb{R} \,. \end{aligned} \tag{11.9.1}$$

(11-9.a) ⚀ (15 min.) State the spatially semi-discrete evolution (an "ordinary differential equation" on $\mathbb{R}^{\mathbb{Z}}$) that arises from a conservative finite-volume discretization of (11.9.1) on an equidistant spatial mesh

$$\mathcal{M} := \{]x_{j-1}, x_j[: x_j := jh,\ j \in \mathbb{Z} \} \,,$$

with cell size $h > 0$, when we use a 2-point numerical flux function $(v,w) \mapsto F(v,w)$. Resort to the 1-point Gauss quadrature rule ("midpoint rule") to deal with the source term.

SOLUTION for (11-9.a) → 11-9-1-0:sl1.pdf ▲

(11-9.b) ⚁ (12 min.) [depends on Sub-problem (11-9.a)]

[Lecture → Code 11.3.2.9] presents a templated C++ function realizing $-h\cdot$ the right-hand-side vectorfield of the ODE arising from a conservative finite-volume spatial semidiscretization of (11.9.1) with $s \equiv 0$ and on a finite section of $\mathcal{M}$. The source code for this function can be found in the file `consformevl.h`.

In the file `conslawwithsource.h` implement the templated C++ function

```
template <typename FunctionF, typename SourceFunction>
  Eigen::VectorXd fluxdiffsource(
    const Eigen::VectorXd &mu, FunctionF &&F, SourceFunction &&s,
    double h);
```

that extends `fluxdiff()` so that it can deal with conservation laws of the form (11.9.1). The vector `mu` passes cell avarages, the arguments `F` and `s` are functor objects encoding the numerical flux function F and the source function s. The parameter `h` is supposed to specify the meshwidth h. Again, this function should return $-h\cdot$ the right-hand-side vectorfield for semi-discrete evolution. As in `fluxdiff()` constant continuation of the states beyond the finite interval is assumed.

SOLUTION for (11-9.b) → 11-9-2-0:sol2.pdf ▲

(11-9.c) ⚄ (15 min.) [depends on Sub-problem (11-9.a)]

We consider the conservative finite-volume semidiscretization of (11.9.1) as derived in Sub-problem (11-9.a). We take for granted that

- the numerical flux function $F : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is consistent with f and monotone.

Definition [Lecture → Def. 11.3.5.5]. Monotone numerical flux function

A 2-point numerical flux function $F = F(v,w)$ is called monotone, if

F is an increasing function of its first argument v $(\forall w)$

and

F is a decreasing function of its second argument w $(\forall v)$.

- the source function $s : \mathbb{R} \to \mathbb{R}$ satisfies $s(0) = 0$.

Show that under these assumptions

$$u_0(x) \geq 0 \quad \forall x \in \mathbb{R} \quad \Longrightarrow \quad \mu_j(t) \geq 0 \quad \forall j \in \mathbb{Z}\,, \quad \forall t \in [0,T]\,.$$

HIDDEN HINT 1 for (11-9.c) → 11-9-3-0:h31.pdf

SOLUTION for (11-9.c) → 11-9-3-1:sd5.pdf ▲

In the sequel we focus on the special flux function

$$f(u) := \exp(u) - u\,, \quad u \in \mathbb{R}\,, \tag{11.9.5}$$

and the source function

$$s(u) := -u\,, \quad u \in \mathbb{R}\,. \tag{11.9.6}$$

(11-9.d) ⚀ (10 min.) Write down the entropy solution of the Riemann problem

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= 0 \quad \text{in} \quad \mathbb{R} \times \mathbb{R}\,, \\ u(x,0) &= \begin{cases} 1 & \text{for } x < 0\,, \\ -1 & \text{for } x \geq 0\,. \end{cases} \end{aligned} \tag{11.9.7}$$

SOLUTION for (11-9.d) → 11-9-4-0:scp3.pdf ▲

(11-9.e) ⚁ (15 min.) Now give the entropy solution of the "inverted" Riemann problem

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= 0 \quad \text{in} \quad \mathbb{R} \times \mathbb{R}\,, \\ u(x,0) &= \begin{cases} -1 & \text{for } x < 0\,, \\ 1 & \text{for } x \geq 0\,. \end{cases} \end{aligned} \tag{11.9.8}$$

SOLUTION for (11-9.e) → 11-9-5-0:s5.pdf ▲

(11-9.f) ⚂ (15 min.) In the file `conslawwithsource.cc` implement a C++ function

```
double godnfn(double v, double w);
```

that provides the Godunov numerical flux [Lecture → Section 11.3.4.4] arising from the flux function f as given in (11.9.5).

SOLUTION for (11-9.f) → 11-9-6-0:.pdf ▲

(11-9.g) ⚄ (25 min.) [depends on Sub-problem (11-9.b) and Sub-problem (11-9.f)]

Based on the function `fluxdiffsource()` in the file `conslawwithsource.h` finish the implementation of the C++ function

```
template <typename U0Functor>
Eigen::VectorXd traceMass(U0Functor &&u0, unsigned int N);
```

that uses the conservative finite-volume spatial discretization of Sub-problem (11-9.a) based on the 2-point Godunov flux available through `godnfn()` from Sub-problem (11-9.f), combined with explicit Euler timestepping to solve (11.9.1) for the special choices (11.9.5) and (11.9.6) on the finite spatial interval $[-5,10]$, for the time period $[0,3]$, on a mesh with `N` cells, and with initial data $u_0 \in C^0(\mathbb{R})$ passed through the functor object `u0`.

Assuming

$$0 \le u_0(x) \le 1 \quad \forall x \in \mathbb{R} \quad , \quad u_0(x) = 0 \quad \text{for} \quad x \notin]0,1[\, ,$$

choose the maximal uniform timestep $\tau > 0$ compatible with the CFL condition [Lecture → Section 11.4.2].

The function should return an approximation of the "total mass"

$$m(t) := \int_{\mathbb{R}} u(x,t) \, \mathrm{d}x \, , \quad 0 \le t \le 3 \, , \tag{11.9.13}$$

for every timestep in a vector $\approx [m(0), m(\tau), m(2\tau), \ldots, m(M\tau)]^\top$, where M is the total number of timesteps, $M := 3/\tau$.

HIDDEN HINT 1 for (11-9.g) → `11-9-7-0:h61.pdf`

HIDDEN HINT 2 for (11-9.g) → `11-9-7-1:h62.pdf`

SOLUTION for (11-9.g) → `11-9-7-2:s6.pdf` ▲

(11-9.h) ⚄ (10 min.) Derive an explicit formula for the total mass $m(t)$ as defined in (11.9.13), where $u = u(x,t)$ is the entropy solution of (11.9.1) with the choices (11.9.5) and (11.9.6) and for

$$u_0(x) = \begin{cases} 1 & \text{for} \quad 0 \le x < 1 \, , \\ 0 & \text{elsewhere.} \end{cases}$$

SOLUTION for (11-9.h) → `11-9-8-0:.pdf` ▲

End Problem 11-9 , 117 min.

Problem 11-10: Flux-Limited Finite-Volume Methods for Scalar Conservation Laws

An alternative to the method-of-lines approach, which separates spatial and temporal discretization are fully discrete conservative finite-volume schemes, which will be examined in this problem. They involve numerical fluxes, which may also depend on the timestep size. This paves the way for designing special higher(= 2nd)-order methods.

You have to know the material from [Lecture → Section 11.2.1], [Lecture → Section 11.3.2], and [Lecture → Section 11.5.2]. Implementation in C++ will rely on EIGEN.
▷ problem name/problem code folder: FluxLimitedFV

We consider the Cauchy problem for a generic scalar conservation law in one spatial dimension:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= 0 \qquad \text{on} \quad \mathbb{R} \times \mathbb{R}_0^+ , \\ u(x,0) &= u_0(x) \quad \forall x \in \mathbb{R} , \end{aligned} \tag{11.10.1}$$

where $f : \mathbb{R} \to \mathbb{R}$ is a smooth flux function and $u_0 \in L^\infty(\mathbb{R})$ is given and constant outside a bounded spatial interval.

We discretize (11.10.1) on an equidistant infinite spatio-temporal mesh with spatial meshwidth $h > 0$ and temporal meshwidth (timestep size) $\tau > 0$. We set

$$x_j := hj , \quad x_{j+\frac{1}{2}} := \tfrac{1}{2}(x_j + x_{j+1}) , \quad j \in \mathbb{Z} \quad , \quad t_k := \tau k , \quad k \in \mathbb{N}_0 .$$

(11-10.a) ⚀ (10 min.) Write $u = u(x,t)$ for the unique entropy solution of (11.10.1). The dual cell averages

$$\rho_j^{(k)} := \frac{1}{h} \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t_k) \,\mathrm{d}x , \quad j \in \mathbb{Z} , \quad k \in \mathbb{N}_0 , \tag{11.10.2}$$

satisfy a recursion

$$\rho_j^{(k+1)} = \rho_j^{(k)} - \gamma(\widetilde{f}_{j+\frac{1}{2}} - \widetilde{f}_{j-\frac{1}{2}}) , \quad \gamma := \frac{\tau}{h} , \quad j \in \mathbb{Z}, \, k \in \mathbb{N}_0 , \tag{11.10.3}$$

with suitably defined fluxes $\widetilde{f}_{j+\frac{1}{2}}, \widetilde{f}_{j-\frac{1}{2}}$. Tick the correct formula(s) for those:

- ○ $\widetilde{f}_{\ell+\frac{1}{2}} = \int_{t_k}^{t_{k+1}} f(u(x_\ell,t)) \,\mathrm{d}t ,$
- ○ $\widetilde{f}_{\ell+\frac{1}{2}} = \frac{1}{\tau} \int_{t_k}^{t_{k+1}} \int_{x_{\ell-\frac{1}{2}}}^{x_{\ell+\frac{1}{2}}} \frac{\partial}{\partial x} f(u(x,t)) \,\mathrm{d}x\mathrm{d}t , \quad \ell = j-1, j .$
- ○ $\widetilde{f}_{\ell+\frac{1}{2}} = f(u(x_{\ell+\frac{1}{2}}, t_{k+1})) - f(u(x_{\ell+\frac{1}{2}}, t_k)) ,$
- ○ $\widetilde{f}_{\ell+\frac{1}{2}} = \frac{1}{\tau} \int_{t_k}^{t_{k+1}} f(u(x_{\ell+\frac{1}{2}}, t)) \,\mathrm{d}t ,$

SOLUTION for (11-10.a) → 11-10-1-0:sol1.pdf ▲

(11-10.b) ⚂ (10 min.) In order to discretize (11.10.1) on a uniform space-time mesh with spatial meshwidth $h > 0$ and timestep $\tau > 0$ we can follow [Lecture → Section 11.3.2] and [Lecture → Section 11.4.1], adopt the method-of-lines approach,

(I) cast (11.10.1) into a spatially semi-discrete conservation form using a *2-point* numerical flux $F = F(v,w)$,

(II) and apply explicit Euler timestepping to the resulting semi-discrete evolution.

This results in a recursion

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \gamma\left(f_{j+\frac{1}{2}} - f_{j-\frac{1}{2}}\right), \quad j \in \mathbb{Z}, \quad k \in \mathbb{N}_0. \tag{11.10.5}$$

Give a concrete expression for the **flux term** $f_{\ell+\frac{1}{2}}$, $\ell \in \mathbb{Z}$, as a function of suitable states $\mu_i^{(k)}$, $i \in \mathbb{Z}$.

SOLUTION for (11-10.b) → 11-10-2-0:sl2.pdf ▲

A **flux-limited finite-volume discretization** of (11.10.1) on an equidistant space-time grid employs the recursion

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \gamma\left(f_{j+\frac{1}{2}} - f_{j-\frac{1}{2}}\right), \quad j \in \mathbb{Z}, \quad k \in \mathbb{N}_0. \tag{11.10.5}$$

with the following definition of the flux terms: For every $\ell \in \mathbb{Z}$ we set

$$f_{\ell+\frac{1}{2}} := F_{\mathrm{GD}}(v,w) + f^{\mathrm{D}}_{\ell+\frac{1}{2}}, \quad v := \mu_\ell^{(k)}, \quad w := \mu_{\ell+1}^{(k)}, \tag{11.10.6a}$$

$$f^{\mathrm{D}}_{\ell+\frac{1}{2}} := \begin{cases} \frac{1}{2}|\dot{s}|(1-\gamma|\dot{s}|)\varphi(\theta)(w-v) & \text{, if } v \neq w, \\ 0 & \text{, if } v = w, \end{cases} \tag{11.10.6b}$$

with

$$\dot{s} := \frac{f(w)-f(v)}{w-v} \quad \text{(Rankine-Hugoniot speed)}, \tag{11.10.6c}$$

$$\theta := \begin{cases} \dfrac{\mu_\ell^{(k)} - \mu_{\ell-1}^{(k)}}{\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}} & \text{, if } \dot{s} \geq 0, \\ \dfrac{\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}}{\mu_{\ell+2}^{(k)} - \mu_{\ell+1}^{(k)}} & \text{, if } \dot{s} < 0. \end{cases} \tag{11.10.6d}$$

Here, F_{GD} is the 2-point **Godunov numerical flux** for the flux function f as introduced in [Lecture → Section 11.3.4.4], [Lecture → (11.3.4.33)], and $\varphi : \mathbb{R} \cup \{\pm\infty\} \to \mathbb{R}$ is a so-called **flux limiter function**, which has to have finite limits, when its argument tends to $\pm\infty$. If a division by zero occurs in the definition (11.10.6d) of θ, then, formally we set $\theta := \pm\infty$, the sign determined by the numerator in (11.10.6d).

(11-10.c) ⚂ (20 min.) In this sub-problem we consider the Cauchy problem for scalar linear advection

$$\begin{aligned} \frac{\partial u}{\partial t} + \beta\frac{\partial u}{\partial x} &= 0 \quad \text{on } \mathbb{R} \times \mathbb{R}_0^+, \\ u(x,0) &= u_0(x) \quad \forall x \in \mathbb{R}, \end{aligned} \quad \text{with } \beta > 0. \tag{11.10.7}$$

State the fully discrete evolution (11.10.5) for (11.10.7) with flux term according to (11.10.6) and with a generic flux limiter function φ.

SOLUTION for (11-10.c) → 11-10-3-0:slv3.pdf ▲

(11-10.d) ⚂ (25 min.) [depends on Sub-problem (11-10.c)]

In the file `fluxlimitedfv.h` finish the implementation of the C++ function

```
template <typename FLUXLIM = std::function<double(double)>
Eigen::VectorXd fluxlimAdvection(
double beta, const Eigen::VectorXd &mu0,
double h, double tau, unsigned int nb_timesteps,
FLUXLIM &&phi = [](double /*theta*/) { return 1.0; });
```

that implements the flux-limited finite-volume discretization of (11.10.7) according to (11.10.6) taking for granted $\beta \geq 0$. The arguments provide β, $\mu_j^{(0)}$ ($j \in \{1,\ldots,N\}$, $N \in \mathbb{N}$), $h > 0$, $\tau > 0$, the total number of timesteps to be conducted, and the flux limiter function φ.

The argument `mu0` also defines the finite spatial mesh with nodes $x_j := hj$, $j = 0,\ldots,N-1$, $N :=$`mu0.size()`.

As in [Lecture → Code 11.3.2.9] *constant continuation* of the states beyond the extent of the spatial mesh is to be employed. You may assume $N > 2$.

The function is supposed to return the state vector $\mu_j^{(M)}$, $j \in \{1,\ldots,N\}$, at final time $T = \tau M$, $M :=$ `nb_timesteps`.

HIDDEN HINT 1 for (11-10.d) → `11-10-4-0:dbz.pdf`

SOLUTION for (11-10.d) → `11-10-4-1:s4.pdf` ▲

(11-10.e) ⚂ (10 min.) The following table lists the (approximate, computed by means of the local mid-point quadrature rule) error norm $\|u(\cdot,1) - u_M\|_{L^1(\mathbb{R})}$, where u is the solution of (11.10.7) with $\beta = 1$ and u_M stands for the piecewise constant reconstruction of the flux-limited finite-volume solution computed with the function `fluxlimAdvection()` from Sub-problem (11-10.d) with $M \in \mathbb{N}$ timesteps over the time interval $]0,1[$ using $h = 1.2\tau$, $\tau := 1/M$, and a sufficiently large spatial mesh to capture all non-constant parts of the solution for all times.

The initial state vectors $\mu_j^{(0)}$, $j \in \mathbb{Z}$, are obtained by sampling at the mesh nodes the functions

$$u_0^A(x) = \begin{cases} 0 & \text{for } x \leq 1\,, \\ \sin^2(\frac{\pi}{2}(x-1)) & \text{for } 1 \leq x \leq 2\,, \\ 1 & \text{for } x \geq 2\,, \end{cases} \quad \text{or} \quad u_0^B(x) = \begin{cases} 0 & \text{for } x \leq 1\,, \\ 1 & \text{for } 1 \leq x \leq 2\,, \\ 0 & \text{for } x \geq 2\,. \end{cases}$$

Moreover, the computations relied on the flux limiter function

$$\varphi(\theta) := \frac{|\theta| + \theta}{1 + |\theta|}\,, \quad \theta \in \mathbb{R}\,. \tag{11.10.11}$$

M	L^1-error for $u_0^?$	L^1-error for $u_0^?$
10	0.15109	0.00450
20	0.10108	0.00108
40	0.05614	0.000271
80	0.03578	6.46e-05
160	0.02064	1.53e-05
320	0.01283	3.64e-06
640	0.00755	8.71e-07
1280	0.00465	2.10e-07

Describe qualitatively and quantitatively the empiric asymptotic behavior of the error norms given in the two columns of the table.

Decide, which column belongs to which initial data (Fill in 'A' or 'B'):

Middle column: ☐

Right column: ☐

SOLUTION for (11-10.e) → 11-10-5-0:s5.pdf

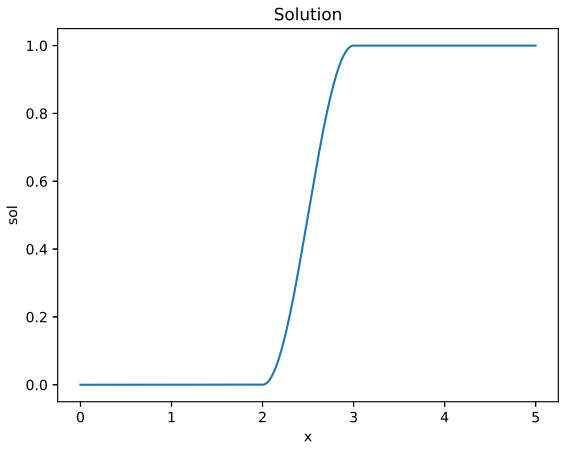


Fig. 200

FV-solution, $M = 1280$, $u_0 = u_0^A$

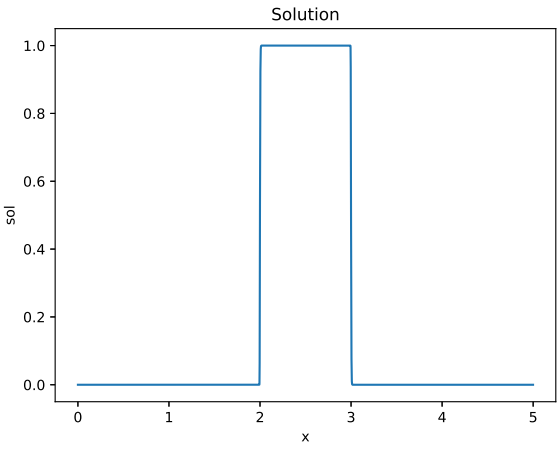


Fig. 201

FV-solution, $M = 1280$, $u_0 = u_0^B$

▲

(11-10.f) ⚁ (15 min.) In the file `fluxlimitedfv.h` inside the function `fluxlimBurgers()` complete the implementation of the lambda function

```
auto godnfnburgers = [](double v,double w) -> double {};
```

that realizes the Godunov numerical flux F_{GD} for the Burgers flux function $f(u) = \frac{1}{2}u^2$.

SOLUTION for (11-10.f) → 11-10-6-0:sl5a.pdf ▲

(11-10.g) ⚂ (25 min.) [depends on Sub-problem (11-10.f)]

In the file `fluxlimitedfv.h` supplement the missing parts of the code for the C++ function

```
template <typename FLUXLIM = std::function<double(double)>
Eigen::VectorXd fluxlimBurgers(
const Eigen::VectorXd &mu0,
double h, double tau, unsigned int nb_timesteps,
FLUXLIM &&phi = [](double /*theta*/) { return 1.0; });
```

which is to use the flux-limited finite-volume discretization according to (11.10.6) to solve Cauchy problems for Burgers equation. The purpose of the arguments and the meaning of the returned vector are the same as for `fluxlimAdvection()` from Sub-problem (11-10.d).

As in Sub-problem (11-10.d) constant continuation of state data beyond the spatial interval covered by the mesh is to be employed.

SOLUTION for (11-10.g) → 11-10-7-0:sol6.pdf ▲

End Problem 11-10 , 115 min.

Problem 11-11: Conservation laws with non-linear density

In the standard form of a conservation law the density is used as state variable and also serves as the main unknown in the evolution problem. Yet, other variables may be used as state and the standard form of the conservation law may not be available directly. In this problem we study that case in one dimension.

This problem involves some coding in C++ using the EIGEN library.
▷ problem name/problem code folder: TransformedConsLaw

We consider the following one-dimensional Cauchy problem: We seek $z : \mathbb{R}\times]0,T[\to\mathbb{R}$, $z = z(x,t)$, with final time $T > 0$, satisfying

$$\begin{aligned} \frac{\partial}{\partial t}\rho(z) + \frac{\partial}{\partial x}g(z) = 0 \quad \text{in} \quad \mathbb{R}\times]0,T[\,, \\ z(x,0) = z_0(x) \quad \forall x \in \mathbb{R}\,. \end{aligned} \tag{11.11.1}$$

Here,

- both $g : \mathbb{R} \to \mathbb{R}$ and $\rho : \mathbb{R} \to \mathbb{R}$ are smooth functions,
- and ρ is strictly increasing.

We assume a minimal steepness of ρ:

$$\exists \gamma > 0: \quad \rho'(\xi) \geq \gamma \quad \text{for all values } \xi \text{ that } z \text{ can attain.} \tag{11.11.2}$$

(11-11.a) ⚀ (15 min.) If z solves (11.11.1), then $u := \rho \circ z$ (which means $u(x,t) := \rho(z(x,t))$) solves a one-dimensional scalar conservation law in standard form $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}f(u) = 0$. What is the flux function $f : \mathbb{R} \to \mathbb{R}$?

$f(u) =$.

SOLUTION for (11-11.a) → 11-11-1-0:tcls1.pdf ▲

(11-11.b) ⚁ (15 min.) Which of the following assertions are true for compactly supported z_0 and all ρ, g satisfying the above properties?

			true	false
(A)	$z^- \leq z_0(x) \leq z^+$	$\Rightarrow \quad z^- \leq z(x,t) \leq z^+ \quad \forall (x,t)$	○	○
(B)	$\int_{-\infty}^{\infty} z(x,t)\,\mathrm{d}x = \int_{-\infty}^{\infty} z_0(x)\,\mathrm{d}x \quad \forall t \in]0,T[$		○	○
(C)	$\int_{-\infty}^{\infty} \lvert z(x,t)\rvert\,\mathrm{d}x \leq \int_{-\infty}^{\infty} \lvert z_0(x)\rvert\,\mathrm{d}x \quad \forall t \in]0,T[$		○	○
(D)	z_0 has three local extrema	$\Rightarrow \quad x \mapsto z(x,t)$ has at most three local extrema	○	○
(E)	$z_0 \in C^\infty(\mathbb{R}) \quad \Rightarrow \quad x \mapsto z(x,t) \in C^0(\mathbb{R})$		○	○

SOLUTION for (11-11.b) → 11-11-2-0:tcls2.pdf ▲

(11-11.c) ⚄ (25 min.) [depends on Sub-problem (11-11.a)]

Assume that z_0 attains a constant value z^* outside $]-1,1[$. What is the maximal spatial interval $I = I(t)$ such that $x \mapsto z(x,t)$ can be different from z^* for $x \in I$. Express I depending on g, t, z_0, and the bounds from (11.11.2).

$I(t) =$.

SOLUTION for (11-11.c) → 11-11-3-0:tcl4s.pdf ▲

In a C++ code we describe the Cauchy problem (11.11.1) by an object of the type

```
class NonStdCauchyProblemCL {
 public:
  NonStdCauchyProblemCL() = default;
  virtual ~NonStdCauchyProblemCL() = default;
  double rho(double z) const;  // Function z ↦ ρ(z)
  double drho(double z) const; // Function z ↦ ρ'(z)
  double g(double z) const;    // Function z ↦ g(z)
  double dg(double z) const;   // Function z ↦ g'(z)
  std::pair<double,double> domain() const;
  double T() const;            // Final time T
  double z0(double x) const;   // Initial data x ↦ z_0(x)
 private:
  .....
};
```

The member function have the following meaning: `rho()`, `drho()`, `g()`, and `dg()` provide the functions $z \mapsto \rho(z)$, $z \mapsto \rho'(z)$, $z \mapsto g(z)$, and $z \mapsto g'(z)$, respectively. `domain()` returns bounds of a finite spatial interval, on which the solution is of interest, and `T()` gives the final time. The initial data z_0 are supplied by `z0()`.

For the spatial discretization of (11.11.1)/(**??**) we use a conservative finite volume method on an equidistant spatial mesh employing the (simplified) **Rusanov 2-point numerical flux** [Lecture → Section 11.3.4.2]. For a scalar one-dimensional conservation law in standard form $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0$ that numerical flux function reads

$$F_R(v,w) := \tfrac{1}{2}\big(f(v) + f(w)\big) - \tfrac{1}{2}(w-v) \cdot \max\{|f'(v)|, |f'(w)|\} \,. \tag{11.11.5}$$

We denote by $\mu_j(t)$, $\zeta_j(t)$, $j = 1,\ldots,N$, the values of the piecewise constant functions approximating $x \mapsto u(x,t)$ and $x \mapsto z(x,t)$ on a mesh with N (dual) cells. These values are to satisfy $\mu_j = \rho(\zeta_j)$.

(11-11.d) ⚂ (45 min.) In the file `transformedconslaw.h` complete the implementation of the C++ function

```
template <typename RHOFUNCTOR, typename DRHOFUNCTOR>
  double rhoInverse(double u, double z0,
    RHOFUNCTOR &&rho, DRHOFUNCTOR &&drho,
    double atol = 1E-10, double rtol = 1E-5);
```

This function is supposed to solve the equation $\rho(z) = u$ for z using Newton's method with initial guess `z0`. Correction-based termination is controlled by the absolute tolerance `atol` and relative tolerance `rtol`. The functors `rho` and `drho` must comply with `std::function<double(double)>`.

SOLUTION for (11-11.d) → 11-11-4-0:newtcl.pdf ▲

(11-11.e) ⚄ (45 min.) [depends on Sub-problem (11-11.a)]

In the file `transformedconslaw.h` implement a C++ function template

```
template <class CAUCHYPROBLEM>
  Eigen::VectorXd semiDiscreteRhs(
    const Eigen::VectorXd &mu,
    const Eigen::VectorXd &zeta,
    CAUCHYPROBLEM prb);
```

This function should realize the right-hand-side vector field $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$ of the ordinary differential equation

$$\left[\frac{d\mu_j}{dt}\right]_{j=1}^{N} = \mathbf{f}\Big(\left[\mu_j(t)\right]_{j=1}^{N}\Big) \tag{11.11.7}$$

arising from the conservative spatial finite-volume semi-discretization for $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0$ described above (using the simplified Rusanov 2-point numerical flux).

The arguments `mu` and `zeta` pass the vectors $\left[\mu_j\right]_{j=1}^{N}$ and $\left[\zeta_j\right]_{j=1}^{N}$ giving approximations at current time. As above N is the number of (dual) mesh cells covering the interval returned by the `domain()` member function of `prb`, which is to be an object compatible with **NonStdCauchyProblemCL**. Constant continuation of the states outside that interval is assumed.

SOLUTION for (11-11.e) → `11-11-5-0:tcl5s.pdf` ▲

(11-11.f) ⚄ (45 min.) [depends on Sub-problem (11-11.a) and Sub-problem (11-11.e)]

We combine the spatial semi-discretization introduced above with an explicit Runge-Kutta timestepping scheme described by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ \frac{1}{3} & \frac{1}{3} & 0 & 0 \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \tag{11.11.9}$$

and using uniform timesteps.

The resulting fully discrete conservative finite volume scheme should be realized in the file `transformedconslaw.h` in the C++ function termplate

```
template <class CAUCHYPROBLEM, typename RECORDER =
          std::function<void(double, const Eigen::VectorXd &)>>
  Eigen::VectorXd solveCauchyPrb(
    unsigned int M, unsigned int N, CAUCHYPROBLEM prb,
    RECORDER &&rec = [](
      double /*time*/,
      const Eigen::VectorXd & /*zstate*/) -> void {});
```

The argument `N` passes the number of dual cells, `prb` an object compatible with **NonStdCauchyProblemCL**. The argument `M` specifies the number of equidistant timestep from 0 to final time T.

Of course, the function `semiDiscreteRhs()` from Sub-problem (11-11.e) should be used. To invert the function ρ use the C++ function `rhoInverse()` from Sub-problem (11-11.d) with a smartly chosen initial guess and correction-based termination with absolute tolerance 10^{-10} and relative tolerance 10^{-5}.

The returned vector is to contain an approximation of $\left[\zeta_j(T)\right]_{j=1}^N$. In every timestep the current approximation of $\left[\zeta_j(t_k)\right]_{j=1}^N$, $t_k := k\frac{T}{M}$, $k = 0,\ldots,M$, should be given to the object `rec` along with t_k.

SOLUTION for (11-11.f) → `11-11-6-0:tclsp6.pdf` ▲

End Problem 11-11 , 190 min.

Problem 11-12: Finite Volumme Methods for 1D Damped Wave Equation

In this problem, along the lines of [Lecture → Ex. 11.6.1.5], we first recast the linear damped wave equation in one spatial dimension as a linear system of conservation laws with source term. Then we derive finite volume methods based on upwinding and the Lax-Friedrich numerical flux and study the solution of Riemann problems produced by them. We also examine boundary conditions.

This problem addresses concepts and techniques introduced in [Lecture → Section 11.6.1], [Lecture → Section 11.6.3.1], and [Lecture → Section 11.6.3.2].
▷ problem name/problem code folder: LinHypDampWaveSys

As a slight generalization of [Lecture → (11.6.1.4)] we consider the Cauchy problem for the constant-coefficient **scalar linear demped wave equation** in 2nd-order form: find a function $p : \mathbb{R} \times \mathbb{R}^+ \to \mathbb{R}$, $p = p(x,t)$, solving

$$\frac{\partial^2 p}{\partial t^2}(x,t) + r\frac{\partial p}{\partial t}(x,t) - \frac{\partial}{\partial x}\left(c^2\frac{\partial p}{\partial x}(x,t)\right) = 0 \quad \text{in} \quad \mathbb{R}\times\mathbb{R}^+ , \tag{11.12.1a}$$

$$p(x,0) = p_0(x) , \quad \frac{\partial p}{\partial t}(x,0) = v_0(x) , \quad x \in \mathbb{R} , \tag{11.12.1b}$$

with damping coefficient $r > 0$, wave speed $c > 0$, and given initial pressure $p_0 : \mathbb{R} \to \mathbb{R}$ and initial velocity $v_0 : \mathbb{R} \to \mathbb{R}$. The initial data are assumed to be constant outside a compact interval $[a,b] \subset \mathbb{R}$.

(11-12.a) ⚀ (15 min.) If p has physical dimension $[p] = 1\mathrm{Nm}^{-2}$ (SI units), then what are the physical dimensions of the coefficients r and c?

SOLUTION for (11-12.a) → 11-12-1-0:dweqs1.pdf ▲

(11-12.b) ⚁ (20 min.) For $p_0(x) = \sin(\omega x)$, $\omega \in \mathbb{R}$, and $v_0 \equiv 0$ determine the *bounded* classical solution of (11.12.1).

HIDDEN HINT 1 for (11-12.b) → 11-12-2-0:dweqh1.pdf

SOLUTION for (11-12.b) → 11-12-2-1:dwes1a.pdf ▲

(11-12.c) ⚂ (15 min.) As in [Lecture → Ex. 11.6.1.5] introducing the new unknown functions

$$w(x,t) = \frac{\partial p}{\partial x}(x,t) \quad , \quad v(x,t) = \frac{\partial p}{\partial t}(x,t) , \tag{Lecture → (11.6.1.7)}$$

converts (11.12.1) into a Cauchy problem for a linear system of conservation laws [Lecture → Def. 11.6.1.1] with **source term**

$$\frac{\partial \mathbf{u}}{\partial t}(x,t) + \mathbf{A}\frac{\partial \mathbf{u}}{\partial x}(x,t) = \mathbf{g}(\mathbf{u}(x,t)) \quad \text{in} \quad \mathbb{R}\times\mathbb{R}^+ . \tag{11.12.4}$$

Define $\mathbf{u} = \mathbf{u}(x,t)$, find the matrix $\mathbf{A}$, and a suitable expression for the function $\mathbf{g} : \mathbb{R}^2 \to \mathbb{R}^2$.

SOLUTION for (11-12.c) → 11-12-3-0:dweqs2.pdf ▲

(11-12.d) ⚂ (20 min.) [depends on Sub-problem (11-12.c)]

Show that the **energy**

$$E(t) := \tfrac{1}{2}\int_{\mathbb{R}} \mathbf{u}(x,t)^\top \mathbf{D}\mathbf{u}(x,t)\,\mathrm{d}x , \quad \mathbf{D} := \begin{bmatrix} 1 & 0 \\ 0 & c^2 \end{bmatrix} , \tag{11.12.5}$$

is non-increasing, if $(x,t) \mapsto \mathbf{u}(x,t)$ solves the linear hyperbolic system (11.12.4) derived in Sub-problem (11-12.c).

SOLUTION for (11-12.d) → 11-12-4-0:dweqs3.pdf ▲

(11-12.e) ⚂ (10 min.) What is the **domain of influence** of $[a,b] \subset \mathbb{R}$ in the Cauchy problem for the linear hyperbolic system (11.12.4) derived in Sub-problem (11-12.c).

HIDDEN HINT 1 for (11-12.e) → 11-12-5-0:dweq4h.pdf

SOLUTION for (11-12.e) → 11-12-5-1:dweqs4.pdf ▲

(11-12.f) ⚄ (20 min.) Taking the cue from [Lecture → Section 11.6.3.1] derive the (infinite-dimensional) ordinary differential equation arising from a finite-volume spatial semi-discretization of (11.12.4) with given two-point numerical flux $\widehat{\mathbf{F}} : \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}^2$ and on an equidistant (infinite) spatial mesh with meshwidth $h > 0$.

HIDDEN HINT 1 for (11-12.f) → 11-12-6-0:dweqh5.pdf

SOLUTION for (11-12.f) → 11-12-6-1:dweqs5.pdf ▲

(11-12.g) ⚂ (20 min.) In the file `linhypdampwavesys.h` implement a templated C++ function

```
template <typename FFunctor, typename gFunctor>
Eigen::MatrixXd linsysoderhs(
  const Eigen::MatrixXd &mu, double h,
  FFunctor &&numflux, gFunctor &&g);
```

that provides the right-hand side for the ordinary differential equation derived in Sub-problem (11-12.f). The argument `numflux` passes the numerical flux function $\widehat{\mathbf{F}}$, `g` the function $\mathbf{g}$, and `h` the spatial meshwidth.

Of course, the infinite sequence of dual cell averages has to be truncated in the same way as in [Lecture → Code 11.3.2.9]. That code can serve as a template.

Note that we assume that the state vectors $\boldsymbol{\mu}_j$ approximating averages over dual cells are supposed to be stored in the columns of a matrix $\left[\boldsymbol{\mu}_{-n}(t), \ldots, \boldsymbol{\mu}_m(t)\right]$.

SOLUTION for (11-12.g) → 11-12-7-0:dweqs6.pdf ▲

(11-12.h) ⚂ (30 min.) [depends on Sub-problem (11-12.g)]

In the file `linhypdampwavesys.h` implement the C++ function

```
template <typename FFunctor,
          typename gFunctor,
          typename u0Functor,
          typename RECORDER = std::function<void(const
             Eigen::MatrixXd &)>
Eigen::MatrixXd fvEulLinSys(
  double a, double b, double T, unsigned int N, unsigned int M,
  u0Functor &&u0, FFunctor &numflux, gFunctor &&f,
  RECORDER &&recorder = [](const Eigen::MatrixXd &) -> void {});
```

that implements a full discretization of the linear system of conservation laws (11.12.4) on the spatial interval $[a,b] \subset \mathbb{R}$

- using a finite volumne discretization in space on an equidistant mesh with N cells,
- based on a numerical flux function $\widehat{\mathbf{F}}$ passed through `numflux`,
- using M uniform timestep of the explicit Euler method over the time interval $[0,T]$.

The initial data are supplied through the functor `u0`, compatible with

`std::function<Eigen::VectorXd(double)>`. The `recorder` argument is given the state matrix $[\boldsymbol{\mu}_{-n}(t),\ldots,\boldsymbol{\mu}_m(t)]$, $N = m+n+1$, in every timestep.

SOLUTION for (11-12.h) → `11-12-8-0:.pdf` ▲

(11-12.i) ⚄ In the file `linhypdampwavesys.h` realize the C++ function

```
template <typename u0Functor,
          typename RECORDER = std::function<void(const
            Eigen::MatrixXd &)>
Eigen::Matrix<double, 2, Eigen::Dynamic> evlDampWave(
  double c, double r, double T, unsigned int N,
  u0Functor &&u0, Fluxtype nf_selector,
  RECORDER &&recorder = [](const Eigen::MatrixXd &) -> void {});
```

which, based on `fvEulLinSys()` from Sub-problem (11-12.h), solves the damped linear wave equation (11.12.1) (in the form (11.12.4)) with coefficients c and r for a duration $T > 0$. The functor `u0`, compatible with cppcodestd::function<Eigen::Vector2d(double)>, provides the initial data $\mathbf{u}_0 = \mathbf{u}_0(x)$.

It can be taken for granted that $\mathbf{u}_0$ is *supported in* $[0,1]$. Then chose the spatial domain for the simulation such the domain of influence of $[0,1]$ at time T is just contained in it. The argument `N` gives the number of cells of an equidistant mesh covering that interval.

The size of the timestep should be chosen as large as possible as allowed by the **CFL-condition**.

The numerical flux to be used is selected by the argument `nf_selector` of type

```
enum Fluxtype { LF, UW };
```

Here `LF` means that the **Lax-Friedrichs numerical flux** [Lecture → (11.6.3.21)] is used, whereas `nf_selector = UW` switches on the **upwind/Godunov numerical flux** [Lecture → (11.6.3.17)].

SOLUTION for (11-12.i) → `11-12-9-0:dweqs8.pdf` ▲

(11-12.j) ⚄ (30 min.) In the file `linhypdampwavesys.cc` write a C++ function

```
void visWavSol(double c, double r, double T, unsigned int N,
  Fluxtype nf_selector)
```

that invokes `evlDampWave()` to solve (11.12.1) (coefficients c, r passed in `c`, `r`) with

$$p_0(x) = \begin{cases} x & \text{for } 0 \le x \le \frac{1}{2}\,, \\ 1-x & \text{for } \frac{1}{2} \le x \le 1\,, qquad v_0 \equiv 0\,. \\ 0 & \text{elsewhere,} \end{cases}$$

over the time interval $[0,T]$.

Write the data for the piecewise constant approximation of $p(x,t)$ to file in the form of a large matrix (size given by the number of spatial dual cells times the number of timesteps) and then, from within the C++ function, call the supplied PYTHON script `visWavSol.p` to create an animated rendering.

From `main()` call `visWavSol()` with $c = 1$, $r = 0$ and $c = 1$, $r = 1$, $N = 500$, $T = 2$, and either numerical flux. Describe your observations.

SOLUTION for (11-12.j) → `11-12-10-0:dweqs9.pdf` ▲

(11-12.k) ⚄ (45 min.) In the file `linhypdampwavesys.cc` write a C++ function

```
Eigen::Matrix<double, Eigen::Dynamic, 2> trackEnergysWavSol(
  double c, double r, double T, unsigned int N, Fluxtype
    nf_selector);
```

that computes the energy E as defined in (11.12.5) of the fully discrete finite-volumne/explicit Euler solution of (11.12.4)/(11.12.1). It is to return the matrix $[\tau i, E(\tau i)]_{i=0}^{M}$, M the total number of timesteps, $\tau > 0$ the timestep size.The arguments of `trackEnergysWavSol()` have the same meaning as those of `visWavSol()`.

Write the result matrix to file and use the supplied PYTHON script `envis.p` to create a plot of the approximation of $t \mapsto E(t)$ over $[0, T]$. Do this for both available numerical fluxes and comment on what you observe.

SOLUTION for (11-12.k) → `11-12-11-0:dweq10s.pdf` ▲

End Problem 11-12 , 225 min.

Problem 11-13: The p-System and its Finite Volume Discretization

This problem studies a translation-invariant non-linear system of conservation laws with two-dimensional state space ($m = 2$). In class the main example was the shallow water equations (SWE) [Lecture → Ex. 11.6.2.13], but now we consider so so-called p-system. In particular we devise finite-volume schemes for it.

Rqquires familiarity with [Lecture → Section 11.6.2], [Lecture → Section 11.6.3.1], and [Lecture → Section 11.6.3.3] and involves implementation in C++.
▷ problem name/problem code folder: FVPsystem

In this problem the spotlight is on the Cauchy problem for the so-called **p-system** of non-linear conservation laws in one spatial dimension,

$$\frac{\partial v}{\partial t} - \frac{\partial w}{\partial x} = 0 \quad , \quad \frac{\partial w}{\partial t} + \frac{\partial p(v)}{\partial x} = 0 \quad \text{in} \quad \mathbb{R} \times \mathbb{R}^+ \, , \tag{11.13.1}$$

where $p : \mathcal{U}_v \subset \mathbb{R} \to \mathbb{R}$ is a smooth function. The p-system is a model of the flow of a compressible fluid through porous media described in Lagrangian coordinates and $p(v)$ can be read as pressure [LeV02, Sect. 2.13]. For $p(v) = a^2/v$, $a > 0$, (11.13.1) is a model for isothermal flow with constant speed of sound a.

(11-13.a) ⚀ (10 min.) Defining the state vector $\mathbf{u}(x,t) := \begin{bmatrix} v(x,t) \\ w(x,t) \end{bmatrix}$ rewrite (11.13.1) in the standard form

$$\frac{\partial \mathbf{u}}{\partial t}(x,t) + \frac{\partial \mathbf{F}(\mathbf{u})}{\partial x}(x,t) = 0 \quad \text{on} \quad D \subset \mathbb{R} \times \mathbb{R}^+ \, , \qquad \text{[Lecture → (11.6.2.3)]}$$

fitting the definition of a non-linear system of conservation laws [Lecture → Def. 11.6.2.2].

SOLUTION for (11-13.a) → 11-13-1-0:psyss1.pdf ▲

(11-13.b) ⚁ Determine necessary and sufficient conditions on the function $p = p(v)$ so that the p-system is **strictly hyperbolic** in the sense of [Lecture → Ass. 11.6.2.6],

Assumption [Lecture → Ass. 11.6.2.6]. Strict hyperbolicity

The Jacobian $D\mathbf{F}(\mathbf{u})$ of the flux function $\mathbf{F} : \mathbb{R}^m \to \mathbb{R}^m$ in [Lecture → (11.6.2.3)] must have m *distinct real eigenvalues* for any $\mathbf{u} \in U$.

SOLUTION for (11-13.b) → 11-13-2-0:psyss2.pdf ▲

In the sequel we will only consider cases where the pressure function $p = p(v)$ is such that the p-system is strictly hyperbolic, see Sub-problem (11-13.b).

(11-13.c) ⚂ (20 min.) Use the **Rankine-Hugoniot jump condition** [Lecture → (11.6.2.20)] of [Lecture → Thm. 11.6.2.19] to show that a shock/discontinuity connecting two constant states $\mathbf{u}_l = \begin{bmatrix} v_l \\ w_l \end{bmatrix} \in \mathbb{R}^2$ and $\mathbf{u}_r = \begin{bmatrix} v_r \\ w_r \end{bmatrix} \in \mathbb{R}^2$ must satisfy

$$w_r = w_l \pm \sqrt{-(p(v_r) - p(v_l))(v_r - v_l)} \, . \tag{11.13.6}$$

SOLUTION for (11-13.c) → 11-13-3-0:psyss3.pdf ▲

(11-13.d) ⚃ (15 min.) [depends on Sub-problem (11-13.c)]

What is the speed of propagation of the shock in the situation studied in the previous sub-problem, that is the case of two constant states $\mathbf{u}_l$ and $\mathbf{u}_r$ separated by the shock. How does the shock speed relate to the eigenvalues of $\mathrm{D}\mathbf{F}(\mathbf{u}_l)$ and $\mathrm{D}\mathbf{F}(\mathbf{u}_r)$?

SOLUTION for (11-13.d) → 11-13-4-0:fvps4.pdf ▲

(11-13.e) ⚂ (30 min.) Recall the notion of a **Hugoniot locus** of a state $\mathbf{u}^*$ from [Lecture → § 11.6.2.21]. Write a PYTHON program that plots the Hugoniot locus $\mathcal{H}(\mathbf{u}^*)$ for the choice $p(v) = -e^v$ and $\mathbf{u}^* = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ for values $-3 \leq v \leq 5$ of the first components of the other state.

SOLUTION for (11-13.e) → 11-13-5-0:psys5s.pdf ▲

(11-13.f) ⚄ (30 min.) By numerically solving a non-linear system of equations[1], determine the two-shock solutions of the Riemann problem for the p-system with $p(v) = -e^v$ and left and right states $\mathbf{u}_l = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$ and $\mathbf{u}_r = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$.

HIDDEN HINT 1 for (11-13.f) → 11-13-6-0:psysh6.pdf

HIDDEN HINT 2 for (11-13.f) → 11-13-6-1:psysh62.pdf

SOLUTION for (11-13.f) → 11-13-6-2:psyss6.pdf ▲

(11-13.g) ⚄ (20 min.) [depends on Sub-problem (11-13.f)]

Do the *all-shock* Riemann solutions found in Sub-problem (11-13.f) satisfy the **Lax entropy condition** of [Lecture → Def. 11.6.2.29]?

Sketch the structure of the solution in the $x-t$-plane. Also indicate the relevant 1-characteristics and 2-characteristics as in [Lecture → Fig. 615], [Lecture → Fig. 616], [Lecture → Fig. 617], and [Lecture → Fig. 618].

SOLUTION for (11-13.g) → 11-13-7-0:psyss7].pdf ▲

(11-13.h) ⚄ (15 min.) [depends on Sub-problem (11-13.a) and Sub-problem (11-13.b)]

We want [Lecture → Ass. 11.6.2.8] to be satisfied: For the p-system (11.13.1) determine a condition on the function $p = p(v)$ so that the eigenvectors $\mathbf{r}_i(\mathbf{u})$ and eigenvalues $\lambda_i(\mathbf{u})$, $i = 1, 2$, of $\mathrm{D}\mathbf{F}(\mathbf{u})$ satisfy

$$\mathbf{grad}\,\mathbf{u}\lambda_i(\mathbf{u}) \cdot \mathbf{r}_i(\mathbf{u}) \neq 0 \quad \text{for all possible states} \quad \mathbf{u}\,, \tag{11.13.11}$$

The strict hyperbolicity of the p-system can be taken for granted, see Sub-problem (11-13.b).

SOLUTION for (11-13.h) → 11-13-8-0:psyss8.pdf ▲

(11-13.i) ⚄ (20 min.) Show that

$$\mathbf{A}_R(\mathbf{u}_l, \mathbf{u}_r) = \begin{bmatrix} 0 & -1 \\ \frac{p(v_{r,1}) - p(v_{l,1})}{v_{r,1} - v_{l,1}} & 0 \end{bmatrix}, \quad \mathbf{u}_l = \begin{bmatrix} v_l \\ w_l \end{bmatrix}, \quad \mathbf{u}_r = \begin{bmatrix} v_r \\ w_r \end{bmatrix}, \tag{11.13.15}$$

is a **Roe matrix** for the p-system (11.13.1) according to [Lecture → Def. 11.6.3.30]. The p-system is assumed to be strictly hyperbolic, recall Sub-problem (11-13.b).

SOLUTION for (11-13.i) → 11-13-9-0:psys9s.pdf ▲

(11-13.j) ⚂ (15 min.) [depends on Sub-problem (11-13.i)]

State the formula for the **Lax-Friedrich/Rusanov numerical flux** [Lecture → (11.6.3.48)] for the p-system with $p(v) = -e^v$ and the Roe matrix from (11.13.15).

SOLUTION for (11-13.j) → 11-13-10-0:psyslfs.pdf ▲

[1] You may use `scipy.optimize.fsolve()` for this purpose

(11-13.k) ⚄ (20 min.) Derive concrete formulas for the **HLLE numerical flux** [Lecture → (11.6.3.38)] for the p-system with $p(v) = -e^v$ and the Roe matrix according to (11.13.15).

SOLUTION for (11-13.k) → 11-13-11-0:psyshlles.pdf ▲

(11-13.l) ⚄ (30 min.) [depends on Sub-problem (11-13.k)]

In the file `fvpsystem.h` implement a C++ function

```
template <typename u0Functor,
          typename RECORDER = std::function<void(const
            Eigen::MatrixXd &)>>
Eigen::MatrixXd evlExpPSystem(
  double a, double b, double T, unsigned int N, unsigned int M,
  u0Functor &&u0,
  RECORDER recorder = [] (const Eigen::MatrixXd &) -> void {});
```

that numerically solves the Cauchy problem for the p-system (11.13.1) with $p(v) = -e^v$, final time $T > 0$, on the truncated spatial domain $[a, b]$ with

- a finite volume discretization in space on an equidistant mesh with $N \in \mathbb{N}$ cells and based on the HLLE numerical flux,
- explicit Euler timestepping, using $M \in \mathbb{N}$ uniform timesteps over the period $[0, T]$,
- initial data provided by a function $\mathbf{u}_0 : \mathbb{R} \to \mathbb{R}^2$ passed through `u0`.

The function is to return the approximate dual cell states at final time T in the columns of a matrix. In each timestep that matrix of states is to be given to the passed `recorder` object.

SOLUTION for (11-13.l) → 11-13-12-0:psys11asol.pdf ▲

(11-13.m) ⚄ (30 min.) [depends on Sub-problem (11-13.l)]

Extend the `main()` function in the file `fvpsystem_main.cc` in the following direction. Use the function `evlExpPSystem` from Sub-problem (11-13.k) to approximate the solution $v = v(x,t)$ of the Riemann problem for the p-system (11.13.1) with $p(v) = -e^v$ and initial data

$$v(x,0) = \begin{cases} 1 & \text{for } x < 0\,, \\ 3 & \text{for } x \geq 0 \end{cases}\,, \quad w(x,0) = \begin{cases} 1 & \text{for } x < 0\,, \\ 4 & \text{for } x \geq 0 \end{cases} \quad \forall x \in \mathbb{R}\,.$$

Simulate until final time $T = 2$ and truncate the spatial domain to $[-10, 10]$. $N = 1000$ spatial cells and choose a suitable timestep size.

Write the values of the piecewise constant approximation of $v(x,t)$ to file as a matrix with M rows and N columns (M the number of timesteps). This matrix can be read by the supplied PYTHON script `vispsys.p` , which will generate an *animated rendering* of the numerical solution.

SOLUTION for (11-13.m) → 11-13-13-0:psyss12.pdf ▲

References

[LeV02] R. J. LeVeque. *Finite volume methods for hyperbolic problems*. Cambridge Texts in Applied Mathematics. Cambridge: Cambridge University Press, 2002, pp. xx+558. DOI: 10.1017/CBO9780511791253 (cit. on p. 459).

End Problem 11-13 , 255 min.

Problem 11-14: Finite Volume Method for the Shallow Water Equations

[Lecture → Ex. 11.6.2.13] presented the shallow water equations as an example of a non-linear hyperbolic system of conservation laws in one spatial dimension and with a two-dimensional state space. In this project we tackle Cauchy problems for the shallow water equations by means for finite volume methods, relying on the concept of local Roe linearization [Lecture → § 11.6.3.28].

This problem specializes the ideas and concepts of [Lecture → Section 11.6.3.3]. It involves implementation of the numerical methods in C++.
▷ problem name/problem code folder: FVMShallowWater

[Lecture → Ex. 11.6.2.13] introduced the shallow water equations (SWE) as a simple model for the depth-uniform flow of an incompressible fluid in a straight channel. In physical variables $h = h(x,t)$ (fluid height) and $v = v(x,t)$ (fluid velocity) the non-dimensional[2] shallow water equations read

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}(vh) = 0 \quad \text{on} \quad \mathbb{R}\times]0,T[\,, \qquad \text{[Lecture → (11.6.2.14a)]}$$

$$\frac{\partial}{\partial t}(hv) + \frac{\partial}{\partial x}(hv^2 + \tfrac{1}{2}h^2) = 0 \quad \text{on} \quad \mathbb{R}\times]0,T[\,, \qquad \text{[Lecture → (11.6.2.14b)]}$$

whereas in conserved variables $\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} := \begin{bmatrix} h \\ hv \end{bmatrix}$ they become a non-linear two-dimensional system of conservation laws,

$$\frac{\partial \mathbf{u}}{\partial t}(x,t) + \frac{\partial}{\partial x} \underbrace{\begin{bmatrix} u_2 \\ u_2^2/u_1 + \frac{1}{2}u_1^2 \end{bmatrix}}_{=:\mathbf{F}(\mathbf{u})}(x,t) = 0 \quad \text{on} \quad \mathbb{R}\times]0,T[\,, \tag{11.14.1}$$

on the state space $U := \mathbb{R}^+ \times \mathbb{R}$. The function $\mathbf{F} : U \to \mathbb{R}^2$ is the flux function, *cf.* [Lecture → Def. 11.6.2.2].

(11-14.a) ⚂ (15 min.) As in [Lecture → § 11.6.2.5] we write $\lambda_1(\mathbf{u}) < \lambda_2(\mathbf{u})$ for the two eigenvalues of $D\mathbf{F}(\mathbf{u})$ and $\mathbf{r}_1(\mathbf{u})$, $\mathbf{r}_2(\mathbf{u})$ for the associated eigenvectors, all regarded as functions of $\mathbf{u}$.

[Lecture → Ass. 11.6.2.8] introduced the concepts of genuine non-linearity/linear degeneracy.

Assumption [Lecture → Ass. 11.6.2.8]. Genuine non-linearity/linear degeneracy

The eigenvectors $\mathbf{r}_i(\mathbf{u})$ and associated eigenvalues $\lambda_i(\mathbf{u})$ of $D\mathbf{F}(\mathbf{u})$ must satisfy

$$\mathbf{grad}_{\mathbf{u}}\,\lambda_i(\mathbf{u}) \cdot \mathbf{r}_i(\mathbf{u}) \neq 0 \quad \forall \mathbf{u} \in U \quad \textbf{or} \quad \mathbf{grad}_{\mathbf{u}}\,\lambda_i(\mathbf{u}) \cdot \mathbf{r}_i(\mathbf{u}) = 0 \quad \forall \mathbf{u} \in U\,.$$

Compute and simplify as much as possible the expressions $\mathbf{grad}_{\mathbf{u}}\,\lambda_i(\mathbf{u}) \cdot \mathbf{r}_i(\mathbf{u})$ for the concrete case of the shallow water equations (11.14.1).

$$\mathbf{grad}_{\mathbf{u}}\,\lambda_i(\mathbf{u}) \cdot \mathbf{r}_i(\mathbf{u}) = \begin{cases} \qquad\qquad & \text{for } i = 1\,, \\ \qquad\qquad & \text{for } i = 2\,. \end{cases}$$

SOLUTION for (11-14.a) → 11-14-1-0:swes1.pdf

▲

[2] Reference physical units are chosen so that earth's gravity acceleration becomes $\equiv 1$.

(11-14.b) ⚂ (45 min.) In the file `fvmshallowwater.cc` write a C++ function

```
bool isPhysicalTwoShockSolution(Eigen::Vector2d ul,
  Eigen::Vector2d us, Eigen::Vector2d ur);
```

that checks numerically, whether the three state vectors $\mathbf{u}_l \in \mathbb{R}^2$, $\mathbf{u}^* \in \mathbb{R}^2$, and $\mathbf{u}_r \in \mathbb{R}^2$ passed in `ul`, `us`, and `ur`, are the constant left, middle, and right state of a *physical* 2-shock solution of the **Riemann problem** for the shallow water equations,

$$\frac{\partial \mathbf{u}}{\partial t} + \frac{\partial}{\partial x}\begin{bmatrix} u_2 \\ u_2^2/u_1 + \frac{1}{2}u_1^2 \end{bmatrix} = 0 \quad \text{on} \quad \mathbb{R}\times]0,T[\,, \quad \mathbf{u}(x,0) = \begin{cases} \mathbf{u}_l & \text{for} \quad x < 0\,, \\ \mathbf{u}_r & \text{for} \quad x \geq 0\,. \end{cases} \tag{11.14.4}$$

By "physical" we mean that the *3-state/2-shock solution*

(i) is a weak solution [Lecture → Def. 11.6.2.11] of the Cauchy problem for the shallow water equations (11.14.1), and

(ii) and that for every shock the **Lax entropy condition** [Lecture → Def. 11.6.2.29] is satisfied.

HIDDEN HINT 1 for (11-14.b) → `11-14-2-0:sweh2.pdf`

SOLUTION for (11-14.b) → `11-14-2-1:swelec.pdf`

▲

(11-14.c) ⚂ (60 min.) In the file `fvmshallowwater.cc` implement a C++ function

```
Eigen::Vector2d numfluxLFSWE(Eigen::Vector2d v, Eigen::Vector2d w);
```

that computes the **Lax-Friedrichs numerical flux** for the shallow water equations as defined in [Lecture → (11.6.3.48)].

SOLUTION for (11-14.c) → `11-14-3-0:swes3.pdf`

▲

(11-14.d) ⚂ (30 min.) [depends on Sub-problem (11-14.c)]

In the file `fvmshallowwater.cc` complete the C++ function

```
Eigen::Vector2d numfluxHLLESWE(Eigen::Vector2d v, Eigen::Vector2d
  w);
```

that is supposed to compute the **HLLE numerical flux** for the shallow water equations according to [Lecture → (11.6.3.38)].

SOLUTION for (11-14.d) → `11-14-4-0:swes4.pdf`

▲

In the file `fvmshallowwater.h` you find a templated function

```
template <
    typename STATE, typename TIMESTEPPER, typename NUMFLUX,
    typename RECORDER = std::function<void(double, const
      std::vector<STATE> &)>>
void FVMEvlGeneric(
    double a, double b, double T, TIMESTEPPER &&ts_control,
    std::vector<STATE> &u, NUMFLUX &&NF,
    RECORDER &&rec);
```

that provides a "generic" implementation of a fully discrete finite volume method for a 1D system of conservation laws. The parameters are

- `a`, `b` ≙ bounds for the finite spatial interval,
- `T` ≙ the final time $T > 0$,
- `ts` a functor of type

  ```
  std::function<double(double, std::vector<STATE>)>
  ```

  that is meant for *timestep control*. In every timestep it receives the meshwidth and the current vector of states and is supposed to return a proposal for the size of the next timestep.
- `u` ≙ vector of initial states $\vec{\boldsymbol{\mu}}^{(0)}$,
- `NF` ≙ functor with signature

  ```
  std::function<STATE(STATE, STATE)>
  ```

  communicating the chosen *numerical flux function*,
- `rec` ≙ object that can be used for processing/storing the state vectors $\vec{\boldsymbol{\mu}}^{(j)}$.

The function implements a fully discrete finite volume method on an equidistant spatial mesh and with **explicit Euler** timestepping, see [Lecture → Ex. 11.4.1.9]. It employs constant continuation of the states beyond the endpoints of the finite spatial interval.

The complete code of the function `FVMEvlGeneric()` is given in Code 11.14.18

C++ code 11.14.18: Generic fully discrete FVM: `FVMEvlGeneric()`

```
template <
    typename STATE, typename TIMESTEPPER, typename NUMFLUX,
    typename RECORDER = std::function<void(double, const std::vector<STATE> &)>>
void FVMEvlGeneric(
    double a, double b, double T, TIMESTEPPER &&ts_control,
    std::vector<STATE> &u, NUMFLUX &&NF,
    RECORDER &&rec = [](double /*t*/,
                        const std::vector<STATE> & /*u*/) -> void {}) {
  size_t N = u.size();  // number of cells = number of (spatial) unknowns
  double h = (b - a) / N;
  // Main timestepping loop
  std::vector<STATE> u_tmp{u};
  rec(0.0, u);
  double t = 0.0;
  do {
    u_tmp.swap(u);
    double tau = std::min(ts_control(h, u_tmp), T - t);  // Local timestep size
    // Constant continuation on the left
    STATE F_minus;
    // Implements discrete evolution [Lecture → (11.4.1.10)]
    STATE F_plus = NF(u_tmp[0], u_tmp[0]);
    for (int j = 0; j < N - 1; ++j) {
      F_minus = F_plus;
      F_plus = NF(u_tmp[j], u_tmp[j + 1]);
      u[j] = u_tmp[j] - tau / h * (F_plus - F_minus);
    }
    F_minus = F_plus;
    // Constant continuation on the right
    F_plus = NF(u_tmp[N - 1], u_tmp[N - 1]);
    u[N - 1] = u_tmp[N - 1] - tau / h * (F_plus - F_minus);
```

```
32      rec(t, u);
33      t += tau;
34    } while (t < T);
35  }
```

(11-14.e) ⚄ (30 min.) The **Courant-Friedrichs-Levy (CFL) condition** also applies to non-linear system of conservation laws: the ratio of meshwidth h and timestep size τ must not exceed the maximum speed of propagation, $\tau \leq \dot{s}_{\max} h$. The maximum speed of propagation at timestep k is estimated according to

$$\dot{s}_{max} := \max_{j=1,\ldots,N} \max_{i=1,2} \left| \lambda_i \left(\left(\vec{\mu}^{(k)} \right)_j \right) \right| , \tag{11.14.19}$$

where $\vec{\mu}^{(k)}$ is the current sequence of approximate cell averages.

In the file `fvmshallowwater.h` you find the function

```
template <typename U0FUNCTOR, typename NUMFLUX>
std::vector<Eigen::Vector2d> solveSWE(
  double T, unsigned int N, NUMFLUX &&NF,
  U0FUNCTOR &&u0, std::ostream *out);
```

that uses `FVMEvlGeneric()` to compute an approximate solution of the shallow water equation (11.14.1) on $[-1,1]$ until final time $T > 0$ and with initial conditions passed through `u0`.

The implementation is complete except for the timestep control which you are supposed to supplement in the form of a suitable λ-function `ts_ctrl`.

SOLUTION for (11-14.e) → `11-14-5-0:swes7.pdf`

▲

End Problem 11-14 , 180 min.

Problem 11-15: Finite Volume Methods for Isentropic Euler Equations

The so-called isentropic appropximation provides a simplified continuum model for adiabatic and inviscid fluid flow under the assumption of constant entropy. In one spatial dimension the resulting isentropic Euler Equations constitute a non-linear hyperbolic system of conservation laws with a two-dimensional state space. In this project we examine these equations and develop finite volume methods (FVMs) for their numerical solution.

This problem is based on the contents of [Lecture → Section 11.6.2.1] and [Lecture → Section 11.6.3.3]. A C++ implementation of the FVM is included.
▷ problem name/problem code folder: FVMIsentropicEuler

We consider the Cauchy problem for the one-dimensional **isentropic Euler equations** of gas dynamics [Zhe01]

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \frac{\partial (\rho v)}{\partial x} &= 0 \\ \frac{\partial (\rho v)}{\partial t} + \frac{\partial}{\partial x}\Big(\rho v^2 + p(\rho)\Big) &= 0 \end{aligned} \quad \text{on} \quad \mathbb{R} \times \mathbb{R}^+ , \tag{11.15.1}$$
$$\rho(x,0) = \rho_0(x) \quad , \quad v(0,x) = v_0(x) , \quad x \in \mathbb{R} ,$$

where $\rho = \rho(x.t)$ is the fluid density, $v = v(x,t)$ is the fluid velocity, and $p : \mathbb{R}^+ \to \mathbb{R}$ is a given smooth "pressure" function that must satisfy

$$p'(\rho) > 0 \quad \forall \rho > 0 , \tag{11.15.2a}$$
$$\rho p''(\rho) + 2p'(\rho) > 0 \quad \forall \rho > 0 . \tag{11.15.2b}$$

(11-15.a) ⚀ (5 min.) For which choice of the pressure function p do the isentropic Euler equations agree with the shallow water equations as given in [Lecture → (11.6.2.14)]?

SOLUTION for (11-15.a) → 11-15-1-0:.pdf

▲

(11-15.b) ⚁ (15 min.) By introducing a suitable state vector $\mathbf{u} = \mathbf{u}(x,t)$ in terms of $\rho = \rho(x,t)$ and $v = v(x,t)$ cast (11.15.1) into the standard form

$$\frac{\partial \mathbf{u}}{\partial t}(x,t) + \frac{\partial \mathbf{F}(\mathbf{u})}{\partial x}(x,t) = 0 \quad \text{on} \quad D \subset \mathbb{R} \times \mathbb{R}^+ , \qquad [\text{Lecture} \to (11.6.2.3)]$$

of a non-linear system of conservation laws according to [Lecture → Def. 11.6.2.2].

$$\mathbf{u}(x,t) = \begin{bmatrix} \square \\ \square \end{bmatrix} , \quad (x,t) \in \mathbb{R} \times \mathbb{R}^+ ,$$

$$\mathbf{F}(\mathbf{u}) = \begin{bmatrix} \square \\ \square \end{bmatrix} , \quad u \in U := \square .$$

SOLUTION for (11-15.b) → 11-15-2-0:iseqs1.pdf

▲

(11-15.c) ⚁ (30 min.) [depends on Sub-problem (11-15.b)]

Show that the non-linear system of conservation laws [Lecture → (11.6.2.3)] arising from the isentropic Euler equations (11.15.1) is strictly hyperbolic in the sense of [Lecture → Ass. 11.6.2.6].

SOLUTION for (11-15.c) → 11-15-3-0:iseqs2.pdf

▲

(11-15.d) ⚄ (30 min.) A key tool for the construction of numerical fluxes for non-linear systems of conservation laws are Roe matrices, whose general definition for [Lecture → (11.6.2.3)] is as follows:

Definition [Lecture → Def. 11.6.3.30]. Roe matrix

A matrix-valued function $\mathbf{A}_R : U \times U \to \mathbb{R}^{m,m}$ is a **Roe matrix** for the flux function $\mathbf{F} : U \to \mathbb{R}^m$, if

(i) $\mathbf{A}_R(\mathbf{u},\mathbf{u}) = \mathrm{D}\mathbf{F}(\mathbf{u})$ for all $\mathbf{u} \in U$ (consistency),
(ii) $\mathbf{A}_R(\mathbf{v},\mathbf{w})(\mathbf{w}-\mathbf{v}) = \mathbf{F}(\mathbf{w}) - \mathbf{F}(\mathbf{v})$ for all $\mathbf{v},\mathbf{w} \in U$ (correct shock speed),
(iii) $\mathbf{A}_R(\mathbf{v},\mathbf{w})$ has m distinct real eigenvalues (strict hyperbolicity).

The following four matrices are proposed as Roe matrices for the isentropic Euler equations. They are all expressed in the conserved variables found in Sub-problem (11-15.c):

$$\mathbf{A}_R^1(\mathbf{v},\mathbf{w}) = \begin{bmatrix} 0 & 1 \\ -\left(\frac{\sqrt{w_1}v_2+\sqrt{v_1}w_2}{\sqrt{w_1}v_1+\sqrt{v_1}w_1}\right)^2 + p\left(\frac{v_1+w_1}{2}\right) & \frac{\sqrt{w_1}v_2+\sqrt{v_1}w_2}{\sqrt{w_1}v_1+\sqrt{v_1}w_1} \end{bmatrix}, \tag{11.15.6a}$$

$$\mathbf{A}_R^2(\mathbf{v},\mathbf{w}) = \begin{bmatrix} 0 & 1 \\ -\left(\frac{\sqrt{w_1}v_2+\sqrt{v_1}w_2}{\sqrt{w_1}v_1+\sqrt{v_1}w_1}\right)^2 + p'\left(\frac{v_1+w_1}{2}\right) & 2\frac{\sqrt{w_1}v_2+\sqrt{v_1}w_2}{\sqrt{w_1}v_1+\sqrt{v_1}w_1} \end{bmatrix}, \tag{11.15.6b}$$

$$\mathbf{A}_R^3(\mathbf{v},\mathbf{w}) = \begin{bmatrix} 0 & 1 \\ -\left(\frac{\sqrt{v_1}v_2+\sqrt{w_1}w_2}{\sqrt{v_1}v_1+\sqrt{w_1}w_1}\right)^2 + p'\left(\frac{v_1+w_1}{2}\right) & 2\frac{\sqrt{v_1}v_2+\sqrt{w_1}w_2}{\sqrt{v_1}v_1+\sqrt{w_1}w_1} \end{bmatrix}, \tag{11.15.6c}$$

$$\mathbf{A}_R^4(\mathbf{v},\mathbf{w}) = \begin{bmatrix} 0 & 1 \\ -\left(\frac{v_2+w_2}{v_1+w_1}\right)^2 + p'\left(\frac{v_1+w_1}{2}\right) & 2\frac{v_2+w_2}{v_1+w_1} \end{bmatrix}, \tag{11.15.6d}$$

where $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$, $\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$. You can assume that for $a,b \in \mathbb{R}$ we have that $p'(\frac{a+b}{2})(a-b) = p(a) - p(b)$. Fill in the following table

	Matrix $\mathbf{A}_R^*$ is a Roe matrix	
$\mathbf{A}_R^1$ from (11.15.6a)	◯ YES	◯ NO
$\mathbf{A}_R^2$ from (11.15.6b)	◯ YES	◯ NO
$\mathbf{A}_R^3$ from (11.15.6c)	◯ YES	◯ NO
$\mathbf{A}_R^4$ from (11.15.6d)	◯ YES	◯ NO

HIDDEN HINT 1 for (11-15.d) → 11-15-4-0:roeh1.pdf

SOLUTION for (11-15.d) → 11-15-4-1:iseqroe.pdf

▲

(11-15.e) ⚂ (60 min.) [depends on Sub-problem (11-15.d)]

In the file `fvmisentropiceuler.h` complete the C++ function

```
template <typename PFUNCTOR>
Eigen::Vector2d numfluxHLLEIseEul(PFUNCTOR &&p, PFUNCTOR &&pd,
                                  Eigen::Vector2d v,
                                  Eigen::Vector2d w);
```

that is supposed to evaluate the 2-point **HLLE numerical flux** $\widehat{\mathbf{F}}_{\mathrm{HLLE}} = \widehat{\mathbf{F}}_{\mathrm{HLLE}}(\mathbf{v},\mathbf{w})$ for the isentropic Euler equations according to [Lecture → (11.6.3.38)]. The parameters `v` and `w` pass the two states $\mathbf{v},\mathbf{w} \in U \subset\in \mathbb{R}^2$.

SOLUTION for (11-15.e) → `11-15-5-0:swes4.pdf`

▲

The goal is to implement the MUSCL higher-order finite volume scheme [Lecture → Section 11.5.3] with **minmod** slope reconstruction [Lecture → Def. 11.5.2.9] for the isentropic Euler equations. As a starting point we have the C++ function

```
template <typename FFUNCTOR, typename SLOPEFUNCTOR>
Eigen::VectorXd slopelimfluxdiff_scalar(const Eigen::VectorXd &mu,
                                        FFUNCTOR &&F,
                                        SLOPEFUNCTOR &&slopes);
```

from [Lecture → Code 11.5.1.5] that realizes $h\cdot$ (h the spatial meshwidth) the right hand side of the ordinary differential equation arising from the spatial conservative finite volume discretization of the Cauchy problem for a general 1D *scalar* conservation law,

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad &\text{in } \mathbb{R}\times]0,T[\,, \\ u(x,0) = u_0(x) \quad &\text{in } \mathbb{R}\,, \end{aligned} \qquad \text{[Lecture → (11.2.2.1)]}$$

on an equidistant spatial mesh.

The parameter `mu` passes the finite vector of initial cell averages, `F` must contain a functor of type `std::function<double(double,double)>` for a suitable **2-point numerical flux**, and `slopes` should provide a functor of type `std::function<double(double,double,double)>` giving $h\cdot$ the local slope of the piecewise linear reconstruction. Constant continuation of the states beyond the endpoints of the computational interval is employed.

C++ code 11.15.14: Code for `slopelimfluxdiff_scalar`, *cf.* [Lecture → Code 11.5.1.5]

```
2  template <typename FFUNCTOR, typename SLOPEFUNCTOR>
3  Eigen::VectorXd slopelimfluxdiff_scalar(const Eigen::VectorXd &mu, FFUNCTOR &&F,
4                                          SLOPEFUNCTOR &&slopes) {
5    size_t n = mu.size(); // Number of active dual grid cells
6    Eigen::VectorXd sigma = Eigen::VectorXd::Zero(n); // Vector of slopes
7    Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);    // Flux differences
8
9    // Computation of slopes σ_j, uses μ_0 = μ_1,
10   // μ_{N+1} = μ_N, which amounts to constant extension of states
11   // beyond domain of influence [a,b] of non-constant intial data.  Same
12   // technique has been applied in [Lecture → Code 11.3.2.9]
13   sigma[0] = slopes(mu[0], mu[0], mu[1]);
14   for (int j = 1; j < n - 1; ++j)
15     sigma[j] = slopes(mu[j - 1], mu[j], mu[j + 1]);
16   sigma[n - 1] = slopes(mu[n - 2], mu[n - 1], mu[n - 1]);
17
18   // Compute linear reconstruction at endpoints of dual cells [Lecture →
         (11.5.1.2)]
```

```
  Eigen::VectorXd nup = mu + 0.5 * sigma;
  Eigen::VectorXd num = mu - 0.5 * sigma;

  // As in [Lecture → Code 11.3.2.10]: constant continuation of data outside
  // [a,b]!
  fd[0] = F(nup[0], num[1]) - F(mu[0], num[0]);
  for (int j = 1; j < n - 1; ++j)
    // see [Lecture → (11.3.2.8)]
    fd[j] = F(nup[j], num[j + 1]) - F(nup[j - 1], num[j]);
  fd[n - 1] = F(nup[n - 1], mu[n - 1]) - F(nup[n - 2], num[n - 1]);
  return fd;
}
```

(11-15.f) ⚀ (30 min.) In the file `fvmisentropiceuler.h` implement the C++ function

```
template <typename FFUNCTOR, typename SLOPEFUNCTOR>
Eigen::MatrixXd slopelimfluxdiff_sys(
  const Eigen::MatrixXd &mu, FFUNCTOR &&F,
  SLOPEFUNCTOR &&slopes);
```

that has the same purpose as `slopelimfluxdiff_scalar`, but this time for non-linear *systems* of conservation laws [Lecture → (11.6.2.3)]. The sequence of cell states is now stored in the columns of an $m \times N$-matrix, m the dimension of state space, N the number of cells. The signatures of the functor type for the numerical flux and the slope reconstruction have to be adapted accordingly.

In the beginning copy the code of `slopelimfluxdiff_scalar()` into the function body of `slopelimfluxdiff_sys()` and then modify it.

SOLUTION for (11-15.f) → `11-15-6-0:iseqxs.pdf`

▲

(11-15.g) ⚄ (60 min.) [depends on Sub-problem (11-15.e)]

In the file `fvmisentropiceuler.h` finish the implementation of the templated C++ function

```
template <typename PFUNCTOR, typename RECORDER = std::function<
                                 void(double, const
                                   Eigen::MatrixXd &)> >
Eigen::MatrixXd musclIseEul(
  double a, double b, double T,
  const Eigen::MatrixXd &mu0_mat, PFUNCTOR &&p,
  PFUNCTOR &&dp, RECORDER &&rec);
```

that uses the **minmod-limited MUSCL scheme** (with 2nd-order explicit Heun timestepping [Lecture → (11.5.3.2)]) with **HLLE numerical flux** to solve Cauchy problems for the 1D isentropic Euler equations with pressure function p and its derivative p' passed through the functor objects `p` and `dp`.

- The spatial computational domain should be truncated to $[a,b]$, $a,b \in \mathbb{R}$ passed through the arguments `a` and `b`.
- Use an equidistant spatial mesh.
- Pass the initial state vectors as columns of the matrix `mu0_mat`.
- Employ constant continuation of the states beyond $[a,b]$, as is done in `slopelimfluxdiff_sys()`.

Moreover, the timestep size should be chosen *adaptively* in order to respect the **CFL-condition**. If $\left(\boldsymbol{\mu}_j^{(k)}\right)_{j=1}^N$ is the finite sequence of state vectors $\in \mathbb{R}^2$ obtained after the j-th timestep, $k \in \mathbb{N}_0$, then the timestep size for the next timestep should be chosen as

$$\tau := \frac{h}{2\dot{s}_{\max}}\,, \quad \dot{s}_{\max} := \max_{j=1,\ldots,N} \max\left\{ |\lambda_1(\boldsymbol{\mu}_j^{(k)})|, |\lambda_2(\boldsymbol{\mu}_j^{(k)})| \right\}\,. \tag{11.15.16}$$

Here $\lambda_1(\mathbf{u})$ and $\lambda_2(\mathbf{u})$ are the two real eigenvalues of $D\mathbf{F}(\mathbf{u})$, $\mathbf{u} \in U$.

SOLUTION for (11-15.g) $\rightarrow$ 11-15-7-0:.pdf

▲

References

[Zhe01] Yuxi Zheng. *Systems of conservation laws*. Vol. 38. Progress in Nonlinear Differential Equations and their Applications. Birkhäuser Boston, Inc., Boston, MA, 2001, pp. xvi+317. DOI: 10.1007/978-1-4612-0141-0 (cit. on p. 466).

End Problem 11-15 , 230 min.

Chapter 12

Finite Elements for the Stokes Equation

Problem 12-1: Viscous Flow Simulation with Taylor-Hood FEM

The Stokes equations either in variational saddle point form [Lecture → (12.2.2.19)] or in strong form as a boundary value problem [Lecture → (12.2.3.5)] provide a mathematical model for the stationary flow of an incompressible, viscous fluid in a container under the action of a stirring force. In this project we want to simulate the flow in a pipe with the fluid velocity prescribed at inlet and outlet boundary parts. This is modeled by the Stokes equations with non-zero Dirichlet boundary conditions for the velocity. We rely on the Taylor-Hood finite element method introduced in [Lecture → Section 12.3.4].

This project heavily relies on LEHRFEM++. Thus almost all the material covered in [Lecture → Section 2.7] will be relevant, in particular [Lecture → § 2.7.4.13] and [Lecture → § 2.7.4.11]. Of course, you must also know the contents of [Lecture → Section 12.3.4].
▷ problem name/problem code folder: StokesPipeFlow

Actually, the flow of a viscous fluid in a pipe requires a full three-dimensional description. However, in the interest of tractability we restrict ourselves to a dimensionally reduced 2D model. In order to obtain a bounded computational domain we have to cut out a short section of the pipe and prescribe the fluid velocity at inlet and outlet boundary parts

The geometric setting:

- computational domain $\Omega \subset \mathbb{R}^2$
- inlet boundary part Γ_{in}
- outlet boundary part Γ_{out}
- no-slip boundary part Γ_0

$\overline{\Gamma}_{\text{in}} \cap \overline{\Gamma}_{\text{out}} = \emptyset$

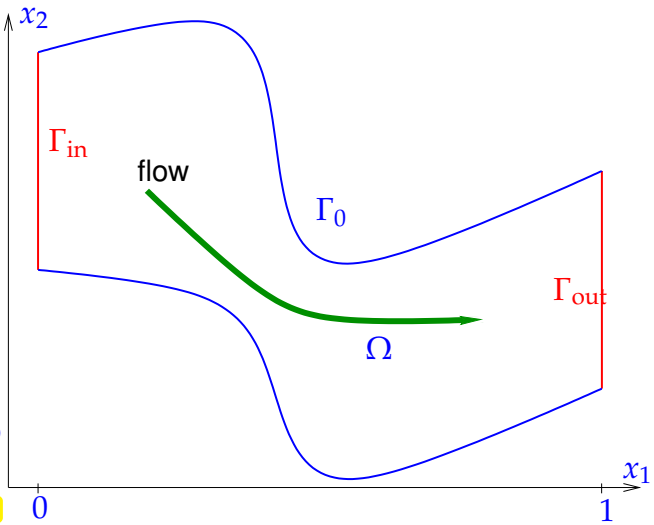


Fig. 219

The the unknown fluid velocity $\mathbf{v}$ and the pressure p can be obtained by solving the (non-dimensional) Stokes "pure Dirichlet" boundary value problem:

$$-\Delta \mathbf{v} - \mathbf{grad}\, p = \mathbf{0} \quad , \quad \operatorname{div} \mathbf{v} = 0 \quad \text{in} \quad \Omega \,, \tag{12.1.1a}$$

$$\int_\Omega p \,\mathrm{d}\boldsymbol{x} = 0 \quad , \quad \mathbf{v} = \mathbf{g} \quad \text{on} \quad \Gamma := \partial\Omega \,. \tag{12.1.1b}$$

The given boundary data $\mathbf{g} : \Gamma \to \mathbb{R}^2$ have to be admissible in the sense that $\mathbf{g}$ has to be continuous. Otherwise the Dirichlet fail to be compatible with the weak formulation of (12.1.1), remember [Lecture → § 1.9.0.6].

(12-1.a) ⚁ (20 min.) We now consider a straight pipe, $\Omega :=]0,1[\times]0,1[$, and, following the separation-of-variables idea, make the assumptions that

- the velocity is directed in x_1-direction and varies only in x_2-direction : $\mathbf{v}(\boldsymbol{x}) = v(x_2)\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, and
- the pressure is a function of x_1 alone: $p = p(x_1)$.

Compute the family of solutions of (12.1.1a) satisfying these assumptions.

SOLUTION for (12-1.a) → 12-1-1-0:spf:s1.pdf ▲

We think of truncating the pipe as chopping off infinitely long straight pieces, in which we have the flow pattern derived in Sub-problem (12-1.a). We also make the assumption that inlet and outlet are straight and parallel to the x_2-direction,

$$\Gamma_{\text{in}} := \{0\}\times]a_i, b_i[\quad , \quad \Gamma_{\text{out}} := \{1\}\times]a_o, b_o[\,, \quad a_*, b_* \in \mathbb{R} \,, \quad a_* < b_* \,. \tag{12.1.7}$$

On both inlet and outlet we impose transversal velocity with parabolic profile ($\boldsymbol{x} = \left[\begin{smallmatrix}x_1\\x_2\end{smallmatrix}\right]$),

$$\begin{aligned} \mathbf{g}(\boldsymbol{x}) &= (x_2 - a_i)(b_i - x_2)\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \quad \text{for} \quad \boldsymbol{x} \in \Gamma_{\text{in}} \,, \\ \mathbf{g}(\boldsymbol{x}) &= v_0(x_2 - a_o)(b_o - x_2)\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right] \quad \text{for} \quad \boldsymbol{x} \in \Gamma_{\text{out}} \,, \end{aligned} \tag{12.1.8}$$

with $v_0 \in \mathbb{R}$, and we set $\mathbf{g} \equiv \mathbf{0}$ on the remaining part Γ_0 of the boundary $\partial\Omega$. Note that this construction ensures $\mathbf{g} \in (C^0(\partial\Omega))^2$ and that the tangential component of the velocity field $\mathbf{v}$ on $\partial\Omega$ will vanish everywhere.

(12-1.b) ⚂ (20 min.) Determine the constant $v_0 \in \mathbb{R}$ in (12.1.8).

HIDDEN HINT 1 for (12-1.b) → 12-1-2-0:spfh2.pdf

SOLUTION for (12-1.b) → 12-1-2-1:spf:s2.pdf ▲

(12-1.c) ⚁ In [Lecture → § 12.1.0.7] we learned that the power dissipated by a viscous fluid flowing in $\Omega \subset \mathbb{R}^2$ with time-independent velocity $\mathbf{v} \in (H^1(\Omega))^2$ is

$$P_{\text{diss}} = \int_\Omega |\operatorname{curl} \mathbf{v}(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x} \,, \quad \operatorname{curl} \mathbf{v}(\boldsymbol{x}) := \frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1} \,, \quad \mathbf{v} = \left[\begin{smallmatrix}v_1\\v_2\end{smallmatrix}\right] \,, \quad \boldsymbol{x} = \left[\begin{smallmatrix}x_1\\x_2\end{smallmatrix}\right] \,.$$

[Lecture → (12.1.0.8)]

Show that, if $\mathbf{v} \in (H^1(\Omega))^2$ solves (12.1.1) and the Dirichlet data $\mathbf{g}$ are purely normal, $\mathbf{g}(\boldsymbol{x}) \parallel \boldsymbol{n}(\boldsymbol{x})$, $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ the exterior unit normal vector field, the dissipated power can be expressed as

$$P_{\text{diss}} = \int_{\partial\Omega} p(\boldsymbol{x})\, \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \,. \tag{12.1.12}$$

HIDDEN HINT 1 for (12-1.c) → 12-1-3-0:spfhsp3.pdf

SOLUTION for (12-1.c) → 12-1-3-1:.pdf ▲

(12-1.d) ⚄

The file `pipe.geo` shown beside contains a description of the pipe geometry in GMSH format.

Use GMSH to generate a triangular mesh of the pipe domain Ω and save the mesh to the file `pipe.mesh`.

Details can be found in [Lecture → § 2.7.1.4] and in the GMSH

Note the parameter `h`, which can be used to control the resolution of the mesh generated by GMSH.

```
// Gmsh description of pipe geometry
// Target mesh size
h = 0.01;
// Corners
Point(1) = { 0, 1, 0, h };
Point(2) = { 0, 0.5, 0, h };
Point(3) = { 0.33, 0.5, 0, h };
Point(4) = { 0.33, 0, 0, h };
Point(5) = { 1, 0, 0, h };
Point(6) = { 1, 0.5, 0, h };
Point(7) = { 0.66, 0.5, 0, h };
Point(8) = { 0.66, 1, 0, h };
// Edges
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 5};
Line(5) = {5, 6};
Line(6) = {6, 7};
Line(7) = {7, 8};
Line(8) = {8, 1};
// Domain
Curve Loop(1) = {1, 2, 3, 4, 5, 6, 7, 8};
Plane Surface(1) = {1};
```

SOLUTION for (12-1.d) → `12-1-4-0:spf:s4.pdf` ▲

Now we want to simulate the flow in the pipe based on the Taylor-Hood FEM [Lecture → Section 12.3.4] realized in LEHRFEM++. This method employs a finite element Galerkin discretization of the augemented Stokes variational saddle point problem; seek a velocity field $\mathbf{v} \in (H^1_0(\Omega))^2$ with $\mathbf{v}|_{\partial\Omega} = \mathbf{g}$, a pressure field $p \in L^2(\Omega)$, and a Lagrange multiplier $\lambda \in \mathbb{R}$

$$\begin{aligned} \int_\Omega \mathrm{D}\mathbf{v} : \mathrm{D}\mathbf{w}\,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div}\mathbf{w}\, p\,\mathrm{d}\boldsymbol{x} &= 0 \quad \forall \mathbf{w} \in (H^1_0(\Omega))^2\,, \\ \int_\Omega \operatorname{div}\mathbf{v}\, q\,\mathrm{d}\boldsymbol{x} \;+\; \lambda \int_\Omega q\,\mathrm{d}\boldsymbol{x} &= 0 \quad \forall q \in L^2(\Omega)\,, \\ \int_\Omega p\,\mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned} \tag{12.1.17}$$

Here and in the sequel, $\mathbf{g} \in (C^0(\partial\Omega))^2$ is used the denote the Dirichlet data for the velocity on the whole boundary.

Recall that the Taylor-Hood (P2-P1) finite element method employs the following trial and test spaces,

- for the velocity $\mathbf{v}$: $(\mathcal{S}^0_2(\mathcal{M}))^2 \subset (H^1(\Omega))^2$, and
- for the pressure p: $\mathcal{S}^0_1(\mathcal{M}) \subset L^2(\Omega)$.

In addition, vanishing trace on $\partial\Omega$ is imposed on the trial functions for the velocity, which are taken from $\mathcal{S}^0_{2,0}(\mathcal{M})$ to honor the principle "do not test where the solution is known" introduced in [Lecture → Section 1.8], [Lecture → § 1.8.0.1].

This leads to the discrete variational problem; seek $\mathbf{v}_h \in \mathcal{S}_2^0(\mathcal{M})$, $\mathbf{v}_h|_{\partial\Omega} = \mathsf{I}_2\mathbf{g}$, $p_h \in \mathcal{S}_1^0(\mathcal{M})$, $\lambda \in \mathbb{R}$,

$$\begin{aligned} \int_\Omega \mathrm{D}\mathbf{v}_h : \mathrm{D}\mathbf{w}_h \,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div}\mathbf{w}_h\, p_h \,\mathrm{d}\boldsymbol{x} &= 0 \quad \forall \mathbf{w}_h \in (\mathcal{S}_{2,0}^0(\mathcal{M}))^2 \,, \\ \int_\Omega \operatorname{div}\mathbf{v}_h\, q_h \,\mathrm{d}\boldsymbol{x} \;+\; \lambda \int_\Omega q_h \,\mathrm{d}\boldsymbol{x} &= 0 \quad \forall q \in \mathcal{S}_1^0(\mathcal{M}) \,, \\ \int_\Omega p_h \,\mathrm{d}\boldsymbol{x} &= 0 \quad . \end{aligned} \tag{12.1.18}$$

The operator I_2 is an interpolation operator onto $\mathcal{S}_2^0(\mathcal{M})$ based on point evaluations in the interpolation nodes (nodes of $\mathcal{M}$ and midpoints of edges of $\mathcal{M}$).

As global shape functions for $\mathcal{S}_2^0(\mathcal{M})$ and $\mathcal{S}_1^0(\mathcal{M})$ we use the standard *nodal basis functions* as described in [Lecture → Ex. 2.6.1.2] and [Lecture → Section 2.4.3], respectively.

> We opt for a **monolithic** representation of the full discrete/trial and test space $\mathcal{S}_2^0(\mathcal{M}) \times \mathcal{S}_1^0(\mathcal{M}) \times \mathbb{R}$ using a single **lf::assemble::DofHandler** object in the LehrFEM++ code.

Remark. An alternative code design relies separate objects of type **lf::uscalfe::FeSpaceLagrangeO2** and **lf::uscalfe::FeSpaceLagrangeO1**, respectively, for the velocity and pressure finite element spaces.

You may also follow this design when working on this project, but you will not be able to consult the master solution for guidance.

A strange "feature" = bug of LehrFEM++ has to be dealt with !

Remark 12.1.19 (Reordering of edge-associated indices of global shape functions by lf::assemble::UniformFEDofHandler) In the documentation of **lf::assemble::DofHandler** we read:

> However, the consistency of the local numbering of dofs has to be ensured. In two dimensions this is an issue for edges only and can be resolved by taking into account the orientation (direction) of the edge: dofs are ordered along the edge and those "closer to endpoint 0" are numbered first. In 3D many more situations have to be dealt with.

Let us look at the implementation in `dofhandler.cc`, where the arrays of indices to be returned by the `GlobalDofIndices()` member function are prepared in the array `dofs_`:

```
// Collect indices of interior shape functions of edges
// Internal ordering will depend on the orientation of the edge
auto edge_orientations = cell_p->RelativeOrientations();
auto edges = cell_p->SubEntities(1);
// Loop over edges
const size_type no_edges_cell = cell_p->RefEl().NumSubEntities(1);
for (int ed_sub_idx = 0; ed_sub_idx < no_edges_cell; ed_sub_idx++) {
  const glb_idx_t edge_idx = mesh_->Index(*edges[ed_sub_idx]);
  const glb_idx_t edge_int_dof_offset =
  edge_idx * num_dofs_[kEdgeOrd] + num_ext_dof_edge;
  // Copy indices of shape functions from edges to cell
  // The order, in which they are copied depends on the relative orientation
  // of the edge w.r.t.  the cell
  switch (edge_orientations[ed_sub_idx]) {
    case lf::mesh::Orientation::positive: {
      for (int j = 0; j < no_int_dof_edge; j++) {
        dofs_[kCellOrd][cell_dof_offset++] =
        dofs_[kEdgeOrd][edge_int_dof_offset + j];
```

```
19        }
20        break;
21      }
22      case lf::mesh::Orientation::negative: {
23        for (int j = static_cast<int>(no_int_dof_edge - 1); j >= 0; j--) {
24          dofs_[kCellOrd][cell_dof_offset++] =
25          dofs_[kEdgeOrd][edge_int_dof_offset + j];
26        }
27        break;
28      }
29    } // end switch
30  }
```

As a consequence, when there are two global shape functions associated with an interior edge, as it is done in the monolithic approach to the numbering, the two adjacent triangles may see receive *different numbers* for the same corresponding local shape functions. For instance, this is what the `GlobalDofIndices()` member function returns for a simple mesh of the unit square with four triangles and one interior point (see test "PrintDof" in the file `stokespipeflow_test.cc`):

```
TRIA 0:  15 dofs = [0 1 2 3 4 5 12 13 14 15 16 23 24 19 20 ]
TRIA 1:  15 dofs = [3 4 5 6 7 8 12 13 14 21 22 27 28 24 23 ]
TRIA 2:  15 dofs = [6 7 8 9 10 11 12 13 14 25 26 29 30 28 27 ]
TRIA 3:  15 dofs = [9 10 11 0 1 2 12 13 14 17 18 20 19 30 29 ]
```

The numbers of the local shape functions for the velocity components have been swapped! This will happen, when the orientation of an edge does not match the numbering of the vertices of a triangle, that is, if its relative orientation is negative.

As a clumsy fix, the current reference implementation checks the relative orientation of edges and, if a negative relative orientation is detected, swaps the local numbers again. ⌟

(12-1.e) ⚀ (20 min.) Write down the list of local shape functions (LSF) $\left(b_K^1,\ldots,b_{+K}^Q\right)$, $Q\in\mathbb{N}$,of the compound finite element space

$$V_{0,h} := \mathcal{S}_2^0(\mathcal{M})\times\mathcal{S}_2^0(\mathcal{M})\times\mathcal{S}_1^0(\mathcal{M})\,,\quad \mathcal{M}\text{ a triangular mesh,} \tag{12.1.20}$$

on a triangle K in terms of its barycentric coordinate functions $\lambda_1,\lambda_2,\lambda_3$.

The ordering of the local shape functions should follow LEHRFEM++'s numbering convention as defined in [Lecture → § 2.7.4.11]. Global shape functions associated with the *same mesh entity* are to ordered in the following fashion:

1. velocity x_1-component - 2. velocity x_2-component - [3. pressure value].

SOLUTION for (12-1.e) → `12-1-5-0:.pdf` ▲

(12-1.f) ⚁ (10 min.) What is $\dim V_{0,h}$ expressed in terms of the number $\sharp\mathcal{M}$ of cells of $\mathcal{M}$, the number $\sharp\mathcal{E}(\mathcal{M})$ of edges of $\mathcal{M}$, and the number $\sharp\mathcal{V}(\mathcal{M})$ of nodes of the mesh?

SOLUTION for (12-1.f) → `12-1-6-0:spf:s5.pdf` ▲

(12-1.g) ⚂ Using the notation b_K^i, $i=1,\ldots,Q$, $K\in\mathcal{M}$ a triangle, for the local shape functions found in Sub-problem (12-1.e), give expressions for the entries of the element matrix related to the discrete variational problem (12.1.18) without taking into account the last equation and the Lagrange multiplier.

SOLUTION for (12-1.g) → `12-1-7-0:spf:s7.pdf`The structure of this variational problem is re-

flected in the block structure of the element matrix $\mathbf{A}_K \in \mathbb{R}^{15,15}$:

$$\mathbf{A}_K = \begin{bmatrix} \mathbf{A}_1 & \mathbf{O}_{6,6} & \mathbf{B}_1^\top \\ \mathbf{O}_{6,6} & \mathbf{A}_2 & \mathbf{B}_2^\top \\ \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{O}_{3,3} \end{bmatrix}, \quad \begin{aligned} \mathbf{A}_1 &:= \left[\int_K \mathbf{grad}\, b_K^j \cdot \mathbf{grad}\, b_K^i \,\mathrm{d}\boldsymbol{x} \right]_{i,j\in\{1,4,7,10,12,14\}}, \\ \mathbf{A}_2 &:= \left[\int_K \mathbf{grad}\, b_K^j \cdot \mathbf{grad}\, b_K^i \,\mathrm{d}\boldsymbol{x} \right]_{i,j\in\{2,5,8,11,13,15\}}, \\ \mathbf{B}_1 &:= \left[\int_K \frac{\partial b_K^j}{\partial x_1} b_K^i \,\mathrm{d}\boldsymbol{x} \right]_{\substack{i\in\{3,6,9\} \\ j\in\{1,4,7,10,12,14\}}}, \\ \mathbf{B}_2 &:= \left[\int_K \frac{\partial b_K^j}{\partial x_1} b_K^i \,\mathrm{d}\boldsymbol{x} \right]_{\substack{i\in\{3,6,9\} \\ j\in\{2,5,8,11,13,15\}}}. \end{aligned} \tag{12.1.23}$$

Note that the ordering of the local shape functions that gives rise to the block structure of the matrix $\mathbf{A}_K$ as displayed in (12.1.23) does *not match* the ordering used by LEHRFEM++ !

▲

(12-1.h) ⚄ (90 min.) [depends on Sub-problem (12-1.g)]

The following class fits the concept of **ENTITY_MATRIX_PROVIDER** and is supposed to compute very element matrices for the Taylor-Hood FEM as derived in Sub-problem (12-1.g).

```
class TaylorHoodElementMatrixProvider {
  public:
  using ElemMat = Eigen::Matrix<double, 15, 15>;
  TaylorHoodElementMatrixProvider(const
     TaylorHoodElementMatrixProvider &) =
  delete;
  TaylorHoodElementMatrixProvider(TaylorHoodElementMatrixProvider
     &&) noexcept =
  default;
  TaylorHoodElementMatrixProvider &operator=(
  const TaylorHoodElementMatrixProvider &) = delete;
  TaylorHoodElementMatrixProvider &operator=(
  TaylorHoodElementMatrixProvider &&) = delete;
  TaylorHoodElementMatrixProvider() = default;
  virtual ~TaylorHoodElementMatrixProvider() = default;
  [[nodiscard]] bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  [[nodiscard]] ElemMat Eval(const lf::mesh::Entity &cell);
  private:
  ElemMat AK_;
};
```

Implement the `Eval()` member function in the file `stokespipeflow.cc`.

SOLUTION for (12-1.h) → `12-1-8-0:.pdf` ▲

(12-1.i) ⚄ (15 min.) The bottom equation in (12.1.17) and the Lagrange multiplier give rise to an extra row and column of the Galerkin matrix. Give a formula for their entries using the notations $\{b_h^i\}_{i=1}^N$ for set of global shape functions of $(\mathcal{S}_2^0(\mathcal{M}))^2$ and $\{s_h^i\}_{i=1}^M$ for the set of tent functions ($\hat{=}$ global shape functions of $\mathcal{S}_1^0(\mathcal{M})$) on $\mathcal{M}$.

SOLUTION for (12-1.i) → 12-1-9-0:spf:s9.pdf ▲

(12-1.j) ⚂ (15 min.) As pointed out above, we intend to use a single **lf::assemble::DofHandler** object represent the local→global index mapping for all the global shape functions spanning $V_{0,h}$. Together with the helper class **TaylorHoodElementMatrixProvider** such a **lf::assemble::DofHandler** object can then be used to build the full Galerkin matrix via `assembleMatrixLocally()`.

Complete the following code meant for the initialization of that **lf::assemble::DofHandler** object, see [Lecture → § 2.7.4.16].

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(),  },
  {lf::base::RefEl::kSegment(),  },
  {lf::base::RefEl::kTria(),  },
  {lf::base::RefEl::kQuad(),  }});
```

Here `mesh_p` is a pointer to the underlying triangular mesh.

SOLUTION for (12-1.j) → 12-1-10-0:spf:9as.pdf ▲

(12-1.k) ⚄ (30 min.) In the file `stokespipeflow.cc` complete the code of the C++ function

```
lf::assemble::COOMatrix<double>
buildTaylorHoodGalerkinMatrix(
const lf::assemble::DofHandler &dofh);
```

which is supposed to return the *full* Galerking matrix for the Taylor-Hood FEM for the augmented Stokes variational saddle point problem (12.1.18). Here "full" means that boundary conditions are not taken into account and that the finite element space $(\mathcal{S}_2^0(\mathcal{M}))^2$ should be used instead of $(\mathcal{S}_{2,0}^0(\mathcal{M}))^2$.

The argument `dofh` supplies the **lf::assemble::DofHandler** object providing the local→global index mapping for all involved finite element spaces combined, refer to Sub-problem (12-1.j).

Of course, you should rely on the **ENTITY_MATRIX_PROVIDER** class **TaylorHoodElementMatrixProvider** from Sub-problem (12-1.h) and LEHRFEM++'s assembly facility `lf::assemble::AssembleMatrixLocally()`, see [Lecture → Ex. 2.7.4.22].

This time the zero mean constraint and the scalar Lagrange multiplier have to be included, recall Sub-problem (12-1.i).

SOLUTION for (12-1.k) → 12-1-11-0:spf:SA.pdf ▲

(12-1.l) ⚄ (60 min.) In the file `stokespipeflow.h` please finish the implementation of the templated C++ function

```
template <typename gFunctor>
Eigen::VectorXd solvePipeFlow(
const lf::assemble::DofHandler &dofh, gFunctor &&g);
```

This function takes as argument `g` a functor of type **std**`::function<Eigen::Vector2d(Eigen::Vecto` which supplies the boundary values $\mathbf{g} = \mathbf{g}(\boldsymbol{x})$ for the velocity. The function should return the basis expansion coefficient vector for the Taylor-Hood finite element solution of (12.1.1) (with Dirichlet boundary conditions $\mathbf{v} = \mathbf{g}$ everywhere on $\partial\Omega$).

Note that that vector comprises the "degrees of freedom" for both velocity and pressure matching the

monolithic treatment of the Taylor-Hood finite element space in the **lf::assemble::DofHandler** object passed through `dofh`.

HIDDEN HINT 1 for (12-1.l) → `12-1-12-0:spfhB.pdf`

HIDDEN HINT 2 for (12-1.l) → `12-1-12-1:spfHB2.pdf`

SOLUTION for (12-1.l) → `12-1-12-2:spf:Bs.pdf` ▲

(12-1.m) ⚀ (30 min.) In [ASS21, Sect. 3] the authors claim that for every vector $\boldsymbol{d} \in \mathbb{R}^2$ the functions

$$\mathbf{v}_{\boldsymbol{d}}(\boldsymbol{x}) := \frac{1}{4\pi}\left(-\log\|\boldsymbol{x}\|\boldsymbol{d} + \frac{\boldsymbol{x}(\boldsymbol{x}\cdot\boldsymbol{d})}{\|\boldsymbol{x}\|^2}\right) \quad , \quad p_{\boldsymbol{d}}(\boldsymbol{x}) := -\frac{1}{2\pi}\frac{\boldsymbol{x}\cdot\boldsymbol{d}}{\|\boldsymbol{x}\|^2} \, , \quad \boldsymbol{x} \neq \mathbf{0} \, , \tag{12.1.33}$$

solve the homogeneous Stokes equations in $\mathbb{R}^2 \setminus \{\mathbf{0}\}$,

$$-\Delta \mathbf{v}_{\boldsymbol{d}} - \mathbf{grad}\, p_{\boldsymbol{d}} = 0 \quad \text{in} \quad \mathbb{R}^2 \setminus \{\mathbf{0}\} \, . \tag{12.1.34}$$

Verify this claim.

Remark. The two functions are representatives of so-called **fundamental solutions** for the Stokes operator.

SOLUTION for (12-1.m) → `12-1-13-0:spfCs.pdf` ▲

(12-1.n) ⚀ (45 min.) In the file `stokespipeflow.cc` finish the impleentation of the LEHRFEM++-based C++ function

```
void testCvgTaylorHood();
```

This function is supposed to do the following:

1. Generate a coarse triangular test mesh of the unit square $]0,1[^2$ using the function call

   `lf::mesh::test_utils::GenerateHybrid2DTestMesh(3,1.0/3.0).`

2. Starting from this mesh call

   `lf::refinment::GenerateMeshHierarchyByUniformRefienment()`

   to set up a **lf::refinement::MeshHierarchy** object representing hierarchy of meshes created by regular refinement, see `ellbvp_linfe_demo.cc` .

3. Loop over all meshes of the sequence and for each

   - initialize a **lf::assemble::DofHandler** object as in Sub-problem (12-1.j) and
   - invoke `solvePipeFlow()` from Sub-problem (12-1.l) to compute the total basis expansion coefficient vector for the Taylor-Hood solution of the Stokes BVP with Dirichlet data matching the following *exact solution* of (12.1.1):

   $$\mathbf{v}(\boldsymbol{x}) = \mathbf{v}_{\boldsymbol{d}}(\boldsymbol{x} + \left[\begin{smallmatrix}0.5\\0.5\end{smallmatrix}\right]) \, , \quad p(\boldsymbol{x}) = p_{\boldsymbol{d}}(\boldsymbol{x} + \left[\begin{smallmatrix}0.5\\0.5\end{smallmatrix}\right]) \quad \text{as in (12.1.33) with} \quad \boldsymbol{d} = \begin{bmatrix}2\\1\end{bmatrix} .$$

4. Writing $\mathbf{v}_\ell \in \mathcal{S}_2^0(\mathcal{M}_\ell)$, $p_\ell \in \mathcal{S}_1^0(\mathcal{M}_\ell)$ for the Taylor-Hood finite element solution on level ℓ, $\ell = 0, \ldots, L$, the function computes the error norms

   $$|\mathbf{v} - \mathbf{v}_\ell|_{H^1(\Omega)} \quad , \quad \|\mathbf{v} - \mathbf{v}_\ell\|_{L^2(\Omega)} \quad \text{and} \quad \|p - p_\ell\|_{L^2(\Omega)} \, ,$$

5. and it prints a table of these error norms for $\ell = 0, \ldots, 7$.

Most of the code has already been written, Items 4. being the only exception.

HIDDEN HINT 1 for (12-1.n) → 12-1-14-0:spferrnorms.pdf

SOLUTION for (12-1.n) → 12-1-14-1:spf:DS.pdf ▲

Now we return to the specific pipe flow setting described in the beginning, that is the Stokes problem (12.1.1), the geometric setting drawn in Fig. 219, and the Dirichlet boundary values given according to (12.1.7), (12.1.8).

(12-1.o) ⚀ (20 min.) In the file `stokespipeflow.cc` complete the code for the LEHRFEM++-based C++ function

```
void visualizeTHPipeFlow(const char *meshfile = "pipe.msh", const
   char *outfile = "pipeflow");
```

which reads a mesh from `pipe.msh`, invokes your implementation of `solvePipeFlow()` from Subproblem (12-1.l) and then writes two `.vtk` files that can be read by PARAVIEW to plot the computed velocity field and pressure.

The bulk of this function has already been implemented. All you have to do is to insert a suitable lambda function for the boundary values $\mathbf{g}$ according to (12.1.7), (12.1.8).

Use PARAVIEW to create a streamline plot or a vector field plot of the velocity field.

HIDDEN HINT 1 for (12-1.o) → 12-1-15-0:vtkwriter.pdf

SOLUTION for (12-1.o) → 12-1-15-1:cppE.pdf ▲

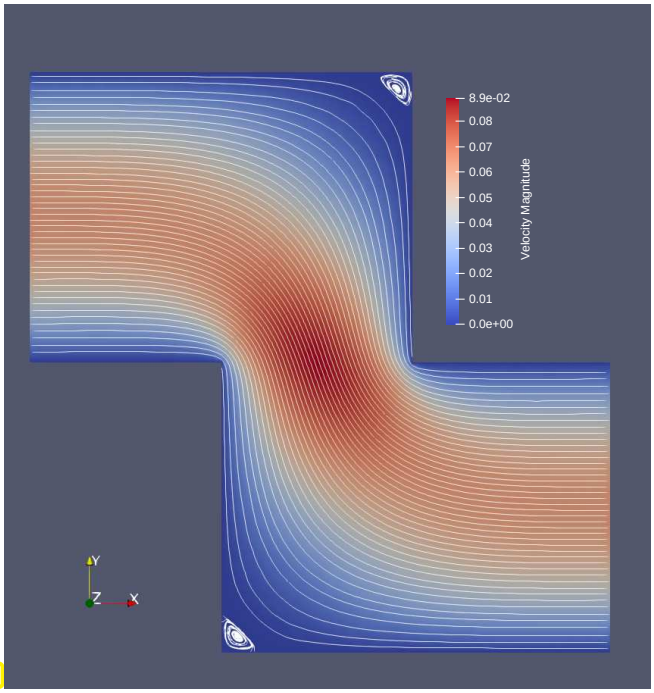


Fig. 223

◁ Streamlines of flow velocity field computed by means of Taylor-Hood FEM.

Note
- the high density of the streamlines at the re-entrant corners; the velocity field has a has a singularity of "corner-simngular-function-type" there [Lecture → Ex. 3.4.0.3];
- the small eddies in the corners; in fact there is an infinite cascade of ever smaller and weaker corner eddies [LD14, Sect. 12.3].

Instructions for visualization with PARAVIEW:

1. Import the `.vtk`-file to PARAVIEW. This file contains the first (u1) and second (u2) component of the velocity. It also contains the pressure (p). In the top menu, you can select which scalar array to visualize.
2. Generate a **Calculator** filter with result `u1*iHat+u2*jHat+0*kHat`. This is necessary, because PARAVIEW only allows contour plots for 3D vector fields.

Special instructions for plotting stream lines:

3. Use the filter **StreamTracer** and adjust the line such that it covers the whole cross-section of the the pipe.
4. Under 'properties' of the **StreamTracer** filter, find the section 'Coloring' and select 'Solid Color'.
5. of the **StreamTracer** filter, find the section 'Line Parameters' and try different values of 'Resolution'.
6. Under 'properties' of the **StreamTracer** filter, find the section 'Streamline Parameters' and play with 'Maximum Streamline Length'.

Vector plot of the velocity vector field $\mathbf{v}_h$ as computed by Taylor-Hood FEM. ▷

To visualize the velocity as vetor field in PARAVIEW after steps 1. and 2. above

3. Use the filter **Glyph**.
4. Under 'properties' of the **Glyph** filter, find the section 'Scale' and change 'Scale Array' to 'Result' and change 'Scale Factor' to your liking.
5. Under 'properties' of the **StreamTracer** filter, find the section 'Coloring' and select 'Solid Color'.

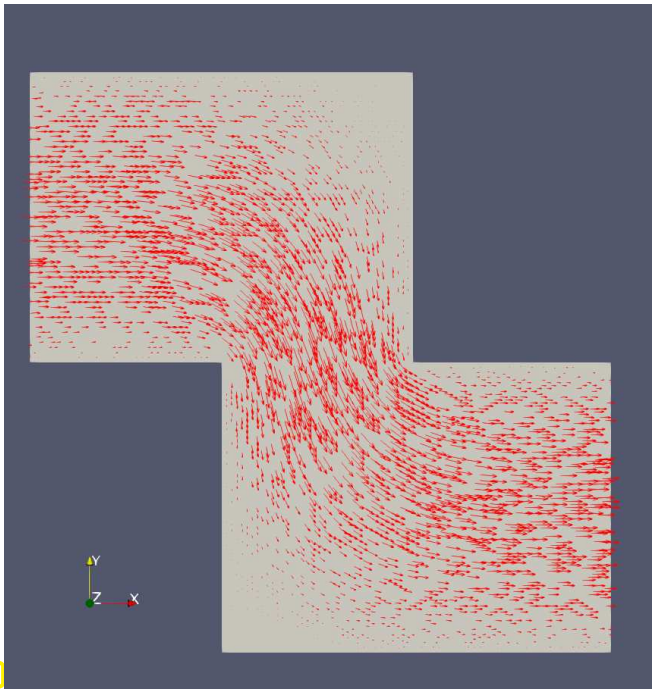


Fig. 224

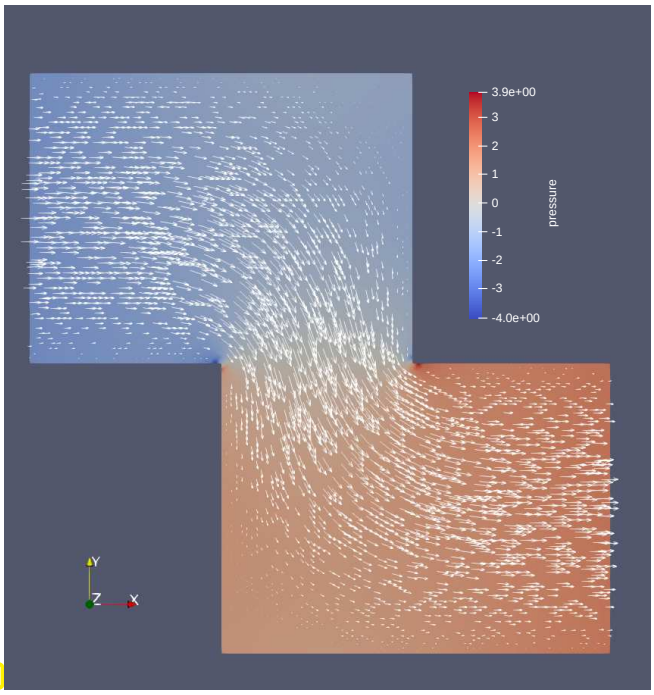


Fig. 225

◁ Color plot visualization of the computed pressure field p_h.

(12-1.p) ⚄ (30 min.) [depends on Sub-problem (12-1.h)]

Again in `stokespipeflow.cc` realize a C++ function

```
double computeDissipatedPower(const char *meshfile = "pipe.msh");
```

that computes the Taylor-Hood finite element solution for the pipe flow setting of this project based on the mesh read from the file whose name is passed through `meshfile`, and then computes the dissipated power P_{diss} as defined in [Lecture → (12.1.0.8)], see Sub-problem (12-1.c). That value P_{diss} should be returned.

SOLUTION for (12-1.p) → `12-1-16-0:cppF.pdf` ▲

(12-1.q) ⚄ (30 min.) In the file `stokespipeflow.cc` write a C++ function

```
double computeDissipatedPowerBd(const char *meshfile = "pipe.msh");
```

with exactly the same parameters and purpose as `computeDissipatedPower()` from Sub-problem (12-1.p).

However, this time you should use the alternative formula (12.1.12) **!**

This means, as an approximation of the dissipated power this function should return

$$P_{\mathrm{diss},h} := \int_{\partial\Omega} p_h(\boldsymbol{x})\mathbf{v}_h(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \,.$$

Of course, you may exploit the specific setting for this project, in particular (12.1.7) and (12.1.8) to simplify the implementation.

SOLUTION for (12-1.q) → `12-1-17-0:cppG.pdf` ▲

FEM-computed dissipated power(s) on `mesh*.mesh` , $* \in \{1,2,3,4,5,6\}$, shape-regular, quasi-uniform sequence $\mathcal{M}_\ell$, $\ell = 1,\ldots,6$, of triangular meshes generated by GMSH, $H_\ell \sim 0.1 \cdot 2^{-\ell}$:

level	P_{vol}	P_{bd}
1	0.0734	0.0785
2	0.0785	0.0807
3	0.0808	0.0820
4	0.0821	0.0827
5	0.0827	0.0830
6	0.0830	0.0831

References

[ASS21] Carlos J. S. Alves, Rodrigo G. Serrão, and Ana L. Silvestre. "Fundamental solutions for the Stokes equations: numerical applications for 2D and 3D flows". In: *Appl. Numer. Math.* 170 (2021), pp. 55–73. DOI: 10.1016/j.apnum.2021.07.011 (cit. on p. 478).

[LD14] William E. Langlois and Michel O. Deville. *Slow viscous flow*. Second. Springer, Cham, 2014, pp. xvi+324. DOI: 10.1007/978-3-319-03835-3 (cit. on p. 479).

End Problem 12-1 , 435 min.

Problem 12-2: The MINI Element for Stokes

In [ABF84] the authors proposed a simple *stable* conforming finite element pair [Lecture → Def. 12.3.2.7] for the 2D Stokes variational saddle point problem. For the pressure they use piecewise linear Lagrangian finite element functions, whereas the velocity components are approximated by means of **bubble augmented** piecewise linear Lagrangian finite element functions.

This project relies on LEHRFEM++ and knowledge about the Galerkin discretization of Stokes BVPs as imparted in [Lecture → Section 12.3].
▷ problem name/problem code folder: StokesMINIElement

As we have learned in [Lecture → Section 12.3.2], **stable pairs** of finite element spaces [Lecture → Def. 12.3.2.7] are essential for a viable Galerkin finite element discretization of the saddle point variational formulations of boundary value problems for the Stokes equations.

A very simple h-uniformly stable and conforming pair of finite element spaces on 2D triangular meshes is provided by the so-called **MINI element FEM** [ABF84].

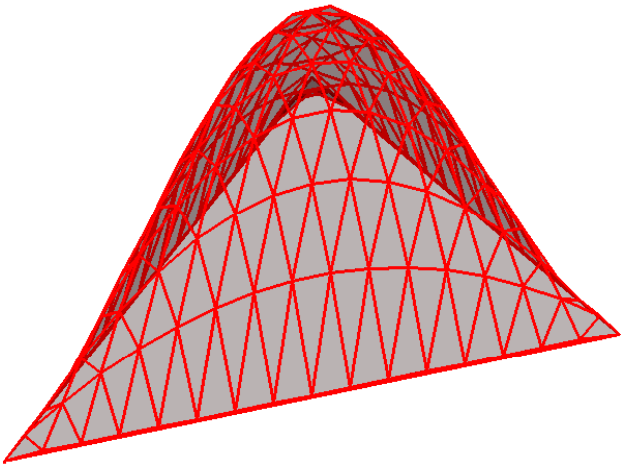


Fig. 226

Its key ingredients are **cubic bubble functions**. For a non-degenerate triangle $K \subset \mathbb{R}^2$ the associated cubic bubble function $\beta_K : K \to \mathbb{R}$ is defined as

$$\beta_K := \lambda_1 \lambda_2 \lambda_3 \,, \tag{12.2.1}$$

where λ_1, λ_2, and λ_3 are the three barycentric coordinate functions for K [Lecture → § 2.4.5.2].

◁ Mesh plot of the graph of the cubic bubble function for a triangle.

Let $\Omega \subset \mathbb{R}^2$ be a polygonally bounded computational domain endowed with a triangular mesh $\mathcal{M}$. The **bubble enriched** lowest-order Lagrangian finite element space is defined as

$$\mathcal{BS}_1^0(\mathcal{M}) := \mathcal{S}_1^0(\mathcal{M}) + \mathrm{Span}\{\beta_K : K \in \mathcal{M}\} \,, \tag{12.2.2}$$

and its subspace of functions vanishing on $\partial\Omega$ is

$$\mathcal{BS}_{1,0}^0(\mathcal{M}) := \mathcal{BS}_1^0(\mathcal{M}) \cap H_0^1(\Omega) = \mathcal{S}_{1,0}^0(\mathcal{M}) + \mathrm{Span}\{\beta_K : K \in \mathcal{M}\} \tag{12.2.3}$$

The MINI element finite element method for the Stokes BVP with non-slip boundary conditions

$$-\Delta \mathbf{v} - \mathbf{grad}\, p = \mathbf{0} \quad , \quad \mathrm{div}\, \mathbf{v} = 0 \quad \text{in} \quad \Omega \quad , \quad \mathbf{v} = \mathbf{g} \quad \text{on} \quad \partial\Omega \,, \tag{12.2.4}$$

$\mathbf{g} \in (C^0(\partial\Omega))^2$ given, then amounts to the following linear variational problem.

Seek $\mathbf{v}_h \in \left(\mathcal{BS}_1^0(\mathcal{M})\right)^2$, $\mathbf{v}_h|_{\partial\Omega} = \mathbf{g}_h$, $p_h \in \mathcal{S}_1^0(\mathcal{M})$, and $\lambda \in \mathbb{R}$ such that

$$\begin{aligned} \int_\Omega \mathrm{D}\mathbf{v}_h : \mathrm{D}\mathbf{w}_h \,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div} \mathbf{w}_h \, p_h \,\mathrm{d}\boldsymbol{x} &= 0\,, \\ \int_\Omega \operatorname{div} \mathbf{v}_h \, q \,\mathrm{d}\boldsymbol{x} \;+\; \lambda \int_\Omega q_h \,\mathrm{d}\boldsymbol{x} &= 0\,, \\ \int_\Omega p_h \,\mathrm{d}\boldsymbol{x} &= 0 \end{aligned} \tag{12.2.5}$$

for all $\mathbf{w}_h \in \left(\mathcal{BS}_{1,0}^0(\mathcal{M})\right)^2$, $q_h \in \mathcal{S}_1^0(\mathcal{M})$, where $\mathbf{g}_h : \partial\Omega \to \mathbb{R}$ is the piecewise linear interpolant of $\mathbf{g}$ on $\mathcal{M}|_{\partial\Omega}$, *cf.* [Lecture → Ex. 2.7.6.5].

Throughout, as global shape functions for $\mathcal{S}_1^0(\mathcal{M})$ we use the standard nodal basis of tent functions as in [Lecture → Section 2.4.3], and for $\mathcal{BS}_1^0(\mathcal{M})$ the global shape functions are all tent functions plus the bubble functions for every cell of the mesh.

(12-2.a) ⚂ (10 min.) We write

- $\mathcal{V}_i(\mathcal{M})$ for the set of interior nodes of $\mathcal{M}$, $\mathcal{V}_\partial(\mathcal{M})$ for the set of mesh nodes on $\partial\Omega$, $\mathcal{V}(\mathcal{M}) := \mathcal{V}_i(\mathcal{M}) \cup \mathcal{V}_\partial(\mathcal{M})$,
- $\mathcal{E}_i(\mathcal{M})$ for the set of interior edges of $\mathcal{M}$, $\mathcal{E}_\partial(\mathcal{M})$ for the set of mesh edges $\subset \partial\Omega$, $\mathcal{E}(\mathcal{M}) := \mathcal{E}_i(\mathcal{M}) \cup \mathcal{E}_\partial(\mathcal{M})$,
- and $\mathcal{M}$ also stands for the set of cells of $\mathcal{M}$.

Express the dimensions of the finite element spaces in terms of the cardinalities of some of these sets,

$$\dim \mathcal{BS}_1^0(\mathcal{M}) = \boxed{} \quad , \quad \dim \mathcal{BS}_{1,0}^0(\mathcal{M}) = \boxed{} \,.$$

Use the ♯-operator to extract the number of elements of a finite set.

SOLUTION for (12-2.a) → 12-2-1-0:minis1.pdf

▲

(12-2.b) ⚄ (20 min.) [depends on Sub-problem (12-2.a)]

The linear system of equations arising from the MINI finite element Galerkin discretization of (12.2.5) has the block structure

$$\begin{bmatrix} \mathbf{A}_x & \mathbf{O} & \mathbf{B}_x^\top & \mathbf{0} \\ \mathbf{O} & \mathbf{A}_y & \mathbf{B}_y^\top & \mathbf{0} \\ \mathbf{B}_x & \mathbf{B}_y & \mathbf{O} & \mathbf{c} \\ \mathbf{0}^\top & \mathbf{0}^\top & \mathbf{c}^\top & 0 \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\mu}}_x \\ \vec{\boldsymbol{\mu}}_y \\ \vec{\boldsymbol{\pi}} \\ \lambda \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\gamma}}_x \\ \vec{\boldsymbol{\gamma}}_y \\ \mathbf{0} \\ 0 \end{bmatrix}, \quad \begin{aligned} &\mathbf{A}_x, \mathbf{A}_y \in \mathbb{R}^{n,n}\,, \\ &\mathbf{B}_x, \mathbf{B}_y \in \mathbb{R}^{M,n}\,, \\ &\mathbf{c} \in \mathbb{R}^M\,, \end{aligned} \quad \begin{aligned} n &:= \dim \mathcal{BS}_{1,0}^0(\mathcal{M})\,, \\ M &:= \dim \mathcal{S}_1^0(\mathcal{M})\,. \end{aligned} \tag{12.2.9}$$

The vector of unknowns collects all *unknown* basis expansion coefficients for (the components of) $\mathbf{v}_h$ and p_h, whereas the right-hand side vector arises from the prescribed values of the velocity on $\partial\Omega$.

Give a *sharp upper bound* for the number of non-zero entries of the blocks of the system matrix of (12.2.9) in terms of their cardinalities. You may use the notations for the various sets of mesh entities

introduced in Sub-problem (12-2.a).

$$\mathrm{nnz}(\mathbf{A}_x) = \mathrm{nnz}(\mathbf{A}_y) \leq \boxed{},$$

$$\mathrm{nnz}(\mathbf{B}_x) = \mathrm{nnz}(\mathbf{B}_y) \leq \boxed{},$$

$$\mathrm{nnz}(\mathbf{c}) \leq \boxed{}.$$

SOLUTION for (12-2.b) → 12-2-2-0:minis2.pdf

▲

Somebody else has left you an old LEHRFEM++-based C++ code implementing another conforming finite element Galerkin discretization of the variational problem underlying (12.2.5). This FEM will be called "simple FEM" in the sequel. The code adheres to a *monolithic* design in the sense that a single **lf::assemble::DofHandler** object is used to encode the local→global mappings for all finite element spaces combined.

The definition of the **ENTITY_MATRIX_PROVIDER** for that simple FEM is listed in the following code

C++ code 12.2.11: Definition of ENTITY_MATRIX_PROVIDER class for simple finite element method

```
class SimpleFEMElementMatrixProvider {
  public:
  using ElemMat = Eigen::Matrix<double, 9, 9>;
  SimpleFEMElementMatrixProvider(const SimpleFEMElementMatrixProvider &) =
  delete;
  SimpleFEMElementMatrixProvider(SimpleFEMElementMatrixProvider &&) noexcept =
  default;
  SimpleFEMElementMatrixProvider &operator=(
  const SimpleFEMElementMatrixProvider &) = delete;
  SimpleFEMElementMatrixProvider &operator=(SimpleFEMElementMatrixProvider &&) =
  delete;
  SimpleFEMElementMatrixProvider() = default;
  virtual ~SimpleFEMElementMatrixProvider() = default;
  [[nodiscard]] bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  [[nodiscard]] ElemMat Eval(const lf::mesh::Entity &cell);
};
```

The well-documented implementation of the `Eval()` member function is given next.

C++ code 12.2.12: Definition of the `Eval()` member function of the class SimpleFEMElementMatrixProvider.

```
lf::mesh::Entity& cell) {
SimpleFEMElementMatrixProvider::ElemMat SimpleFEMElementMatrixProvider::Eval(
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
  "Unsupported cell type " << cell.RefEl());
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions, see
  // [Lecture → Code 2.4.5.11]. First fetch vertices of the triangle
  auto endpoints = lf::geometry::Corners(*(cell.Geometry()));
  Eigen::Matrix<double, 3, 3> X;  // temporary matrix
```

```
  X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
  X.block<3, 2>(0, 1) = endpoints.transpose();
  // This matrix contains grad λ_i in its columns
  const auto G{X.inverse().block<2, 3>(1, 0)};
  // Inner products of gradients of barycentric coordinate functions
  // (Scaled) element matrix for −Δ
  const Eigen::Matrix3d L{G.transpose() * G};
  // Fixed-size matrix containing the element matrix
  ElemMat AK;
  AK.setZero();
  // indices of local shape function for various solution components
  constexpr std::array<Eigen::Index, 3> vx_idx_{0, 3, 6};
  constexpr std::array<Eigen::Index, 3> vy_idx_{1, 4, 7};
  constexpr std::array<Eigen::Index, 3> p_idx_{2, 5, 8};
  // Place the element matrix for −Δ in the element matrix
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      AK(vx_idx_[i], vx_idx_[j]) = L(i, j);
      AK(vy_idx_[i], vy_idx_[j]) = L(i, j);
    }
  }
  // Insert contributions to "pressure parts" of element matrix
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      AK(vx_idx_[j], p_idx_[i]) = AK(p_idx_[i], vx_idx_[j]) = G(0, j) / 3.0;
      AK(vy_idx_[j], p_idx_[i]) = AK(p_idx_[i], vy_idx_[j]) = G(1, j) / 3.0;
    }
  }
  // Finally scale with the area of the triangle
  AK *= area;
  return AK;
}
```

(12-2.c) ⚀ (20 min.) What are the finite element spaces (on $\mathcal{M}$) used in the simple FEM for the approximation of the velocity $\mathbf{v}$ and of the pressure p?

$$\mathbf{v}_h \in \boxed{} \quad , \quad p_h \in \boxed{} \; ?$$

Use the notations from [Lecture → Chapter 2].

HIDDEN HINT 1 for (12-2.c) → 12-2-3-0:mini4ah1.pdf

SOLUTION for (12-2.c) → 12-2-3-1:minis4a.pdf

▲

(12-2.d) ⚅ (15 min.)

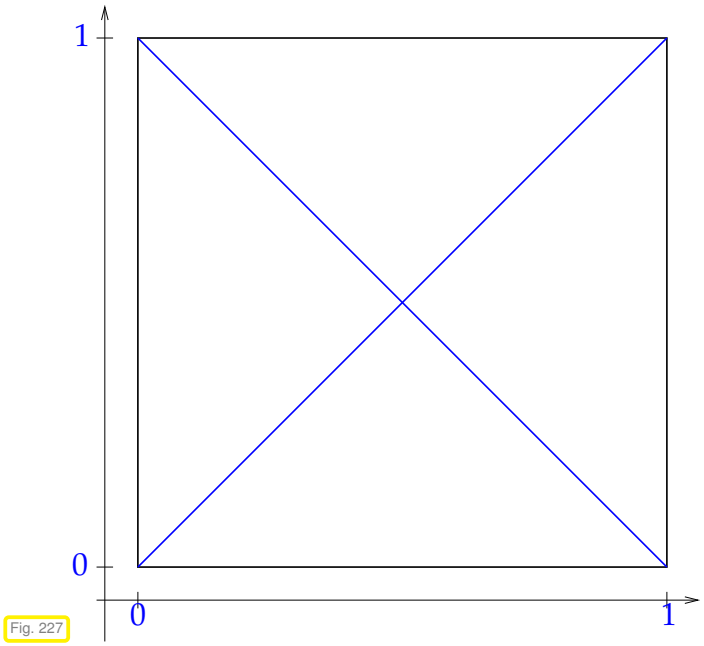


Fig. 227

The full 8×8 Galerkin matrix for the simple FEM and the 4-cell mesh drawn beside is

$$\mathbf{A} := \begin{bmatrix} 4 & 0 & \frac{1}{6} & -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} & 0 & 0 \\ 0 & 4 & \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ -\frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} \\ 0 & 0 & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{6} & \frac{1}{3} & 0 \end{bmatrix}$$

What is the rank of $\mathbf{A}$?

$$\operatorname{rank}(\mathbf{A}) = \boxed{} .$$

SOLUTION for (12-2.d) → 12-2-4-0:minis3.pdf

▲

(12-2.e) ⚄ (30 min.) This is the output of the `main()` program, which solves a Stokes boundary value problem with manufactured solution on a sequence of triangular meshes (created by uniform regular refinement) with the simple FEM presented above.

N	L2 err(**v1**)	L2 err(**v2**)	H1 err(**v1**)	H1 err(**v2**)	L2 err(**p**)
39	0.0262	0.0227	0.188	0.231	4.43e+14
120	0.0134	0.00907	0.235	0.166	5.16e+13
417	0.00185	0.0013	0.0657	0.0506	3.18e+12
1551	0.000385	0.000414	0.0307	0.0307	7.33e+11
5979	0.000371	0.000396	0.0446	0.0481	7.14e+11
23475	0.000119	9.6e−05	0.0253	0.0248	9.91e+10
93027	0.000288	0.000364	0.109	0.117	4.94e+10

The table lists various norms of the discretization error for the velocity components (`v1` and `v2`) and the pressure (`p`). The first column gives the total number of degrees of freedom.

Explain the data contained in the table.

SOLUTION for (12-2.e) → 12-2-5-0:minis5.pdf

▲

(12-2.f) ⚄ (30 min.) For a non-degenerate plane triangle $K \subset \mathbb{R}^2$ evaluate the following integrals

in terms of its area $|K|$ and the gradients of its barycentric coordinate function $g_i := \mathbf{grad}\,\lambda_i$.

(i) $\displaystyle\int_K \beta_K(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} =$,

(ii) $\displaystyle\int_K \mathbf{grad}\,\beta_K(\boldsymbol{x})\cdot\mathbf{grad}\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} =$,

(iii) $\displaystyle\int_K \frac{\partial\beta_K}{\partial x_k}\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} =$, $k = 1,2$.

HIDDEN HINT 1 for (12-2.f) → 12-2-6-0:mini7h1.pdf

HIDDEN HINT 2 for (12-2.f) → 12-2-6-1:mini7h2.pdf

SOLUTION for (12-2.f) → 12-2-6-2:minis7.pdf

▲

(12-2.g) ⚄ (60 min.) [depends on Sub-problem (12-2.f)]

In the files `stokesminielement.h` *and* `stokesminielement.cc` you find the definition and implementation (of the `Eval()` member function) of the class **MINIElementMatrixProvider**.

At this point the code is just a copy of **SimpleFEMElementMatrixProvider**.

Modify the code in such a way that **MINIElementMatrixProvider** becomes a valid **ENTITY_MATRIX_PROVIDER** for the MINI finite element method for (12.2.5).

HIDDEN HINT 1 for (12-2.g) → 12-2-7-0:mini6h1.pdf

SOLUTION for (12-2.g) → 12-2-7-1:minicode6.pdf

▲

(12-2.h) ⚂ (20 min.) At this point the C++ function `testCvgMINIFEM()` is just a copy of `testCvgSimpleFEM()` which conducts convergence test for the Simple FEM on a sequence of meshes created by uniform regular refinement and based on a manufactured solution.

Adapt the code so that it carries out the same convergence test for the MINI Stokes FEM.

You need to make *small changes* to only a part of the function, which is listed next.

C++ code 12.2.23: Part of `testCvgMINIFEM()` to be modified

```
// ********** Part III: Solving on a single mesh **********
// Initialize dof handler for simple FEM
lf::assemble::UniformFEDofHandler dofh(lev_mesh_p,
                                       {{lf::base::RefEl::kPoint(), 3},
                                        {lf::base::RefEl::kSegment(), 0},
                                        {lf::base::RefEl::kTria(), 0},
                                        {lf::base::RefEl::kQuad(), 0}});
// Build and solve the linear system with trace of the exact velocity
// solution as Dirichlet data.
// Total number of d.o.f.  in monolithic FE spaces
size_t n = dofh.NumDofs();
std::cout << "Computing with " << n << " d.o.f.s, assembling LSE .. "
          << std::flush;
// Obtain full Galerkin matrix in triplet format
```

```cpp
// Full Galerkin matrix in triplet format taking into account the zero mean
// constraint on the pressure.
lf::assemble::COOMatrix<double> A(n + 1, n + 1);
// Set up computation of element matrix
SimpleFEMElementMatrixProvider themp{};
// Assemble full Galerkin matrix for simple Stokes FEM
lf::assemble::AssembleMatrixLocally(0, dofh, dofh, themp, A);

// Add bottom row and right column corresponding to Lagrange multiplier
for (const lf::mesh::Entity* cell : dofh.Mesh()->Entities(0)) {
  LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                "Only implemented for triangles");
  // Obtain area of triangle
  const double area = lf::geometry::Volume(*cell->Geometry());
  // The pressure GSFs are associated with the nodes
  const std::span<const lf::mesh::Entity* const> nodes{
      cell->SubEntities(2)};
  // Loop over nodes
  for (const lf::mesh::Entity* node : nodes) {
    // Area of the cell
    // Obtain index of tent function associated with node
    // All indices of global shape functions sitting at node
    std::span<const lf::assemble::gdof_idx_t> dof_idx{
        dofh.InteriorGlobalDofIndices(*node)};
    LF_ASSERT_MSG(dof_idx.size() == 3, "Node must carry 3 dofs!");
    // The index of the tent function is the third one
    const lf::assemble::gdof_idx_t tent_idx = dof_idx[2];
    A.AddToEntry(n, tent_idx, area / 3.0);
    A.AddToEntry(tent_idx, n, area / 3.0);
  }
}
// Auxiliary right-hnad side vector
Eigen::VectorXd phi(A.cols());
phi.setZero();
// Impose Dirichlet boundary conditions
const std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh.Mesh();
// Flag nodes located on the boundary
auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 2)};
// Flag vector for d.o.f.  on the boundary
std::vector<std::pair<bool, double>> ess_dof_select(n + 1, {false, 0.0});
// Visit nodes on the boundary
for (const lf::mesh::Entity* node : mesh_p->Entities(2)) {
  if (bd_flags(*node)) {
    // Indices of global shape functions sitting at node
    std::span<const lf::assemble::gdof_idx_t> dof_idx{
        dofh.InteriorGlobalDofIndices(*node)};
    // Position of node
    const Eigen::Vector2d pos{Corners(*(node->Geometry())).col(0)};
    // Dirichlet data
    const Eigen::Vector2d g_val{v_ex(pos)};
    // x-component of the velocity
    ess_dof_select[dof_idx[0]] = {true, g_val[0]};
    // y-component of the velocity
    ess_dof_select[dof_idx[1]] = {true, g_val[1]};
  }
}
// modify linear system of equations
lf::assemble::FixFlaggedSolutionComponents<double>(
    [&ess_dof_select](lf::assemble::glb_idx_t dof_idx)
```

```
        -> std::pair<bool, double> { return ess_dof_select[dof_idx]; },
    A, phi);
// Assembly completed: Convert COO matrix A into CRS format using
    Eigen's
// internal conversion routines.
Eigen::SparseMatrix<double> A_crs{A.makeSparse()};

std::cout << "done. Solving ..... " << std::flush;
// Solve linear system using Eigen's sparse direct elimination
// Examine return status of solver in case the matrix is singular
Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
solver.compute(A_crs);
LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
// This is the coefficient vector for the FE solution; Dirichlet
// boundary conditions are included
const Eigen::VectorXd res = solver.solve(phi);
LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
std::cout << "done. |dof vector| = " << res.norm() << std::endl;
```

SOLUTION for (12-2.h) → 12-2-8-0:minis7.pdf

▲

(12-2.i) ⚀ (25 min.) From [ABF84] we learn the following estimate

Theorem 12.2.25. inf-sup condition for MINI FEM

With a constant $C > 0$ depending only on the shape-regularity and quasi-uniformity of the triangular mesh $\mathcal{M}$ there holds

$$\sup_{\mathbf{w}_h \in \left(\mathcal{BS}^0_{1,0}(\mathcal{M})\right)^2} \frac{\left|\int_\Omega \operatorname{div} \mathbf{w}_h(\boldsymbol{x})\, q_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right|}{|\mathbf{v}_h|_{H^1(\Omega)}} \geq C\|q_h\|_{L^2(\Omega)} \quad \forall q_h \in \mathcal{S}^0_1(\mathcal{M})\,. \tag{12.2.26}$$

Consider a sequence of triangular meshes $\mathcal{M}_\ell$, $\ell \in \mathbb{N}$ created by uniform regular refinement and denote by $(\mathbf{v}_\ell, p_\ell) \in \left(\mathcal{BS}^0_{1,0}(\mathcal{M}_\ell)\right)^2 \times \mathcal{S}^0_1(\mathcal{M}_\ell)$ the MINI FEM solution of a Stokes BVP (12.2.4) with exact solution $(\mathbf{v}, p) \in H^3(\Omega) \times H^3(\Omega)$. Predict the asymptotic convergence of the following error norms.

$$|\mathbf{v}_\ell - \mathbf{v}|_{H^1(\Omega)} = O\left(\boxed{}\right) \quad \text{for} \quad h_\ell \to 0\,,$$

$$\|p_\ell - p\|_{L^2(\Omega)} = O\left(\boxed{}\right) \quad \text{for} \quad h_\ell \to 0\,,$$

where h_ℓ is the mesh width of $\mathcal{M}_\ell$.

SOLUTION for (12-2.i) → 12-2-9-0:minnis7.pdf

▲

References

[ABF84] D. N. Arnold, F. Brezzi, and M. Fortin. "A stable finite element for the Stokes equations". In: *Calcolo* 21.4 (1984), pp. 337–344. DOI: 10.1007/BF02576171 (cit. on pp. 482, 489).

[CB22] Andrea Cioncolini and Daniele Boffi. "Superconvergence of the MINI mixed finite element discretization of the Stokes problem: an experimental study in 3D". In: *Finite Elem. Anal. Des.* 201 (2022), Paper No. 103706, 12. DOI: 10.1016/j.finel.2021.103706.

End Problem 12-2 , 230 min.

Problem 12-3: Taylor-Hood FEM for Stokes BVP

The Taylor-Hood finite element method of [Lecture → Section 12.3.4] is the most popular conforming stable finite element discretization of the Stokes variational saddle point problem [Lecture → (12.2.2.44)]. In this project we implement it in LEHRFEM++ using different **lf::fe::ScalarFESpace** objects to handle the two components of the velocity and the pressure, a **non-monolithic** approach to finite element discretization.

Beside the contents of [Lecture → Section 11.6.3.3] this project requires familiarity with LEHRFEM++.
▷ problem name/problem code folder: TaylorHoodNonMonolithic

The **Taylor-Hood finite element method (TH-FEM)** from [Lecture → Section 12.3.4] can be used to solve a boundary value problem for the Stokes equations with no-slip boundary conditions [Lecture → (12.2.3.5)],

$$-\Delta \mathbf{v} - \mathbf{grad}\, p = \mathbf{f} \quad , \quad \operatorname{div} \mathbf{v} = 0 \quad \text{in} \quad \Omega \quad , \quad \mathbf{v} = \mathbf{0} \quad \text{on} \quad \partial\Omega \, , \tag{12.3.1}$$

on a polygonally bounded computational domain $\Omega \subset \mathbb{R}^2$ and for given driving force field $\mathbf{f} \in \left(C^0(\overline{\Omega})\right)$. Computing the TH-FEM solution of (12.3.1) amounts to solving the following discrete variational problem: seek $\mathbf{v}_h \in \left(\mathcal{S}^0_{2,0}(\mathcal{M})\right)^2$, $p_h \in \mathcal{S}^0_1(\mathcal{M})$,

$$\begin{aligned} \int_\Omega \mathrm{D}\mathbf{v}_h : \mathrm{D}\mathbf{w}_h \,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div} \mathbf{w}_h \, p_h \,\mathrm{d}\boldsymbol{x} \;&=\; \int_\Omega \mathbf{f} \cdot \mathbf{w}_h, \mathrm{d}\boldsymbol{x} \qquad \forall \mathbf{w}_h \in (\mathcal{S}^0_{2,0}(\mathcal{M}))^2 \, , \\ \int_\Omega \operatorname{div} \mathbf{v}_h \, q_h \,\mathrm{d}\boldsymbol{x} \;&=\; 0 \qquad\qquad\qquad\quad \forall q \in \mathcal{S}^0_1(\mathcal{M}) \, , \end{aligned} \tag{12.3.2}$$

where $\mathcal{M}$ is a *triangular* mesh of Ω. We can split the velocity field $\mathbf{v}$ in its two Cartesian components, $\mathbf{v} = \left[\begin{smallmatrix} v_x \\ v_y \end{smallmatrix}\right]$, and state the TH-FEM discrete variational problem in terms of these: seek $v_{x,h} \in \mathcal{S}^0_{2,0}(\mathcal{M})$, $v_{y,h} \in \mathcal{S}^0_{2,0}(\mathcal{M})$, $p_h \in \mathcal{S}^0_1(\mathcal{M})$ such that

$$\begin{aligned} \int_\Omega \begin{bmatrix} \mathbf{grad}^\top v_{x,h} \\ \mathbf{grad}^\top v_{y,h} \end{bmatrix} : \begin{bmatrix} \mathbf{grad}^\top w_{x,h} \\ \mathbf{grad}^\top w_{y,h} \end{bmatrix} \mathrm{d}\boldsymbol{x} + \int_\Omega \operatorname{div} \begin{bmatrix} w_{x,h} \\ w_{y,h} \end{bmatrix} p_h \,\mathrm{d}\boldsymbol{x} \;&=\; \int_\Omega \mathbf{f} \cdot \begin{bmatrix} w_{x,h} \\ w_{y,h} \end{bmatrix} \mathrm{d}\boldsymbol{x} \quad \forall \begin{array}{l} w_{x,h} \in \mathcal{S}^0_{2,0}(\mathcal{M}) \, , \\ w_{y,h} \in \mathcal{S}^0_{2,0}(\mathcal{M}) \, , \end{array} \\ \int_\Omega \operatorname{div} \begin{bmatrix} v_{x,h} \\ v_{y,h} \end{bmatrix} q_h \,\mathrm{d}\boldsymbol{x} \;&=\; 0 \qquad\qquad\qquad\qquad \forall q_h \in \mathcal{S}^0_1(\mathcal{M}) \, . \end{aligned} \tag{12.3.3}$$

Relying on the standard "nodal" bases $\mathfrak{B}_2 := \{b^1_h, \ldots, b^N_h\}$, $\mathfrak{B}_1 := \{q^1_h, \ldots, q^M_h\}$, $N, M \in \mathbb{N}$, of $\mathcal{S}^0_{2,0}(\mathcal{M})$ and $\mathcal{S}^0_1(\mathcal{M})$, respectively, see [Lecture → Ex. 2.6.1.2] and [Lecture → Section 2.4.3], we can convert (12.3.3) into a linear system of equations

$$\begin{bmatrix} \mathbf{A} & \mathbf{O} & \mathbf{B}_x^\top \\ \mathbf{O} & \mathbf{A} & \mathbf{B}_y^\top \\ \mathbf{B}_x & \mathbf{B}_y & \mathbf{O} \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{v}}_x \\ \vec{\boldsymbol{v}}_y \\ \vec{\boldsymbol{\pi}} \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}_x \\ \vec{\boldsymbol{\varphi}}_y \\ \mathbf{0} \end{bmatrix} \tag{12.3.4}$$

for the basis expansion coefficient vectors $\vec{\boldsymbol{v}}_x$ of $v_{x,h}$, $\vec{\boldsymbol{v}}_y$ of $v_{y,h}$, and $\vec{\boldsymbol{\pi}}$ of p_h.

(12-3.a) ⚀ (15 min.) We introduce notations for the following sets of mesh entities:

- $\mathcal{V}_i(\mathcal{M}) \mathrel{\hat{=}}$ set of interior nodes of $\mathcal{M}$, $\mathcal{V}_\partial(\mathcal{M}) \mathrel{\hat{=}}$ set of mesh nodes on $\partial\Omega$, $\mathcal{V}(\mathcal{M}) := \mathcal{V}_i(\mathcal{M}) \cup \mathcal{V}_\partial(\mathcal{M})$,

- $\mathcal{E}_i(\mathcal{M}) \triangleq$ set of interior edges of $\mathcal{M}$, $\mathcal{E}_\partial(\mathcal{M}) \triangleq$ set of mesh edges $\subset \partial\Omega$, $\mathcal{E}(\mathcal{M}) := \mathcal{E}_i(\mathcal{M}) \cup \mathcal{E}_\partial(\mathcal{M})$,

and as usual we write $\mathcal{M}$ also for the set of mesh cells.

Express the dimensions N and M in terms of the cardinalities of the sets $\mathcal{V}_i(\mathcal{M})$, $\mathcal{V}_\partial(\mathcal{M})$, $\mathcal{E}_i(\mathcal{M})$, $\mathcal{E}_\partial(\mathcal{M})$, and $\mathcal{M}$.

$N =$ ☐ , $M =$ ☐ .

Use the operator $\sharp$ to designate the cardinality (= number of elements) of a finite set.

SOLUTION for (12-3.a) → 12-3-1-0:thfems1.pdf

▲

(12-3.b) ⚂ (20 min.) We call $\mathbf{M}$ a $X_h \times Y_h$**-Galerkin matrix** for the bilinear form $c : X \times Y \to \mathbb{R}$, if $\mathbf{M}$ arises from the Galerkin discretization of c using the trial space $X_h \subset X$ and test space $Y_h \subset Y$, both equipped with suitable bases.

Complete the following statements on the matrices occurring in (12.3.4):

(i) $\mathbf{A}$ is the ☐ × ☐ -Galerkin matrix for the bilinear form

$$(u,v) \mapsto \int_\Omega \boxed{} \,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1(\Omega)\,.$$

(ii) $\mathbf{B}_x$ is the ☐ × ☐ -Galerkin matrix for the bilinear form

$$(u,q) \mapsto \int_\Omega \boxed{} \,\mathrm{d}\boldsymbol{x}\,, \quad u \in H^1(\Omega),\ q \in L^2(\Omega)\,.$$

(iii) $\mathbf{B}_y$ is the ☐ × ☐ -Galerkin matrix for the bilinear form

$$(u,q) \mapsto \int_\Omega \boxed{} \,\mathrm{d}\boldsymbol{x}\,, \quad u \in H^1(\Omega),\ q \in L^2(\Omega)\,.$$

SOLUTION for (12-3.b) → 12-3-2-0:thfems2.pdf

▲

(12-3.c) ⚄ (25 min.) [depends on Sub-problem (12-3.b)]

In the figure beside the black line shows $\partial\Omega$.
The red lines mark the (interior) edges of a triangular mesh of Ω with

$$\begin{aligned}\sharp\mathcal{M} &= 17\,,\\ \sharp\mathcal{V}_\partial(\mathcal{M}) &= 9\,,\\ \sharp\mathcal{V}_i(\mathcal{M}) &= 5\,,\\ \sharp\mathcal{E}_\partial(\mathcal{M}) &= 9\,,\\ \sharp\mathcal{E}_i(\mathcal{M}) &= 21\,.\end{aligned}$$

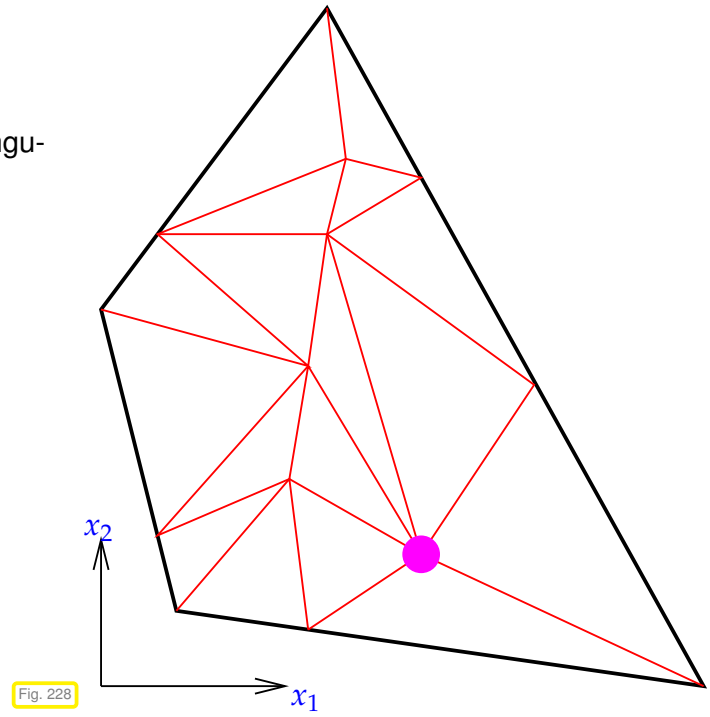


Fig. 228

What is the size of the matrix $\mathbf{B}_x$?

$\mathbf{B}_x \in \mathbb{R}^{\square,\square}$.

What is the number of non-zero entries in the *column* of $\mathbf{B}_x$ induced by a global shape function associated with the node • in Fig. 228.

$$\mathrm{nnz}((\mathbf{B}_x)_{:,\bullet}) = \square\ ?$$

What is the number of non-zero entries in the *row* of $\mathbf{B}_x$ induced by a global shape function associated with the node • in Fig. 228.

$$\mathrm{nnz}((\mathbf{B}_x)_{\bullet,:}) = \square\ ?$$

What is the number of non-zero entries of the matrix $\mathbf{B}_x$ for this mesh $\mathcal{M}$?

$$\mathrm{nnz}(\mathbf{B}_x) = \square\,.$$

SOLUTION for (12-3.c) → 12-3-3-0:thfem3s.pdf

▲

The LEHRFEM++-based C++ function, defined in `taylorhoodnonmonolithic.h`,

```
template <typename FFUNCTOR>
  std::tuple<Eigen::SparseMatrix<double>,
             Eigen::SparseMatrix<double>,
             Eigen::SparseMatrix<double>,
             Eigen::VectorXd, Eigen::VectorXd>
    buildStokesLSE(std::shared_ptr<const lf::mesh::Mesh> mesh_p,
                   FFUNCTOR&& force);
```

initializes the matrix blocks $\mathbf{A}, \mathbf{B}_x, \mathbf{B}_y$ and right-hand side vector segments $\vec{\varphi}_x, \vec{\varphi}_y$ of (12.3.4) and returns the tuple

$$(\mathbf{A}, \mathbf{B}_x, \mathbf{B}_y, \vec{\varphi}_x, \vec{\varphi}_y).$$

The parameter `mesh_p` passes a pointer to an **lf::mesh::Mesh** object describing a triangular mesh, and `force` is a functor with signature `sdt::function<Eigen::Vector2d(Eigen::Vector2d)>`, which provides the force field $\mathbf{f}$.

(12-3.d) ⚂ (20 min.) `taylorhoodnonmonolithic.h` contains an incomplete implementation of the C++ function `buildStokesLSE()`. whose first part builds a precursor of the matrix block $\mathbf{A}$ in triplet (COO) format and stores it in the **lf::assemble::COOMatrix** object `A_velo_comp`. It is called a precursor, because boundary conditions are not yet taken into account, that is, `A_velo_comp` contains a full $\mathcal{S}_2^0(\mathcal{M}) \times \mathcal{S}_2^0(\mathcal{M})$-Galerkin matrix.

Supplement the missing arguments of the two incomplete function calls.

SOLUTION for (12-3.d) → `12-3-4-0:ths4s.pdf`

▲

(12-3.e) ⚄ (30 min.) [depends on Sub-problem (12-3.b)]

The following class fits the concept of **ENTITY_MATRIX_PROVIDER** and when used in conjunction with `lf::assemble::AssembleMatrixLocally()` is supposed to yield the precursor matrices, often also called full Galerkin matrices, that is, the Galerkin matrices without boundary conditions, for the matrix blocks $\mathbf{B}_x$ and $\mathbf{B}_y$ in(12.3.4).

```
enum DirFlag : Eigen::Index { X_Dir = 0, Y_Dir = 1 };
class THBElementMatrixProvider {
 public:
  using ElemMat = Eigen::Matrix<double, 3, 6>;
  // Main constructor
  THBElementMatrixProvider(const THBElementMatrixProvider &) =
      delete;
  THBElementMatrixProvider(THBElementMatrixProvider &&) noexcept =
      default;
  THBElementMatrixProvider &operator=(const
     THBElementMatrixProvider &) =
       delete;
  THBElementMatrixProvider &operator=(THBElementMatrixProvider &&)
     = delete;
  THBElementMatrixProvider(DirFlag dirflag) : dirflag_(dirflag) {}
  virtual ~THBElementMatrixProvider() = default;
  [[nodiscard]] bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  [[nodiscard]] ElemMat Eval(const lf::mesh::Entity &cell);

 private:
  DirFlag dirflag_;  // Chooses partial derivative in bilinear form
};
```

The `dirflag` argument to the constructor determines whether the `Eval()` method returns the element matrix for $\mathbf{B}_x$ (`dirfalg == X_Dir`) or $\mathbf{B}_y$ (`dirflag = Y_Dir`).

In the file `taylorhoodnonmonolithic.cc` complete the code of the `Eval()` member function.

To compute the element matrix you have to integrate quadratic polynomials over a triangle. Use the local quadrature formula that relies on the edge midpoints as evaluation nodes:

$$\int_K \varphi(\lambda_1(\boldsymbol{x}),\lambda_2(\boldsymbol{x}),\lambda_3(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} = \tfrac{1}{3}|K|\Big(\varphi(\tfrac{1}{2},\tfrac{1}{2},0)+\varphi(0,\tfrac{1}{2},\tfrac{1}{2})+\varphi(\tfrac{1}{2},0,\tfrac{1}{2})\Big)\,. \tag{12.3.13}$$

HIDDEN HINT 1 for (12-3.e) → 12-3-5-0:th5h1.pdf

SOLUTION for (12-3.e) → 12-3-5-1:.pdf

▲

The **conjugate gradient Uzawa algorithm** [Bra07, IV.§5, 5.1] is an iterative solution method for saddle point linear systems of equations. For the general saddle point linear system

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^\top \\ \mathbf{B} & \mathbf{O} \end{bmatrix} \begin{bmatrix} \vec{\mu} \\ \vec{\pi} \end{bmatrix} = \begin{bmatrix} \vec{\varphi} \\ \vec{\gamma} \end{bmatrix} \tag{12.3.17}$$

this algorithm is defined through the following pseudocode:

Pseudocode 12.3.18: CG-Uzawa algorithm for solving (12.3.17) **iteratively**

```
[Vector, Vector] = CGUzawa(
  const Matrix A ∈ ℝ^{N,N}, const Matrix B ∈ ℝ^{M,N},
  const Vector φ⃗ ∈ ℝ^N, const Vector γ⃗ ∈ ℝ^N,
  double rtol, double atol) {
    Vector π⃗ ∈ ℝ^M; π⃗ := 0; // Approximates solution component π⃗
    Vector μ⃗ := A^{-1}φ⃗ ∈ ℝ^N; // Approximates solution component μ⃗
    Vector δ⃗ := Bμ⃗ − γ⃗ ∈ ℝ^M;
    Vector ρ⃗ := −δ⃗;
    const double rn0 := ‖ρ⃗‖_2;
    double rn = r0;
    Vector ω⃗ ∈ ℝ^N;
    Vector η⃗ ∈ ℝ^N;
    do { // Main iteration loop
      ω⃗ := B^⊤δ⃗;
      η⃗ := A^{-1}ω⃗;
      double α = (rn * rn) / ω⃗^⊤η⃗;
      π⃗ := π⃗ + αδ⃗;
      μ⃗ := μ⃗ − αη⃗;
      ρ⃗ := γ⃗ − Bμ⃗;
      double rn1 = ‖ρ‖_2;
      double β = (rn1*rn1)/(rn*rn);
      rn := rn1;
      δ⃗ := −ρ⃗ + βδ⃗;
    }
    while ((rn ≥ rn0 * rtol) and (rn ≥ atol)); // Termination
    return { μ⃗, π⃗ }; // Return approximate solution vectors
}
```

This code implements a residual based termination rule controlled by two tolerance parameters. It uses initial guess $= \mathbf{0}$ for the $\vec{\pi}$-component of the solution.

(12-3.f) ⚄ (40 min.) In the file `taylorhoodnonmonolithic.h` finish the implementation of the templated C++ function

```
template <typename RECORDER>
std::tuple<Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd>
  CGUzawa(
    const Eigen::SparseMatrix<double> &A,
    const Eigen::SparseMatrix<double> &B_x,
    const Eigen::SparseMatrix<double> &B_y,
    const Eigen::VectorXd &phi_x,
    const Eigen::VectorXd &phi_y,
    double rtol, double atol,  unsigned int itmax, RECORDER &&rec);
```

It realizes the conjugate gradient Uzawa method from Code 12.3.18 in C++ and for the particular saddle point linear system (12.3.4), whose matrix blocks $\mathbf{A}$, $\mathbf{B}_x$, and $\mathbf{B}_y$ are passed through `A`, `B_x`, and `B_y`, while the right-hand side vectors $\vec{\varphi}_x$ and $\vec{\varphi}_y$ correspond to `phi_x` and `phi_y`. As in Code 12.3.18 termination is controlled by `atol` and `rtol`. The argument `rec` supplies an optional object matching

```
std::function<void(const Eigen::VectorXd &, const Eigen::VectorXd &,
                   const Eigen::VectorXd &)>
```

The function should return the final approximations for $\vec{\mu}_x$, $\vec{\mu}_y$, and $\vec{\pi}$.

SOLUTION for (12-3.f) → `12-3-6-0:thxs.pdf`

▲

(12-3.g) ⚄ (30 min.)

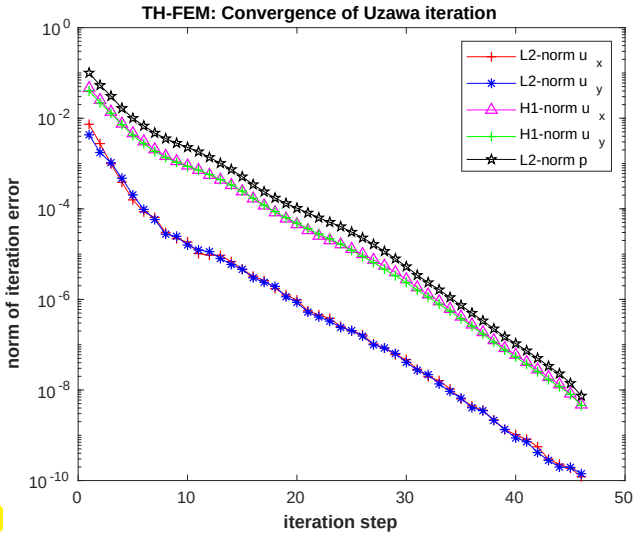


Fig. 230

◁ This semi-logarithmic plot shows various (function space) norms of the approximate Taylor-Hood finite element solutions as reconstructed from the coefficient vectors during the conjugate gradient Uzawa iteration of Code 12.3.18.

The data hint that the conjugate gradient Uzawa method *converges linearly* [NumCSE course → Def. 8.2.2.1], that is norms of the iteration error decrease by some fixed factor in every step of the iteration, for instance,

$$\left\| v_{x,h}^{(k+1)} - v_{x,h} \right\|_{H^1(\Omega)} \leq q \left\| v_{x,h}^{(k)} - v_{x,h} \right\|_{H^1(\Omega)} \quad \forall k \,, \tag{12.3.20}$$

for some $0 \leq q < 1$, chosen optimally (the rate of convergence). Here a superscript (k) designates the iterates.

Assume linear convergence of the conjugate gradient Uzawa iteration with *sharp* rate $0 < q < 1$ independent of the mesh. Also assume that we use the conjugate gradient Uzawa method to solve the saddle point linear systems of equations arising from the Taylor-Hood FEM for a Stokes boundary value problem (12.3.1) with a *smooth solution*.

On a given "nice" and rather fine triangular mesh $\mathcal{M}_0$ assume that we carry out the minimum amount of iterations such that the iteration error in the $H^1(\Omega)$-norm of the velocity is equal to the finite element discretization error $\|\mathbf{v} - \mathbf{v}_h\|_{H^1(\Omega)}$ with $\mathbf{v}_h \in \mathcal{S}^0_{2,0}(\mathcal{M}_0)$ the TH-FEM solution. Then we want to do the same simulation also on a mesh $\mathcal{M}_1$ created by **uniform regular refinement** of $\mathcal{M}_0$. How many more iteration steps does one have to carry out on $\mathcal{M}_1$ in order to keep the balance between iteration error and discretization error also on $\mathcal{M}_1$? Throughout we use the initial guess $= \mathbf{0}$ and sharpness of asymptotic a-priori error estimates can be taken for granted.

Number of additional iteration steps = ,

SOLUTION for (12-3.g) → 12-3-7-0:thsty.pdf

▲

References

[Bra07] Dietrich Braess. *Finite elements. Theory, Fast Solvers, and Applications in Elasticity Theory*. Third. Cambridge: Cambridge University Press, 2007, p. 365. DOI: 10.1017/CBO9780511618635 (cit. on p. 495).

End Problem 12-3 , 180 min.

Problem 12-4: FEM for Stokes BVPs

This problem revisits the variational formulation of the Stokes problem with no-slip boundary condition and also its Galerkin finite-element discretization, where the key notion is that of a **stable finite element pair**.

This problem does not involve any implementation and builds upon [Lecture → Section 12.2.2] and [Lecture → Section 12.3.2], [Lecture → Section 12.3.3].
▷ problem name/problem code folder: StokesVPFEM

In this problem we consider the flow of a viscous fluid in a container occupying the domain $\Omega \subset \mathbb{R}^2$. Velocity $\mathbf{v} : \Omega \to \mathbb{R}^2$ and pressure $p : \Omega \to \mathbb{R}$ can be obtained as solutions of the Stokes boundary value problem ($\mu > 0$, $\mathbf{f} \in \left(L^2(\Omega)\right)^2$)

$$\begin{aligned} -\mu\Delta\mathbf{v} - \mathbf{grad}\, p &= \mathbf{f} \quad \text{in} \quad \Omega\,, \\ \operatorname{div}\mathbf{v} &= 0 \quad \text{in} \quad \Omega\,, \\ \int_\Omega p\,\mathrm{d}\boldsymbol{x} &= 0 \\ \mathbf{v} &= 0 \quad \text{on} \quad \partial\Omega\,. \end{aligned} \qquad \text{([Lecture → (12.2.3.5)])}$$

(12-4.a) ⚂ (10 min.) Fill in the boxes so that the following becomes a valid weak/variational formulation for [Lecture → (12.2.3.5)]:

Seek $\mathbf{v} \in (H_0^1(\Omega))^d$, $p \in L^2(\Omega)$, $\lambda \in \mathbb{R}$ such that

$$\begin{aligned} \int_\Omega \boxed{}\,\mathrm{d}\boldsymbol{x} &= \boxed{}\,, \\ \int_\Omega \boxed{}\,\mathrm{d}\boldsymbol{x} &= 0\,, \\ \int_\Omega p\,\mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned}$$

for all $\mathbf{w} \in (H_0^1(\Omega))^2$, $q \in L^2(\Omega)$.

SOLUTION for (12-4.a) → 12-4-1-0:svps1.pdf ▲

For the remainder of this problem we assume that
the pair of finite element spaces (U_h, Q_h), $U_h \subset \left(H_0^1(\Omega)\right)^2$, $Q_h \in L^2_*(\Omega)$ (*) is a **stable finite element pair**.

(*) $L^2_*(\Omega) := \left\{v \in L^2(\Omega) : \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0\right\}$

Definition [Lecture → Def. 12.3.2.7]. Stable finite element pair

A pair of finite element spaces $U_h \subset \boldsymbol{H}^1_0(\Omega)$, $Q_h \subset L^2_*(\Omega)$ is a **stable finite element pair**, if the solution $(\mathbf{v}_h, p_h) \in U_h \times Q_h$ of the discrete variational Stokes problem [Lecture → (12.3.0.4)] exists, is unique, and satisfies

$$\exists C > 0: \quad \|\mathbf{v}_h\|_{\mathsf{a}} + \|p_h\|_{L^2(\Omega)} \leq C \sup_{\mathbf{w} \in (H^1_0(\Omega))^d} \frac{|\int_\Omega \mathbf{f}(\boldsymbol{x})\mathbf{w}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}|}{\|\mathbf{w}\|_{\mathsf{a}}} \quad \forall \mathbf{f} \in L^2(\Omega)\,,$$

where the constant $C > 0$ may depend only on Ω, the coefficient μ, and the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$.

(12-4.b) ⚀ (5 min.) What can we say about how the dimensions of the two spaces are related?

$$\dim U_h \quad \square \quad \dim Q_h\,.$$

Fill in "<", "≤", "=", "≥", or ">".

SOLUTION for (12-4.b) → 12-4-2-0:svpss2.pdf ▲

(12-4.c) ⚄ (10 min.) Fill in the boxes with meaningful expressions to complete the statement of the theorem:

Theorem [Lecture → Thm. 12.3.2.15]. inf-sup condition

The finite element spaces $U_h \subset \boldsymbol{H}^1_0(\Omega)$, $Q_h \subset L^2_*(\Omega)$ provide a stable finite element pair (→ [Lecture → Def. 12.3.2.7]) for the Stokes problem in variational form, if there is a constant $\beta > 0$ depending only on Ω and the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$ such that

$$\sup_{\mathbf{w}_h \in U_h} \frac{\square}{\square} \geq \beta \square \quad \forall q_h \in \square\,.$$

([Lecture → (12.3.3.25)])

Note. All symbols you use must be standard notations or must have been defined somewhere.

SOLUTION for (12-4.c) → 12-4-3-0:svpsol4.pdf ▲

(12-4.d) ⚄ (5 min.) On a finer mesh we obtain larger finite element spaces

$$\widetilde{U}_h \in \left(H^1_0(\Omega)\right)^2, \quad U_h \subset \widetilde{U}_h \quad \text{and} \quad \widetilde{Q}_h \subset L^2_*(\Omega)\,, \quad Q_h \subset \widetilde{Q}_h\,.$$

Which of the pairs $(U_h, \widetilde{Q}_h)$, $(\widetilde{U}_h, Q_h)$, $(\widetilde{U}_h, \widetilde{Q}_h)$ is/are guaranteed to provide a stable finite element pair for the Stokes variational problem?

Pair of spaces	Guaranteed to be a stable pair?			
$(U_h, \widetilde{Q}_h)$	YES	◯	NO	◯
$(\widetilde{U}_h, Q_h)$	YES	◯	NO	◯
$(\widetilde{U}_h, \widetilde{Q}_h)$	YES	◯	NO	◯

SOLUTION for (12-4.d) → 12-4-4-0:.pdf ▲

(12-4.e) ⚄ (10 min.) Let $\mathcal{M}$ be a triangular mesh of Ω. Theoretical results tell us that

$$\mathcal{S}^0_{2,0}(\mathcal{M}) \text{ and } \mathcal{S}^{-1}_0(\mathcal{M}) \text{ is a } \textit{stable finite element pair}$$

in the sense of [Lecture → Def. 12.3.2.7], *cf.* [Lecture → Exp. 12.3.2.1]. Remember that $\mathcal{S}^{-1}_0(\mathcal{M})$ is the space of $\mathcal{M}$-piecewise constant functions.

We also know that the exact solution $(\mathbf{v}, p)$ of the Stokes boundary value problem is $C^\infty(\overline{\Omega})$-smooth.

Given a sequence of triangular meshes created by *uniform regular refinement* and writing $\mathbf{v}_h \in \mathcal{S}^0_{2,0}(\mathcal{M})$ and $p_h \in \mathcal{S}^{-1}_0(\mathcal{M})$ for the finite element solutions of the Stokes variational problem, give a sharp *asymptotic* bound for the natural norm of the finite element discretization error

$$\|\mathbf{v} - \mathbf{v}_h\|_{H^1(\Omega)} + \|p - p_h\|_{L^2(\Omega)} = O\Big(\quad\Big) \quad \text{for meshwidth } h_{\mathcal{M}} \to 0\,.$$

SOLUTION for (12-4.e) → 12-4-5-0:svpcvgs.pdf ▲

End Problem 12-4, 40 min.

Problem 12-5: Stabilized P1-FEM for Stokes

Numerical tests show that lowest order Lagrangian finite elements for both velocity components and the pressure do not provide a stable finite element pair for the Stokes variational saddle point problem. Remedies in the form of **stabilized finite element methods** have been proposed. Those introduce an extra discrete pressure coupling term into the Stokes variational saddle point problem. In this project we implement and study the stabilization introduced in [BH08].

This problem assumes familiarity with LEHRFEM++ and the saddle point variational formulation of the Stokes problem from [Lecture → Section 12.2.2].
▷ problem name/problem code folder: StokesStabP1FEM

We consider the Stokes boundary value problem with zero Dirichlet boundary conditions on a bounded 2D domain $\Omega \subset \mathbb{R}^2$. We equip Ω with a *triangular* finite element mesh $\mathcal{M}$. Then the stabilized P1-FEM of [BH08] reads:

Seek $\mathbf{v}_h \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2$, $p_h \in \mathcal{S}^0_1(\mathcal{M})$

$$\begin{aligned} \int_\Omega \mathrm{D}\mathbf{v}_h(\boldsymbol{x}) : \mathrm{D}\mathbf{w}_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div}\mathbf{w}_h(\boldsymbol{x})\, p_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{f}(\boldsymbol{x})\cdot\mathbf{w}_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \operatorname{div}\mathbf{v}_h(\boldsymbol{x})\, q_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;-\; \int_\Omega \mathsf{I}_1(p_h q_h)(\boldsymbol{x}) - p_h(\boldsymbol{x}) q_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned} \tag{12.5.1}$$

for all $\mathbf{w}_h \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2$, $q_h \in \mathcal{S}^0_1(\mathcal{M})$.

We demand that the force field $\mathbf{f}$ is square-integrable, $\mathbf{f} \in \left(L^2(\Omega)\right)^2$. Further, in (12.5.1), I_1 is the **"nodal" interpolation operator** onto $\mathcal{S}^0_1(\mathcal{M})$ as introduced in [Lecture → Section 3.3.2], [Lecture → Def. 3.3.2.1].

Definition [Lecture → Def. 3.3.2.1]. Linear interpolation in 2D

The linear interpolation operator $\mathsf{I}_1 : C^0(\bar{\Omega}) \mapsto \mathcal{S}^0_1(\mathcal{M})$ is defined by

$$\mathsf{I}_1 u \in \mathcal{S}^0_1(\mathcal{M}) \quad , \quad \mathsf{I}_1 u(\boldsymbol{p}) = u(\boldsymbol{p}) \quad \forall \boldsymbol{p} \in \mathcal{V}(\mathcal{M})\,.$$

(12-5.a) ⚄ (2 min.) What problem arises in the statement of the following variational generalized saddle point problem?

"Seek $\mathbf{v} \in \left(H^1_0(\Omega)\right)^2$, $p \in H^1(\Omega)$

$$\begin{aligned} \int_\Omega \mathrm{D}\mathbf{v}(\boldsymbol{x}) : \mathrm{D}\mathbf{w}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div}\mathbf{w}(\boldsymbol{x})\, p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{f}(\boldsymbol{x})\cdot\mathbf{w}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \operatorname{div}\mathbf{v}(\boldsymbol{x})\, q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;-\; \int_\Omega \mathsf{I}_1(pq)(\boldsymbol{x}) - p(\boldsymbol{x}) q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned}$$

for all $\mathbf{w} \in \left(H^1_0(\Omega)\right)^2$, $q \in H^1(\Omega)$."

SOLUTION for (12-5.a) → `12-5-1-0:ssts1.pdf` ▲

Let $\mathfrak{B} = \{b^1_h, \ldots, b^N_h\}$, N = number of nodes of $\mathcal{M}$, be the "tent function" basis of $\mathcal{S}^0_1(\mathcal{M})$ [Lecture → Section 2.4.3] and $\mathfrak{B}_0 = \left\{\mathring{b}^1_h, \ldots, \mathring{b}^n_h\right\} \subset \mathfrak{B}$, n = number of interior nodes of $\mathcal{M}$, the set of tent functions

associated with interior nodes. As sets of **global shape functions** (GSFs) we use

- $\mathfrak{B}_V := \left\{\mathbf{b}_h^1,\ldots,\mathbf{b}_h^{2n}\right\} := \left\{\begin{bmatrix}\mathring{b}_h^1\\0\end{bmatrix},\ldots,\begin{bmatrix}\mathring{b}_h^n\\0\end{bmatrix},\begin{bmatrix}0\\\mathring{b}_h^1\end{bmatrix},\ldots,\begin{bmatrix}0\\\mathring{b}_h^n\end{bmatrix}\right\}$ for $\left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2$, and
- $\mathfrak{B}$ for $\mathcal{S}^0_1(\mathcal{M})$.

Then (12.5.1) is equivalent to the linear system of equations

$$\begin{bmatrix}\mathbf{A} & \mathbf{B}^\top\\ \mathbf{B} & -\mathbf{S}\end{bmatrix}\begin{bmatrix}\vec{\boldsymbol{\nu}}\\ \vec{\boldsymbol{\pi}}\end{bmatrix} = \begin{bmatrix}\vec{\boldsymbol{\varphi}}\\ \mathbf{0}\end{bmatrix}, \tag{12.5.2}$$

where $\vec{\boldsymbol{\nu}} \in \mathbb{R}^{2n}$ is the $\mathfrak{B}_V$-basis expansion coefficient vector for $\mathbf{v}_h \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2$, and $\vec{\boldsymbol{\pi}} \in \mathbb{R}^N$ that for $p_h \in \mathcal{S}^0_1(\mathcal{M})$ with respect to the basis $\mathfrak{B}$.

(12-5.b) ⚄ (3 min.) Write down a non-zero kernel vector $\vec{\boldsymbol{\zeta}} \in \mathbb{R}^{2n+N}$ of the matrix of (12.5.2).

$$\vec{\boldsymbol{\zeta}} = \begin{bmatrix}\square\\ \square\\ \square\end{bmatrix}.$$

The partitioning of the vector matches the block partitioning in (12.5.2).

SOLUTION for (12-5.b) → 12-5-2-0:ststs2.pdf ▲

We aim for an implementation of the stabilized P1-FEM for Stokes in LEHRFEM++. Beside the numbering conventions of LEHRFEM++ we have to decide about the order of the three global shape functions sitting in every node of $\mathcal{M}$.

Ordering convention for node-associated GSFs

We order the global shape functions associated with the i-th node according to the rule

1. First comes the GSF $\begin{bmatrix}b_h^i\\0\end{bmatrix}$ for the x_1-component of the velocity,
2. then comes the GSF $\begin{bmatrix}0\\b_h^i\end{bmatrix}$ for the x_2-component of the velocity, and
3. last comes the tent-function GSF b_h^i for the pressure.

(12-5.c) ⚄ (12 min.) The following class fits the LEHRFEM++ concept of **ENTITY_MATRIX_PROVIDER** and is supposed to compute the element matrix for the monolithic stabilized P1 Stokes FEM.

C++ code 12.5.4: Declaration of ENTITY_MATRIX_PROVIDER class for the computation of the element matrix for the monolithic stabilized P1 Stokes FEM

```
2  class P1StabFEMElementMatrixProvider {
3   public:
4    using ElemMat = Eigen::Matrix<double, 9, 9>;
5    P1StabFEMElementMatrixProvider(const P1StabFEMElementMatrixProvider &) =
6        delete;
7    P1StabFEMElementMatrixProvider(P1StabFEMElementMatrixProvider &&) noexcept =
8        default;
9    P1StabFEMElementMatrixProvider &operator=(
10       const P1StabFEMElementMatrixProvider &) = delete;
```

```
  P1StabFEMElementMatrixProvider &operator=(P1StabFEMElementMatrixProvider &&) =
      delete;
  P1StabFEMElementMatrixProvider() = default;
  virtual ~P1StabFEMElementMatrixProvider() = default;
  [[nodiscard]] bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  [[nodiscard]] ElemMat Eval(const lf::mesh::Entity &cell);

 private:
  ElemMat MK_;
};
```

Write valid and appropriate C++ code in the blanks in the following code listing to complete this implementation of the `Eval()` method of the class **P1StabFEMElementMatrixProvider**.

```
P1StabFEMElementMatrixProvider::ElemMat P1StabFEMElementMatrixProvider::Eval(
    const lf::mesh::Entity& cell) {
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions and store them
  // in the columns of the 2×3-matrix G
  const Eigen::MatrixXd dpt = (Eigen::MatrixXd(2, 1) << 0.0, 0.0).finished();
  const Eigen::Matrix<double, 2, 3> G =
      cell.Geometry()->JacobianInverseGramian(dpt).block(0, 0, 2, 2) *
    (Eigen::Matrix<double, 2, 3>(2,3) << -1, 1, 0, -1, 0, 1).finished();
  // Compute the element matrix for −Δ and S_1^0.
  Eigen::Matrix<double, 3, 3> L = area * G.transpose() * G;

  MK_.setZero();
  // Arrays for local index remapping
  const std::array<Eigen::Index, 3> vx_idx{0, 3, 6};
  const std::array<Eigen::Index, 3> vy_idx{1, 4, 7};
  const std::array<Eigen::Index, 3> p_idx{2, 5, 8};

  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      MK_(vx_idx[i], vx_idx[j]) = MK_(vy_idx[i], vy_idx[j]) =
          [                                                ]
    }
  }
  // Fill entries related to the B-blocks of the element matrix
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      MK_(p_idx[i], vx_idx[j]) = MK_(vx_idx[j], p_idx[i]) =
          [                                                ];
      MK_(p_idx[i], vy_idx[j]) = MK_(vy_idx[j], p_idx[i]) =
          [                                                ];
    }
  }
  // Fill 3×3 block due to stabilization terms
  Eigen::Matrix3d SK;
  SK(0, 0) = [                                                ];
  SK(1, 1) = [                                                ];
  SK(2, 2) = [                                                ];
```

```
38   SK(0, 1) =                                    ;
39   SK(0, 2) =                                    ;
40   SK(2, 1) =                                    ;
41   for (int i = 0; i < 3; ++i) {
42     for (int j = 0; j < 3; ++j) {
43       MK_(p_idx[i], p_idx[j]) = -SK(i, j);
44     }
45   }
46   return MK_;
47 }
```

SOLUTION for (12-5.c) → 12-5-3-0:stss4.pdf ▲

(12-5.d) ⚄ (4 min.) Write $(\mathbf{v}, p) \in \left(H_0^1(\Omega)\right)^2 \times L^2(\Omega)$ for the solution of the Stokes boundary value problem

$$-\Delta \mathbf{v} - \mathbf{grad}\, p = \mathbf{f}\,, \quad \operatorname{div} \mathbf{v} = 0 \quad \text{in} \quad \Omega\,,$$
$$\int_\Omega p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad , \quad \mathbf{v} = 0 \quad \text{on} \quad \partial\Omega\,,$$

and $(\mathbf{v}_h, p_h) \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2 \times \mathcal{S}^0_1(\mathcal{M})$ for the solution of (12.5.1), for which we also assume vanishing mean of the pressure component.

The authors of [BH08] prove that

$$|\mathbf{v} - \mathbf{v}_h|_{H^1(\Omega)} + \|p - p_h\|_{L^2(\Omega)} \leq C \left(\inf_{\mathbf{w}_h \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2} |\mathbf{v} - \mathbf{w}_h|_{H^1(\Omega)} + \inf_{q_h \in \mathcal{S}^0_1(\mathcal{M})} \|p - q_h\|_{L^2(\Omega)} + h^2_{\mathcal{M}} \right)\,, \quad (12.5.12)$$

with $C > 0$ independent of $\mathbf{v}$, p, and the meshwidth $h_{\mathcal{M}}$ of $\mathcal{M}$ (only dependent on Ω and the shape-regularity of $\mathcal{M}$).

Based on this result and assuming $\mathbf{v} \in \left(H^2(\Omega)\right)^2$, $p \in H^2(\Omega)$, make a theoretical prediction for the asymptotic convergence of the following norms of the finite element discretization errors for $h_{\mathcal{M}} \to 0$ and for sequences of uniformly shape-regular meshes:

$$|\mathbf{v} - \mathbf{v}_h|_{H^1(\Omega)} = O\Big(\quad\Big) \quad \text{and} \quad \|p - p_h\|_{L^2(\Omega)} = O\Big(\quad\Big) \quad \text{for} \quad h_{\mathcal{M}} \to 0\,.$$

SOLUTION for (12-5.d) → 12-5-4-0:.pdf ▲

(12-5.e) ⚄ (4 min.) For the domain $\Omega =]0,1[^2$, the manufactured solution

$$\mathbf{v}(\boldsymbol{x}) = \begin{bmatrix} -\sin^2(\pi x_1)\sin(2\pi x_2) \\ \sin(2\pi x_1)\sin^2(\pi x_2) \end{bmatrix} \quad , \quad p(\boldsymbol{x}) = x_1 + x_2 - 1$$

and on sequences of triangular meshes generated by *uniform regular refinement*, the following table gives various norms of the discretization error for the stabilized P1-FEM for (12.5.1) as presented above.

```
N(d.o.f.)   L2err(v1)   L2err(v2)   H1err(v1)   H1err(v2)      L2err(p)
----------------------------------------------------------------------
```

39	0.282	0.219	2.47	2.07	0.316
120	0.0799	0.0769	1.29	1.2	0.302
417	0.0223	0.0216	0.681	0.641	0.143
1551	0.00575	0.00556	0.345	0.325	0.0494
5979	0.00145	0.0014	0.173	0.163	0.0159
23475	0.000362	0.00035	0.0864	0.0816	0.00514
93027	9.06e−05	8.76e−05	0.0432	0.0408	0.00171

Here `N(d.o.f.)` gives the total number of degrees of freedom, `L2err(v1)` is the $L^2(\Omega)$-norm of the discretization error for the x_1-component of the velocity, `L2err(v2)` that for the x_2-component, `H1err(v1)`, `H1err(v2)` are the $|\cdot|_{H^1(\Omega)}$-norms of the discretization error in the velocity components, and `L2err(p)` is the $L^2(\Omega)$-norm of the pressure error.

Describe qualitatively and quantitatively the empiric asymptotic convergence of the following error norms for $h_{\mathcal{M}} \to 0$:

$\|\mathbf{v} - \mathbf{v}_h\|_{L^2(\Omega)}$: ,

$|\mathbf{v} - \mathbf{v}_h|_{H^1(\Omega)}$: ,

$\|p - p_h\|_{L^2(\Omega)}$: .

SOLUTION for (12-5.e) → 12-5-5-0:ststs4.pdf ▲

References

[BH08] Roland Becker and Peter Hansbo. "A simple pressure stabilization method for the Stokes equation". In: *Comm. Numer. Methods Engrg.* 24.11 (2008), pp. 1421–1430. DOI: 10.1002/cnm.1041 (cit. on pp. 501, 504).

End Problem 12-5 , 25 min.

Chapter 13

Finite-Element Exterior Calculus (FEEC)

Problem 13-1: Incidence Matrices of a 2D Finite Element Mesh

In [Lecture → Section 13.2.1.2] we saw that the exterior derivative of cochains [Lecture → Def. 13.2.1.16] has a matrix representation through the (signed) **incidence matrices** of the (oriented) mesh. This problem is devoted to handling incidence matrices in LEHRFEM++ and it practices accessing topological information about a mesh in LEHRFEM++ and dealing with sparse matrices in EIGEN.

Implementation based on LEHRFEM++, background information is given in [Lecture → Section 2.7.2.2]. This problem is important as an introduction to the use of the LEHRFEM++ library, in particular concerning the data type **lf::mesh::Mesh** and **lf::mesh::Entity**.
▷ problem name/problem code folder: IncidenceMatricesFEEC

We are given an **oriented 2D mesh** $\mathcal{M}$ [Lecture → Def. 13.2.1.1] of a (polygonal) domain $\Omega \subset \mathbb{R}^2$ with triangular or quadrilateral cells forming the set of its 2-facets $\mathcal{F}_2(\mathcal{M})$. As we have learned in [Lecture → Section 2.7.2] this is the kind oh hybrid meshes LEHRFEM++ can represent internally and handle through **lf::mesh::Mesh** objects. As in [Lecture → (13.2.1.8)] we write $N_\ell := \sharp\mathcal{F}_\ell(\mathcal{M})$ for the number of ℓ-facets of $\mathcal{M}$, $\ell \in \{0,1,2\}$, nodes for $\ell = 0$, edges for $\ell = 1$, and cells for $\ell = 2$. Note that ℓ stands for the dimensions of a mesh entity, from which its co-dimension k, as frequently used in LEHRFEM++ can be computed via $k = 2 - \ell$.

We assume the ℓ-facets are numbered consecutively from 1 to N_ℓ. In LEHRFEM++ this numbering is given by the `Index` method of **lf::mesh::Mesh**, see [Lecture → Section 2.7.2.1].

In [Lecture → Section 2.7.2.1] we saw that in LEHRFEM++ a mesh is represented by objects of type **lf::mesh::Mesh** and that the member function `lf::mesh::Mesh::Index()` provides a consecutive numbering of all mesh entities of a specific (co-)dimension. We will indicate the number of a mesh entity by a subscript and write

$$\mathcal{F}_1(\mathcal{M}) = \{E_1,\ldots,E_{N_1}\}\,,\quad \mathcal{F}_0(\mathcal{M}) = \{\boldsymbol{p}_1,\ldots,\boldsymbol{p}_{N_0}\}\,,\quad \mathcal{F}_2(\mathcal{M}) = \{K_1,\ldots,K_{N_2}\}\,. \tag{13.1.1}$$

The `SubEntities()` member function of **lf::mesh::Entity** returns an array of sub-entities and, thus, fixes their ordering [Lecture → § 2.7.2.14]. In particular, this defines an ordering of the nodes spanning every ℓ-facet and, thus, its **inner orientation**, *cf.* [Lecture → § 13.1.2.1] and [Lecture → § 13.1.2.2]. The inner orientation of the mesh nodes themselves is set to $+1$.

The ordering of its nodes of an ℓ-facet f, $\ell = 1,2$, implies an **induced orientation** of its boundary ∂f [Lecture → § 13.1.4.25]; every subset of nodes can be endowed with an inherited order. We visualize this for cells with numbered nodes, with green arrows indicating the induced orientation of edges.

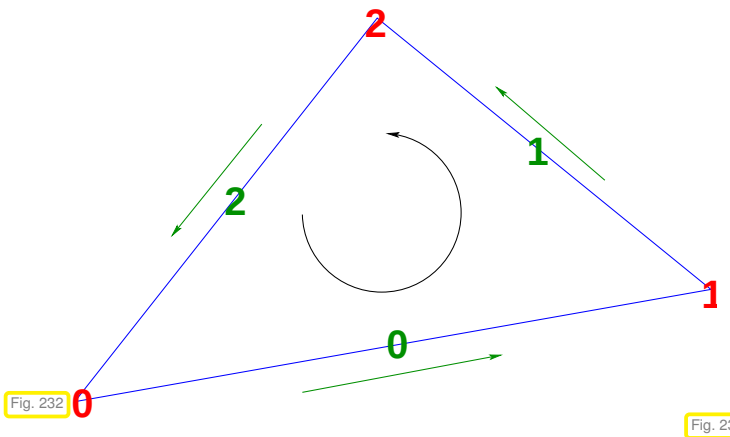


Fig. 232

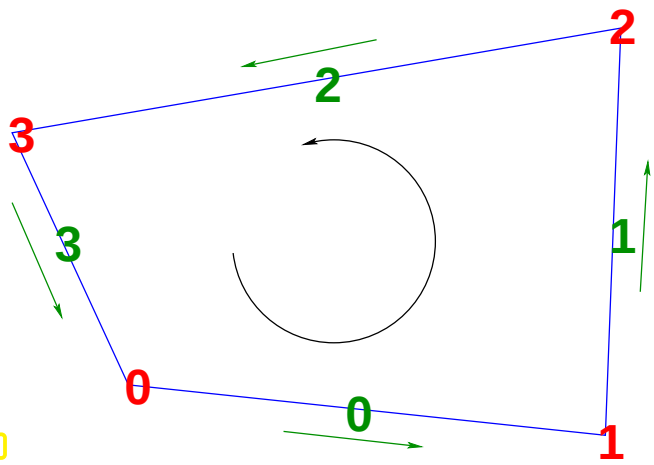


Fig. 233

This defines the **relative orientation** [Lecture → Def. 13.2.1.13] $o_r(E,K) \in \{-1,1\}$ of an edge E contained in the boundary of a cell K, $E \subset \overline{K}$. A special case are edges, for which we set

$$o_r(\boldsymbol{p},E) = \begin{cases} 1 & \text{, if node } \boldsymbol{p} \text{ is the first endpoint of edge } E\text{ ,} \\ -1 & \text{, if node } \boldsymbol{p} \text{ is the second endpoint of edge } i\text{ ,} \\ 0 & \text{, if node } \boldsymbol{p} \text{ is no endpoint of edge } i\text{ ,} \end{cases} \quad \boldsymbol{p} \in \mathcal{F}_0(\mathcal{M}),\ E \in \mathcal{F}_1(\mathcal{M})\ .$$

With (relative) orientations fixed and all ℓ-facets numbered, the two incidence matrices $\mathbf{D}_0 \in \{-1,0,+1\}^{N_1,N_0}$ (node-edge incidence matrix) and $\mathbf{D}_1 \in \{-1,0,1\}^{N_2,N_1}$ (face-cell incidence matrix) can be defined according to [Lecture → (13.2.1.18)], *cf.* [Lecture → Ex. 13.2.1.20].

(13-1.a) ⚀ (10 min.) Using the notations from (13.1.1) give formulas for the entries of $\mathbf{D}_0$ and $\mathbf{D}_1$.

SOLUTION for (13-1.a) → 13-1-1-0:icms0.pdf ▲

(13-1.b) ⚁ (15 min.) Explain why

- $\mathbf{D}_0$ is also called a **discrete gradient** (matrix), while
- $\mathbf{D}_1$ is referred to as **discrete divergence** (matrix).

SOLUTION for (13-1.b) → 13-1-2-0:incms0q.pdf ▲

(13-1.c) ⚀ (10 min.) What is the number of non-zero entries in each row of $\mathbf{D}_0$ and $\mathbf{D}_1$, respectively?

HIDDEN HINT 1 for (13-1.c) → 13-1-3-0:h1.pdf

SOLUTION for (13-1.c) → 13-1-3-1:s1.pdf ▲

(13-1.d) ⚀ (10 min.) [depends on Sub-problem (13-1.c)]

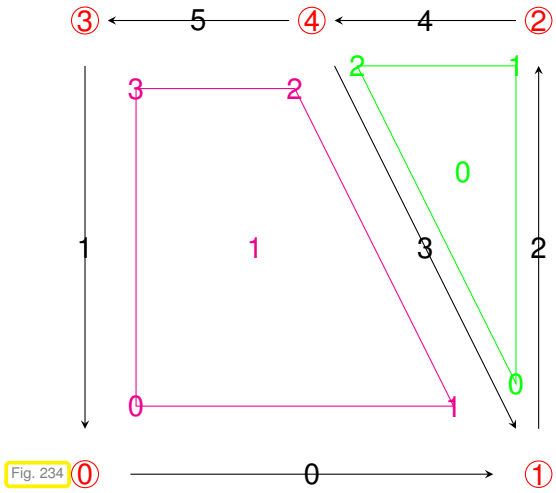
What is the topological meaning of the number of non-zero entries in the columns of $\mathbf{D}_0$ and $\mathbf{D}_1$, respectively?

SOLUTION for (13-1.d) → 13-1-4-0:s2.pdf ▲

(13-1.e) ⚁ (15 min.)

Compute the incidence matrices $\mathbf{D}_0$ and $\mathbf{D}_1$ for the mesh depicted in Fig. 234 beside.

In red $\hat{=}$ numbers of nodes, in black $\hat{=}$ numbers of edges, orientations of edges indicated by arrows, local numbers of corners of cells in magenta. Cell indices are printed inside the cell.



SOLUTION for (13-1.e) → 13-1-5-0:s2a.pdf ▲

In the following sub-problems we rely on EIGEN's sparse matrix data type **Eigen::SparseMatrix<T>**, see EIGEN documentation and [NumCSE course → Section 2.7.2]. It will be useful to know that triplet initialization of a sparse matrix can be skipped, if one knows the (maximum) number of non-zero entries per row in advance. This information can be passed to EIGEN through the member function `Eigen::SparseMatrix<T>::reserve()` as in [NumCSE course → Code 2.7.2.1]. The following code illustrates its use.

C++ code 13.1.3: `reserve()`-based initializatin of a sparse matrix in EIGEN ↬ GitLab

```
2  const int n = 20, m = 10;
3  // Set up zero sparse matrix with row major storage format
4  // This format is essential for being able to set the maximal
5  // number of non-zero entries per row.
6  Eigen::SparseMatrix<int, Eigen::RowMajor> X(n, m);
7  // Reserve space for at most nnz_row non-zero entries per row
8  const std::size_t nnz_row = 3;
9  X.reserve(Eigen::VectorXi::Constant(n, nnz_row));
10 // Initialize nnz_row entries per row
11 for (int row_idx = 0; row_idx < n; ++row_idx) {
12   for (int k = 0; k < nnz_row; ++k) {
13     X.coeffRef(row_idx, (row_idx * k) % m) += 1;
14   }
15 }
```

(13-1.f) ⚀ (30 min.) In the file `incidencematrices.cc` write a C++ function

```
Eigen::SparseMatrix<int> computeEdgeVertexIncidenceMatrix(
    const lf::mesh::Mesh &mesh);
```

which computes the sparse matrix $\mathbf{D}_0$ for the mesh passed in `mesh` efficiently.

The `Index()` method of **lf::mesh::Mesh** provides the numbering of entities which underlies indexing of the entries of the incidence matrices.

SOLUTION for (13-1.f) → 13-1-6-0:s3.pdf ▲

(13-1.g) ⚀ (30 min.) Now implement a C++ function

```
Eigen::SparseMatrix<int> computeCellEdgeIncidenceMatrix(
    const lf::mesh::Mesh &mesh);
```

which returns the cell-edge incidence matrix $\mathbf{D}_1$ for the mesh encoded in `mesh`. Again rely on the

numbering of mesh entities supplied by the `Index()` method of **lf::mesh::Mesh**.

HIDDEN HINT 1 for (13-1.g) → 13-1-7-0:h14.pdf

SOLUTION for (13-1.g) → 13-1-7-1:s4.pdf ▲

(13-1.h) ⚄ (20 min.) For the concrete case of the gradient matrix $\mathbf{D}_0$ and the divergence matrix $\mathbf{D}_1$ prove the following special case of [Lecture → Thm. 13.2.1.21] ($\mathbf{O}$ is the zero matrix).

Corollary 13.1.6. Cochain complex property

For any oriented 2D mesh holds

$$\mathbf{D}_1\mathbf{D}_0 = \mathbf{O}\ , \tag{13.1.7}$$

with $\mathbf{D}_0$/$\mathbf{D}_1$ the node-edge/face-cell (signed) incidence matrix of the mesh.

HIDDEN HINT 1 for (13-1.h) → 13-1-8-0:h5.pdf

SOLUTION for (13-1.h) → 13-1-8-1:s5.pdf ▲

(13-1.i) ⚄ (20 min.) [depends on Sub-problem (13-1.g), Sub-problem (13-1.f)]

Inspired by the statement of Cor. 13.1.6 and based on your functions `computeEdgeVertexIncidenceMatrix()` and `computeCellEdgeIncidenceMatrix()` implement in LEHRFEM++ a testing function.

```
bool testZeroIncidenceMatrixProduct(const lf::mesh::Mesh &mesh);
```

that returns **true**, whenever the two incidence matrices of the 2D hybrid mesh satisfy the relationship asserted in Cor. 13.1.6.

SOLUTION for (13-1.i) → 13-1-9-0:s5.pdf ▲

(13-1.j) ⚄ (30 min.) Define the **discrete Hodge-Laplacian** matrix

$$\mathbf{L} := \mathbf{D}_1^\top\mathbf{D}_1 + \mathbf{D}_0\mathbf{D}_0^\top \in \mathbb{R}^{N_1,N_1}\ . \tag{13.1.10}$$

Using elementary linear algebra show that

$$\mathcal{R}(\mathbf{D}_0) = \mathcal{N}(\mathbf{D}_1) \quad\Leftrightarrow\quad \mathcal{N}(\mathbf{L}) = \{\mathbf{0}\}\ . \tag{13.1.11}$$

Remark. [Lecture → Thm. 13.2.1.22], together with [Lecture → Thm. 13.1.4.81], asserts that $\mathcal{R}(\mathbf{D}_0) = \mathcal{N}(\mathbf{D}_1)$ hinges on *topological properties* of Ω, namely that *every* closed curve $\gamma \subset \Omega$ is the boundary of a subdomain $\Sigma \subset \Omega$, $\gamma = \partial\Sigma$. Now with the Hodge-Laplacian matrix we have a purely algebraic probe for this topological property. One just needs to inspect to kernel of a simple matrix of integers to find out about a rather "visual" property of Ω.

HIDDEN HINT 1 for (13-1.j) → 13-1-10-0:incmxhi.pdf

HIDDEN HINT 2 for (13-1.j) → 13-1-10-1:incmhx2.pdf

SOLUTION for (13-1.j) → 13-1-10-2:incmxs.pdf ▲

(13-1.k) ⚄ (30 min.) Next code an efficient C++ function

```
Eigen::SparseMatrix<int> computeHodgeLaplaceMatrix(const
    lf::mesh::Mesh &mesh);
```

which computes the integer matrix $\mathbf{L}$ as defined in (13.1.10) and returns it in EIGEN's sparse matrix format.

HIDDEN HINT 1 for (13-1.k) → 13-1-11-0:incmhy1.pdf

SOLUTION for (13-1.k) → 13-1-11-1:incmsy.pdf ▲

End Problem 13-1 , 220 min.

Problem 13-2: Mixed FEM for the 2D Hodge-Laplacian

The Hodge-Laplacian is a generalization of the Laplace operator for functions (= 0-forms) to differential forms of arbitrary degree. In this problem we study its finite element discretization based on a special mixed variational formulation from [AFW06]. We will rely on Whitney finite element spaces and examine all aspects ranging from the well-posedness of the continuous and discrete variational problems through details of the local computations. Various aspects of FEEC will be addressed.

For this project Knowledge about vector proxies, exterior calculus and Whitney finite element spaces and is essential; study [Lecture → § 13.1.4.107], [Lecture → § 13.1.4.49], and [Lecture → Section 13.2.2.3] befreohand. The second part of this project has a fcous on implementation in LEHRFEM++.
▷ problem name/problem code folder: HodgeLaplacian2D

Let $\Omega \subset \mathbb{R}^2$ be a bounded *connected* polygonal domain in 2D Euclidean space. Throughout, we use the following L^2-type inner products on spaces of differential forms (which are to be defined almost everywhere on Ω)

$$(\pi,\kappa)_{L^2\Lambda^0(\Omega)} := \int_\Omega \mathrm{v.p.}(\omega)(\boldsymbol{x})\ \mathrm{v.p.}(\eta)(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \qquad \forall \omega,\eta \in L^2\Lambda^0(\Omega)\,, \tag{13.2.1a}$$

$$(\omega,\eta)_{L^2\Lambda^1(\Omega)} := \int_\Omega \mathrm{v.p.}(\omega)(\boldsymbol{x})\cdot\mathrm{v.p.}(\eta)(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \qquad \forall \omega,\eta \in L^2\Lambda^1(\Omega)\,, \tag{13.2.1b}$$

$$(\rho,\sigma)_{L^2\Lambda^2(\Omega)} := \int_\Omega \mathrm{v.p.}(\rho)(\boldsymbol{x})\ \mathrm{v.p.}(\sigma)(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \qquad \forall \rho,\sigma \in L^2\Lambda^2(\Omega)\,. \tag{13.2.1c}$$

Here $\mathrm{v.p.}(\cdot)$ denotes 2D Euclidean vector proxies as introduced in [Lecture → § 13.1.3.20]. Recall from [Lecture → (13.1.3.21)] that there are two conventions (Options I & II) for defining proxy vectorfields for differential 1-forms in 2D. However, in (13.2.1b) both will induce the same inner product on $L^2\Lambda^1(\Omega)$, because the respective proxy vectorfields just differ by a rotation by $\frac{\pi}{2}$.

Choice of proxy vectorfields

Throughout this project we use "Option I" proxy vectorfields for 1-forms on Ω, see [Lecture → (13.1.3.21)]. Vector proxies for 0-forms and 2-forms follow the conventions presented in [Lecture → § 13.1.3.20].

The central object in this project is the following variational problem (for notations see [Lecture → Def. 13.1.5.2]), a boundary value problem for the **Hodge-Laplacian** operator in weak form.

Seek $\omega \in H\Lambda^1(\mathrm{d},\Omega)$ and $\pi \in H\Lambda^0(\mathrm{d},\Omega)$ such that

$$\begin{aligned} (\mathrm{d}_1\omega,\mathrm{d}_1\omega')_{L^2\Lambda^2(\Omega)} + (\omega',\mathrm{d}_0\pi)_{L^2\Lambda^1(\Omega)} &= (\varphi,\omega')_{L^2\Lambda^1(\Omega)} && \forall \omega' \in H\Lambda^1(\mathrm{d},\Omega)\,,\\ (\omega,\mathrm{d}_0\pi')_{L^2\Lambda^1(\Omega)} - (\pi,\pi')_{L^2\Lambda^0(\Omega)} &= 0 && \forall \pi' \in H\Lambda^0(\mathrm{d},\Omega)\,. \end{aligned} \tag{13.2.3}$$

In (13.2.3) φ is a given 1-form $\in L^2\Lambda^1(\Omega)$.

(13-2.a) ⊡ (10 min.) What does "$\omega \in C^0\Lambda^\ell(\Omega)$", $\ell \in \{0,1,2\}$, mean?

SOLUTION for (13-2.a) → 13-2-1-0:hlns.pdf ▲

(13-2.b) ⚄ (20 min.) To become more familiar with 2D Euclidean vector proxies for 1-forms, answer the following question:

What is the vector proxy incarnation of the **exterior product** $\wedge : C^0\Lambda^1(\Omega)\times C^0\Lambda^1(\Omega) \to C^0\Lambda^2(\Omega)$

according to [Lecture → Def. 13.1.4.102], when using the following "vector proxy mapping conventions" for ℓ-forms, *cf.* [Lecture → § 13.1.3.20]:

- $\ell = 0$: $u := \mathrm{v.p.}(\omega) : \Omega \to \mathbb{R}$ such that $\omega(\boldsymbol{x}) = u(\boldsymbol{x})$, $\omega \in C^0\Lambda^0(\Omega)$, $\boldsymbol{x} \in \Omega$,
- $\ell = 1$: $\mathbf{u} := \mathrm{v.p.}(\omega) : \Omega \to \mathbb{R}^2$ such that $\omega(\boldsymbol{x})(\mathbf{v}) = \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{v}$, $\mathbf{v} \in \mathbb{R}^2$, $\boldsymbol{x} \in \Omega$, $\omega \in C^0\Lambda^1(\Omega)$ (Option I in [Lecture → (13.1.3.21)]),
- $\ell = 2$, $q := \mathrm{v.p.}(\omega) : \Omega \to \mathbb{R}$ such that $\omega(\boldsymbol{x})(\mathbf{v}, \mathbf{w}) = q(\boldsymbol{x}) \det(\mathbf{v}\,\mathbf{w})$, $\mathbf{v}, \mathbf{w} \in \mathbb{R}^2$, $\boldsymbol{x} \in \Omega$, $\omega \in C^0\Lambda^2(\Omega)$ [Lecture → (13.1.3.22)],

HIDDEN HINT 1 for (13-2.b) → 13-2-2-0:hlhm1.pdf

HIDDEN HINT 2 for (13-2.b) → 13-2-2-1:hlmh2.pdf

SOLUTION for (13-2.b) → 13-2-2-2:hlms.pdf ▲

(13-2.c) ⚀ (10 min.) Parallel to what was done in [Lecture → § 13.1.5.7], define the spaces $H\Lambda^\ell(\mathrm{d}, \Omega)$ in terms of 2D Euclidean vector proxies.

HIDDEN HINT 1 for (13-2.c) → 13-2-3-0:hl0sh.pdf

SOLUTION for (13-2.c) → 13-2-3-1:hls0.pdf ▲

(13-2.d) ⚁ (15 min.) Let $\gamma \subset \Omega$ be a curve given by the parameterization $\gamma : [0,1] \to \Omega$. For $\omega \in C^0\Lambda^1(\Omega)$ express the integral $\int_\gamma \omega$ in terms of the 2D Euclidean vector proxy $\mathrm{v.p.}(\omega)$ of ω.

SOLUTION for (13-2.d) → 13-2-4-0:hl1s.pdf ▲

(13-2.e) ⚀ (10 min.) Define $H_0\Lambda^1(\mathrm{d}, \Omega)$ in terms of 2D Euclidean vector proxies.

SOLUTION for (13-2.e) → 13-2-5-0:hls23.pdf ▲

(13-2.f) ⚅ (20 min.) [depends on Sub-problem (13-2.b) and Sub-problem (13-2.d)]

The general **integration by parts** formula

$$\int_{\partial\Gamma} \omega \wedge \eta = \int_\Gamma \mathrm{d}_\ell\omega \wedge \eta + (-1)^\ell \omega \wedge \mathrm{d}_k\eta \quad \forall \omega \in C^1\Lambda^\ell(\Omega),\ \eta \in C^1\Lambda^k(\Omega)\ ,$$

([Lecture → (13.1.4.124)])

can be specialized to ($\ell = 1, k = 0$)

$$\int_{\partial\Omega} \omega \wedge \eta = \int_\Omega \mathrm{d}_1\omega \wedge \eta - \omega \wedge \mathrm{d}_0\eta \quad \forall \omega \in C^1\Lambda^1(\Omega),\ \eta \in C^0\Lambda^0(\Omega)\ . \tag{13.2.12}$$

Express (13.2.12) in terms of vector proxies $\mathbf{u} := \mathrm{v.p.}(\omega)$ and $p := \mathrm{v.p.}(\eta)$.

SOLUTION for (13-2.f) → 13-2-6-0:hls4.pdf ▲

(13-2.g) ⚀ (10 min.) State the variational problem (13.2.3) in terms of 2D Euclidean vector proxies $\mathbf{u} := \mathrm{v.p.}(\omega)$, $p := \mathrm{v.p.}(\pi)$, $\mathbf{f} := \mathrm{v.p.}(\varphi)$.

SOLUTION for (13-2.g) → 13-2-7-0:hls4.pdf ▲

(13-2.h) ⚅ (30 min.) [depends on Sub-problem (13-2.f) and Sub-problem (13-2.g)]

State the boundary value problem encoded by (13.2.3) in *strong/PDE form* and for vector proxies $\mathbf{u} := \mathrm{v.p.}(\omega)$, $p := \mathrm{v.p.}(\pi)$, $\mathbf{f} := \mathrm{v.p.}(\varphi)$. Existence and uniqueness of solutions of (13.2.3) and sufficient smoothness of $\mathbf{u}$ and p can be taken for granted.

HIDDEN HINT 1 for (13-2.h) → 13-2-8-0:hls5h.pdf

SOLUTION for (13-2.h) → 13-2-8-1:hl5s.pdf ▲

(13-2.i) ⚂ (15 min.) [depends on Sub-problem (13-2.h)]

Let $\mathbf{u} := \mathrm{v.p.}(\omega) \in \left(C^2(\overline{\Omega})\right)^2$, $p := \mathrm{v.p.}(\eta) \in C^1(\overline{\Omega})$ be the proxy vectorfields for a solution (ω, η) of (13.2.3). Besides set $\mathbf{f} := \mathrm{v.p.}(\varphi)$. Show that

$$-\mathbf{\Delta u} = \mathbf{f} \quad \text{in} \quad \Omega\,, \tag{13.2.18}$$

where $\mathbf{\Delta}$ designates the **vectorial Laplacian**, that is, the standard scalar Laplacian Δ applied separately to the components of a vectorfield.

SOLUTION for (13-2.i) → 13-2-9-0:hls6x.pdf ▲

Given Hilbert spaces U and Q, *continuous* bilinear forms $\mathsf{a} : U \times U \to \mathbb{R}$, $\mathsf{b} : U \times Q \to \mathbb{R}$, and $\mathsf{c} : Q \times Q \to \mathbb{R}$, the following is a **generalized variational saddle point problem (GSPP)** (with *continuous* right hand side linear functionals $\ell : U \to \mathbb{R}$, $g : Q \to \mathbb{R}$):

$$\text{Seek} \begin{array}{l} v \in U \\ p \in Q \end{array} : \begin{array}{rcll} \mathsf{a}(v,w) & + & \mathsf{b}(w,p) & = \ell(w) \quad \forall w \in U\,, \\ \mathsf{b}(v,q) & - & \mathsf{c}(p,q) & = g(q) \quad \forall q \in Q\,, \end{array} \tag{13.2.19}$$

(13-2.j) ⚀ (10 min.) Fit the concrete linear variational problem (13.2.3) for the Hodge-Laplacian into the class of generalized variational saddle point problems by identifying the concrete spaces, bilinear forms and linear forms.

SOLUTION for (13-2.j) → 13-2-10-0:hl7s.pdf ▲

In [BBF13, Sect. 4.3.1] we find the following theoretical result, which is an extension of [Lecture → Thm. 12.2.2.23]. Again we write

$$\mathcal{N}(\mathsf{b}) := \{v \in U : \; b(v,q) = 0 \; \forall q \in Q\} \subset U\,, \qquad \text{[Lecture → (13.3.3.38)]}$$

and also need the "kernel of the adjoint of b"

$$\mathcal{N}^*(\mathsf{b}) := \{q \in Q : \; b(v,q) = 0 \; \forall v \in U\} \subset Q\,. \tag{13.2.20}$$

Theorem 13.2.21. LBB-conditions for GSPPs

If the following four conditions are satisfied,

(i) the bilinear form a *is* $\mathcal{N}(\mathsf{b})$*-elliptic (**ellipticity on the kernel**)*

$$\exists \alpha > 0: \quad |\mathsf{a}(v,v)| \geq \alpha \|v\|_U^2 \quad \forall v \in \mathcal{N}(\mathsf{b}) \,, \tag{LBB1}$$

(ii) the bilinear form b *satisfies the **inf-sup condition***

$$\exists \beta > 0: \quad \sup_{v \in U} \frac{|\mathsf{b}(v,p)|}{\|v\|_U} \geq \beta \inf_{q \in \mathcal{N}^*(\mathsf{b})} \|q - p\|_Q \quad \forall p \in Q \,, \tag{LBB2}$$

(iii) the bilinear form c *is symmetric and positive semidefinite* [*Lecture* → *Def. 1.2.3.24*]*,*

$$\mathsf{c}(q,q) \geq 0 \quad \forall q \in Q \,, \tag{13.2.22}$$

(iv) and satisfies the ellipticity estimate

$$\exists \gamma > 0: \quad |\mathsf{c}(q,q)| \geq \gamma \|q\|_Q^2 \quad \forall q \in \mathcal{N}^*(\mathsf{b}) \,, \tag{13.2.23}$$

then the generalized variational saddle point problem (13.2.19) *possesses a unique solution* $(v,p) \in U \times Q$*, which satisfies*

$$\|v\|_U + \|p\|_Q \leq C \left(\sup_{w \in U} \frac{|\ell(w)|}{\|w\|_U} + \sup_{q \in Q} \frac{|g(q)|}{\|q\|_Q} \right) \,, \tag{13.2.24}$$

with $C > 0$ *independent of* ℓ *and* g*.*

In the next few sub-problems we verify the assumptions of this theorem for the Hodge-Laplacian generalized variational saddle point problem (13.2.3).

(13-2.k) ⚀ (10 min.) What is $\mathcal{N}^*(\mathsf{b})$ in the case of the Hodge-Laplacian generalized variational saddle point problem (13.2.3).

HIDDEN HINT 1 for (13-2.k) → `13-2-11-0:hlh8.pdf`

SOLUTION for (13-2.k) → `13-2-11-1:hls7a.pdf` ▲

(13-2.l) ⚅ (15 min.) [depends on Sub-problem (13-2.k)]

Confirm that (LBB2) holds true for the Hodge-Laplacian generalized variational problem (13.2.3).

HIDDEN HINT 1 for (13-2.l) → `13-2-12-0:hlh7ah.pdf`

HIDDEN HINT 2 for (13-2.l) → `13-2-12-1:hlh7a2.pdf`

SOLUTION for (13-2.l) → `13-2-12-2:hls8s.pdf` ▲

(13-2.m) ⚀ (10 min.) [depends on Sub-problem (13-2.k)]

Explain why (13.2.22) and (13.2.23) are satisfied for the Hodge-Laplacian variational saddle point problem (13.2.3).

SOLUTION for (13-2.m) → `13-2-13-0:hls9s.pdf` ▲

The most challenging assumption of Thm. 13.2.21 is (LBB1). Indeed, it will not hold in general, but hinges on topological properties of Ω. We encountered a similar situation with magentostatic boundary value problems as presented in [Lecture → Section 13.3.3]. Therefore, from now we assume simple topology.

Assumption 13.2.29. Simple topology of Ω

We assume that every closed curve $\subset \Omega$ is the boundary of a subdomain of Ω.

According to [Lecture → Thm. 13.1.4.81] (and its extension to Sobolev spaces of differential forms) under this assumption

$$\left\{\omega \in H\Lambda^1(\mathsf{d},\Omega) : \ \mathsf{d}_1\omega = 0\right\} = \mathsf{d}_0 H\Lambda^0(\mathsf{d},\Omega) \ , \tag{13.2.30}$$

expressed through vector proxies

$$\{\mathbf{u} \in \boldsymbol{H}(\mathrm{curl}_{2D},\Omega) : \ \mathrm{curl}_{2D}\,\mathbf{u} = 0\} = \mathbf{grad}\, H^1(\Omega) \ . \tag{13.2.31}$$

(13-2.n) ⚂ (10 min.) Sketch a simple bounded domain $\Omega \subset \mathbb{R}^2$ for which Ass. 13.2.29 is violated.

HIDDEN HINT 1 for (13-2.n) → 13-2-14-0:HStoph.pdf

SOLUTION for (13-2.n) → 13-2-14-1:anntopcv.pdf ▲

(13-2.o) ⚂ (10 min.) Show that the bilinear form

$$\mathsf{a} : H\Lambda^1(\mathsf{d},\Omega) \times H\Lambda^1(\mathsf{d},\Omega) \to \mathbb{R} \quad , \quad \mathsf{a}(\omega,\omega') := \left(\mathsf{d}_1\omega, \mathsf{d}_1\omega'\right)_{L^2\Lambda^2(\Omega)} \ ,$$

is symmetric *positive definite* according to [Lecture → Def. 1.2.3.27] on $\mathcal{N}(\mathsf{b})$, if Ass. 13.2.29 holds true.

SOLUTION for (13-2.o) → 13-2-15-0:hlsA.pdf ▲

The Galerkin discretization of a generalized variational saddle point problem (13.2.19) follows standard lines and starts with replacing the spaces U and Q with finite-dimensional subspaces $U_h \subset U$ and $Q_h \subset Q$, respectively.

$$\text{Seek} \quad \begin{matrix} v_h \in U_h \\ p_h \in Q_h \end{matrix} : \quad \begin{matrix} \mathsf{a}(v_h,w_h) & + & \mathsf{b}(w_h,p_h) & = & \ell(w_h) & \forall w_h \in U_h \ , \\ \mathsf{b}(v_h,q_h) & - & \mathsf{c}(p_h,q_h) & = & g(q_h) & \forall q_h \in Q_h \ , \end{matrix} \tag{13.2.33}$$

An extension of [Lecture → Thm. 13.3.3.41] remains valid. Analoguous to [Lecture → (13.3.3.40)] and (13.2.20) we define

$$\begin{aligned} \mathcal{N}_h(\mathsf{b}) &:= \{v_h \in U_h : \ b(v_h,q_h) = 0 \ \forall q_h \in Q_h\} \subset U_h \ , && [\text{Lecture} \to (13.3.3.40)] \\ \mathcal{N}_h^*(\mathsf{b}) &:= \{q_h \in Q_h : \ b(v_h,q_h) = 0 \ \forall v_h \in U_h\} \subset U_h \ . && (13.2.34) \end{aligned}$$

Theorem 13.2.35. Galerkin discretization error estimate for generalized discrete variational saddle point problems

We consider a generalized variational saddle point problem (13.2.19) *complying with the assumptions of Thm. 13.2.21. If*

$$\exists \alpha_h > 0: \quad |\mathsf{a}(v_h, v_h)| \geq \alpha_h \|v_h\|_U^2 \quad \forall v_h \in \mathcal{N}_h(\mathsf{b}) \,, \tag{LBB1$_h$}$$

$$\exists \beta_h > 0: \quad \sup_{v_h \in U_h} \frac{|\mathsf{b}(v_h, q_h)|}{\|v\|_U} \geq \beta_h \|q_h\|_Q \quad \forall q_h \in Q_h \,, \tag{LBB2$_h$}$$

$$\mathsf{c}(q_h, q_h) \geq 0 \quad \forall q_h \in Q_h \,, \tag{13.2.36}$$

$$\exists \gamma_h > 0: \quad |\mathsf{c}(q_h, q_h)| \geq \gamma_h \|q_h\|_Q^2 \quad \forall q_h \in \mathcal{N}_h^*(\mathsf{b}) \,, \tag{13.2.37}$$

then the discrete variational saddle point problem (13.2.33) *possesses a unique solution* $(v_h, p_h) \in U_h \times Q_h$*, which satisfies the Galerkin discretization error estimate*

$$\|v - v_h\|_U + \|p - p_h\|_Q \leq C \left(\inf_{u_h \in U_h} \|v - u_h\|_U + \inf_{q_h \in Q_h} \|p - q_h\|_Q \right) \,, \tag{13.2.38}$$

where $(v, p) \in U \times Q$ *solves* (13.2.19) *and with a constant* $C > 0$ *depending only on the norms of the bilinear forms and* α_h*,* β_h*, and* γ_h*.*

For Step I of the finite element Galerkin discretization of (13.2.3) we employ suitable Whitney finite element spaces [Lecture → Section 13.2.2.3].

Given an *oriented* triangular mesh $\mathcal{M}$ of Ω [Lecture → Def. 13.2.1.1] we use as trial/test spaces

$$\mathcal{W}^1(\mathcal{M}) \subset H\Lambda^1(\mathsf{d}, \Omega) \; [\sim \boldsymbol{H}(\mathrm{curl}_{2D}, \Omega)] \quad , \quad \mathcal{W}^0(\mathcal{M}) \subset H\Lambda^0(\mathsf{d}, \Omega) \; [\sim H^1(\Omega))] \,.$$

This results in a discrete variational problem: Seek $\omega_h \in \mathcal{W}^1(\mathcal{M})$ and $\pi_h \in \mathcal{W}^0(\mathcal{M})$ such that

$$\begin{array}{rcccll} (\mathsf{d}_1 \omega_h, \mathsf{d}_1 \omega_h')_{L^2\Lambda^2(\Omega)} & + & (\omega_h', \mathsf{d}_0 \pi_h)_{L^2\Lambda^1(\Omega)} & = & (\varphi, \omega_h')_{L^2\Lambda^1(\Omega)} & \forall \omega_h' \in \mathcal{W}^1(\mathcal{M}) \,, \\ (\omega_h, \mathsf{d}_0 \pi_h')_{L^2\Lambda^1(\Omega)} & - & (\pi_h, \pi_h')_{L^2\Lambda^0(\Omega)} & = & 0 & \forall \pi_h' \in \mathcal{W}^0(\mathcal{M}) \,. \end{array} \tag{13.2.39}$$

In Step II of the finite element Galerkin discretization we equip

- $\mathcal{W}^0(\mathcal{M})$ with the standard basis $\left\{\beta_p^0\right\}_{p \in \mathcal{F}_0(\mathcal{M})}$ of *node-associated* tent functions, and
- $\mathcal{W}^1(\mathcal{M})$ with the basis $\left\{\beta_e^1\right\}_{e \in \mathcal{F}_1(\mathcal{M})}$ of *edge-associated* Whitney 1-forms [Lecture → Fig. 695].

An ordering of the basis forms based on a numbering of nodes and edges of $\mathcal{M}$,

$$\mathcal{F}_0(\mathcal{M}) = \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_M\} \,, \quad M := \sharp \mathcal{F}_0(\mathcal{M}) \quad , \quad \mathcal{F}_1(\mathcal{M}) = \{e_1, \ldots, e_N\} \,, \quad N := \sharp \mathcal{F}_1(\mathcal{M}) \,,$$

is assumed. Thus (13.2.39) can be converted into an equivalent linear system of equations

$$\begin{array}{rcccl} \mathbf{A}\vec{\omega} & + & \mathbf{B}^\top \vec{\pi} & = & \vec{\varphi} \,, \\ \mathbf{B}\vec{\omega} & - & \mathbf{C}\vec{\pi} & = & 0 \,, \end{array} \quad \Leftrightarrow \quad \begin{bmatrix} \mathbf{A} & \mathbf{B}^\top \\ \mathbf{B} & -\mathbf{C} \end{bmatrix} \begin{bmatrix} \vec{\omega} \\ \vec{\pi} \end{bmatrix} = \begin{bmatrix} \vec{\varphi} \\ 0 \end{bmatrix}$$

for the basis expansion coefficient vectors $\vec{\omega}$ and $\vec{\pi}$ of ω_h and π_h, respectively.

(13-2.p) ⚀ (10 min.) Give formulas for the entries of the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ in Problem 13-2.

SOLUTION for (13-2.p) → `13-2-16-0:hlSB.pdf` ▲

(13-2.q) ⚂ (15 min.) [depends on Sub-problem (13-2.p)]

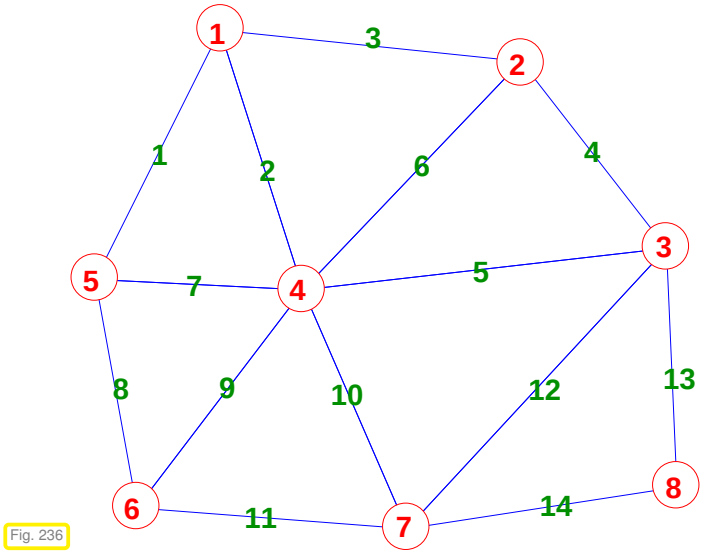


Fig. 236

◁ For the triangular mesh drawn beside the nodes are numbered in red, the edges in green.

Sketch the pattern of possible non-zero entries of the matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ from Problem 13-2 for this small triangular mesh (Indicate non-zero matrix entries with $*$).

SOLUTION for (13-2.q) → 13-2-17-0:.pdf ▲

We want to implement the finite element Galerkin discretization of (13.2.3) described before in LEHRFEM++. We opt for a **monolithic approach** that is, the full matrix of the linear system of equations Problem 13-2 is stored in one big sparse matrix.

(13-2.r) ⚂ To represent the local→global mapping for the shape functions of the envisaged finite element method, we rely on LEHRFEM++'s **lf::assemble::UniformFEDofHandler** class. How does a suitable object of that class have to be initialized?

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(),      },
  {lf::base::RefEl::kSegment(),           },
  {lf::base::RefEl::kTria(),              },
  {lf::base::RefEl::kQuad(),              });
```

SOLUTION for (13-2.r) → 13-2-18-0:HLsD.pdf ▲

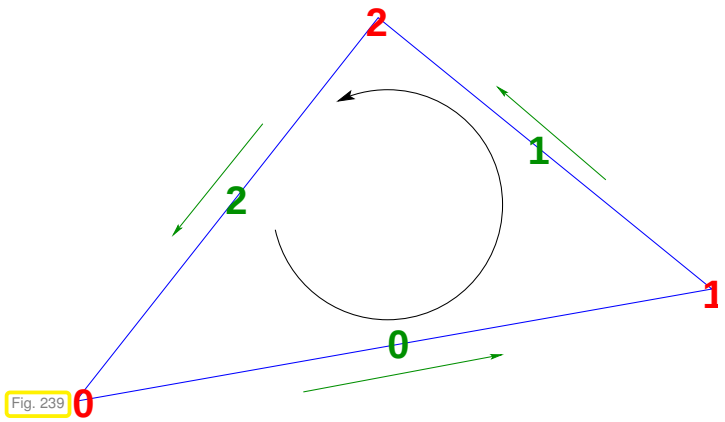


Fig. 239

We now examine the **local shape functions** on a triangle $K \in \mathcal{M}$. We rely on the local numbering of vertices and edges of K used in LEHRFEM++ and also on the induced orientation of the edges, which is the orientation used for the definition of local Whitney 1-forms.

For the classical lowest-order Lagrangian finite element space $\mathcal{W}^0(\mathcal{M}) = \mathcal{S}_1^0(\mathcal{M})$ the local shape functions coincide with the barycentric coordinate functions λ_1, λ_2, and λ_3 of K [Lecture → § 2.4.5.2].

The local shape function for $\mathcal{W}^1(\mathcal{M})$ were constructed in [Lecture → § 13.2.2.15]:

$$\begin{aligned} \beta^1_{1,K} &= \lambda_1 \mathrm{d}_0\lambda_2 - \lambda_2 \mathrm{d}_0\lambda_1 \quad [\,\check{\beta}^1_{1,K}(\boldsymbol{x}) = \lambda_1(\boldsymbol{x})\,\mathbf{grad}\,\lambda_2(\boldsymbol{x}) - \lambda_2(\boldsymbol{x})\,\mathbf{grad}\,\lambda_1(\boldsymbol{x})\,]\,, \\ \beta^1_{2,K} &= \lambda_2 \mathrm{d}_0\lambda_3 - \lambda_3 \mathrm{d}_0\lambda_2 \quad [\,\check{\beta}^1_{2,K}(\boldsymbol{x}) = \lambda_2(\boldsymbol{x})\,\mathbf{grad}\,\lambda_3(\boldsymbol{x}) - \lambda_3(\boldsymbol{x})\,\mathbf{grad}\,\lambda_2(\boldsymbol{x})\,]\,, \\ \beta^1_{3,K} &= \lambda_3 \mathrm{d}_0\lambda_1 - \lambda_1 \mathrm{d}_0\lambda_3 \quad [\,\check{\beta}^1_{3,K}(\boldsymbol{x}) = \lambda_3(\boldsymbol{x})\,\mathbf{grad}\,\lambda_1(\boldsymbol{x}) - \lambda_1(\boldsymbol{x})\,\mathbf{grad}\,\lambda_3(\boldsymbol{x})\,]\,. \end{aligned} \qquad \text{(13.2.40)}$$

(in 2D Euclidean vector proxies)

(13-2.s) ⚃ (15 min.) For a general triangle $K \in \mathcal{M}$ compute the function proxy of $\mathrm{d}_2\beta^1_{i,K}$, $i = 1,2,3$.

HIDDEN HINT 1 for (13-2.s) → 13-2-19-0:HLEh1.pdf

SOLUTION for (13-2.s) → 13-2-19-1:HLsE.pdf ▲

(13-2.t) ⚂ (10 min.) [depends on Sub-problem (13-2.s)]

For a triangle $K \in \mathcal{M}$ compute the matrix

$$\mathbf{A}_K := \left[\left(\mathrm{d}_1\beta^1_{j,K}, \mathrm{d}_1\beta^1_{i,K}\right)_{L^2\Lambda^2(K)}\right]_{i,j=1}^3 = \left[\int_K \mathrm{curl}_{2D}\,\check{\beta}_{j,K}(\boldsymbol{x})\;\mathrm{curl}_{2D}\,\check{\beta}^1_{i,K}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^3 \in \mathbb{R}^{3,3}\,. \qquad \text{(13.2.43)}$$

SOLUTION for (13-2.t) → 13-2-20-0:hlE1s.pdf ▲

(13-2.u) ⚄ (15 min.) Let $\mathbf{L}_K$ be the $\mathcal{S}^0_1(\mathcal{M})$ element matrix for a triangle $K \in \mathcal{M}$ and the Laplacian, that is [Lecture → § 2.4.5.5]

$$\mathbf{L}_K := \left[\int_K \mathbf{grad}\,\lambda_i(\boldsymbol{x}) \cdot \mathbf{grad}\,\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^3 \in \mathbb{R}^{3,3}\,. \qquad \text{(13.2.45)}$$

Express the matrix

$$\mathbf{B}_K := \left[\int_K \check{\beta}^1_{j,K}(\boldsymbol{x}) \cdot \mathbf{grad}\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^3 \in \mathbb{R}^3 \qquad \text{(13.2.46)}$$

in terms of entries of $\mathbf{L}_K$.

SOLUTION for (13-2.u) → 13-2-21-0:.pdf ▲

(13-2.v) ⚃ (15 min.) For an arbitrary triangle compute the entries of the $\mathcal{S}^0_1(\mathcal{M})$ "element mass matrix"

$$\mathbf{C}_K := \int_K \lambda_i(\boldsymbol{x})\,\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad i,j = 1,2,3\,. \qquad \text{(13.2.48)}$$

HIDDEN HINT 1 for (13-2.v) → 13-2-22-0:LFE11.pdf

SOLUTION for (13-2.v) → 13-2-22-1:.pdf ▲

Let $\mathbf{M}_K \in \mathbb{R}^{6,6}$, $K \in \mathcal{M}$ a triangle, denote the "monolithic" $\mathcal{W}^0(\mathcal{M}) \times \mathcal{W}^1(\mathcal{M})$ **element matrix** [Lecture → Def. 2.7.4.5] for the finite element Galerkin scheme for (13.2.3) to be implemented in this project. The next goal is to realize a class matching the concept of an **ENTITY_MATRIX_PROVIDER** ➙ (documentation) that can supply $\mathbf{M}_K$. The declaration of this class is listed next.

C++ code 13.2.51: Declaration of ENTITY_MATRIX_PROVIDER class for the computation of the element matrix for the monolithic Whitney FEM discretization of the Hodge-Laplacian variational problem ⇢ GitLab

```
class HodgeLaplacian2DElementMatrixProvider {
 public:
  // The size of the element matrix is 6×6.
  using ElemMat = Eigen::Matrix<double, 6, 6>;
  HodgeLaplacian2DElementMatrixProvider(
      const HodgeLaplacian2DElementMatrixProvider &) = delete;
  HodgeLaplacian2DElementMatrixProvider(
      HodgeLaplacian2DElementMatrixProvider &&) noexcept = default;
  HodgeLaplacian2DElementMatrixProvider &operator=(
      const HodgeLaplacian2DElementMatrixProvider &) = delete;
  HodgeLaplacian2DElementMatrixProvider &operator=(
      HodgeLaplacian2DElementMatrixProvider &&) = delete;
  HodgeLaplacian2DElementMatrixProvider() = default;
  virtual ~HodgeLaplacian2DElementMatrixProvider() = default;
  [[nodiscard]] bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  // Computation of element matrix M_K:  two versions
  [[nodiscard]] ElemMat Eval(const lf::mesh::Entity &cell);
  // Alternative implementation created for debugging purposes
  [[nodiscard]] ElemMat Eval_ref(const lf::mesh::Entity &cell);

 private:
  ElemMat MK_;
};
```

Element matrices invariably rely on an agreed numbering of local shape functions.

Numbering of local shape functions

The ordering of the local shape functions for the $\mathcal{W}^1(\mathcal{M}) \times \mathcal{W}^0(\mathcal{M})$ finite element method for (13.2.3) follows LEHRFEM++'s numbering convention as put forth in [Lecture → § 2.7.4.11].

(13-2.w) ⚄ (15 min.) According to

Definition [Lecture → Def. 2.7.4.5]. Element (stiffness) matrix and element (load) vector

Given a mesh entity $K \in \mathcal{M}$ and local shape functions $\{b_K^1, \ldots, b_K^Q\}$, $Q = Q(K) \in \mathbb{N}$, we introduce

$$\text{the element (stiffness) matrix} \quad \mathbf{A}_K := \left[\mathsf{a}_K(b_K^j, b_K^i)\right]_{i,j=1}^{Q} \in \mathbb{R}^{Q,Q},$$

$$\text{and the element (load) vector} \quad \vec{\boldsymbol{\varphi}}_K := \left[\ell_K(b_K^i)\right]_{i=1}^{Q} \in \mathbb{R}^{Q}.$$

an element matrix is based on a local(ized) bilinear form a_K and an ordered set of local shape functions.

What is that local bilinear form and ordered set of local shape functions underlying the element matrix $\mathbf{M}_K$?

SOLUTION for (13-2.w) → 13-2-23-0:HLGAs.pdf ▲

(13-2.x) ⚄ (10 min.) [depends on Sub-problem (13-2.t), Sub-problem (13-2.u), Sub-problem (13-2.v),

Sub-problem (13-2.w)]

Write the element matrix $\mathbf{M}_K$ as 2×2 block matrix with 3×3 blocks derived from $\mathbf{A}_K$, $\mathbf{B}_K$, and $\mathbf{C}_K$.

SOLUTION for (13-2.x) → 13-2-24-0:HLsG1.pdf ▲

(13-2.y) ⚄ (30 min.) [depends on Sub-problem (13-2.t), Sub-problem (13-2.u), Sub-problem (13-2.v), Sub-problem (13-2.w)]

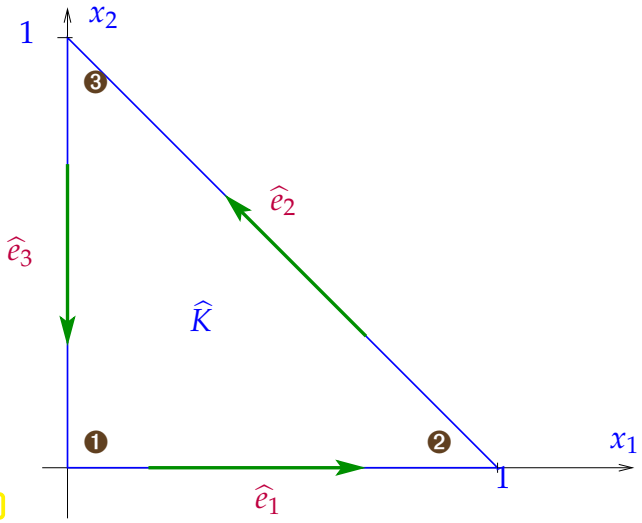


Fig. 241

In order to be able to set up a unit test, "with pen and paper" compute $\mathbf{M}_K$ for the "unit triangle" $K := \widehat{K}$ with vertices

$$\widehat{\boldsymbol{a}}^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \widehat{\boldsymbol{a}}^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \widehat{\boldsymbol{a}}^3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

◁ Numbering and induced orientiations of edges are chosen in agreement with LEHRFEM++'s conventions.

SOLUTION for (13-2.y) → 13-2-25-0:HLSG2.pdf ▲

(13-2.z) ⚄ (10 min.) The entries of $\mathbf{M}_K$ are first computed using local shape functions associated with nodes and edges of K, the latter of which carry the **induced orientation** [Lecture → § 13.2.1.10].

However, that induced orientation need not match the **intrinsic orientation** of the edge in $\mathcal{M}$. Therefore we have to correct for this mismatch by

multiplying the row and column of $\mathbf{M}_K$ belonging to the local shape function $\beta^1_{i,K}$ with -1, in case the induced orientation of the i-th edge of K is different from its intrinsic orientation.

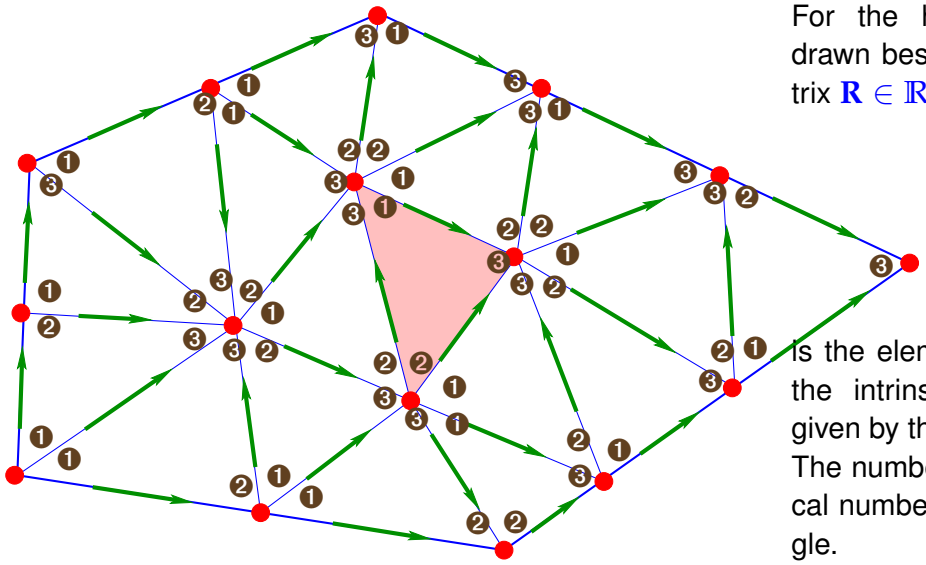


Fig. 242

For the highlighted cell of the mesh drawn beside write down a *diagonal* matrix $\mathbf{R} \in \mathbb{R}^{6,6}$ such that

$$\widetilde{\mathbf{M}}_K = \mathbf{R}^\top \mathbf{M}_K \mathbf{R}$$

is the element matrix taking into account the intrinsic orientations of the edges given by the green arrows →.
The numbers ❶, ❷, and ❸ specify the local numbering of the vertices of the triangle.

$$\mathbf{R} = \operatorname{diag}\left(\quad \right).$$

SOLUTION for (13-2.z) → 13-2-26-0:HLsH.pdf ▲

(13-2.aa) ⚂ (45 min.) [depends on Sub-problem (13-2.t), Sub-problem (13-2.u), Sub-problem (13-2.v), Sub-problem (13-2.x)]

In the file `hodgelaplacian2d.cc` implement the member function `Eval()` of the class of **HodgeLaplacian2DElementMatrixProvider** listed in Code 13.2.51.

SOLUTION for (13-2.aa) → `13-2-27-0:HLSIs.pdf` ▲

(13-2.ab) ⚂ (15 min.)

```
N      L2 err(u)  L2 err(p)
----------------------------
40     0.314      0.612
139    0.164      0.176
517    0.0838     0.0477
1993   0.0422     0.0123
7825   0.0212     0.00311
31009  0.0106     0.000779
123457 0.00529    0.000195
492673 0.00265    4.87e-05
```

If your implementation of the `Eval()` member function of **HodgeLaplacian2DElementMatrixProvider** is correct and you run the main program for this project you should obtain the error table shown on the left.

The 'N' column reports the total number of degrees of freedom, that is $N = \dim \mathcal{W}^0(\mathcal{M}) + \dim \mathcal{W}^1(\mathcal{M})$.

The table's two rightmost columns give the errors

$$\|\omega - \omega_h\|_{L^2\Lambda^1(\Omega)} \quad \text{and} \quad \|\pi - \pi_h\|_{L^2\Lambda^0(\Omega)}\,,$$

respectively, for the Whitney finite element solutions $(\omega_h, \pi_h) \in \mathcal{W}^1(\mathcal{M}) \times \mathcal{W}^0(\mathcal{M})$ of the discrete variational problem (13.2.39) and for a sequence of triangular meshes created by uniform regular refinement through `lf::refinement::GenerateMeshHierarchyByUniformRefinemnt()`.

The method of manufactured solutions was applied [Lecture → § 3.8.0.9]. The data for this numerical experiment were (in 2D Euclidean vector proxies $\mathbf{u} := \mathrm{v.p.}(\omega)$, $p := \mathrm{v.p.}(\pi)$, Option I)

$$\mathbf{f}(\boldsymbol{x}) = 2\pi^2 \begin{bmatrix} \sin(\pi x_1)\cos(\pi x_2) \\ \cos(\pi x_1)\sin(\pi x_2) \end{bmatrix} \quad \blacktriangleright \quad \begin{aligned} \mathbf{u}(\boldsymbol{x}) &= \begin{bmatrix} \sin(\pi x_1)\cos(\pi x_2) \\ \cos(\pi x_1)\sin(\pi x_2) \end{bmatrix}, \\ p(\boldsymbol{x}) &= \operatorname{div} \mathbf{u}(\boldsymbol{x}) = 2\pi \cos(\pi x_1)\cos(\pi x_2)\,. \end{aligned}$$

Obviously, both $\mathbf{u}$ and p are smooth on $\overline{\Omega}$.

Describe qualitatively and quantitatively the observed asymptotic convergence of the L^2-errors tabulated above for $N \to \infty$. Do the observations agree with the predictions of the a-priori error estimates of [Lecture → Thm. 13.3.3.41] combined with the best approximation estimates for Whitney finite element spaces given in [Lecture → Thm. 13.2.2.61].

SOLUTION for (13-2.ab) → `13-2-28-0:HLLse.pdf` ▲

Visualizing the program output with PARAVIEW:

1. Invoke `paraview 2dhl.vtk` (Will load the data file, but this can also be done through `File->Open`).
2. Two data arrays `p` and `u` should show up in the data array information tab on the left.
3. Select data source in the data toolbar (below the action toolbar)
4. In the left sidebar click "properties" and "Apply".
5. Let's visualize the `p`-data as a surface plot:
   (a) `Filters->Alphabetical->WarpByScalar` & "Apply"

(b) Select surface rendering mode in data toolbar, for instant "Surface with Edges".

(c) You may activate "Data axis grid" in the left sidebar.

6. Now let's visualize the vector field stored in `u`:

   (a) In the pipeline browser select data file and in the data toolbar menu `u`.

   (b) `Filters-Alphabetical->Glyph`, Scale Array = `u`, "Scale by Magnitude", and set scale factor.

7. The properties section of the sidebar offers many options to change axes, etc.

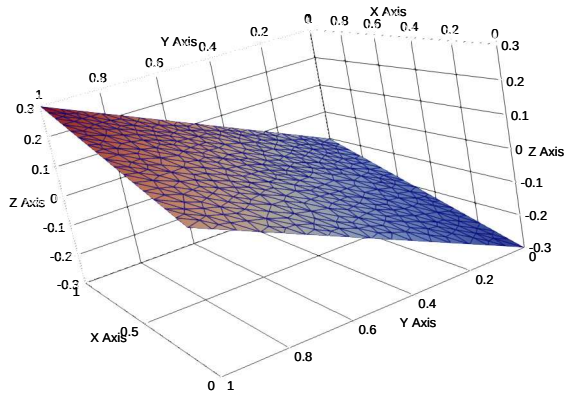


Fig. 245

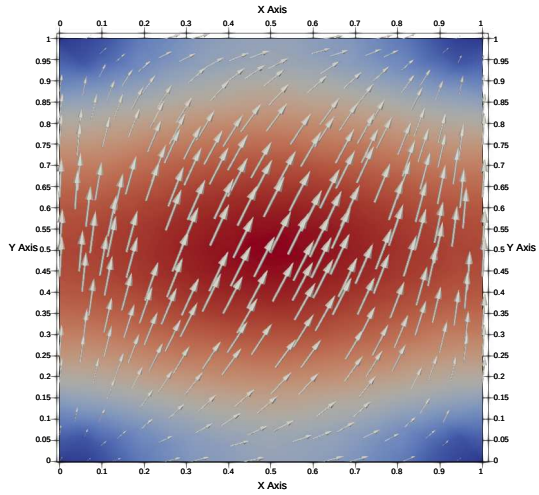


Fig. 246

References

[AFW06] Douglas N. Arnold, Richard S. Falk, and Ragnar Winther. "Finite element exterior calculus, homological techniques, and applications". In: *Acta Numer.* 15 (2006), pp. 1–155. DOI: 10.1017/S0962492906210018 (cit. on p. 511).

[BBF13] Daniele Boffi, Franco Brezzi, and Michel Fortin. *Mixed finite element methods and applications*. Vol. 44. Springer Series in Computational Mathematics. Springer, Heidelberg, 2013, pp. xiv+685. DOI: 10.1007/978-3-642-36519-5 (cit. on p. 513).

End Problem 13-2 , 410 min.

Problem 13-3: Whitney Finite Elements for Magnetostatic BVP in Two Dimensions

In an x_3-translation-invariant setting with in-plane source currents the resulting magnetic fields can be computed approximately by solving a boundary value problem (BVP) for the in-plane component of the magnetic vector potential on the two-dimensional cross-sectional domain. In this problem we examine the finite element Galerkin discretization of that BVP by means of Whitney finite elements and using the Coulomb gauge.

This problem relies on [Lecture → Section 13.2.2.3] and [Lecture → Section 13.3.3.2]. General knowledge about the finite element method is taken for granted as well as familiarity with LEHRFEM++.
▷ problem name/problem code folder: MagStat2D

We consider linear magnetostatics as in [Lecture → Section 13.3.3]. Assuming that

- none of the fields depends on the coordinate x_3,
- the material is linear, local, and **isotropic**, that is, the permeability tensor can be represented by a uniformly positive bounded function $\mu : \Omega \to \mathbb{R}^+$, and
- that the third component of the source current vanishes,

we arrive at the following Coulomb-gauged vector-potential-based variational formulation on a domain $\Omega \subset \mathbb{R}^2$, here stated in terms of 2D Euclidean vector proxies (according to Option I).

Seek $\mathbf{a} \in \boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega)$, $p \in H^1_0(\Omega)$ such that

$$\begin{aligned} \int_\Omega \frac{1}{\mu(\boldsymbol{x})} \mathrm{curl}_{2D}\, \mathbf{a}(\boldsymbol{x})\, \mathrm{curl}_{2D}\, \mathbf{a}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \mathbf{a}'(\boldsymbol{x}) \cdot \mathbf{grad}\, p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{j}(\boldsymbol{x}) \cdot \mathbf{a}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \mathbf{a}(\boldsymbol{x}) \cdot \mathbf{grad}\, p'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned} \tag{13.3.1}$$

for all $\mathbf{a}' \in \boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega)$, $p' \in H^1_0(\Omega)$ and for given $\mathbf{j} \in \left(L^2(\Omega)\right)^2$.

Above we have used the Sobolev space of 2D vector fields with square-integrable 2D scalar rotation curl_{2D} and vanishing tangential trace,

$$\boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega) := \left\{ \mathbf{v} \in \left(L^2(\Omega)\right)^2 : \mathrm{curl}_{2D}\, \mathbf{v} := \frac{\partial v_2}{\partial x_1} - \frac{\partial v_1}{\partial x_2} \in L^2(\Omega),\ \mathbf{v} \cdot \boldsymbol{t} = 0 \text{ on } \partial\Omega \right\}. \tag{13.3.2}$$

(13-3.a) ⚀ (4 min.) Assuming sufficient smoothness of a solution $(\mathbf{a}, p)$ of (13.3.1) and of $\boldsymbol{x} \mapsto \mu(\boldsymbol{x})$, derive the PDE(s) and boundary conditions satisfied by $\mathbf{a}$ and p.

$$\square = \square \quad \text{in } \Omega\,,$$
$$\square = \square \quad \text{in } \Omega\,,$$
$$\square \quad \text{on } \partial\Omega\,.$$

HIDDEN HINT 1 for (13-3.a) → 13-3-1-0:ms2dh1.pdf

SOLUTION for (13-3.a) → 13-3-1-1:ms2ds1.pdf ▲

(13-3.b) ⚄ (3 min.) Let $(\mathbf{a}, p) \in \boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega) \times H_0^1(\Omega)$ be a solution of (13.3.1). Describe a function space $Z \subset \left(L^2(\Omega)\right)^2$ such that

$$p = 0 \quad \Longleftrightarrow \quad \mathbf{j} \in Z \, .$$

$$Z = \left\{ \mathbf{j} \in \left(L^2(\Omega)\right)^2 : \boxed{\phantom{xxxx

(13-3.c) ⚂ (3 min.) The finite element discretization of (13.3.1) leads to a **discrete saddle point variational problem**. Crucial for its well-posedness is the satisfaction of **discrete LBB conditions** as stated in [Lecture → Thm. 13.3.3.41].

For any $q_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ specify a concrete $\mathbf{v}_h = \mathbf{v}_h(q_h) \in \mathcal{W}^1_0(\mathcal{M})$, dependent on q_h, such that

$$\|\mathbf{v}_h\|_{L^2(\Omega)} + \|\mathrm{curl}_{2D}\,\mathbf{v}_h\|_{L^2(\Omega)} \le c\|q_h\|_{H^1(\Omega)} \quad \text{and} \quad \frac{\int_\Omega \mathbf{v}_h(\boldsymbol{x})\cdot\mathbf{grad}\,q_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}{\|\mathbf{v}_h\|_{L^2(\Omega)} + \|\mathrm{curl}_{2D}\,\mathbf{v}_h\|_{L^2(\Omega)}} \ge \frac{1}{c}\|q_h\|_{H^1(\Omega)}$$

with a constant $c > 0$ independent of q_h:

$$\mathbf{v}_h = \boxed{}\,.$$

Remark. If such a $\mathbf{v}_h$ can be found, [Lecture → (LBB2$_h$)] in [Lecture → Thm. 13.3.3.41] will be fulfilled.

SOLUTION for (13-3.c) → 13-3-3-0:ms2dlbbs.pdf ▲

(13-3.d) ⚂ (3 min.) The chosen finite element method leads to a linear system of equations in saddle point form

$$\begin{bmatrix} \mathbf{O} & \mathbf{B} \\ \mathbf{B}^\top & \mathbf{A} \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\pi}} \\ \vec{\boldsymbol{\alpha}} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \vec{\boldsymbol{\psi}} \end{bmatrix}, \tag{13.3.9}$$

where

- $\vec{\boldsymbol{\pi}}$ is the basis expansion coefficient of the finite element approximation for p, and
- $\vec{\boldsymbol{\alpha}}$ is the same for $\mathbf{a}$.

Give formulas for the entries of the matrices $\mathbf{B}$ and $\mathbf{A}$ and determine their sizes

$$(\mathbf{A})_{i,j} = \boxed{},\quad i = 1,\ldots,\boxed{},\ j = 1,\ldots,\boxed{},$$
$$(\mathbf{B})_{i,j} = \boxed{},\quad i = 1,\ldots,\boxed{},\ j = 1,\ldots,\boxed{}.$$

SOLUTION for (13-3.d) → 13-3-4-0:mssps.pdf ▲

(13-3.e) ⚂ (3 min.) [depends on Sub-problem (13-3.d)]

Give an upper bound, as sharp as possible, of the number of non-zero entries of $\mathbf{A}$ in terms of the numbers N_0, N_1, and $N_2 := \sharp\mathcal{F}_2(\mathcal{M})$ (number of cells of the mesh).

$$\mathrm{nnz}(\mathbf{A}) \le \boxed{}\,.$$

SOLUTION for (13-3.e) → 13-3-5-0:mg2d3s.pdf ▲

We realize the Whitney FEM for (13.3.1) in LEHRFEM++ following a **monolithic approach** that is, the full matrix of the linear system of equations (13.3.9) is stored in one big sparse matrix and a single **lf::assemble::DofHandler** object is in charge of managing all degrees of freedom. In the monolithic approach assembly will also rely on a single **ENTITY_MATRIX_PROVIDER** class, which is defined as follows:

C++ code 13.3.10: Declaration of ENTITY_MATRIX_PROVIDER class for the computation of the element matrix for the monolithic Whitney FEM discretization of the vector-potential-based 2D magnetostatic saddle point variational problem

```
template <lf::mesh::utils::MeshFunction MESH_FUNCTION>
class MagStat2DElementMatrixProvider {
 public:
  // The size of the element matrix is 6×6.
  using ElemMat = Eigen::Matrix<double, 6, 6>;
  MagStat2DElementMatrixProvider(const MagStat2DElementMatrixProvider &) =
      delete;
  MagStat2DElementMatrixProvider(MagStat2DElementMatrixProvider &&) noexcept =
      default;
  MagStat2DElementMatrixProvider &operator=(
      const MagStat2DElementMatrixProvider &) = delete;
  MagStat2DElementMatrixProvider &operator=(MagStat2DElementMatrixProvider &&) =
      delete;
  MagStat2DElementMatrixProvider() = delete;
  virtual ~MagStat2DElementMatrixProvider() = default;

  MagStat2DElementMatrixProvider(MESH_FUNCTION mu) : mu_(mu) {}

  [[nodiscard]] bool isActive(const lf::mesh::Entity & /*cell*/) {
    return true;
  }
  [[nodiscard]] ElemMat Eval(const lf::mesh::Entity &cell);

 private:
  ElemMat MK_;
  MESH_FUNCTION mu_;
};
```

(13-3.f) ⚁ (2 min.) To represent the local→global mapping for the shape functions of the $\mathcal{W}_0^1(\mathcal{M}) \times \mathcal{S}_{1,0}^0(\mathcal{M})$-based finite element method, we rely on a single instance of type **lf::assemble::UniformFEDofHandler**. How does a suitable object of that class have to be initialized?

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(),      },
  {lf::base::RefEl::kSegment(),           },
  {lf::base::RefEl::kTria(),              },
  {lf::base::RefEl::kQuad(),              });
```

SOLUTION for (13-3.f) → 13-3-6-0:HLsD.pdf ▲

(13-3.g) ⚄ (10 min.) Complete the following code implementing the `Eval()` member function of **MagStat2DElementMatrixProvider**. To deal with the possibly non-constant permeability $\mu = \mu(\boldsymbol{x})$ the 3-point local quadrature formula

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \tfrac{1}{3}|K| \sum_{\ell=1}^{3} f(\boldsymbol{m}_\ell^K)\,, \quad K \in \mathcal{M}\,, \tag{13.3.11}$$

is used, where $\boldsymbol{m}_1^K$, $\boldsymbol{m}_2^K$, and $\boldsymbol{m}_3^K$ are the midpoints of the edges of the triangle K.

```
template <lf::mesh::utils::MeshFunction MESH_FUNCTION>
typename MagStat2DElementMatrixProvider<MESH_FUNCTION>::ElemMat
MagStat2DElementMatrixProvider<MESH_FUNCTION>::Eval(
  const lf::mesh::Entity &cell) {
  // Retrieve value of coefficient μ at the center of gravity
  // of the triangle
  const Eigen::MatrixXd mpc{
      (Eigen::MatrixXd(2, 1) << 1.0 / 3.0, 1.0 / 3.0).finished()};
  const auto muvals{mu_(cell, mpc)};
  const double muval = muvals[0];
  // Area of the triangle
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions and store them
  // in the columns of the 2×3-matrix G
  const Eigen::MatrixXd dpt = (Eigen::MatrixXd(2, 1) << 0.0, 0.0).finished();
  const Eigen::Matrix<double, 2, 3> G =
      cell.Geometry()->JacobianInverseGramian(dpt).block(0, 0, 2, 2) *
    (Eigen::Matrix<double, 2, 3>(2,3) << -1, 1, 0,
                                         -1, 0, 1).finished();
  // Compute the element matrix for −Δ and FE space S_1^0,
  // see also [Lecture → Code 2.4.5.13].
  Eigen::Matrix<double, 3, 3> L = area * G.transpose() * G;
  Eigen::Matrix3d B;
  B.col(0) = (L.col(1) - L.col(0)) / 3.0;
  B.col(1) = (L.col(2) - L.col(1)) / 3.0;
  B.col(2) = (L.col(0) - L.col(2)) / 3.0;
  // Set blocks of element matrix
  MK_.block(0, 0, 3, 3) =                                          ;
  MK_.block(0, 3, 3, 3) =                                          ;
  MK_.block(3, 0, 3, 3) =                                          ;
  MK_.block(3, 3, 3, 3) =                                          ;
  // Correct for orientation mismatch
  auto relor = cell.RelativeOrientations();
  for (int k = 0; k < 3; ++k) {
    if (relor[k] == lf::mesh::Orientation::negative) {
      ;
      ;
    }
  }
  return MK_;
}
```

SOLUTION for (13-3.g) → 13-3-7-0:msqs.pdf ▲

(13-3.h) ⚄ (10 min.) In the file `magstat2d.h` you can find a C++ function

```
template <lf::mesh::utils::MeshFunction MESH_FUNCTION_J,
          lf::mesh::utils::MeshFunction MESH_FUNCTION_MU>
Eigen::VectorXd solveMagStat2DBVP(
  const lf::assemble::DofHandler &dofh,
  MESH_FUNCTION_MU mu, MESH_FUNCTION_J j_source);
```

which computes the finite element solution of (13.3.1):

- The argument `dofh` passes a **lf::assemble::DofHandler** object which also contains information about the mesh.
- The parameters `mu` and `j_source` supply functor objects for the permeability $\boldsymbol{x} \mapsto \mu(\boldsymbol{x})$ and the source current $\boldsymbol{x} \mapsto j(\boldsymbol{x})$.
- The function returns the basis expansion coefficient vector for the finite element solution of the monolithic Whitney FEM.

The implementation of the function has to be completed in order to handle the zero essential boundary conditions correctly.

The provided code relies on the two functions `buildMagStat2DGalerkinMatrix()` and `computeMagStat2DRhsVector()`, which provide the *full* matrix and right-hand side vector of the finite element linear system of equations for the monolithic Whitney FEM for (13.3.1). "Full" means that the underlying finite element space is $\mathcal{W}^1(\mathcal{M}) \times \mathcal{S}_1^0(\mathcal{M})$; essential boundary conditions have not yet been taken into account.

SOLUTION for (13-3.h) → `13-3-8-0:ms2ds6.pdf` ▲

End Problem 13-3 , 38 min.

SOLUTION of (0-1.c):

By running `NumPDESetup_mastersolution` you should obtain the following output

```
Dummy "homework problem" for NumPDE course
NumPDESetup: master solution code
42, 42, 42, 42, 42, 42, 42, 42, 42, 42
```

while running `NumPDESetup_mysolution` prints

```
Dummy "homework problem" for NumPDE course
NumPDESetup: student solution code
0, 0
```

C++ code 0.1.1: Master solution for some EIGEN-based function ⇝ GitLab

```
2  Eigen::VectorXd dummyFunction(double x, int n) {
3    // Appears only in mastersolution
4    std::cout << "NumPDESetup: master solution code" << std::endl;
5    return Eigen::VectorXd::Constant(n, x);
6  }
```

C++ code 0.1.2: Driver code with `main()` function ⇝ GitLab

```
2  int main(int /*argc*/, char** /*argv*/) {
3    std::cout << "Dummy \"homework problem\" for NumPDE course" << std::endl;
4    Eigen::VectorXd v = NumPDESetup::dummyFunction(42.0, 10);
5
6    const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
7                                           Eigen::DontAlignCols, ", ", "\n");
8    std::cout << v.transpose().format(CSVFormat) << std::endl;
9    return 0;
10 }
```

However, when you run `NumPDESetup_mysolution`, then the following template version of the function `dummyFunction()` is used.

C++ code 0.1.3: Template for `dummyFunction()` ⇝ GitLab

```
2  Eigen::VectorXd dummyFunction(double x, int n) {
3    // Appears only in mysolution and templates
4    std::cout << "NumPDESetup: student solution code" << std::endl;
5    // This is not a meaningful solution
6    // Please remove this line and put your code here!
7    return Eigen::Vector2d::Zero();
8  }
```

SOLUTION of (0-1.e):

C++ code 0.1.4: Triplet initialization of a sparse matrix and use of sparse direct solver
➥ GitLab

```cpp
void demo_EigenTripletInit(unsigned int n) {
  const double diag = 2.0;
  const double offd = -1.0;
  // Initialize tridiagonal sparse matrix via COO (Triplet) format
  std::vector<Eigen::Triplet<double>> X_COO;
  X_COO.emplace_back(0, 0, diag);
  X_COO.emplace_back(0, 1, offd);
  for (int j = 1; j < n - 1; ++j) {
    X_COO.emplace_back(j, j - 1, offd);
    X_COO.emplace_back(j, j, diag);
    X_COO.emplace_back(j, j + 1, offd);
  }
  X_COO.emplace_back(n - 1, n - 1, diag);
  X_COO.emplace_back(n - 1, n - 2, offd);
  // Build sparse matrix
  Eigen::SparseMatrix<double> X(n, n);
  X.setFromTriplets(X_COO.begin(), X_COO.end());
  // Solve linear system
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(X);
  if (solver.info() != Eigen::Success) {
    throw std::runtime_error("Could not factorize X");
  }
  std::cout << "x = "
            << solver.solve(Eigen::VectorXd::Constant(n, 1.0)).transpose()
            << std::endl;
}
```

SOLUTION of (0-1.f):

C++ code 0.1.5: Code running test based on GoogleTest framework ➙ GitLab

```
2  TEST(NumPDESetup, dummyFunction) {
3    double x = 1.0;
4    int n = 2;
5    // Invoke the function to be tested
6    Eigen::Vector2d v = NumPDESetup::dummyFunction(x, n);
7    // The expected solution
8    Eigen::Vector2d v_ref = {1.0, 1.0};
9    // Check whether function call produces correct result using the
       Google test
10   // framework
11   double tol = 1.0e-8;
12   ASSERT_NEAR(0.0, (v - v_ref).lpNorm<Eigen::Infinity>(), tol);
13 }
```

Of course, the implementation of `dummyFunction()` in Code 0.1.1 in the case `SOLUTION=0` just returns the zero vector, recall Code 0.1.3, which makes the test fail. Hence, we get the following output:

```
Running main() from
    /scratch/users/ralfh/.hunter/_Base/d45d77d/3ae2c49/4368ee8/
Build/GTest/Source/googletest/src/gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from NumPDESetup
[ RUN      ] NumPDESetup.dummyFunction
NumPDESetup: student solution code
/scratch/users/ralfh/NOSAVE/numpde/Numcourses/NumPDE/npdeflipped/
NPDECODES/homeworks/NumPDESetup/mysolution/test/numpdesetup_test.cc:29:
    Failure
The difference between 0.0 and (v - v_ref).lpNorm<Eigen::Infinity>() is 1,
    which exceeds tol, where
0.0 evaluates to 0,
(v - v_ref).lpNorm<Eigen::Infinity>() evaluates to 1, and
tol evaluates to 1e-08.
[  FAILED  ] NumPDESetup.dummyFunction (0 ms)
[----------] 1 test from NumPDESetup (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] NumPDESetup.dummyFunction
1 FAILED TEST
```

SOLUTION of (0-2.a):

Calling `g++` from the command line:

```
build $ g++ --version
g++ (GCC) 13.2.1 20231011 (Red Hat 13.2.1-4)
Copyright (C) 2023 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

SOLUTION of (0-2.b):

Checking the currently installed version of the debugger.

```
build $ gdb --version
GNU gdb (GDB) Fedora Linux 14.1-3.fc38
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

SOLUTION of (0-2.c):

An excerpt from the **cmake** configuration:

```
....
CMAKE_BUILD_TYPE                    Debug
CMAKE_COLOR_MAKEFILE                ON
CMAKE_CXX_COMPILER                  /usr/bin/g++
CMAKE_CXX_COMPILER_AR               /usr/bin/gcc-ar
CMAKE_CXX_COMPILER_RANLIB           /usr/bin/gcc-ranlibq
CMAKE_CXX_FLAGS
CMAKE_CXX_FLAGS_DEBUG               -g
....
```

SOLUTION of (0-2.d):

The installation command:

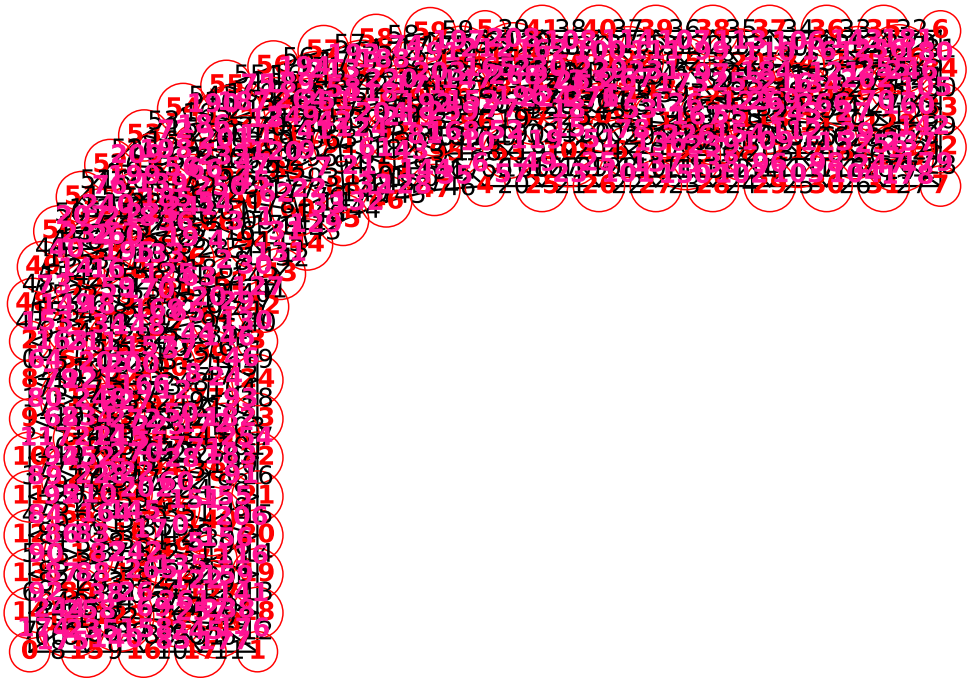
```
git clone https://github.com/dmillard/eigengdb.git
cd eigengdb
python setup.py install # Make sure to use system python which matches GDB
python bin/eigengdb_register_printers
```

SOLUTION of (0-2.e):

If all goes well the code should print

```
enter of codim-0 entities = 1.47275 2.54429
Center of codim-1 entities = 1.47804 2.53784
Center of codim-2 entities = 1.49024 2.52361
Wrote .../ljoint.csv
```

and a “messy” plot should be stored in .../mesh.eps.



HINT 1 for (0-2.f): A quite extensive documentation of GDB an be found here, while instructions for operating the LLDB debugger can be found in this document by Till Ehrengruber. ⌟

SOLUTION of (0-2.f):

```
(gdb) break DebuggingWitGDB::ReadAndOutputMesh(char const*)
Breakpoint 1 at 0x13179: file .../debuggingwithgdb.cc, line 15.
(gdb) break debuggingwithgdb.cc:29
Breakpoint 2 at 0x13359: file .../debuggingwithgdb.cc, line 29.
(gdb) break debuggingwithgdb.cc:37 if (mesh.Index(*entity) == 10)
Breakpoint 3 at 0x1349f: file .../debuggingwithgdb.cc, line 37
(gdb) info breakpoints
Num     Type           Disp Eb Address            What
1       breakpoint     keep y  0x0000000000013179 in
    DebuggingWithGDB::ReadAndOutputMesh(char const*)
                                                  at .../debuggingwithgdb.cc:15
2       breakpoint     keep y  0x0000000000013359 in
    DebuggingWithGDB::ReadAndOutputMesh(char const*)
                                                  at .../debuggingwithgdb.cc:29
3       breakpoint     keep y  0x000000000001349f in
    DebuggingWithGDB::ReadAndOutputMesh(char const*)
                                                  at .../debuggingwithgdb.cc:37
stop only if (mesh.Index(*entity) == 10)
```

SOLUTION of (0-2.g):

```
(gdb) run
Starting program: .../DebuggingWithGDB_mysolution
[Thread debugging using libthread_db enabled]
Using host libthead_db library "/lib/x86_64-linux-gnu/libthread_db.so.1"

Breakpoint 1, DebuggingWithGDB::ReadAndOutputMesh
    (filename=0x555555d73560<lf::base::RefEl::ncoords_segment_dynamic_>
    "\220_\330UUU")
     at .../debuggingwithgdb.cc:15
15       void ReadAndOutputMesh(const char *filename) {
(gdb) step
16         if (filename != nullptr) {
(gdb) print filename
$2 = 0x555555ae0114 "../ljoint.msh"
(gdb) continue
Continuing.

Breakpoint 2, DebuggingWithGDB::ReadAndOutputMesh (filename=0x555555ae0114
    "../ljoint.msh")
     at .../debuggingwithgdb.cc:29
29             Eigen::VectorXd c{Eigen::VectorXd::Zero(mesh.DimWorld())};
(gdb) print codim
$3 = 0
(gdb) continue
Breakpoint 3, DebuggingWithGDB::ReadAndOutputMesh (filename=0x555555ae0114
    "../ljoint.msh")
     at .../debuggingwithgdb.cc:37
37               c += vertices.rowwise().sum() / no_vertices;
(gdb) print vertices
$3 = Eigen::Matrix<double,2,3,ColMajor> (data ptr: 0x555555db23b0)
[[1.83196313 2.10154112 1.96637639]
 [3.2096386  3.21403431 3.40581993]]
```

SOLUTION of (10-1.a):

Recall from [Lecture → Section 1.8] the rule "Don't test where the solution in known". By integration by parts with test functions $v \in H^1_{\Gamma_D}(\Omega) := \{v \in H^1(\Omega) : \ v|_{\Gamma_D} = 0\}$ (taking into account the Dirichlet boundary conditions), we get

$$\int_\Omega (\mathbf{grad}\, u - u\,\mathbf{grad}\,\Psi)\cdot\mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} v \underbrace{(\mathbf{n}\cdot\mathbf{grad}\, u - u\mathbf{n}\cdot\mathbf{grad}\,\Psi)}_{=\mathbf{j}\cdot\boldsymbol{n}} \,\mathrm{d}S = \int_\Omega v\, f \,\mathrm{d}\boldsymbol{x}\,.$$

The boundary term is 0 on both Γ_D (due to $v|_{\Gamma_D} = 0$) and Γ_I (due to (10.1.7b)), and we end up with

$$\int_\Omega (\mathbf{grad}\, u - u\,\mathbf{grad}\,\Psi)\cdot\mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f\, v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_{\Gamma_D}(\Omega)\,. \tag{10.1.8}$$

Remember that $u = u_0 + \tilde{g}$, with $u_0 \in H^1_{\Gamma_D}(\Omega)$ and offset function $\tilde{g} \in H^1(\Omega)$ an extension of the Dirichlet data g from Γ_D into Ω, $g|_{\Gamma_D} = g$ with g from (10.1.7a). This is the offset function technique from [Lecture → § 1.9.0.6].

Remark. Setting $\mathbf{v}(\boldsymbol{x}) := \mathbf{grad}\,\Psi(\boldsymbol{x})$ we can write (10.1.6) as

$$-\Delta u + \mathrm{div}(u\mathbf{v}) = f \quad \text{in} \quad \Omega\,. \tag{10.1.9}$$

This is a stationary convection-diffusion equation that we have already seen in [Lecture → Eq. (10.1.2.8)], with $\mathrm{div}(u\mathbf{v})$ as the "convective term". In [Lecture → Section 10.2] we did not apply integration by parts to the convective terms. Why are we doing this now?

If we forgo integration by parts on the second term, we arrive at

$$\int_\Omega \mathbf{grad}\, u\cdot\mathbf{grad}\, v + \mathrm{div}(u\,\mathbf{grad}\,\Psi)v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_{\Gamma_D}(\Omega)\,.$$

Recall that we may only assume $\Psi \in H^1(\Omega) \cap C^0(\overline{\Omega})$. For instance, Ψ could be a piecewise linear finite-element function. Then its gradient would be piecewise constant and the divergence of $u\,\mathbf{grad}\,\Psi$ may not be well-defined. Conversely, the function $u\,\mathbf{grad}\,\Psi\cdot\mathbf{grad}\, v$ will always be integrable for all $u, v \in H^1(\Omega)$, because $\mathbf{grad}\,\Psi$ is bounded.

HINT 1 for (10-1.j): Refresh yourself on , in particular on .

SOLUTION of (10-1.j):

For $\tau \approx 0$, cancellation will affect the computation of the denominator $e^\tau - 1$. This can be avoided by replacing e^τ with a Taylor polynomial:

$$e^\tau \approx 1 + \tau + \tfrac{1}{2}\tau^2 + \tfrac{1}{6}\tau^3 \,.$$

C++ code 10.1.23: Sub-problem (10-1.j): Implementation of `Bernoulli()` ➔ GitLab

```
2  double Bernoulli(double tau) {
3    double res = 0;
4    // In order to avoid cancellation, use Taylor polynomial approximation
       of
5    // the denominator for small arguments
6    if (std::abs(tau) < 1e-10) {
7      res = 1.0;
8    } else if (std::abs(tau) < 1e-3) {
9      res = 1.0 / (1.0 + (0.5 + 1.0 / 6.0 * tau) * tau);
10   } else {
11     // No cancellation for large arguments:  use formula defining the
         Bernoulli
12     // function
13     res = tau / (std::exp(tau) - 1.0);
14   }
15   return res;
16 }
```

SOLUTION of (10-1.k):

C++ code 10.1.24: Sub-problem (10-1.k): Implementation of `CompBeta()` ↪ GitLab

```cpp
// REVISE: Does not match specification
std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<double>> CompBeta(
    const lf::uscalfe::FeSpaceLagrangeO1<double>& fe_space,
    const Eigen::VectorXd& mu) {
  // data set over all edges of the mesh.
  auto beta_p = lf::mesh::utils::make_CodimMeshDataSet(fe_space.Mesh(), 1, 1.0);
  // Get the DOF handler for the FE space
  const lf::assemble::DofHandler& dofh = fe_space.LocGlobMap();
  // compute beta(e) for all edges of the mesh
  for (const lf::mesh::Entity* edge : fe_space.Mesh()->Entities(1)) {
    // Compute the DOF indices of the vertices of the edge
    auto idxs = dofh.GlobalDofIndices(*edge);
    (*beta_p)(*edge) =
        std::exp(mu(idxs[1])) * Bernoulli(mu(idxs[1]) - mu(idxs[0]));
  }

  return beta_p;
}
```

SOLUTION of (10-1.l):

C++ code 10.1.25: Sub-problem (10-1.l): Implementation of constructor of ExpFittedEMP.
➥ GitLab

```
  explicit ExpFittedEMP(
      std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
      Eigen::VectorXd mu)
      : fe_space_(fe_space), beta_(CompBeta(*fe_space, mu)), mu_(mu) {}
```

C++ code 10.1.26: Sub-problem (10-1.l): Implementation of `Eval()` method of ExpFittedEMP
➥ GitLab

```
Eigen::Matrix3d ExpFittedEMP::Eval(const lf::mesh::Entity& cell) {
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Only 2D triangles are supported.");

  // Evaluate the element matrix A_K
  Eigen::Matrix3d AK = laplace_provider_.Eval(cell).block<3, 3>(0, 0);
  Eigen::Matrix3d result;
  // Get the values of beta on the edges of the triangle.
  // by the Lehrfem++ numbering convention
  // b = [beta(e_0), beta(e_1), beta(e_2)]' = [\beta_{1,2}, \beta_{2,3},
  // \beta_{1,3}]'
  Eigen::Vector3d b = beta_loc(cell);

  // evaluate the element matrix using the formula in subproblem h)
  result << -AK(0, 1) * b(0) - AK(0, 2) * b(2), AK(0, 1) * b(0),
      AK(0, 2) * b(2), AK(0, 1) * b(0), -AK(0, 1) * b(0) - AK(1, 2) * b(1),
      AK(1, 2) * b(1), AK(0, 2) * b(2), AK(1, 2) * b(1),
      -AK(0, 2) * b(2) - AK(1, 2) * b(1);

  Eigen::Vector3d mu_exp = (-mu_loc(cell)).array().exp();
  result *= mu_exp.asDiagonal();
  return std::move(result);
}
```

HINT 1 for (10-1.m): For a related implementation refer to the example code in `ellbvp_linfe_demo` ➜ GITHUB. ⌟

SOLUTION of (10-1.m):

C++ code 10.1.27: Sub-problem (10-1.m): Implementation of `solveDriftDiffusionDirBVP()` ↬ GitLab

```
template <typename FUNC_F, typename FUNC_G>
Eigen::VectorXd solveDriftDiffusionDirBVP(
    std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    const Eigen::VectorXd& mu, FUNC_F&& func_f, FUNC_G&& func_g) {
  const lf::assemble::DofHandler& dofh{fe_space->LocGlobMap()};
  lf::base::size_type N_dofs = dofh.NumDofs();

  lf::mesh::utils::MeshFunctionGlobal mf_f{func_f};
  lf::mesh::utils::MeshFunctionGlobal mf_g{func_g};

  Eigen::VectorXd sol = Eigen::VectorXd::Ones(N_dofs);

  // Assembly of the System Matrix based on our ExpFittedEMP class
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  ExpFittedEMP elmat_builder(fe_space, mu);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);

  // Assembly of the RHS
  Eigen::VectorXd phi(N_dofs);
  phi.setZero();
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      elvec_builder(fe_space, mf_f);
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // Fixing solution components according to essential Dirichlet
     boundary
  // conditions
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fe_space->Mesh(), 1)};

  const lf::fe::ScalarReferenceFiniteElement<double>* rsf_edge =
      fe_space->ShapeFunctionLayout(lf::base::RefEl::kSegment());

  LF_ASSERT_MSG(rsf_edge != nullptr, "FE specification for edges missing.");

  // Fetch flags and values for dofs on the Dirichlet edges
  auto bd_flags_val{lf::fe::InitEssentialConditionFromFunction(
      *fe_space,
      [&bd_flags](const lf::mesh::Entity& edge) -> bool {
        return bd_flags(edge);
      },
      mf_g)};

  // Eliminate Dirichlet dofs from linear system
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&bd_flags_val](lf::assemble::glb_idx_t idx) {
        return bd_flags_val[idx];
      },
      A, phi);

  // Solve the sparse LSE
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
```

```
52      Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
53
54      solver.compute(A_crs);
55      LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
56      sol = solver.solve(phi);
57      LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
58
59      return sol;
60  }
```

SOLUTION of (10-1.n):

No. of dofs	L2 error
2704	0.000335482
10404	8.66205e-05
40804	2.29505e-05
161604	6.65167e-06

We observe that a fourfold increase of the dimension N of the finite-element space ("no. of d.o.f.s") leads to a decrease of the $L^2(\Omega)$-norm of the discretization error approximately by a factor of 4.

This hints at asymptotic algebraic convergence with rate 1 as $N \to \infty$ in the case of successive regular refinement:

$$\|u - u_h\|_{L^2(\Omega)} = O(N^{-1}) \quad \text{for} \quad N \to \infty \,.$$

The algebraic convergence rate in terms of the number of degrees of freedom N found by linear regression as in [Lecture → Code 3.2.2.10] is 0.9597 for $N \to \infty$.

HINT 1 for (10-1.b): Use the product rule for the gradient

$$\mathbf{grad}(u\,v)(\boldsymbol{x}) = \mathbf{grad}\,u(\boldsymbol{x})\,v(\boldsymbol{x}) + u(\boldsymbol{x})\,\mathbf{grad}\,v(\boldsymbol{x})\,, \quad u,v \in C^1(\overline{\Omega})\,. \tag{10.1.12}$$

⌟

SOLUTION of (10-1.b):

Directly from the product rule (10.1.12):

$$\mathbf{grad}[\exp(-\Psi(\boldsymbol{x}))\,u(\boldsymbol{x})] = -\exp(-\Psi(\boldsymbol{x}))\,\mathbf{grad}\,\Psi(\boldsymbol{x})\,u(\boldsymbol{x}) + \exp(-\Psi(\boldsymbol{x}))\,\mathbf{grad}\,u(\boldsymbol{x})\ .$$

The exponentials will cancel.

HINT 1 for (10-1.c): The flux $\mathbf{j}$ directly occurs in the variational formulation. Make use of (10.1.11). ⌟

SOLUTION of (10-1.c):

Recalling the flux law

$$\mathbf{j}(\boldsymbol{x}) = -\,\mathbf{grad}\, u(\boldsymbol{x}) + u(\boldsymbol{x})\,\mathbf{grad}\,\Psi(\boldsymbol{x})$$

we can recast (10.1.8) as

$$u \in g + H^1_{\Gamma_D}(\Omega): \quad -\int_\Omega \mathbf{j}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_{\Gamma_D}(\Omega)\ .$$

Then we can plug in (10.1.11) and get: Seek $u \in H^1(\Omega)$, $u|_{\Gamma_D} = g$, such that

$$\int_\Omega \exp(\Psi)\ \mathbf{grad}\, w\ \mathbf{grad}\, v\, \mathrm{d}\boldsymbol{x} = \int_\Omega v\, f\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_{\Gamma_D}(\Omega)\ .$$

HINT 1 for (10-1.d): Appeal to the variational formulation found in Sub-problem (10-1.c). ⌟

SOLUTION of (10-1.d):

The arguments of [Lecture → Section 1.2.3] apply, because under the assumption $\Psi \in C^0(\overline{\Omega})$ the coefficient function $\kappa(\boldsymbol{x}) = \exp(\Psi(\boldsymbol{x}))$ is uniformly positive on any bounded domain.

HINT 1 for (10-1.e): A precise formula for $\mathbf{A}_K$ is given in [Lecture $\rightarrow$ Eq. (2.4.5.8)]. Recall that the row sums of $\mathbf{A}_K$ vanish. ⌟

SOLUTION of (10-1.e):

By their very definition the barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ of the triangle K form a cardinal basis of the space $\mathcal{P}_1(\mathbb{R}^2)$ of affine-linear functions with respect to point evaluation at the vertices. In particular, for $u_h.v_h \in \mathcal{S}_1^0(\mathcal{M})$

$$u_h|_K = u_h(\boldsymbol{a}_K^1)\lambda_1 + u_h(\boldsymbol{a}_K^2)\lambda_2 + u_h(\boldsymbol{a}_K^3)\lambda_3 \quad , \quad v_h|_K = v_h(\boldsymbol{a}_K^1)\lambda_1 + v_h(\boldsymbol{a}_K^2)\lambda_2 + v_h(\boldsymbol{a}_K^3)\lambda_3 \, . \tag{10.1.14}$$

Since the barycentric coordinate functions are the local shape functions [Lecture → Def. 2.5.3.4] of the finite-element space $\mathcal{S}_1^0(\mathcal{M})$ the entries of the element matrix are

$$(\mathbf{A}_K)_{i,j} = \int_K \mathbf{grad}\, \lambda_j(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \, , \quad i,j \in \{1,2,3\} \, . \tag{10.1.15}$$

This together with (10.1.14) implies that

$$\int_K \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \, \mathrm{d}\boldsymbol{x} = [u_h(\boldsymbol{a}_K^1), u_h(\boldsymbol{a}_K^2), u_h(\boldsymbol{a}_K^3)]\mathbf{A}_K[v_h(\boldsymbol{a}_K^1), v_h(\boldsymbol{a}_K^2), v_h(\boldsymbol{a}_K^3)]^\top \, .$$

In addition, we can write $\mathbf{A}_K$ as the sum of its diagonal part, its lower triangular part and its upper triangular part. The diagonal part can be eliminated using the fact that *the row/column sums of* $\mathbf{A}_K$ *vanish*:

$$\int_K \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \, \mathrm{d}\boldsymbol{x} =$$

$$\begin{bmatrix} u_h(\boldsymbol{a}_K^1) \\ u_h(\boldsymbol{a}_K^2) \\ u_h(\boldsymbol{a}_K^3) \end{bmatrix}^\top \begin{bmatrix} -(\mathbf{A}_K)_{1,2} - (\mathbf{A}_K)_{1,3} & (\mathbf{A}_K)_{1,2} & (\mathbf{A}_K)_{1,3} \\ (\mathbf{A}_K)_{2,1} & -(\mathbf{A}_K)_{2,1} - (\mathbf{A}_K)_{2,3} & (\mathbf{A}_K)_{2,3} \\ (\mathbf{A}_K)_{3,1} & (\mathbf{A}_K)_{3,2} & -(\mathbf{A}_K)_{3,1} - (\mathbf{A}_K)_{3,2} \end{bmatrix} \begin{bmatrix} v_h(\boldsymbol{a}_K^1) \\ v_h(\boldsymbol{a}_K^2) \\ v_h(\boldsymbol{a}_K^3) \end{bmatrix} \, .$$

Then we exploit that symmetry of $\mathbf{A}_K$,

$$\int_K \mathbf{grad}\, u_h \cdot \mathbf{grad}\, v_h \, \mathrm{d}\boldsymbol{x} =$$

$$\begin{bmatrix} u_h(\boldsymbol{a}_K^1) \\ u_h(\boldsymbol{a}_K^2) \\ u_h(\boldsymbol{a}_K^3) \end{bmatrix}^\top \begin{bmatrix} -(\mathbf{A}_K)_{1,2} - (\mathbf{A}_K)_{1,3} & (\mathbf{A}_K)_{1,2} & (\mathbf{A}_K)_{1,3} \\ (\mathbf{A}_K)_{1,2} & -(\mathbf{A}_K)_{1,2} - (\mathbf{A}_K)_{2,3} & (\mathbf{A}_K)_{2,3} \\ (\mathbf{A}_K)_{1,3} & (\mathbf{A}_K)_{2,3} & -(\mathbf{A}_K)_{1,3} - (\mathbf{A}_K)_{2,3} \end{bmatrix} \begin{bmatrix} v_h(\boldsymbol{a}_K^1) \\ v_h(\boldsymbol{a}_K^2) \\ v_h(\boldsymbol{a}_K^3) \end{bmatrix} \, ,$$

and after some rearrangements of terms we get (10.1.13).

HINT 1 for (10-1.f): Use the result of Sub-problem (10-1.e) and that $\boldsymbol{j} = \mathbf{grad}(\{\boldsymbol{x} \mapsto \boldsymbol{j} \cdot \boldsymbol{x}\})$. ⌟

SOLUTION of (10-1.f):

(10.1.16) is immediate, given the solution of Sub-problem (10-1.e) with $u_h(\boldsymbol{x}) = \mathbf{j} \cdot \boldsymbol{x}$, which means

$$u_h(\boldsymbol{a}_K^\ell) = \mathbf{j} \cdot \boldsymbol{a}_K^\ell \,, \quad \ell = 1,2,3 \,.$$

HINT 1 for (10-1.g): Use the expression (10.1.11) for the flux $\mathbf{j}$. ⌟

SOLUTION of (10-1.g):

For the sake of brevity we write $\gamma := \gamma_{j,i}$ and obtain by the chain rule

$$\begin{aligned}\mathbf{j}(u(\gamma(\xi))) \cdot \boldsymbol{d}_{j,i} &= -e^{\Psi(\gamma(\xi))}\,\mathbf{grad}\Big\{\boldsymbol{x} \mapsto e^{-\Psi(\boldsymbol{x})}u(\boldsymbol{x})\Big\}(\gamma(\xi)) \cdot \boldsymbol{d}_{j,i} \\ &= -e^{\Psi(\gamma(\xi))}\frac{d}{d\xi}\Big(e^{-\Psi(\gamma(\xi))}u(\gamma(\xi))\Big) \stackrel{!}{=} \text{const}\,.\end{aligned}$$

Straightforward integration

$$-e^{\Psi(\gamma(\xi))}\frac{d}{d\xi}\Big(e^{-\Psi(\gamma(\xi))}u(\gamma(\xi))\Big) = \text{const}$$

$$\blacktriangleright \qquad \frac{d}{d\xi}\Big(e^{-\Psi(\gamma(\xi))}u(\gamma(\xi))\Big) = -\frac{\text{const}}{\exp(\Psi(\gamma(\xi)))}\,, \quad 0 \le \xi \le 1$$

$$\blacktriangleright \qquad \int_0^1 \frac{d}{d\xi}\Big(e^{-\Psi(\gamma(\xi))}u(\gamma(\xi))\Big)\,\mathrm{d}\xi = \int_0^1 -\frac{\text{const}}{\exp(\Psi(\gamma(\xi)))}\,\mathrm{d}\xi\,,$$

and appealing to the fundamental theorem of calculus [Lecture → Eq. (1.5.1.6)] yields for this constant

$$\boldsymbol{c}_{j,i} \cdot \boldsymbol{d}_{j,i} = \left(\int_0^1 \exp(-\Psi(\gamma(\tau)))\,\mathrm{d}\tau\right)^{-1}\Big(e^{-\Psi(a_K^j)}u(\boldsymbol{a}_K^j) - e^{-\Psi(a_K^i)}u(\boldsymbol{a}_K^i)\Big)\,. \tag{10.1.18}$$

Note that it takes only the values of u at the endpoints of the edge to compute $\boldsymbol{c}_{j,i} \cdot \boldsymbol{d}_{j,i}$.

HINT 1 for (10-1.h): The entries of the element matrix $\mathbf{A}_K^{\exp}$ are defined as

$$(\mathbf{A}_K^{\exp})_{k,\ell} = -\sum_{1\le j<i\le 3} (\mathbf{A}_K)_{i,j}\,(\boldsymbol{c}_{j,i}(\lambda_\ell)\cdot\boldsymbol{d}_{j,i})\left(\lambda_k(\boldsymbol{a}_K^i)-\lambda_k(\boldsymbol{a}_K^j)\right), \quad k,\ell\in\{1,2,3\}, \tag{10.1.20}$$

where $\boldsymbol{c}_{j,i}=\boldsymbol{c}_{j,i}(u)$ has been introduced in (10.1.17). Then replace $\boldsymbol{c}_{j,i}(\lambda_\ell)\cdot\boldsymbol{d}_{j,i}$ with the expression found in Sub-problem (10-1.g), which is constant for each edge. ⌟

HINT 2 for (10-1.h): Of course, for $\Psi \equiv 0$ you should get the element matrix $\mathbf{A}_k$ for $-\Delta$. This can be used for the validation of the formulas you get for the entries of $\mathbf{A}_K^{\exp}$. ⌟

SOLUTION of (10-1.h):

Of course, we have to rely on (10.1.18). To begin with note

$$\begin{aligned}
&\boldsymbol{c}_{1,2}(\lambda_1)\cdot\boldsymbol{d}_{1,2} = e^{-\Psi(a_K^1)}\beta_{1,2}\,, && \boldsymbol{c}_{1,3}(\lambda_1)\cdot\boldsymbol{d}_{1,3} = e^{-\Psi(a_K^1)}\beta_{1,3}\,, && \boldsymbol{c}_{2,3}(\lambda_1)\cdot\boldsymbol{d}_{2,3} = 0\,,\\
&\boldsymbol{c}_{1,2}(\lambda_2)\cdot\boldsymbol{d}_{1,2} = -e^{-\Psi(a_K^2)}\beta_{1,2}\,, && \boldsymbol{c}_{1,3}(\lambda_2)\cdot\boldsymbol{d}_{1,3} = 0\,, && \boldsymbol{c}_{2,3}(\lambda_2)\cdot\boldsymbol{d}_{2,3} = e^{-\Psi(a_K^2)}\beta_{2,3}\,,\\
&\boldsymbol{c}_{1,2}(\lambda_3)\cdot\boldsymbol{d}_{1,2} = 0\,, && \boldsymbol{c}_{1,3}(\lambda_3)\cdot\boldsymbol{d}_{1,3} = -e^{-\Psi(a_K^3)}\beta_{1,3}\,, && \boldsymbol{c}_{2,3}(\lambda_3)\cdot\boldsymbol{d}_{2,3} = e^{-\Psi(a_K^3)}\beta_{2,3}\,.
\end{aligned}$$

This gives us, for instance,

$$\begin{aligned}
(\mathbf{A}_K^{\exp})_{1,1} &= -(\mathbf{A}_K)_{1,2}\boldsymbol{c}_{1,2}(\lambda_1)\cdot\boldsymbol{d}_{1,2}\,(\lambda_1(a_K^2)-\lambda_1(a_K^1))-\\
&\quad (\mathbf{A}_K)_{1,3}\boldsymbol{c}_{1,3}(\lambda_1)\cdot\boldsymbol{d}_{1,3}\,(\lambda_1(a_K^3)-\lambda_1(a_K^1))-\\
&\quad (\mathbf{A}_K)_{2,3}\boldsymbol{c}_{2,3}(\lambda_1)\cdot\boldsymbol{d}_{2,3}\,(\lambda_1(a_K^3)-\lambda_1(a_K^2))\\
&= e^{-\Psi(a_K^1)}(-(\mathbf{A}_K)_{1,2}\beta_{1,2}-(\mathbf{A}_K)_{1,3}\beta_{1,3})\,,\\
(\mathbf{A}_K^{\exp})_{1,2} &= -(\mathbf{A}_K)_{1,2}\boldsymbol{c}_{1,2}(\lambda_2)\cdot\boldsymbol{d}_{1,2}\,(\lambda_1(a_K^2)-\lambda_1(a_K^1))-\\
&\quad (\mathbf{A}_K)_{1,3}\boldsymbol{c}_{1,3}(\lambda_2)\cdot\boldsymbol{d}_{1,3}\,(\lambda_1(a_K^3)-\lambda_1(a_K^1))-\\
&\quad (\mathbf{A}_K)_{2,3}\boldsymbol{c}_{2,3}(\lambda_2)\cdot\boldsymbol{d}_{2,3}\,(\lambda_1(a_K^3)-\lambda_1(a_K^2))\\
&= e^{-\Psi(a_K^2)}\,(\mathbf{A}_K)_{1,2}\beta_{1,2}\,,\\
(\mathbf{A}_K^{\exp})_{1,3} &= -(\mathbf{A}_K)_{1,2}\boldsymbol{c}_{1,2}(\lambda_3)\cdot\boldsymbol{d}_{1,2}\,(\lambda_1(a_K^2)-\lambda_1(a_K^1))-\\
&\quad (\mathbf{A}_K)_{1,3}\boldsymbol{c}_{1,3}(\lambda_3)\cdot\boldsymbol{d}_{1,3}\,(\lambda_1(a_K^3)-\lambda_1(a_K^1))-\\
&\quad (\mathbf{A}_K)_{2,3}\boldsymbol{c}_{2,3}(\lambda_3)\cdot\boldsymbol{d}_{2,3}\,(\lambda_1(a_K^3)-\lambda_1(a_K^2))\\
&= e^{-\Psi(a_K^3)}\,(\mathbf{A}_K)_{1,3}\beta_{1,3}\,.\\
(\mathbf{A}_K^{\exp})_{2,1} &= -(\mathbf{A}_K)_{1,2}\boldsymbol{c}_{1,2}(\lambda_1)\cdot\boldsymbol{d}_{1,2}\,(\lambda_2(a_K^2)-\lambda_2(a_K^1))-\\
&\quad (\mathbf{A}_K)_{1,3}\boldsymbol{c}_{1,3}(\lambda_1)\cdot\boldsymbol{d}_{1,3}\,(\lambda_2(a_K^3)-\lambda_2(a_K^1))-\\
&\quad (\mathbf{A}_K)_{2,3}\boldsymbol{c}_{2,3}(\lambda_1)\cdot\boldsymbol{d}_{2,3}\,(\lambda_2(a_K^3)-\lambda_2(a_K^2))\\
&= e^{-\Psi(a_K^1)}\,(\mathbf{A}_K)_{1,2}\beta_{1,2}\,.\\
(\mathbf{A}_K^{\exp})_{2,2} &= -(\mathbf{A}_K)_{1,2}\boldsymbol{c}_{1,2}(\lambda_2)\cdot\boldsymbol{d}_{1,2}\,(\lambda_2(a_K^2)-\lambda_2(a_K^1))-\\
&\quad (\mathbf{A}_K)_{1,3}\boldsymbol{c}_{1,3}(\lambda_2)\cdot\boldsymbol{d}_{1,3}\,(\lambda_2(a_K^3)-\lambda_2(a_K^1))-\\
&\quad (\mathbf{A}_K)_{2,3}\boldsymbol{c}_{2,3}(\lambda_2)\cdot\boldsymbol{d}_{2,3}\,(\lambda_2(a_K^3)-\lambda_2(a_K^2))\\
&= e^{-\Psi(a_K^2)}(-(\mathbf{A}_K)_{1,2}\beta_{1,2}-(\mathbf{A}_K)_{2,3}\beta_{2,3})\,,
\end{aligned}$$

and the final result

$$\mathbf{A}_K^{\exp} = \begin{bmatrix} -(\mathbf{A}_K)_{1,2}\beta_{1,2}-(\mathbf{A}_K)_{1,3}\beta_{1,3} & (\mathbf{A}_K)_{1,2}\beta_{1,2} & (\mathbf{A}_K)_{1,3}\beta_{1,3}\\ (\mathbf{A}_K)_{1,2}\beta_{1,2} & -(\mathbf{A}_K)_{1,2}\beta_{1,2}-(\mathbf{A}_K)_{2,3}\beta_{2,3} & (\mathbf{A}_K)_{2,3}\beta_{2,3}\\ (\mathbf{A}_K)_{1,3}\beta_{1,3} & (\mathbf{A}_K)_{2,3}\beta_{2,3} & -(\mathbf{A}_K)_{1,3}\beta_{1,3}-(\mathbf{A}_K)_{2,3}\beta_{2,3}\end{bmatrix}\cdot \begin{bmatrix} e^{-\Psi(a_K^1)} & 0 & \\ 0 & e^{-\Psi(a_K^2)} & \\ 0 & 0 & e^{-\Psi(a_K^2)}\end{bmatrix}.$$

HINT 1 for (10-1.i): If ψ is linear on an edge $[\boldsymbol{a},\boldsymbol{b}]$ parameterized by $\gamma : [0,1] \to \mathbb{R}^2$, $\gamma(\xi) = (1-\xi)\boldsymbol{a} + \xi\boldsymbol{b}$, $0 \le \xi \le 1$, then

$$\left(\int_0^1 \exp(-\Psi(\gamma(\tau)))\,\mathrm{d}\tau\right)^{-1} = e^{\Psi_1} B(\Psi_1 - \Psi_2) = e^{\Psi_2} B(\Psi_2 - \Psi_1)\,, \tag{10.1.22}$$

where Ψ_1, Ψ_2 are the values of the potential at the endpoints of the edge. ⌟

SOLUTION of (10-1.i):

To derive (10.1.21) we compute

$$\int_0^1 \exp(\Psi_1(1-\tau)+\Psi_2\tau)\,\mathrm{d}\tau = e^{\Psi_1}\int_0^1 e^{\tau(\Psi_2-\Psi_1)}\,\mathrm{d}\tau = \frac{e^{\Psi_1}}{\Psi_2-\Psi_1}\left(e^{\Psi_2-\Psi_1}-1\right).$$

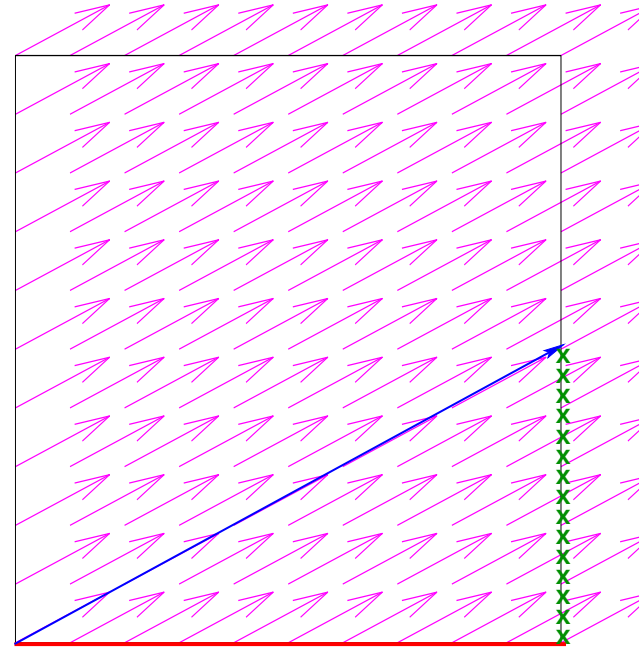
This means that

$$\beta_{j,i} = \left(\int_0^1 \exp(\Psi(\gamma_{j,i}(\tau)))\,\mathrm{d}\tau\right)^{-1} = e^{-\Psi(a_K^j)}B(\Psi(a_K^j)-\Psi(a_K^i)) = e^{-\Psi(a_K^i)}B(\Psi(a_K^i)-\Psi(a_K^j)).$$

Then simply plug (10.1.21) into (10.1.18), which can be inserted into (10.1.20).

SOLUTION of (10-10.a):

Boundary layers for $\epsilon \ll 1$ will emerge where the Dirichlet data, when transported with the flow induced by $\mathbf{v}$ do not match the Dirichlet data imposed on $\partial\Omega$.



The non-zero boundary data from the bottom side $[0,1] \times \{0\}$ are carried to the bottom half of the right side $\{1\} \times [0,1/2]$, where zero Dirichlet boundary conditions are imposed. There we have a mismatch of the transported and imposed boundary values.

The zero boundary condition from the right side $\{1\} \times [0,1]$ is carried to the top side $[0,1] \times \{1\}$ and the top half of the right side, where it meets another boundary zero condition: mismatch, no boundary layer.

Boundary layers will arise next to $\{1\} \times [0,1/2]$

HINT 1 for (10-10.b): Write the formula for $\kappa \equiv 0$, in case you cannot find the full solution. ⌟

SOLUTION of (10-10.b):

We apply the method of characteristics as explained in [Lecture → § 10.2.1.7].

We pick $\boldsymbol{x} \in \Omega$. Since the velocity field is constant, the streamlines are straight lines and the streamline through $\boldsymbol{x}$ is $\tau \mapsto \boldsymbol{x} + \tau \mathbf{v}$, τ in a suitable interval. Write $\boldsymbol{x}_0$ for the nearest point $\in \Gamma_{\text{in}}$ on the downstream part of the streamline and define

$$\varphi(\tau) := u(\boldsymbol{p}(\tau)) \,, \quad \boldsymbol{p}(\tau) := \boldsymbol{x}_0 + \tau \mathbf{v} \,, \quad \tau > 0 : \ \boldsymbol{x}_0 + \tau \mathbf{v} \in \Omega \,. \tag{10.10.5}$$

From the PDE in (10.10.4) we obtain by applying the *chain rule*,

$$\frac{d\varphi}{d\tau}(\tau) = \mathbf{grad}\, u(\boldsymbol{p}(\tau)) \cdot \mathbf{v} = -\kappa(\boldsymbol{p}(\tau))\varphi(\tau) \,. \tag{10.10.6}$$

The Dirichlet boundary condition for u implies

$$\varphi(0) = g(\boldsymbol{x}_0) \,. \tag{10.10.7}$$

Thus we have to solve an initial-value problem for a scalar linear ordinary equation, for which we know a closed-form solution

$$\varphi(\tau) = g(\boldsymbol{x}_0) \cdot \exp(-\int_0^\tau \kappa(\boldsymbol{p}(\xi))\, \mathrm{d}\xi) \,, \quad \tau > 0 \,, \quad \boldsymbol{p}(\tau) \in \Omega \,.$$

Expressed in terms of the solution u we find

$$u(\boldsymbol{x}) = g(\boldsymbol{x}_0) \cdot \exp(-\int_0^\tau \kappa(\boldsymbol{p}(\xi))\, \mathrm{d}\xi) \,, \quad \boldsymbol{x}_0 = \boldsymbol{x} - \tau \mathbf{v} \in \Gamma_{\text{in}} \,, \quad \tau > 0 \,. \tag{10.10.8}$$

In case the equation $\boldsymbol{x}_0 = \boldsymbol{x} - \tau \mathbf{v}$ has several solutions $(\boldsymbol{x}_0, \tau) \in \Gamma_{\text{in}} \times \mathbb{R}^+$, we choose the one with the smallest value of τ.

SOLUTION of (10-10.c):

The variational problem (10.10.9) can be rewritten as

$$u \in \widetilde{g} + V_0: \quad \mathsf{a}(u,w) = 0 \quad \forall w \in V_0 \,,$$

with a bilinear form

$$\mathsf{a}(u,w) := \int_\Omega (\mathbf{v} \cdot \mathbf{grad}\, u(\boldsymbol{x}) + \kappa(\boldsymbol{x})u(\boldsymbol{x}))(\mathbf{v} \cdot \mathbf{grad}\, w(\boldsymbol{x}) + \kappa(\boldsymbol{x})w(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} \,, \quad u,w \in V_0 + H^1(\Omega) \,. \tag{10.10.10}$$

Relying on the considerations of [Lecture → Section 1.4.1], recall, in particular, [Lecture → Thm. 1.4.1.8], we find $J(w) = \frac{1}{2}\mathsf{a}(w,w)$, which means

$$J(w) = \tfrac{1}{2}\int_\Omega (\mathbf{v} \cdot \mathbf{grad}\, w(\boldsymbol{x}) + \kappa(\boldsymbol{x})w(\boldsymbol{x}))^2\,\mathrm{d}\boldsymbol{x} \,.$$

HINT 1 for (10-10.d): The Dirichlet boundary conditions of (10.10.4) have to be taken into account in V_0. ⌟

SOLUTION of (10-10.d):

As in Sub-problem (10-10.c) we abbreviate the left-hand side bilinear form of (10.10.9) as a. The bilinear form a is positive semi-definite and induces an energy (semi-)norm

$$\|w\|_{\mathsf{a}}^2 := \int_\Omega (\mathbf{v}\cdot\mathbf{grad}\, w(\boldsymbol{x}) + \kappa(\boldsymbol{x})w(\boldsymbol{x}))^2\,\mathrm{d}\boldsymbol{x} = \|\mathbf{v}\cdot\mathbf{grad}\, w + \kappa w\|_{L^2(\Omega)}^2\,. \tag{10.10.11}$$

In the spirit of [Lecture → Section 1.3.1] this suggests a first *tentative* definition of a suitable function space

$$V := \left\{w \in L^2(\Omega):\ \|w\|_{\mathsf{a}} < \infty\right\} = \left\{w \in L^2(\Omega):\ \|\mathbf{v}\cdot\mathbf{grad}\, w + \kappa w\|_{L^2(\Omega)} < \infty\right\}.$$

However, this definition fails to meet the requirement that it must not depend on κ. To get rid of κ we recall the triangle inequality

$$\begin{aligned}\|\mathbf{v}\cdot\mathbf{grad}\, w + \kappa w\|_{L^2(\Omega)} &\le \|\mathbf{v}\cdot\mathbf{grad}\, w\|_{L^2(\Omega)} + \|\kappa w\|_{L^2(\Omega)}\\ &\le \|\mathbf{v}\cdot\mathbf{grad}\, w\|_{L^2(\Omega)} + \|\kappa\|_{L^\infty(\Omega)}\|w\|_{L^2(\Omega)}\,.\end{aligned}$$

Hence, a definition that does no longer depend on κ and still yields a space as large as possible is

$$V := \left\{w \in L^2(\Omega):\ \|\mathbf{v}\cdot\mathbf{grad}\, w\|_{L^2(\Omega)} < \infty\right\}\,. \tag{10.10.12}$$

Next, we have to incorporate the essential boundary conditions on Γ_{in}. Heeding the principle "Do not test where the solution is known", we demand that functions in V_0 vanish on Γ_{in}:

$$V_0 := \left\{w \in L^2(\Omega):\ \|\mathbf{v}\cdot\mathbf{grad}\, w\|_{L^2(\Omega)} < \infty,\ w|_{\Gamma_{\mathrm{in}}} = 0\right\}\,. \tag{10.10.13}$$

HINT 1 for (10-10.e): Employ integration by parts in the form of Green's first formula [Lecture → Thm. 1.5.2.7] and the general product rule [Lecture → Lemma 1.5.2.1]. ⌟

SOLUTION of (10-10.e):

Apply Green's first formula

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad \text{[Lecture → Eq. (12.2.3.2)]}$$

with $v := u$ and $\mathbf{j} := \mathbf{v}\kappa(\boldsymbol{x})u$ to obtain

$$\int_\Omega (\mathbf{v}\,\kappa u) \cdot \mathbf{grad}\, u \,\mathrm{d}\boldsymbol{x} = -\int_\Omega u \operatorname{div}(\mathbf{v}\,\kappa u) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u\, \mathbf{v}\,\kappa u \cdot \boldsymbol{n} \,\mathrm{d}S\,.$$

Then use

Lemma [Lecture → Lemma 1.5.2.1]. General product rule

For all $\mathbf{j} \in (C^1(\overline{\Omega}))^d$, $v \in C^1(\overline{\Omega})$ *holds*

$$\operatorname{div}(\mathbf{j}v) = v\ \operatorname{div} \mathbf{j} + \mathbf{j} \cdot \mathbf{grad}\, v \quad \textit{in every point of } \Omega\,. \qquad \text{[Lecture → Eq. (1.5.2.2)]}$$

to manipulate the first integral on the right-hand side:

$$\int_\Omega (\mathbf{v}\,\kappa u) \cdot \mathbf{grad}\, u \,\mathrm{d}\boldsymbol{x} = -\int_\Omega u\,\kappa\,\mathbf{v} \cdot \mathbf{grad}\, u + u^2 \operatorname{div}(\kappa\mathbf{v}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u\, \mathbf{v}\,\kappa u \cdot \boldsymbol{n} \,\mathrm{d}S\,.$$

We see that the initial integral has also emerged on the right-hand side and we move those terms to the left-hand side:

$$\begin{aligned} 2\int_\Omega (\mathbf{v}\,\kappa u) \cdot \mathbf{grad}\, u \,\mathrm{d}\boldsymbol{x} &= \int_\Omega u^2 \operatorname{div}(\kappa\mathbf{v}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u^2\, \kappa(\mathbf{v}\cdot\boldsymbol{n}) \,\mathrm{d}S \\ &= \int_\Omega u^2\, (\mathbf{grad}\,\kappa \cdot \mathbf{v}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u^2\, \kappa(\mathbf{v}\cdot\boldsymbol{n}) \,\mathrm{d}S\,. \end{aligned}$$

$$\begin{aligned} \blacktriangleright \qquad \int_\Omega (\mathbf{v}\,\kappa u) \cdot \mathbf{grad}\, u \,\mathrm{d}\boldsymbol{x} &= \int_\Omega u^2\ \tfrac{1}{2} \operatorname{div}(\kappa\mathbf{v})\ \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u^2\ \tfrac{1}{2}\kappa(\mathbf{v}\cdot\boldsymbol{n})\ \mathrm{d}S \\ &= \int_\Omega u^2\ \tfrac{1}{2}(\mathbf{grad}\,\kappa \cdot \mathbf{v})\ \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u^2\ \tfrac{1}{2}\kappa(\mathbf{v}\cdot\boldsymbol{n})\ \mathrm{d}S\,. \end{aligned}$$

Remark. If $\operatorname{div}(\kappa\mathbf{v}) \geq 0$ and $u \in V_0$ (as defined in (10.10.13)) then this identity means that

$$\begin{aligned} \mathsf{a}(u,u) &= \int_\Omega (\mathbf{v}\cdot\mathbf{grad}\, u + \kappa u)^2 \,\mathrm{d}\boldsymbol{x} \\ &= \|\mathbf{v}\cdot\mathbf{grad}\, u\|^2_{L^2(\Omega)} + \|\kappa u\|^2_{L^2(\Omega)} + 2\int_\Omega \mathbf{v}\cdot\mathbf{grad}\, u\,\kappa u \,\mathrm{d}\boldsymbol{x} \\ &\geq \|\mathbf{v}\cdot\mathbf{grad}\, u\|^2_{L^2(\Omega)} + \|\kappa u\|^2_{L^2(\Omega)}\,, \end{aligned}$$

which implies, if κ is uniformly positive,

$$\exists C = C(\kappa) > 0\colon \quad \mathsf{a}(u,u) \geq C\|u\|^2_{V_0^2} \quad \forall u \in V_0\,.$$

From this estimate we can infer quasi-optimality for finite element Galerkin solutions.

HINT 1 for (10-10.f): The implementation of **LSQAdvectionMatrixProvider** closely follows that of **lf::uscalfe::ReactionDiffusionElementMatrixProvider**, which is the **ENTITY_MATRIX_PROVIDER** type for Lagrangian finite element spaces and the bilinear form

$$\mathsf{a}(u,v) := \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1(\Omega)\,.$$

The coefficient functions $\boldsymbol{\alpha} = \boldsymbol{\alpha}(\boldsymbol{x})$ and $\gamma = \gamma(\boldsymbol{x})$ are stored in the **MeshFunction** data members `alpha_` and `gamma_`. The following code was the starting point for the implementation requested in this sub-problem, see also [Lecture → Code 2.8.3.31].

C++ code 10.10.15: Skeleton implementation of `Eval()` method for lf::uscalfe::ReactionDiffusionElementMatrixProvider. See also `uscalfe/loc_comp_ellbvp.h` of the LEHRFEM++ library.

```
template <base::Scalar SCALAR, mesh::utils::MeshFunction DIFF_COEFF,
          mesh::utils::MeshFunction REACTION_COEFF>
typename lf::uscalfe::ReactionDiffusionElementMatrixProvider<
    SCALAR, DIFF_COEFF, REACTION_COEFF>::ElemMat
ReactionDiffusionElementMatrixProvider<
    SCALAR, DIFF_COEFF, REACTION_COEFF>::Eval(const lf::mesh::Entity &cell) {
  const lf::base::RefEl ref_el{cell.RefEl()};
  const PrecomputedScalarReferenceFiniteElement<SCALAR> &pfe =
      fe_precomp_[ref_el.Id()];
  const lf::geometry::Geometry *geo_ptr = cell.Geometry();
  const dim_t world_dim = geo_ptr->DimGlobal();
  // Gram determinant at quadrature points
  const Eigen::VectorXd determinants(
      geo_ptr->IntegrationElement(pfe.Qr().Points()));
  // Fetch the transformation matrices for the gradients
  const Eigen::MatrixXd JinvT(
      geo_ptr->JacobianInverseGramian(pfe.Qr().Points()));
  // Element matrix
  ElemMat mat(pfe.NumRefShapeFunctions(), pfe.NumRefShapeFunctions());
  mat.setZero();
  // compute values of coefficients alpha, gamma at quadrature points:
  auto alphaval = alpha_(cell, pfe.Qr().Points());
  auto gammaval = gamma_(cell, pfe.Qr().Points());
  // Loop over quadrature points
  for (base::size_type k = 0; k < pfe.Qr().NumPoints(); ++k) {
    const double w = pfe.Qr().Weights()[k] * determinants[k];
    // Matrix of gradients of LSFs in mapped quadrature nodes
    const auto trf_grad(
        JinvT.block(0, 2 * static_cast<Eigen::Index>(k), world_dim, 2) *
        pfe.PrecompGradientsReferenceShapeFunctions()
            .block(0, 2 * k, mat.rows(), 2)
            .transpose());
    // Diffusion coefficients multiplied with gradients of LSFs
    const auto alpha_trf_grad(alphaval[k] * trf_grad);
    mat += w * (trf_grad.adjoint() * alpha_trf_grad +
                (gammaval[k] * pfe.PrecompReferenceShapeFunctions().col(k)) *
                    (pfe.PrecompReferenceShapeFunctions().col(k).adjoint()));
  }
  return mat;
}
```

SOLUTION of (10-10.f):

Our approach is a transformation technique along the lines of [Lecture → Section 2.8.3].

Pick a cell K with reference element $\widehat{K}$ and associated bijective mapping $\boldsymbol{\Phi}_K : \widehat{K} \to K$. Write b_K^j, $j = 1, \ldots,$ for the local shape functions on K, $\widehat{b}^j$, $j = 1, \ldots, Q$, for those on $\widehat{K}$, which are related according to

$$\widehat{b}^j = \boldsymbol{\Phi}_K^* b_K^j , \quad j = 1, \ldots, Q .$$

This distinguishes a parametric finite element space [Lecture → Def. 2.8.3.1]. All Lagrangian finite element spaces available in LEHRFEM++ belong to this class.

Here, the relevant bilinear form is

$$\mathsf{a}(u, w) := \int_\Omega (\mathbf{v} \cdot \mathbf{grad}\, u(\boldsymbol{x}) + \kappa(\boldsymbol{x}) u(\boldsymbol{x}))(\mathbf{v} \cdot \mathbf{grad}\, w(\boldsymbol{x}) + \kappa(\boldsymbol{x}) w(\boldsymbol{x}))\, \mathrm{d}\boldsymbol{x} . \tag{10.10.10}$$

Hence, the entries of the element matrix $\mathbf{A}_K \in \mathbb{R}^{Q,Q}$ for K are

$$(\mathbf{A})_{i,j} = \int_K (\mathbf{v} \cdot \mathbf{grad}\, b_K^j(\boldsymbol{x}) + \kappa(\boldsymbol{x}) b_K^j(\boldsymbol{x}))(\mathbf{v} \cdot \mathbf{grad}\, b_K^i(\boldsymbol{x}) + \kappa(\boldsymbol{x}) b_K^i(\boldsymbol{x}))\, \mathrm{d}\boldsymbol{x} , \quad i, j = 1, \ldots, Q . \tag{10.10.16}$$

As in [Lecture → § 2.8.3.6] we pull the integral back to the reference element $\widehat{K}$:

$$\begin{aligned} (\mathbf{A}_K)_{i,j} = \int_{\widehat{K}} & (\mathbf{v} \cdot (\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\boldsymbol{x}}) + \kappa(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})) \widehat{b}^j(\widehat{\boldsymbol{x}})) \cdot \\ & (\mathbf{v} \cdot (\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))^{-\top} \mathbf{grad}\, \widehat{b}^i(\widehat{\boldsymbol{x}}) + \kappa(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})) \widehat{b}^i(\widehat{\boldsymbol{x}})) \, |\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})| \, \mathrm{d}\widehat{\boldsymbol{x}} , \quad i, j = 1, \ldots, Q . \end{aligned}$$

Next, we apply a local p-point quadrature formula, $p \in \mathbb{N}$, on $\widehat{K}$,

$$\int_{\widehat{K}} \widehat{f}(\widehat{\boldsymbol{x}})\, \mathrm{d}\widehat{\boldsymbol{x}} \approx \sum_{\ell=1}^{p} \widehat{\omega}_\ell \widehat{f}(\widehat{\boldsymbol{\zeta}}_\ell) , \quad \widehat{\omega}_\ell \in \mathbb{R} , \quad \widehat{\boldsymbol{\zeta}}_\ell \in \widehat{K} . \tag{10.10.17}$$

This yields approximate entries of the element matrix

$$\begin{aligned} (\mathbf{A}_K)_{i,j} \approx \sum_{\ell=1}^{p} \widehat{\omega}_\ell & (\mathbf{v} \cdot (\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell) + \kappa(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)) \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)) \cdot \\ & (\mathbf{v} \cdot (\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))^{-\top} \mathbf{grad}\, \widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell) + \kappa(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)) \widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell)) \, |\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)| \end{aligned} \tag{10.10.18}$$

The idea behind the implementation can be conveyed by a more compact notation. Write

$$\mathbf{G}_\ell = \left[\mathbf{grad}\, b_K^j(\boldsymbol{\zeta}_\ell) \right]_{j=1}^{Q} = \left[(\mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell) \right]_{j=1}^{Q} \in \mathbb{R}^{2,Q} \tag{10.10.19}$$

for a matrix whose columns store the gradients $\mathbf{grad}\, b_K^j$ evaluated in a (mapped) quadrature node $\boldsymbol{\zeta}_\ell := \boldsymbol{\Phi}_K$
In addition, abbreviate

$$\boldsymbol{k}_\ell := \left[\kappa(\boldsymbol{\zeta}_\ell)\, b_K^j(\boldsymbol{\zeta}_\ell)) \right]_{j=1}^{Q} = \left[\kappa(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\, \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell) \right]_{j=1}^{Q} \in \mathbb{R}^Q . \tag{10.10.20}$$

By elementary linear algebra we find

$$\mathbf{A}_K = \sum_{\ell=1}^{p} \nu_\ell \boldsymbol{z}_\ell^\top \boldsymbol{z}_\ell , \quad \boldsymbol{z}_\ell := \mathbf{v}^\top \mathbf{G}_\ell + \boldsymbol{k}_\ell^\top , \quad \nu_\ell := \widehat{\omega}_\ell |\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)| . \tag{10.10.21}$$

C++ code 10.10.22: Implementation of `Eval()` member function for LSQAdvectionMatrixProvider

```
template <lf::mesh::utils::MeshFunction REACTION_COEFF>
LSQAdvectionMatrixProvider<REACTION_COEFF>::ElemMat
LSQAdvectionMatrixProvider<REACTION_COEFF>::Eval(const lf::mesh::Entity &cell) {
  // Topological type of the cell
  const lf::base::RefEl ref_el{cell.RefEl()};
  // Obtain precomputed information about values of local shape functions
  // and their gradients at quadrature points.
  const lf::uscalfe::PrecomputedScalarReferenceFiniteElement<double> &pfe =
      fe_precomp_[ref_el.Id()];
  LF_ASSERT_MSG(pfe.isInitialized(), "Precomputed data missing!");
  // Query the shape of the cell
  const lf::geometry::Geometry *geo_ptr = cell.Geometry();
  LF_ASSERT_MSG(geo_ptr != nullptr, "Invalid geometry!");
  LF_ASSERT_MSG((geo_ptr->DimLocal() == 2),
                "Only 2D implementation available!");
  // Physical dimension of the cell
  const lf::base::dim_t world_dim = geo_ptr->DimGlobal();
  // Gram determinant at quadrature points
  const Eigen::VectorXd determinants(
      geo_ptr->IntegrationElement(pfe.Qr().Points()));
  LF_ASSERT_MSG(
      determinants.size() == pfe.Qr().NumPoints(),
      "Mismatch " << determinants.size() << " <-> " << pfe.Qr().NumPoints());
  // Fetch the transformation matrices for the gradients
  const Eigen::MatrixXd JinvT(
      geo_ptr->JacobianInverseGramian(pfe.Qr().Points()));
  LF_ASSERT_MSG(
      JinvT.cols() == 2 * pfe.Qr().NumPoints(),
      "Mismatch " << JinvT.cols() << " <-> " << 2 * pfe.Qr().NumPoints());
  LF_ASSERT_MSG(JinvT.rows() == world_dim,
                "Mismatch " << JinvT.rows() << " <-> " << world_dim);
  // Element matrix
  ElemMat mat(pfe.NumRefShapeFunctions(), pfe.NumRefShapeFunctions());
  mat.setZero();
  // Compute values of κ in the quadrature nodes
  // Φ_K(ζ̂_ℓ).
  auto kappaval = kappa_(cell, pfe.Qr().Points());
  // Loop over quadrature points
  for (lf::base::size_type k = 0; k < pfe.Qr().NumPoints(); ++k) {
    // The weighting factors ν_ℓ
    const double w = pfe.Qr().Weights()[k] * determinants[k];
    // Transformed gradients in G_ℓ, see (10.10.19).
    const auto trf_grad(
        JinvT.block(0, 2 * static_cast<Eigen::Index>(k), world_dim, 2) *
        pfe.PrecompGradientsReferenceShapeFunctions()
            .block(0, 2 * k, mat.rows(), 2)
            .transpose());
    // Shape functions multiplied with kappa, vectors k_ℓ,
    // (10.10.20)
    const auto kvec = kappaval[k] * pfe.PrecompReferenceShapeFunctions().col(k);
    // Transformed gradients multiplied with velocity,
    // v^⊤ G_ℓ
    const auto vTG(velocity_.transpose() * trf_grad);
    // Realization of (10.10.21).
```

```
    const auto zvec = vTG + kvec.transpose();
    mat += w * zvec.transpose() * zvec;
  }
  return mat;
}
```

HINT 1 for (10-10.g): The auxiliary function

```
lf::mesh::utils::AllCodimMeshDataSet<bool> flagEntitiesOnInflow(
    const std::shared_ptr<const lf::mesh::Mesh> &mesh_p,
    Eigen::Vector2d velocity);
```

creates an array of flags that are set to **true** only for mesh entities (edges and nodes) contained in the inflow boundary part $\overline{\Gamma}_{\mathrm{in}}$. ⌟

HINT 2 for (10-10.g): To obtain the basis expansion coefficient vector of a suitable $\widetilde{g}_h$ rely on the LEHRFEM++ utility function `lf::fe::NodalProjection()`. ⌟

SOLUTION of (10-10.g):

We follow the policy outlined in [Lecture → Section 2.7.6] using LEHRFEM++'s utility functions `fixFlagge` or `fixFlaggedSolutionCompAlt()`. To that end we have to devise a functor that

- flags all finite-element basis functions associated with mesh entities contained in $\overline{\Gamma}_{\mathrm{in}}$,
- and also provides their associated basis expansion coefficients of the nodal interpolant of $\widetilde{g}$.

The latter coefficients can be computed by calling `NodalProjection()`.

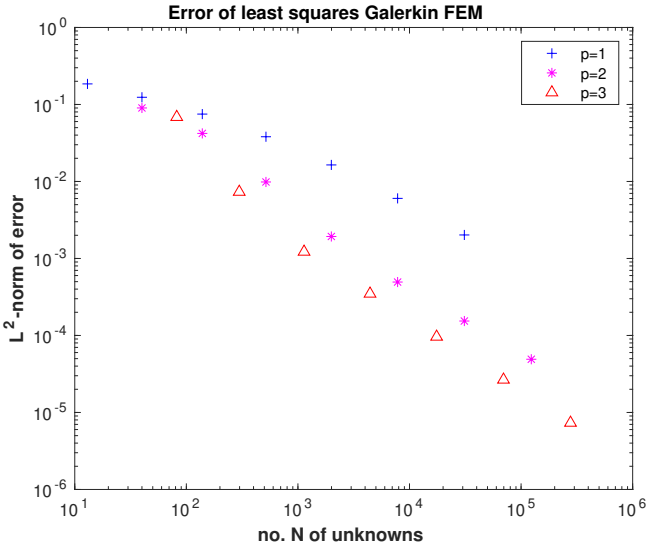
Details on how to use these functions are given in [Lecture → § 2.7.6.13].

C++ code 10.10.23: Complete code of `solveAdvectionDirichletBVP()`

```
2  template <lf::mesh::utils::MeshFunction REACTION_COEFF,
3            lf::mesh::utils::MeshFunction GFUNCTION>
4  Eigen::VectorXd solveAdvectionDirichletBVP(
5      std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>> fe_space,
6      Eigen::Vector2d velocity, const REACTION_COEFF &kappa, const GFUNCTION &g) {
7    // I. Assemble the full Galerkin matrix for the least squares
       variational
8    // formulation Fetch DofHandler
9    const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
10   const size_t N = dofh.NumDofs();
11   // Sparse matrix in triplet format
12   lf::assemble::COOMatrix<double> A_coo(N, N);
13   // Set up ENTITY_MATRIX_PROVIDER
14   LeastSquaresAdvection::LSQAdvectionMatrixProvider lsq_adv_emp(
15       fe_space, velocity, kappa);
16   // Assembly of Galerkin matrix
17   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, lsq_adv_emp, A_coo);
18
19   // II. Enforce Dirichlet boundary conditions on inflow boundary
20   // as in [Lecture → Section 2.7.6]
21   const lf::mesh::utils::AllCodimMeshDataSet<bool> inflow_flags{
22       flagEntitiesOnInflow(fe_space->Mesh(), velocity)};
23   // Right-hand side vector
24   Eigen::VectorXd phi(N);
25   phi.setZero();
26   // Interpolate extended boundary data to obtain g̃_h
27   const Eigen::VectorXd gt_vec = lf::fe::NodalProjection(*fe_space, g);
28   // Eliminate degrees of freedom on the boundary using
29   // LEHRFEM++'s built-in function, see [Lecture → § 2.7.6.13]
30   lf::assemble::FixFlaggedSolutionComponents<double>(
31       [&inflow_flags, &dofh,
32        &gt_vec](lf::assemble::glb_idx_t dof_idx) -> std::pair<bool, double> {
33         const bool on_inflow = inflow_flags(dofh.Entity(dof_idx));
34         if (on_inflow) {
35           return {true, gt_vec[dof_idx]};
36         }
37         return {false, 0.0};
38       },
39       A_coo, phi);
40   // III. Solving linear system of equations using
41   // EIGEN's sparse direct elimination solver
42   Eigen::SparseMatrix<double> A_crs{A_coo.makeSparse()};
```

```
43    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
44    solver.compute(A_crs);
45    Eigen::VectorXd result = Eigen::VectorXd::Constant(N, 3.0);
46    if (solver.info() != Eigen::Success) {
47      throw std::runtime_error("LU decomposition failed");
48    }
49    result = solver.solve(phi);
50    if (solver.info() != Eigen::Success) {
51      throw std::runtime_error("Solving LSE failed");
52    }
53    return result;
54  }
```

SOLUTION of (10-10.h):

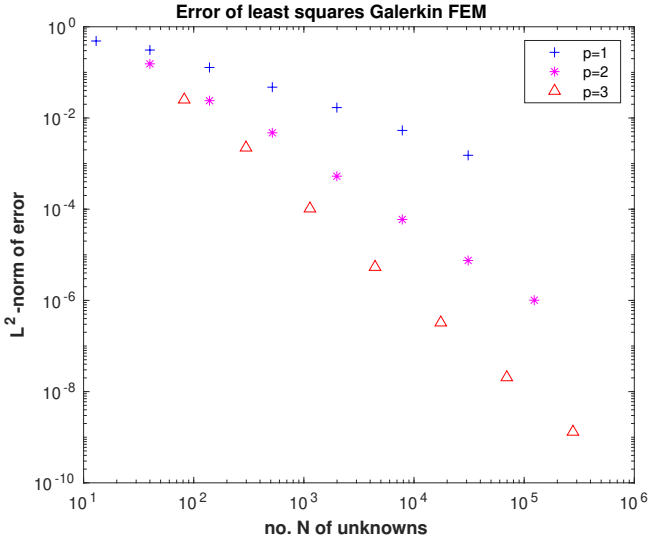


Setting (I) for advection-reaction BVP:

- $\Omega =]0,1[^2$ (unit square domain),
- $\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, $\kappa \equiv 1$
- Boundary data $g : \partial\Omega \to \mathbb{R}$:

$$g(\boldsymbol{x}) = \begin{cases} \sin^2(\pi x_1) & \text{for} \quad x_2 = 0\,, \\ 0 & \text{elsewhere on } \partial\Omega\,. \end{cases}$$

◁ (Approximate))$L^2(\Omega)$-norms of the discretization errors for Lagrangian FEM with $p = 1,2,3$.

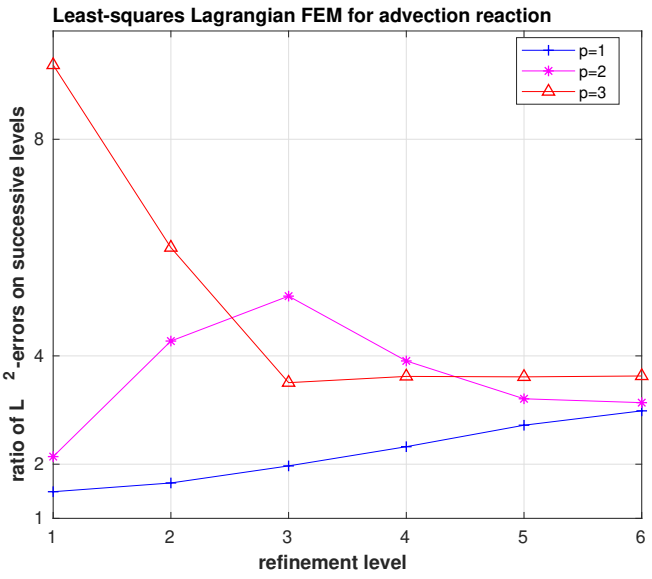


(II) $\kappa \equiv 0$, $\mathbf{v} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}$, *smooth* manufactured solution

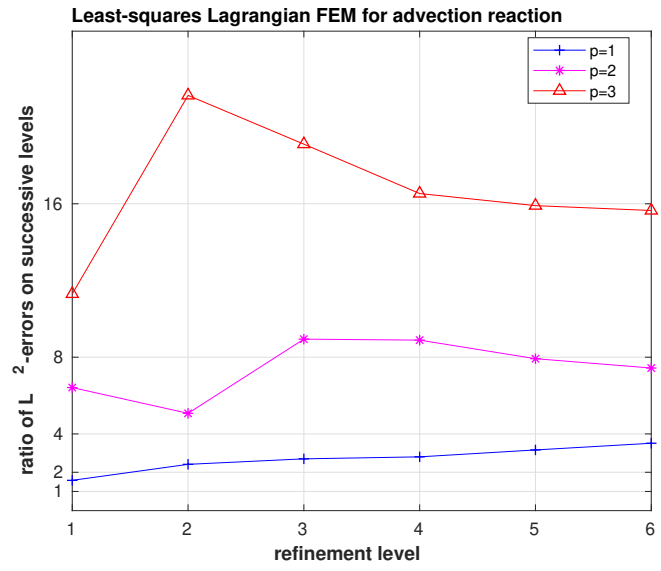
$$u(\boldsymbol{x}) = \cos(\mathbf{v}^\perp \cdot \boldsymbol{x})\,, \quad \boldsymbol{x} \in \Omega\,.$$

(Approximate))$L^2(\Omega)$-norms of the discretization errors for Lagrangian FEM with $p = 1,2,3$. ▷

The following plots display the ratios of the $L^2(\Omega)$-norms of the finite element discretization errors on successive meshes in the refinement hierarchy.



Setting (I)



Setting (II)

For both settings the error points in a *doubly logarithmic plot* are roughly located on straight lines, which hints at asymptotic algebraic convergence [Lecture → § 3.2.2.5].

- Setting (II), smooth solution: The ratios of the $L^2(\Omega)$-norm of the discretization errors on pairs of meshes arising from uniform regular refinement approach 2^{p+1} on fine meshes. Since regular refinement means halving the mesh width $h_{\mathcal{M}}$, we conclude an asymptotic algebraic convergence with rate $p+1$:

$$\|u - u_h\|_{L^2(\Omega)} = O(h_{\mathcal{M}}^{p+1}) \quad \text{for} \quad h_{\mathcal{M}} \to 0\,. \tag{10.10.24}$$

  This behavior of the discretization error matches the asymptotic behavior of the best approximation errors

$$\inf_{w_h \in \mathcal{S}_p^0(\mathcal{M})} \|u - w_h\|_{L^2(\Omega)} \quad \text{for} \quad h_{\mathcal{M}} \to 0\,.$$

  Thus (10.10.24) reflects the quasi-optimality of the finite element Galerkin solution, that is, the discretization error is proportional to the best approximation error.

- Setting (I): Regardless of the polynomial degree p, the rate of asymptotic algebraic convergence (in terms of mesh width $h_{\mathcal{M}}$) seems to edge towards 2 for $h_{\mathcal{M}} \to 0$. Increasing the polynomial degree does not buy us faster convergence!

  The reason is that the exact solution according to (10.10.8) is *not smooth*, because it has a kink along the line $\mathbf{v}\xi$, $\xi \in \mathbb{R}$; the corner of $\partial\Omega$ "propagates into the domain".

HINT 1 for (10-2.a): The velocity fields $\mathbf{v}$ for incompressible fluid flows satisfy $\operatorname{div} \mathbf{v}(\mathbf{x}) = 0$ everywhere, see [Lecture → Thm. 10.1.3.8].

SOLUTION of (10-2.a):

The verification of $\operatorname{div} \mathbf{v} = 0$ can be done by direct computation

$$\operatorname{div}\ \mathbf{v}(\boldsymbol{x}) = -\frac{\partial\{\boldsymbol{x} \mapsto x_2\}}{\partial x_1} + \frac{\partial\{\boldsymbol{x} \mapsto x_1\}}{\partial x_2} = 0\ .$$

HINT 1 for (10-2.b): Remember that the velocity is always tangent to the streamlines. ⌟

SOLUTION of (10-2.b):

In this case the streamline ODE reads

$$\dot{\mathbf{y}} = \mathbf{v}(\mathbf{y}) = \begin{bmatrix} -y_2 \\ y_1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} \,.$$

This is an autonomous linear ODE according to [Lecture → Def. 6.1.1.9] of the form $\dot{\mathbf{y}} = \mathbf{A}\mathbf{y}$, which can be solved by a diagonalization technique as explained in [Lecture → § 6.1.1.8]. The characteristic polynomial of the skew-symmetric matrix $\mathbf{A}$ is $\chi(\tau) = \tau^2 + 1$ with roots $\pm\imath$, and we find

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ -\imath & \imath \end{bmatrix} \begin{bmatrix} \imath & 0 \\ 0 & -\imath \end{bmatrix} \begin{bmatrix} \frac{1}{2} & \frac{1}{2}\imath \\ \frac{1}{2} & -\frac{1}{2}\imath \end{bmatrix} =: \mathbf{S}\mathbf{D}\mathbf{S}^{-1} \,.$$

According to [Lecture → Eq. (6.1.1.14)] the solution of the initial-value problem $\dot{\mathbf{y}} = \mathbf{A}\mathbf{y}$, $\mathbf{y}(0) = \mathbf{y}_0$ is given by

$$\mathbf{y}(t) = \mathbf{S}\exp(\mathbf{D}t)\mathbf{S}^{-1}\mathbf{y}_0 \,, \quad t \in \mathbb{R} \,,$$

which implies the flow map

$$\mathbf{\Phi}^t\mathbf{y} = \mathbf{S}\exp(\mathbf{D}t)\mathbf{S}^{-1}\mathbf{y} = \begin{bmatrix} 1 & 1 \\ -\imath & \imath \end{bmatrix} \begin{bmatrix} e^{\imath t} & 0 \\ 0 & e^{-\imath t} \end{bmatrix} \begin{bmatrix} \frac{1}{2} & \frac{1}{2}\imath \\ \frac{1}{2} & -\frac{1}{2}\imath \end{bmatrix} = \begin{bmatrix} \cos(t) & -\sin(t) \\ \sin(t) & \cos(t) \end{bmatrix} \mathbf{y} \,,$$

which follows from Euler's formula $e^{\imath t} = \cos(t) + \imath\sin(t)$. This flow encodes counter-clockwise rigid rotations around the origin.

Remark. The streamlines of the flow are concentric circles around $\mathbf{0}$.

HINT 1 for (10-2.c): Reading [Lecture → Section 10.2.1] might offer crucial clues. Employ the method of characteristics from [Lecture → § 10.2.1.7] to find the exact solution of the boundary value problem for $\epsilon = 0$. ⌟

SOLUTION of (10-2.c):

From Sub-problem (10-2.b) we know that for $\epsilon = 0$ we face pure transport along streamlines that are concentric circles around $\mathbf{0}$: The flow map describes a rotation with constant angular velocity 1.

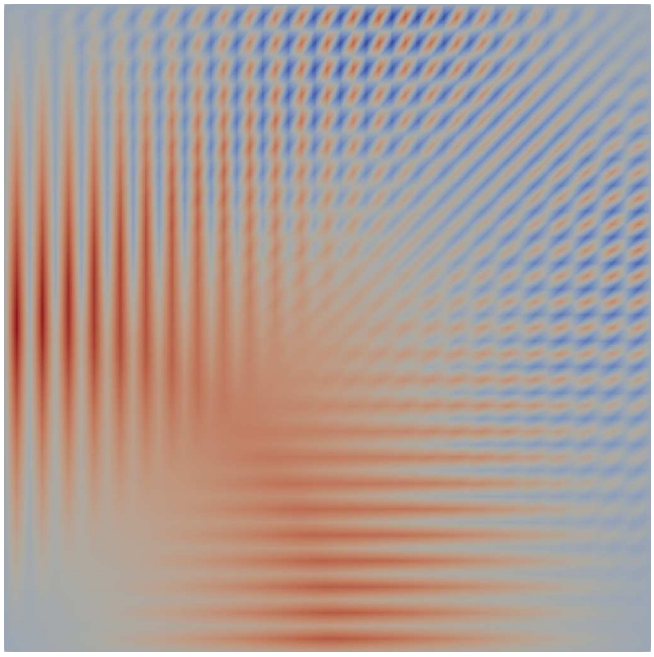
Hence, by the method of characteristics, see formula [Lecture $\rightarrow$ Eq. (10.2.1.8)], the boundary data are merely transported into the domain along concentric circles. Thus the solution is

$$u(\boldsymbol{x}) = \begin{cases} g(\|\boldsymbol{x}\|, 0) & \text{for} \quad \boldsymbol{x} \in \Omega\,, \quad \|\boldsymbol{x}\| < 1\,, \\ 0 & \text{for} \quad \|\boldsymbol{x}\| \geq 1\,, \end{cases} \tag{10.2.3}$$

since we can choose $(x_1, x_2) = (\|\boldsymbol{x}\|, 0)$ as starting point of a streamline on $\partial\Omega$ and there we know the value of u from the Dirichlet boundary data. However, the formula (10.2.3) clashes with the boundary condition $u(\boldsymbol{x}) = 0$ for $x_1 = 0$.

Therefore, for small ϵ, a boundary layer will emerge at the x_2-axis.

SOLUTION of (10-2.d):



◁ Plot of the solution obtained by standard finite-element Galerkin discretization of the boundary-value problem (10.2.1).

We observe that the solution shows strong *oscillations* in the direction of the streamlines.

This is the typical failure of the Galerkin FEM in the case of a singularly perturbed convection-diffusion problem, *cf.* [Lecture → Exp. 10.2.2.12].

SOLUTION of (10-2.f):

We obtain the entries of $\mathbf{B}$ by summing up local contributions based on (10.2.4). For the test function $w_h := b_h^j$, due to the cardinal basis property [Lecture $\rightarrow$ Eq. (2.4.3.5)], the sum just covers the cells adjacent to $\boldsymbol{a}^j$:

$$(\mathbf{B})_{j,i} = \sum_K \frac{|K|}{3} \sum_{\mathbf{a}^l \in K} \Big(\mathbf{v}(\mathbf{a}^l) \cdot (\mathbf{grad}\, b_h^i)^+(\mathbf{a}^l) \Big) b_h^j(\mathbf{a}^l) \,.$$

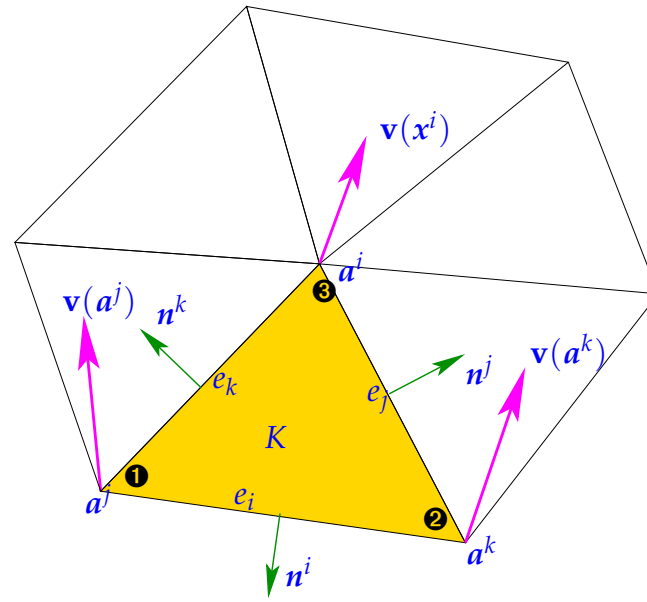
Since $b_h^j(\mathbf{a}^l) = \delta_{j,l}$, we can simplify to

$$(\mathbf{B})_{j,i} = \sum_{\mathbf{a}^j \in K} \frac{|K|}{3} \mathbf{v}(\mathbf{a}^j) \cdot (\mathbf{grad}\, b_h^i)^+(\mathbf{a}^j) = m(\mathbf{a}^j) \Big(\mathbf{v}(\mathbf{a}^j) \cdot (\mathbf{grad}\, b_h^i)^+(\mathbf{a}^j) \Big) \,.$$

HINT 1 for (10-2.g): Note that $\mathbf{B}_K$ is not symmetric, so take extra care in getting the matrix right! ⌟

SOLUTION of (10-2.g):

Let us start with the situation sketched in Fig. 149 as an example. Assume that the numbering of mesh nodes $\in \mathcal{V}(\mathcal{M})$ corresponds to the numbering of the basis functions of $\mathcal{S}_1^0(\mathcal{M})$. The local numbering of the vertices of K is given in the figure, too.



By (10.2.5) the i-th row of $\mathbf{B}$ has only three non-zero entries at positions i, j, and k:

$$(\mathbf{B})_{i,i} = m(a^i)\mathbf{v}(a^i) \cdot \mathbf{grad}\, b_h^i \,,$$
$$(\mathbf{B})_{i,k} = m(a^i)\mathbf{v}(a^i) \cdot \mathbf{grad}\, b_h^k \,,$$
$$(\mathbf{B})_{i,j} = m(a^i)\mathbf{v}(a^i) \cdot \mathbf{grad}\, b_h^j \,,.$$

How can this row of $\mathbf{B}$ arise from cell-oriented assembly as implemented in LEHRFEM++? This can be accomplished by

- initializing the third row of the element matrix $\mathbf{B}_K$ to

$$(\mathbf{B})_{3,:} = m(a^i) \begin{bmatrix} \mathbf{v}(a^i) \cdot \mathbf{grad}\, b_h^j & \mathbf{v}(a^i) \cdot \mathbf{grad}\, b_h^k & \mathbf{v}(a^i) \cdot \mathbf{grad}\, b_h^i \end{bmatrix} ,$$

- and by setting the a^i-related rows of the element matrices of all other cells abutting a^i to zero.

Summing up, only K is used to assembly the i-th row of $\mathbf{B}$ in this case.

The entries of the i-th row of $\mathbf{B}$ are contributed *solely* by the upstream triangle w.r.t. a^i!

Writing a_K^1, a_K^2, a_K^3 for the vertices of K and $\lambda_1, \lambda_2, \lambda_3$ for the associated barycentric coordinate functions, see [Lecture → § 2.4.5.2], the strategy for computing the element matrix $\mathbf{B}_K \in \mathbb{R}^{3,3}$ is the following:

1. Find the local index $\ell \in \{1,2,3\}$ such that $-\mathbf{v}(a^\ell)$ points *into* the triangle K. To that end you may compute the inner products of $-\mathbf{v}(\mathbf{a}^j)$ with exterior normals n^1, n^2, n^3 for the edges of K:

$$K \text{ upstream of } a_K^\ell \;\Leftrightarrow\; \operatorname{sign}\left(\mathbf{v}(a_K^\ell) \cdot n^j\right) = \begin{cases} -1 & \text{for } \ell = j \,, \\ 1 & \text{for } \ell \neq j \,. \end{cases} \tag{10.2.7}$$

   According to [Lecture → Eq. (2.4.5.4)] those are accessible through gradients of barycentric coordinate functions λ_j, $j = 1,2,3$:

$$n^j = -\frac{\mathbf{grad}\,\lambda_j}{\|\mathbf{grad}\,\lambda_j\|} \,.$$

   Hence, we can equivalently state (10.2.7) as

$$K \text{ upstream of } a_K^\ell \;\Leftrightarrow\; \operatorname{sign}\left(\mathbf{v}(a_K^\ell) \cdot \mathbf{grad}\,\lambda_j\right) = \begin{cases} 1 & \text{for } \ell = j \,, \\ -1 & \text{for } \ell \neq j \,. \end{cases} \tag{10.2.8}$$

The gradients of barycentric coordinate function can be computed as demonstrated in [Lecture → Code 2.4.5.11].

2. Taking the cue from (10.2.5) evaluate

$$(\mathbf{B}_K)_{\ell,i} := m(\boldsymbol{a}^\ell)\Big(\mathbf{v}(\boldsymbol{a}^\ell)\,\mathbf{grad}\,\lambda_i\Big)\,, \quad i = 1,2,3\,. \tag{10.2.9}$$

3. Set all other entries of $\mathbf{B}_K$ to zero.

Note that if $-\mathbf{v}(\boldsymbol{a}^j)$ is parallel to an edge, we have to divide the corresponding cell contributions by 2, because an adjacent cell will also contribute (the same) values during assembly.

C++ code 10.2.10: Sub-problem (10-2.g): Implementation of the `Eval()` member function of UpwindConvectionElementMatrixProvider

```
template <typename FUNCTOR>
Eigen::Matrix3d UpwindConvectionElementMatrixProvider<FUNCTOR>::Eval(
    const lf::mesh::Entity &entity) {
  LF_ASSERT_MSG(lf::base::RefEl::kTria() == entity.RefEl(),
                "Function only defined for triangular cells");

  const lf::geometry::Geometry *geo_ptr = entity.Geometry();
  const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_ptr);
  const double area = lf::geometry::Volume(*geo_ptr);
  Eigen::Matrix3d loc_mat;

  // calculate the gradients of the basis functions.
  // See [Lecture → Code 2.4.5.11], [Lecture → Code 2.4.5.13] for
     details.
  Eigen::Matrix3d grad_helper;
  grad_helper.col(0) = Eigen::Vector3d::Ones();
  grad_helper.rightCols(2) = corners.transpose();
  // Matrix with gradients of the local shape functions in its columns
  const Eigen::MatrixXd grad_basis = grad_helper.inverse().bottomRows(2);

  // masses of the corners.
  std::vector<double> local_masses;
  for (const lf::mesh::Entity *sub_ent : entity.SubEntities(2)) {
    local_masses.push_back(masses_(*sub_ent));
  }

  // Compute velocities at corners
  Eigen::MatrixXd velocities(2, 3);
  velocities << v_(corners.col(0)), v_(corners.col(1)), v_(corners.col(2));

  // Matrix of outer normals(not normalized)
  Eigen::MatrixXd n = -grad_basis;

  // compute rows of the local matrix according to the upwind quadrature
     scheme.
  // -v(a^j) points into the triangle K
  // iff the inner product of v(a^j) with the two adjecant outer normals
     is
  // positive.
  // check direction at the first corner of the triangle
  Eigen::Vector2d v0 = velocities.col(0);
  if (v0.dot(n.col(1)) >= 0 && v0.dot(n.col(2)) >= 0) {
    if (v0.dot(n.col(1)) == 0 || v0.dot(n.col(2)) == 0) {
```

```
      loc_mat.row(0) =
          0.5 * local_masses[0] * velocities.transpose().row(0) * grad_basis;
    } else {
      loc_mat.row(0) =
          local_masses[0] * velocities.transpose().row(0) * grad_basis;
    }
  } else {
    loc_mat.row(0) = Eigen::Vector3d::Zero();
  }

  // check direction at the second corner of the triangle.
  Eigen::Vector2d v1 = velocities.col(1);
  if (v1.dot(n.col(0)) >= 0 && v1.dot(n.col(2)) >= 0) {
    if (v1.dot(n.col(0)) == 0 || v1.dot(n.col(2)) == 0) {
      loc_mat.row(1) =
          0.5 * local_masses[1] * velocities.transpose().row(1) * grad_basis;
    } else {
      loc_mat.row(1) =
          local_masses[1] * velocities.transpose().row(1) * grad_basis;
    }
  } else {
    loc_mat.row(1) = Eigen::Vector3d::Zero();
  }

  // check direction at the third corner of the triangle.
  Eigen::Vector2d v2 = velocities.col(2);
  if (v2.dot(n.col(0)) >= 0 && v2.dot(n.col(1)) >= 0) {
    if (v2.dot(n.col(0)) == 0 || v2.dot(n.col(1)) == 0) {
      loc_mat.row(2) =
          0.5 * local_masses[2] * velocities.transpose().row(2) * grad_basis;
    } else {
      loc_mat.row(2) =
          local_masses[2] * velocities.transpose().row(2) * grad_basis;
    }
  } else {
    loc_mat.row(2) = Eigen::Vector3d::Zero();
  }
  return loc_mat;
}
```

Remark. The computation of the gradients of the barycentric coordinate functions will be affected by roundoff errors, which may impact the signs of the inner products $\mathbf{v}(a^i) \cdot \mathbf{grad}\, \lambda_j$ and, thus, the decision whether a the current cell is lying upstream with respect to a vertex. In extreme cases this may make the code behave as if there was no upstream cell for some nodes of the mesh. The corresponding rows of the matrix $\mathbf{B}$ will then contain all zeros!

To avoid this, the following strategy should be implemented: Before assembly initialize an **lf::mesh::utils::C** object of type **CodimMeshDataSet<const lf::mesh::Entity *>**, which, for *every* node of the mesh, except for those on the inflow boundary, contains a pointer to *an* associated upstream cell. To fill this container, loop over all cells and test whether it lies upstream to one of its vertices. Do the test in a way that will avoid failure to detect an upstream situation even when roundoff errors are taken into account.

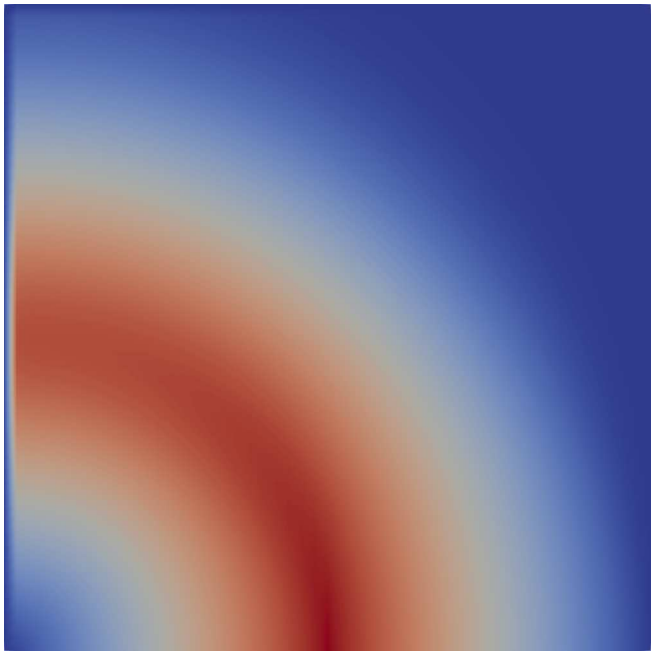
A reference to that **lf::mesh::utils::CodimMeshDataSet** object is then made available to **UpwindConvectionElementMatrixProvider** and queried in each invocation of `Eval()`.

HINT 1 for (10-2.h): Since we have the **ENTITY_MATRIX_PROVIDER** class **UpwindConvectionElementMatrixProvider** at our disposal, we can use the LEHRFEM++ library function `lf::assemble::Ass`
Consult [Lecture → Code 2.7.6.17] on how to use this function in the context of a finite-element discretization of a Dirichlet problem. Also consider using the LEHRFEM++ function `lf::fe::InitEssentialCo`
⌟

HINT 2 for (10-2.h): In order to build the Galerkin matrix for the second-order operator $-\epsilon\Delta$ you can rely on the **ENTITY_MATRIX_PROVIDER** class **lf::uscalfe::ReactionDiffusionElementMatrixProvider**. You may also copy the provider classes from [Lecture → Ex. 2.7.4.33], [Lecture → Code 2.7.4.35], [Lecture → Code 2.7.4.37]. ⌟

SOLUTION of (10-2.h):

After



◁ Plot of the finite-element solution computed by means of the upwind quadrature scheme.

The improvement compared to Fig. 148 is striking: the spurious oscillations have disappeared and the boundary layer is nicely confined to a small neighborhood of the x_2-axis.
The code seems to be able to handle $\epsilon = 0$ without problems, which reflects the robustness of the upwind quadrature scheme with respect to the singular perturbation limit $\epsilon \to 0$.

C++ code 10.2.11: Sub-problem (10-2.g): Solving convection-diffusion BVP based on upwind quadrature scheme.

```
2   // Matrix in triplet format holding Galerkin matrix, zero initially.
3   lf::assemble::COOMatrix<double> A_stable(dofh.NumDofs(), dofh.NumDofs());
4
5   // ASSEMBLE GALERKIN MATRIX
6   // First the part corresponding to the laplacian, computed using
        standard
7   // Galerkin approach
8   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, laplacian_provider,
9                                       A_stable);
10
11  // Next part corresponding to the convection term, computed using
        upwind
12  // quadrature:
13  UpwindQuadrature::UpwindConvectionElementMatrixProvider
14      convection_provider_stable(v, UpwindQuadrature::initializeMasses(mesh_p));
15  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, convection_provider_stable,
16                                      A_stable);
17
18  // RIGHT-HAND SIDE VECTOR
19  Eigen::VectorXd phi_stable(dofh.NumDofs());
20  phi_stable.setZero();
21
22  // IMPOSE DIRICHLET CONDITIONS:
23  // Eliminate Dirichlet dofs from linear system
24  lf::assemble::FixFlaggedSolutionComponents<double>(
25      [&ess_bdc_flags_values](lf::uscalfe::glb_idx_t gdof_idx) {
26        return ess_bdc_flags_values[gdof_idx];
27      },
28      A_stable, phi_stable);
```

```
29
30  // SOLVE LINEAR SYSTEM
31  Eigen::SparseMatrix A_stable_crs = A_stable.makeSparse();
32  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_stable;
33  solver_stable.compute(A_stable_crs);
34  Eigen::VectorXd sol_vec_stable = solver_stable.solve(phi_stable);
35
36  // OUTPUT RESULTS TO VTK FILe
37  // construct mesh function representing finite element solution
38  lf::fe::MeshFunctionFE mf_sol_stable(fe_space, sol_vec_stable);
39  // construct vtk writer
40  lf::io::VtkWriter vtk_writer_stable(mesh_p,
41                                      "upwind_quadrature_solution_stable.vtk");
42  // output data
43  vtk_writer_stable.WritePointData("upwind_quadrature_solution_stable",
44                                   mf_sol_stable);
```

SOLUTION of (10-3.a):

The direct verification that (10.3.9) and (10.3.10) solve (10.3.7) and satisfy the boundary conditions (10.3.8) is an easy exercise in one-dimensional differential calculus.

Here, we present the derivation of these formulas. From (10.3.7) we get:

$$\epsilon w'(\xi) + \widehat{v} w(\xi) = C_1 \quad \Rightarrow \quad \epsilon w'(\xi) = C_1 - \widehat{v} w(\xi) \tag{10.3.11}$$

Using separation of variables, we get:

$$\int \frac{dw}{C_1 - \widehat{v} w} = \frac{1}{\epsilon} \int d\xi \tag{10.3.12}$$

Now, if $\widehat{v} \neq 0$, we can multiply both sides with $-\widehat{v}$ and get

$$\log(C_1 - \widehat{v} w) = -\frac{\widehat{v}}{\epsilon} \xi + \log C_2 \tag{10.3.13}$$

and then

$$w(\xi) = \frac{1}{\widehat{v}} (C_1 - C_2 e^{-\frac{\widehat{v}}{\epsilon} \xi}). \tag{10.3.14}$$

Taking into account the boundary conditions, we determine the values of C_1 and C_2.

If $\widehat{v} = 0$, we simply have to solve

$$\epsilon w'(\xi) = C_1 \, , \tag{10.3.15}$$

which has a linear solution of the form

$$w(\xi) = \frac{C_1}{\epsilon} \xi + C_2. \tag{10.3.16}$$

From the boundary conditions we get $C_1 = \frac{\mu_k - \mu_i}{d_{ik}} \epsilon$ and $C_2 = \mu_i$.

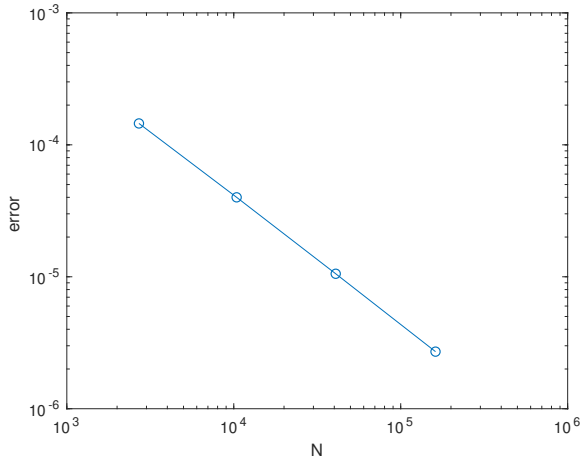
SOLUTION of (10-3.j):

See Code 10.3.32 for the code and Fig. 153 for the convergence plot for $\epsilon = 10^{-6}$. The convergence rate with respect to the number of degrees of freedom is 1.1308.

C++ code 10.3.32: Sub-problem (10-3.j): Implementation of convergence test

```
  // Read in mesh
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  lf::io::GmshReader reader(std::move(mesh_factory), "meshes/mesh.msh");

  std::shared_ptr<lf::mesh::Mesh> mesh_p = reader.mesh();

  auto mesh_seq_p{
      lf::refinement::GenerateMeshHierarchyByUniformRefinemnt(mesh_p, 6)};
  int num_meshes = mesh_seq_p->NumLevels();

  double eps = 1e-6;

  auto v = [](Eigen::Vector2d x) -> Eigen::Vector2d {
    return Eigen::Vector2d(1, 0.5 - x[1]);
  };

  auto u = [](Eigen::Vector2d x) -> double {
    return std::sin(4 * M_PI * x[1]) * std::exp(-x[0]);
  };

  auto f = [&u, eps](Eigen::Vector2d x) -> double {
    return eps * (1.0 - 16.0 * M_PI * M_PI) * u(x) - 2 * u(x) +
           (0.5 - x[1]) * 4 * M_PI * std::cos(4.0 * M_PI * x[1]) *
               std::exp(-x[0]);
  };

  // Convergence study
  std::vector<double> vec_l2error;
  std::vector<double> vec_Ndofs;
  for (int level = 0; level < num_meshes; ++level) {
    // Get the current mesh
    auto cur_mesh = mesh_seq_p->getMesh(level);

    // Create a DOF Hander for the current mesh
    const lf::assemble::UniformFEDofHandler cur_dofh(
        cur_mesh, {{lf::base::RefEl::kPoint(), 1},
                   {lf::base::RefEl::kSegment(), 0},
                   {lf::base::RefEl::kTria(), 0},
                   {lf::base::RefEl::kQuad(), 0}});
    int N_dofs = cur_dofh.NumDofs();

    lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
    UpwindFiniteVolume::ElementMatrixProvider my_mat_provider(v, eps);
    lf::assemble::AssembleMatrixLocally(0, cur_dofh, cur_dofh, my_mat_provider,
                                        A);

    Eigen::VectorXd phi(N_dofs);
    phi.setZero();
    UpwindFiniteVolume::ElementVectorProvider my_vec_provider(f);
```

```
    lf::assemble::AssembleVectorLocally(0, cur_dofh, my_vec_provider, phi);

    // Dirichlet boundary condition
    auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(cur_mesh, 2)};
    auto my_selector = [&cur_dofh, &u, &bd_flags](unsigned int dof_idx) {
      if (bd_flags(cur_dofh.Entity(dof_idx))) {
        const lf::mesh::Entity &entity{cur_dofh.Entity(dof_idx)};
        const lf::geometry::Geometry *geo_p = entity.Geometry();
        const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);
        Eigen::Vector2d corner = corners.col(0);
        return std::make_pair(true, u(corner));
      }
      return std::make_pair(false, 42.0);
    };
    lf::assemble::FixFlaggedSolutionComponents<double>(my_selector, A, phi);

    const Eigen::SparseMatrix<double> A_crs = A.makeSparse();

    // Numerical Solution
    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
    solver.compute(A_crs);
    if (solver.info() != Eigen::Success) {
      printf("Decomposition Failed.\n");
    }
    Eigen::VectorXd sol_vec = solver.solve(phi);

    // Exact Solution
    Eigen::VectorXd exact_sol(N_dofs);
    for (int i = 0; i < N_dofs; ++i) {
      const lf::mesh::Entity &entity{cur_dofh.Entity(i)};
      const lf::geometry::Geometry *geo_p = entity.Geometry();
      const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);
      exact_sol[i] = u(corners.col(0));
    }

    // L2 Error
    double l2_error = (sol_vec - exact_sol).norm() / N_dofs;
    vec_l2error.push_back(l2_error);
    vec_Ndofs.push_back(N_dofs);
  }

  for (int i = 0; i < vec_l2error.size(); ++i) {
    if (i > 0) {
      double conv_rate =
          (std::log(vec_l2error.at(i - 1)) - std::log(vec_l2error.at(i))) /
          (std::log(vec_Ndofs.at(i)) - std::log(vec_Ndofs.at(i - 1)));
      std::cout << "N_dofs: " << vec_Ndofs.at(i)
                << "; L2 Error: " << vec_l2error.at(i)
                << "; Conv. rate: " << conv_rate << std::endl;
    } else {
      std::cout << "N_dofs: " << vec_Ndofs.at(i)
                << "; L2 Error: " << vec_l2error.at(i) << std::endl;
    }
  }
```



◁ Convergence plot for Sub-problem (10-3.j), $\epsilon = 10^{-6}$.

N	L^2-Error	Convergence Rate
21	0.0515566	-
69	0.0315071	0.413983
249	0.0138218	0.642047
945	0.00473387	0.803386
3681	0.00140693	0.892315
14529	0.00038586	0.942265
57729	0.000101264	0.969653

Linear regression applied to the L^2-error norm as a function of the number N of degrees of freedom points to an asymptotic convergence like $O(N^{-1})$ for $N \to \infty$.

In terms of the meshwidth h this means

$$\|u - u_h\|_{L^2(\Omega)} = O(h^2) \quad \text{for} \quad h \to 0 \,.$$

SOLUTION of (10-3.b):

By direct computation from (10.3.9) we obtain

$$J_{ik}(\mu_i,\mu_k) = \frac{\widehat{v}(\mu_k-\mu_i)}{1-e^{-\frac{\widehat{v}}{\epsilon}d_{ik}}} + \widehat{v}\mu_i \tag{10.3.17}$$

for $\widehat{v} \neq 0$. The formula (10.3.10) yields

$$J_{ik}(\mu_i,\mu_k) = \epsilon\frac{\mu_k-\mu_i}{d_{ik}} \tag{10.3.18}$$

for $\widehat{v} = 0$.

Validation: from (10.3.3) we conclude

$$J_{ik}(\mu_i,\mu_k) = -J_{ki}(\mu_k,\mu_i)\ . \tag{10.3.19}$$

We also find for $\widehat{v} \neq 0$, setting $\rho := e^{-\frac{\widehat{v}}{\epsilon}d_{ik}}$, that

$$\begin{aligned} J_{ik}(\mu_i,\mu_k) &= \widehat{v}\left(\mu_i + \frac{\mu_k-\mu_i}{1-\rho}\right) = \widehat{v}\left(\frac{\mu_k-\rho\mu_i}{1-\rho}\right), \\ J_{ki}(\mu_k,\mu_i) &= -\widehat{v}\left(\mu_k + \frac{\mu_i-\mu_k}{1-\rho^{-1}}\right) = -\widehat{v}\,\frac{\rho\mu_i-\mu_k}{\rho-1} = -J_{ik}(\mu_i,\mu_k)\ . \end{aligned}$$

Note that swapping $\boldsymbol{p}_i$ and $\boldsymbol{p}_k$ entails a sign change of $\widehat{v}$!

HINT 1 for (10-3.c): The expressions for $J_{ik}(\mu_i, \mu_k)$ are vulnerable to **cancellation** for $\widehat{v} \approx 0$. Use the recipe “trade cancellation for approximation” as explained in to avoid potentially dangerous cancellation.
⌟

SOLUTION of (10-3.c):

The problem is the formula

$$J_{ik}(\mu_i,\mu_k) = \frac{\widehat{v}(\mu_k-\mu_i)}{1-e^{-\frac{\widehat{v}}{\epsilon}d_{ik}}} + \widehat{v}\mu_i \ . \tag{10.3.17}$$

For $\widehat{v} \approx 0$ there will be massive cancellation in the denominator and the resulting large relative error of the result is automatically amplified through the quotient.

A remedy is to use the following approximation of the exponential function by a Taylor polynomial

$$e^{-t} \approx 1 - t + \tfrac{1}{2}t^2 \ .$$

This yields the cancellation-free formula

$$J_{ik}(\mu_i,\mu_k) \approx \begin{cases} \frac{\widehat{v}(\mu_k-\mu_i)}{1-e^{-\frac{\widehat{v}}{\epsilon}d_{ik}}} + \widehat{v}\mu_i & \text{for} \quad \left|\frac{\widehat{v}}{\epsilon}d_{ik}\right| > \delta \ , \\ \epsilon\frac{\mu_k-\mu_i}{d_{ik}-\frac{1}{2}\frac{\widehat{v}}{\epsilon}d_{ik}^2} + \widehat{v}\mu_i & \text{for} \quad \left|\frac{\widehat{v}}{\epsilon}d_{ik}\right| \leq \delta \ , \end{cases} \tag{10.3.20}$$

for "sufficiently small" $\delta > 0$. This threshold may be set to $\sqrt{\texttt{EPS}} \approx 10^{-8}$, where EPS is the machine precision, $\texttt{EPS} \approx 10^{-16}$.

A more rigorous threshold δ can be obtained by estimating the error of Taylor approximation as in .

C++ code 10.3.21: Sub-problem (10-3.b): Implementation of `computeUpwindFlux()`

```
double computeUpwindFlux(double mui, double muk, double vhat, double dik,
                         double epsilon) {
  double flux = 0;
  const double delta = 1e-8;
  const double exponent = vhat / epsilon * dik;
  if (std::fabs(exponent) > delta) {
    // No risk of cancellation
    flux = vhat * (muk - mui) / (1 - std::exp(-exponent)) + vhat * mui;
  } else {
    // Use taylor approximation to prevent cancelation
    flux = epsilon * (muk - mui) / (dik * (1 - 0.5 * exponent)) + vhat * mui;
  }
  return flux;
}
```

HINT 1 for (10-3.d): The circumcenter of a triangle is the intersection of the perpendicular bisectors of its edges. ⌟

SOLUTION of (10-3.d):

Let $\boldsymbol{m}_1$ and $\boldsymbol{m}_2$ be the midpoints of two edges with direction vectors $\boldsymbol{e}_1, \boldsymbol{e}_2 \in \mathbb{R}^2$. We solve the 2×2 linear system of equations

$$\boldsymbol{m}_1 + \xi_1 \boldsymbol{e}_1^{\perp} = \boldsymbol{m}_2 + \xi_2 \boldsymbol{e}_2^{\perp} ,$$

with $\perp$ designating a rotiation by $\frac{\pi}{2}$. Thus we find the circumcenter

$$\boldsymbol{c} = \boldsymbol{m}_1 + \xi_1 \boldsymbol{e}_1^{\perp} .$$

C++ code 10.3.22: Sub-problem (10-3.d): Implementation of `computeCircumcenters()`

```
2  Eigen::Vector2d computeCircumcenters(const Eigen::Vector2d &a1,
3                                       const Eigen::Vector2d &a2,
4                                       const Eigen::Vector2d &a3) {
5
6    Eigen::Vector2d mp1 = 0.5 * (a1 + a2);
7    Eigen::Vector2d mp2 = 0.5 * (a2 + a3);
8
9    Eigen::Vector2d dir1 = a2 - a1;
10   Eigen::Vector2d dir2 = a3 - a2;
11
12   Eigen::Matrix2d dir;
13   dir << dir1(1), dir2(1), -dir1(0), -dir2(0);
14
15   Eigen::Vector2d p = dir.colPivHouseholderQr().solve(mp2 - mp1);
16
17   Eigen::Vector2d center = mp1 + p(0) * dir.col(0);
18
19   // Barycentric coordinates from
20   // Christer Ericson's 'Real-Time Collision Detection'
21
22   Eigen::Vector2d v0 = a2 - a1, v1 = a3 - a1, v2 = center - a1;
23   double d00 = v0.dot(v0);
24   double d01 = v0.dot(v1);
25   double d11 = v1.dot(v1);
26   double d20 = v2.dot(v0);
27   double d21 = v2.dot(v1);
28
29   double denom = d00 * d11 - d01 * d01;
30   double v = (d11 * d20 - d01 * d21) / denom;
31   double w = (d00 * d21 - d01 * d20) / denom;
32   double u = 1.0 - v - w;
33
34   if (v <= 0 || w <= 0 || u <= 0) {
35     throw std::runtime_error("Obtused triangle!");
36     return Eigen::Vector2d::Zero();
37   }
38   return center;
39
40 }
41 /* SAM_LISTING_BEGIN_3 */
42
43 }  // namespace UpwindFiniteVolume
```

HINT 1 for (10-3.e): For the finite-volume method studied in this problem the unknowns/d.o.f.s are associated with nodes of the primal mesh and potentially non-zero off-diagonal entries of $\mathbf{A}$ are in one-to-one relationship with primal edges. Thus the situation is the same as for a finite-element Galerkin discretization of a scalar second-order elliptic BVP based on $\mathcal{S}_1^0(\mathcal{M})$, which has been presented in [Lecture → Section 2.4]. Thus the considerations of [Lecture → Section 2.4.5.2] apply. ⌟

HINT 2 for (10-3.e): Obviously we can write for the flux (10.3.4) through the dual edge f_{ik}

$$\Psi(\mu_i, \mu_k) = \|\boldsymbol{c}_K - \boldsymbol{m}_{ik}\| J_{ik}(\mu_i, \mu_k) + \|\boldsymbol{c}_{K'} - \boldsymbol{m}_{ik}\| J_{ik}(\mu_i, \mu_k) \ , \tag{10.3.24}$$

when K, K' are the two triangles adjacent to the primal edge $[\boldsymbol{p}_i, \boldsymbol{p}_k]$. ⌟

SOLUTION of (10-3.e):

Note that the coefficients α_{ik}^* are associated with the edge $[\boldsymbol{p}_i, \boldsymbol{p}_k]$ and from (10.3.19) we conclude that $\alpha_{ik}^* = -\alpha_{ki}^*$. We combine (10.3.2) and (10.3.4) and get

$$\sum_{k\in\mathcal{U}_i} |f_{ik}| J(\mu_i,\mu_k) = \sum_{k\in\mathcal{U}_i} |f_{ik}| \left(\alpha_{ik}^i \mu_i + \alpha_{ik}^k \mu_k\right) = \int_{C_i} f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad i = 1,\ldots,N\,.$$

This linear equation corresponds to row i of the matrix $\mathbf{A}$ (before the elimination of boundary d.o.f.s), which implies

$$(\mathbf{A})_{i,k} = |f_{ik}|\alpha_{ik}^k\,, \quad k\in\mathcal{U}_i \quad , \quad (\mathbf{A})_{i,k} = 0\,, \quad k\notin\mathcal{U}_i\cup\{i\} \quad , \quad (\mathbf{A})_{i,i} = \sum_{k\in\mathcal{U}_i} |f_{ik}|\alpha_{ik}^i \tag{10.3.25}$$

As suggested by (10.3.24) we introduce the abbreviation

$$L_{ik}^K := \|\boldsymbol{c}_K - \boldsymbol{m}_{ik}\|\,, \quad 1\le i\neq k\le N\,,$$

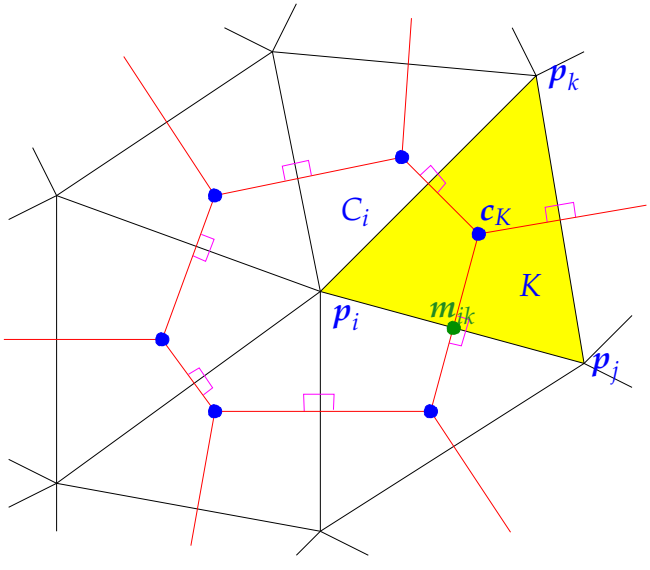
for the length of the part of the dual edge f_{ik} inside the triangle K. Then denoting by K and K' the two triangles adjacent to the primal edge $[\mathbf{p}_i, \mathbf{p}_k]$ we can rewrite

$$(\mathbf{A})_{i,k} = L_{ik}^K \alpha_{ik}^k + L_{ik}^{K'} \alpha_{ik}^k\,, \quad k\in\mathcal{U}_i \quad , \quad (\mathbf{A})_{i,k} = 0\,, \quad k\notin\mathcal{U}_i\cup\{i\}\,. \tag{10.3.26}$$

Thus we have written $(\mathbf{A})_{i,k}$ as a sum of two cell contributions. For the diagonal matrix entry we introduce the set $\mathcal{M}_i$ of triangles sharing the vertex $\boldsymbol{p}_i$. Then we can express

$$(\mathbf{A})_{i,i} = \sum_{K\in\mathcal{M}_i} L_{ik}^K \alpha_{ik}^i + L_{ij}^K \alpha_{ij}^i\,, \tag{10.3.27}$$

where, abusing notation, the summation variable K is supposed to be a triangle with vertices $\boldsymbol{p}_i, \boldsymbol{p}_k, \boldsymbol{p}_j$ in every case. Again, we have written $(\mathbf{A})_{i,i}$ as a sum of cell contributions.



◁ Triangular cell K with vertices

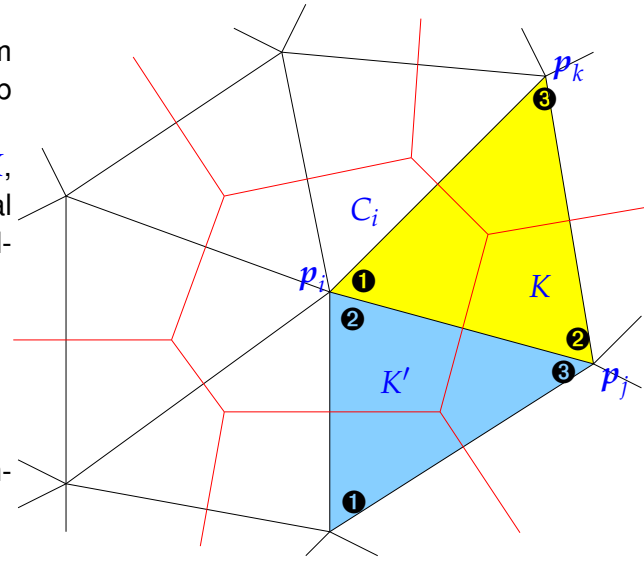
$\boldsymbol{p}_i \leftrightarrow$ vertex 1
$\boldsymbol{p}_j \leftrightarrow$ vertex 2
$\boldsymbol{p}_k \leftrightarrow$ vertex 3

$\boldsymbol{c}_K \mathrel{\hat{=}}$ circumcenter of K
$\boldsymbol{m}_{ik} := \frac{1}{2}(\boldsymbol{p}_i + \boldsymbol{p}_j)$

Now we want to infer the element matrices from (10.3.26) and (10.3.27). Let us recall the relationship of element matrices and the global matrix $\mathbf{A}$:
In the situation sketched beside, with triangles K, K' abutting the edge $[\boldsymbol{p}_i, \boldsymbol{p}_k]$ and the indicated local numbering of vertices, we have for $i \neq k$ (and including d.o.f.s located on $\partial\Omega$)

$$(\mathbf{A})_{i,k} = (\mathbf{A}_K)_{1,2} + (\mathbf{A}_{K'})_{2,3} ,$$
$$(\mathbf{A})_{k,i} = (\mathbf{A}_K)_{2,1} + (\mathbf{A}_{K'})_{3,2} .$$

Note that neither $\mathbf{A}$ nor $\mathbf{A}_K$ and $\mathbf{A}_{K'}$ need to be symmetric.



Let K be a triangle with vertices $\boldsymbol{p}_i$, $\boldsymbol{p}_j$, and $\boldsymbol{p}_k$ (in this order). Then from (10.3.26) and (10.3.27) we read off the element matrix

$$\mathbf{A}_K = \begin{bmatrix} L_{ij}^K \alpha_{ij}^i + L_{ik}^K \alpha_{ik}^i & L_{ij}^K \alpha_{ij}^j & L_{ik}^K \alpha_{ik}^k \\ L_{ji} \alpha_{ji}^i & L_{ji}^K \alpha_{ji}^j + L_{jk}^K \alpha_{jk}^j & L_{jk} \alpha_{jk}^k \\ L_{ki}^K \alpha_{ki}^i & L_{kj}^K \alpha_{kj}^j & L_{ki}^K \alpha_{ki}^k + L_{kj} \alpha_{kj}^k \end{bmatrix} . \tag{10.3.28}$$

SOLUTION of (10-3.f):

We recall the definition (10.3.6) of $\widehat{v}$:

$$\widehat{v} = \widehat{v}(\boldsymbol{m}_{ik}, i, k) := \mathbf{v}(\boldsymbol{m}_{ik}) \cdot \frac{\boldsymbol{p}_k - \boldsymbol{p}_i}{\|\boldsymbol{p}_k - \boldsymbol{p}_i\|} \;, \quad \boldsymbol{m}_{ik} := \tfrac{1}{2}(\boldsymbol{p}_i + \boldsymbol{p}_k) \;, \quad i \neq k \;,$$

For $\widehat{v} \neq 0$ we set $\rho := \exp(-\frac{\widehat{v}}{\epsilon} d_{ik})$ and get from (10.3.17)

$$J_{ik}(\mu_i, \mu_k) = \widehat{v}\left(\mu_i + \frac{\mu_k - \mu_i}{1-\rho}\right) = \left(\frac{\widehat{v}\rho}{\rho - 1}\right)\mu_i + \left(\frac{\widehat{v}}{1-\rho}\right)\mu_k \;,$$

which means

$$\alpha_{ik}^i = \frac{\widehat{v}\rho}{\rho - 1} \quad , \quad \alpha_{ik}^k = \frac{\widehat{v}}{1-\rho} \;. \tag{10.3.29}$$

For $\widehat{v} = 0$ we have

$$J_{ik}(\mu_i, \mu_k) = \epsilon \frac{\mu_k - \mu_i}{d_{ik}} = \left(-\frac{\epsilon}{d_{ik}}\right)\mu_i + \left(\frac{\epsilon}{d_{ik}}\right)\mu_k \;,$$

and immediately read off the coefficients α_{ik}^i and α_{ik}^k.

SOLUTION of (10-3.g):

C++ code 10.3.30: Sub-problem (10-3.g): Implementation of `ElementMatrixProvider::Eval()`

```
template <typename FUNCTOR>
Eigen::Matrix3d ElementMatrixProvider<FUNCTOR>::Eval(
    const lf::mesh::Entity &entity) {
  Eigen::Matrix3d A = Eigen::Matrix3d::Zero();

  LF_VERIFY_MSG(entity.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type!");

  const lf::geometry::Geometry *geo_p = entity.Geometry();
  const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);

  Eigen::Vector2d circumcenter =
      computeCircumcenters(corners.col(0), corners.col(1), corners.col(2));

  Eigen::Matrix3d f = Eigen::Matrix3d::Zero();
  Eigen::Matrix3d d = Eigen::Matrix3d::Zero();
  Eigen::Matrix3d vv = Eigen::Matrix3d::Zero();

  f(0, 1) = (0.5 * (corners.col(0) + corners.col(1)) - circumcenter).norm();
  f(0, 2) = (0.5 * (corners.col(0) + corners.col(2)) - circumcenter).norm();
  f(1, 2) = (0.5 * (corners.col(1) + corners.col(2)) - circumcenter).norm();
  f.triangularView<Eigen::Lower>() = f.transpose();

  d(0, 1) = (corners.col(1) - corners.col(0)).norm();
  d(0, 2) = (corners.col(2) - corners.col(0)).norm();
  d(1, 2) = (corners.col(2) - corners.col(1)).norm();
  d.triangularView<Eigen::Lower>() = d.transpose();

  vv(0, 1) = v_(0.5 * (corners.col(0) + corners.col(1)))
                 .dot((corners.col(1) - corners.col(0)).normalized());
  vv(0, 2) = v_(0.5 * (corners.col(0) + corners.col(2)))
                 .dot((corners.col(2) - corners.col(0)).normalized());
  vv(1, 2) = v_(0.5 * (corners.col(1) + corners.col(2)))
                 .dot((corners.col(2) - corners.col(1)).normalized());
  vv.triangularView<Eigen::Lower>() = -vv.transpose();

  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      if (i != j) {
        if (std::abs(vv(i, j)) > 1e-14) {
          double beta = 1. / (1. - std::exp(-vv(i, j) * d(i, j) / eps_));
          A(i, j) = vv(i, j) * beta * f(i, j);
          A(i, i) += vv(i, j) * (1. - beta) * f(i, j);
        } else {
          A(i, j) = eps_ / d(i, j) * f(i, j);
          A(i, i) -= eps_ / d(i, j) * f(i, j);
        }
      }
    }
  }
  return A;
```

```
53 }
```

HINT 1 for (10-3.h): You will have to find an efficient way to calculate the area $|C_i|$ by summing up cell contributions. ⌟

SOLUTION of (10-3.h):

C++ code 10.3.31: Sub-problem (10-3.h): Implementation of `ElementVectorProvider::Eval()`

```
template <typename FUNCTOR>
Eigen::Vector3d ElementVectorProvider<FUNCTOR>::Eval(
    const lf::mesh::Entity &entity) {
  LF_VERIFY_MSG(entity.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type!");

  const lf::geometry::Geometry *geo_p = entity.Geometry();
  const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);

  Eigen::Vector2d circumcenter =
      computeCircumcenters(corners.col(0), corners.col(1), corners.col(2));

  double a12 = (corners.col(0) - corners.col(1)).norm() *
               (0.5 * (corners.col(0) + corners.col(1)) - circumcenter).norm() /
               4.0;
  double a13 = (corners.col(0) - corners.col(2)).norm() *
               (0.5 * (corners.col(0) + corners.col(2)) - circumcenter).norm() /
               4.0;
  double a23 = (corners.col(1) - corners.col(2)).norm() *
               (0.5 * (corners.col(1) + corners.col(2)) - circumcenter).norm() /
               4.0;

  Eigen::Vector3d result;
  result(0) = (a12 + a13) * f_(corners.col(0));
  result(1) = (a12 + a23) * f_(corners.col(1));
  result(2) = (a13 + a23) * f_(corners.col(2));

  return result;
}
```

SOLUTION of (10-3.i):

The gradient of u is $\mathbf{grad}\, u = (-\sin(4\pi x_2)e^{-x_1}, 4\pi\cos(4\pi x_2)e^{-x_1})^T$. Inserting this expression in (10.3.1) we get:

$$\begin{aligned}\operatorname{div}(\epsilon\,\mathbf{grad}\, u + vu) &= \epsilon\sin(4\pi x_2)e^{-x_1} - \sin(4\pi x_2)e^{-x_1} - 16\pi^2\epsilon\sin(4\pi x_2)e^{-x_1} + \\ &+ \left(\frac{1}{2} - x_2\right)\cos(4\pi x_2)e^{-x_1} - \sin(4\pi x_2)e^{-x_1} = \\ &= \epsilon(1-16\pi^2)u(x) - 2u(x) + \left(\frac{1}{2} - x_2\right)4\pi\cos(4\pi x_2)e^{-x_1} = f(\boldsymbol{x})\end{aligned}$$

HINT 1 for (10-4.a): Do not forget to specify the trial and test space. ⌟

SOLUTION of (10-4.a):

To begin with, note that zero Dirichlet boundary conditions are imposed on the whole boundary. They are essential boundary condition and have to be built into the trial and test space.

We employ the standard approach explained in detail in [Lecture → Section 9.2.2]: we multiply with $w \in H_0^1(\Omega)$ the PDE

$$\frac{\partial u}{\partial t} - \epsilon \Delta u + \mathbf{v} \cdot \mathbf{grad} u = 0 \, .$$

Using integration by parts via Green's formula [Lecture → Thm. 1.5.2.7], we obtain the following spatial variational formulation of (10.4.1): we seek for $t \in (0,T) \to u(t) \in H_0^1(\Omega)$ (**trial space**) such that

$$\begin{cases} \int_\Omega \left(\frac{\partial u}{\partial t} w + \epsilon \mathbf{grad} u \cdot \mathbf{grad} w + (\mathbf{v} \cdot \mathbf{grad} u) w \right) \mathrm{d}\boldsymbol{x} = 0 \quad \forall w \in H_0^1(\Omega) \quad \text{(**test space**)} \, , \\ u(0) = u_0 \in H_0^1(\Omega) \, . \end{cases} \tag{10.4.2}$$

SOLUTION of (10-4.j):

The error (10.4.18) is the maximal error in the nodes of the mesh at final time, which can be regarded as a sampling-based approximation of $\left\| u_h^{(K)} - u(\cdot, T) \right\|_{L^\infty(\Omega)}$. We make the following observations:

- As we refine the spatial mesh (increase M) and reduce the timestep (increase K) the error norm $\mathrm{err}(\mathrm{M}, \mathrm{K})$ decreases until it reaches a saturation level. A similar observation was made in [Lecture $\rightarrow$ Exp. 9.2.8.1]. In that numerical experiment we concluded that probably the total error is the sum of a spatial and a temporal error contribution, see [Lecture $\rightarrow$ § 9.2.8.4].
- When saturation is reached, a further reduction of the timestep makes the error norm grow again. This may be a consequence of the accumulation of spatial (interpolation) errors in the course of timestepping.
- On fine meshes halving the timestep $K \leftarrow \frac{K}{2}$ roughly halves the total error. If K is large halving the meshwidth $M \leftarrow \frac{M}{2}$ makes the error shrink by a factor of 2. This hints at an asymptotic error behavior like

$$\mathrm{err}(M, K) = O(M^{-1} + K^{-1}) = O(h_{\mathcal{M}} + \tau) \quad \text{for} \quad h = M^{-1}, \tau = K^{-1} \to 0 \,.$$

HINT 1 for (10-4.b): As in [Lecture → § 10.2.0.2], use integration by parts via Green's formula and the product rule for the divergence to tease out the sign of the convective term:

$$\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\, u)u\,\mathrm{d}\boldsymbol{x} = -\int_\Omega \mathrm{div}(\mathbf{v}u)u\,\mathrm{d}\boldsymbol{x} = -\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\, u)u + (\mathrm{div}\,\mathbf{v})u^2\,\mathrm{d}\boldsymbol{x}\,. \tag{10.4.3}$$

⌟

SOLUTION of (10-4.b):

To begin with, from (10.4.3) we conclude

$$\int_\Omega (\mathbf{v} \cdot \mathbf{grad}\, u) u \,\mathrm{d}\boldsymbol{x} = -\tfrac{1}{2} \int_\Omega (\operatorname{div} \mathbf{u}) u^2 \,\mathrm{d}\boldsymbol{x} \,. \tag{10.4.4}$$

Since $u \in H_0^1(\Omega)$, we can use $v := u(t)$ as test function in the spatial variational formulation (10.4.2). This idea leads to

$$\begin{aligned} \frac{1}{2}\frac{\partial}{\partial t}\|u(\cdot,t)\|^2_{L^2(\Omega)} = \int_\Omega u(t)\frac{\partial u}{\partial t}(t)\,\mathrm{d}\boldsymbol{x} = \int_\Omega -\epsilon\|\mathbf{grad}\, u\|^2 + \tfrac{1}{2}(\operatorname{div}\mathbf{v})u^2\,\mathrm{d}\boldsymbol{x} = \\ \leq \tfrac{1}{2}\int_\Omega (\operatorname{div}\mathbf{v})u^2\,\mathrm{d}\boldsymbol{x}\,. \end{aligned}$$

If, for some $d_0 > 0$, $\operatorname{div}\mathbf{v}(\boldsymbol{x}) \leq -d_0$ for all $\boldsymbol{x} \in \Omega$, then

$$\frac{\partial}{\partial t}\|u(\cdot,t)\|^2_{L^2(\Omega)} \leq -d_0\|u(\cdot,t)\|^2_{L^2(\Omega)} \,,$$

and we can predict an exponential decay of $\|u(\cdot,t)\|^2_{L^2(\Omega)}$ as $t \to \infty$. This follows by arguments given in the proof of [Lecture → Lemma 9.2.3.8].

HINT 1 for (10-4.c): Your code can exploit the fact that $\Omega =]0,1[^2$. ⌟

SOLUTION of (10-4.c):

According to the definition [Lecture → Def. 6.4.0.9] and the conventions underlying Butcher schemes [Lecture → Eq. (6.4.0.11)], we obtain from (10.4.7) applied to the autonomous ODE $\dot{\boldsymbol{y}} = \mathbf{v}(\boldsymbol{y})$

$$\begin{aligned}\mathbf{k}_1 &= \mathbf{v}(\boldsymbol{y}_0)\,,\\ \mathbf{k}_2 &= \mathbf{v}(\boldsymbol{y}_0 + \tfrac{2}{3}\tau\boldsymbol{k}_1)\,,\\ \boldsymbol{y}_1 &= \boldsymbol{y}_0 + \tau\left(\tfrac{1}{4}\mathbf{k}_1 + \tfrac{3}{4}\mathbf{k}_2\right)\,,\end{aligned}$$

where $\tau := \frac{t}{K} > 0$ is the timestep. After each timestep we check whether the current point $\boldsymbol{y}^{(j)}$ still lies inside Ω. If not return 0, if yes, carry on unless $j = K$.

C++ code 10.4.8: Sub-problem (10-4.c): Implementation of `solveTransport`

```
2  template <typename FUNCTOR_V, typename FUNCTOR_U0>
3  double solveTransport(const Eigen::Vector2d& x, int K, double t, FUNCTOR_V&& v,
4                        FUNCTOR_U0&& u0) {
5    double tau = t / K;     // timestep
6    Eigen::Vector2d y = x;  // starting point
7
8    for (int i = 0; i < K; ++i) {
9      // A single step of Heun's method, expressed by RK increments
10     Eigen::Vector2d k_1 = v(y);
11     Eigen::Vector2d k_2 = v(y - 2. / 3. * tau * k_1);
12     y -= tau / 4. * k_1 + 3. / 4. * tau * k_2;
13
14     // check, if current point is outside the domain
15     if (y(0) < 0. || y(0) > 1. || y(1) < 0. || y(1) > 1.) {
16       return 0.0;
17     }
18   }
19   return u0(y);
20 }
```

SOLUTION of (10-4.d):

We write $\left\{b_j^k\right\}_{k=1}^N$ for the nodal basis of $\mathcal{S}_{1,0}^0(\mathcal{M})$. We start from the expansion $u_h^{(j)} = \sum_{k=1}^N \mu_k^{(j)} b_h^k$. By taking $w_h = b_h^k$, (10.4.9) becomes

$$\int_\Omega \left(\sum_{k=1}^N \mu_k^{(j)} b_h^k - \mathbf{I}_1(\sum_{k=1}^N \mu_k^{(j-1)} b_h^k(\cdot - \tau \mathbf{v}(\cdot))) \right) b_h^l \,\mathrm{d}\boldsymbol{x} + \tau\epsilon \int_\Omega \sum_{k=1}^N \mu_k^{(j)} \,\mathbf{grad}\, b_h^k \cdot \mathbf{grad}\, b_h^l \,\mathrm{d}\boldsymbol{x} = 0 \,.$$

Then, we can write (10.4.9) as follows:

$$(\mathbf{M} + \tau\mathbf{A})\vec{\boldsymbol{\mu}}^{(j)} = \mathbf{f}(\vec{\boldsymbol{\mu}}^{(j-1)}) \,,$$

where $\mathbf{M}$ is the mass matrix, that is, the Galerkin matrix for the $L^2(\Omega)$ inner product, here approximated by means of the composite trapezoidal rule,

$$(\mathbf{M})_{k,l} \left[\quad \leftrightarrow \quad \int_\Omega b_h^k b_h^l \,\mathrm{d}\boldsymbol{x} \right] \quad = h^2 \sum_{i=1}^{M-1} \sum_{j=1}^{M-1} b_h^k \left(\begin{bmatrix} ih \\ jh \end{bmatrix} \right) b_h^l \left(\begin{bmatrix} ih \\ jh \end{bmatrix} \right) = \begin{cases} h^2 & \text{, if } k = l \\ 0 & \text{else,} \end{cases}$$

and $\mathbf{A}$ is the Galerkin matrix for $-\Delta$:

$$(\mathbf{A})_{k,l} = \int_\Omega \epsilon \,\mathbf{grad}\, b_h^k \cdot \mathbf{grad}\, b_h^l \,\mathrm{d}\boldsymbol{x} \,.$$

Note that only the formula for the entries of $\mathbf{M}$ was requested. Its derivation is due to the fact that, using the 2D trapezoidal rule, all approximate integrals of mixed products $b_h^k\, b_h^l$ for $k \neq l$ in $\mathbf{M}$ are 0, as nodal basis functions attain the value 1 on only one vertex and 0 on all the others. Hence, only the diagonal of $\mathbf{M}$ contains nonzero entries. The number of nonzero entries of $\mathbf{M}$ is therefore $(M+1)^2 - 4M = (M-1)^2$, excluding the degrees of freedom on $\partial\Omega$.

HINT 1 for (10-4.e): If the corners of the square are numbered counterclockwise starting with $\begin{bmatrix}0\\0\end{bmatrix}$ the local shape functions are

$$\begin{aligned} b_K^1(\boldsymbol{x}) &= (1-\frac{x_1}{h})(1-\frac{x_2}{h})\;, \\ b_K^2(\boldsymbol{x}) &= \frac{x_1}{h}(1-\frac{x_2}{h})\;, \\ b_K^3(\boldsymbol{x}) &= \frac{x_1}{h}\frac{x_2}{h}\;, \\ b_K^4(\boldsymbol{x}) &= (1-\frac{x_1}{h})\frac{x_2}{h}\;. \end{aligned} \tag{10.4.11}$$

⌟

HINT 2 for (10-4.e): Take into account symmetry! ⌟

SOLUTION of (10-4.e):

The gradients of the local shape functions are

	at $\boldsymbol{a}^1 := \begin{bmatrix}0\\0\end{bmatrix}$	at $\boldsymbol{a}^2 := \begin{bmatrix}h\\0\end{bmatrix}$	at $\boldsymbol{a}^3 := \begin{bmatrix}h\\h\end{bmatrix}$	at $\boldsymbol{a}^4 := \begin{bmatrix}0\\h\end{bmatrix}$
$\mathbf{grad}\, b_K^1(\boldsymbol{x}) = \frac{1}{h}\left[\frac{x_2}{h}-1, \frac{x_1}{h}-1\right]^\top$,	$\frac{1}{h}\begin{bmatrix}-1\\-1\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}-1\\0\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}0\\0\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}0\\-1\end{bmatrix}$
$\mathbf{grad}\, b_K^2(\boldsymbol{x}) = \frac{1}{h}\left[1-\frac{x_2}{h}, -\frac{x_1}{h}\right]^\top$,	$\frac{1}{h}\begin{bmatrix}1\\0\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}1\\-1\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}0\\-1\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}0\\0\end{bmatrix}$
$\mathbf{grad}\, b_K^3(\boldsymbol{x}) = \frac{1}{h}\left[\frac{x_2}{h}, \frac{x_1}{h}\right]^\top$,	$\frac{1}{h}\begin{bmatrix}0\\0\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}0\\1\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}1\\1\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}1\\0\end{bmatrix}$
$\mathbf{grad}\, b_K^4(\boldsymbol{x}) = \frac{1}{h}\left[-\frac{x_2}{h}, 1-\frac{x_1}{h}\right]^\top$.	$\frac{1}{h}\begin{bmatrix}0\\1\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}0\\0\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}-1\\0\end{bmatrix}$	$\frac{1}{h}\begin{bmatrix}-1\\1\end{bmatrix}$

Using the 2D trapezoidal (10.4.10) on a single cell with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3, \boldsymbol{a}^4$ [Lecture → Fig. 115], the element matrix becomes

$$\left[\frac{h^2}{4}\sum_{\ell=1}^{4} \mathbf{grad}\, b_K^j(\boldsymbol{a}^\ell)\cdot\mathbf{grad}\, b_K^i(\boldsymbol{a}^\ell)\right]_{i,j=1}^{4} = \frac{1}{2}\begin{bmatrix}2&-1&0&-1\\-1&2&-1&0\\0&-1&2&-1\\-1&0&-1&2\end{bmatrix}. \tag{10.4.12}$$

Note that the area h^2 of the square offsets the factors $\frac{1}{h}$. As a consequence the element matrix does not depend on the meshwidth!

SOLUTION of (10-4.f):

The algorithm involves the following steps:

1. Compute the indices
$$i := \lfloor \frac{x_1}{h} \rfloor \quad , \quad j := \lfloor \frac{x_2}{h} \rfloor ,$$
and check whether $i, j \in \{0, \ldots, M-1\}$, which indicates that $\boldsymbol{x} \in \Omega$.
2. Compute the local coordinates of the point in the cell
$$\widehat{x}_1 := \frac{x_1 - ih}{h} \quad , \quad \widehat{x}_2 := \frac{x_2 - jh}{h} .$$
3. Find the vector indices $\ell_1, \ell_2, \ell_3, \ell_4$ of the four vertices of the cell in which $\boldsymbol{x}$ is located. Beware, cells adjacent to $\partial\Omega$ require a special treatment, because some of their vertices do not carry global shape functions.
4. Evaluate
$$u_h(\boldsymbol{x}) = (\mathrm{u})_{\ell_1} b^1_{\widehat{K}}(\widehat{\boldsymbol{x}}) + (\mathrm{u})_{\ell_2} b^2_{\widehat{K}}(\widehat{\boldsymbol{x}}) + (\mathrm{u})_{\ell_3} b^3_{\widehat{K}}(\widehat{\boldsymbol{x}}) + (\mathrm{u})_{\ell_4} b^4_{\widehat{K}}(\widehat{\boldsymbol{x}}) ,$$
where b^j_K, $j = 1, 2, 3, 4$, are the reference local shape function from (10.4.12).

C++ code 10.4.13: Sub-problem (10-4.f): Implementation of `evalFEfunction()`

SOLUTION of (10-4.g):

Thanks to the use of the composite trapezoidal rule (10.4.10), we find that

$$(\mathbf{f}(\vec{\eta}))_k = h^2 v_h(\boldsymbol{x}^k - \tau \mathbf{v}(\boldsymbol{x}^k)) \,, \quad k = 1, \ldots, N \,,$$

where $\boldsymbol{x}^k$ is the grid node carrying the k-th basis function, and $v_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$ is the piecewise bi-linear finite element function defined through the coefficient vector $\vec{\eta}$.

C++ code 10.4.14: Sub-problem (10-4.g): Implementation of `semiLagrangeSource`

```
template <typename FUNCTOR>
Eigen::VectorXd semiLagrangeSource(const Eigen::VectorXd& u_old, double tau,
                                   FUNCTOR&& velocity) {
  // Note:  components of coefficient vectors are associated
  // with interior nodes only
  int N = u_old.size();
  // Extract number of cells in one direction
  int root = std::round(std::sqrt(N));
  if (N != root * root) {
    std::cerr << "The number of dofs should be a perfect square!" << std::endl;
  }
  int M = root + 1;
  Eigen::MatrixXd grid = findGrid(M);
  Eigen::VectorXd f(N);

  for (int i = 0; i < grid.cols(); ++i) {
    // Find grid point corresponding to a degree of freedom
    Eigen::Vector2d x = grid.col(i);
    // Determine location of advected gridpoint
    Eigen::Vector2d y = x - tau * velocity(x);
    // Test whether it still lies inside the computational domain
    if (y(0) >= 0. && y(0) <= 1. && y(1) >= 0. && y(1) <= 1.) {
      // Evaluate finite element function from previous timestep
      // at preimage of gridpoint under flow
      f(i) = evalFEfunction(y, u_old);
    } else {
      // Zero, if advected point outside domain
      f(i) = 0.;
    }
  }
  // Finally scale with h^2
  return f / (M * M);  // * 1 (from [0,1]^2) * 4 (from no.  of adjacent
                       // squares) / 4 (from no.  of vertices of square)
}
```

HINT 1 for (10-4.h): The Galerkin matrix $\mathbf{A}$ in (10.4.11) coincides with a scaled **Poisson matrix**, see [Lecture → Eq. (4.1.2.8)]. ⌟

SOLUTION of (10-4.h):

For $\epsilon = 0$ the timestepping scheme boils down to

$$\mathbf{M}\vec{\mu}^{(j)} = \mathbf{f}(\vec{\mu}^{(j-1)}) \,, \quad j = 1,2,3,\ldots \,.$$

From Sub-problem (10-4.d) we know that $\mathbf{M}$ is a h^2-scaled identity matrix.

C++ code 10.4.16: Sub-problem (10-4.h): Implementation of `semiLagrangePureTransport`

SOLUTION of (10-4.i):

SOLUTION of (10-5.j):

From we recall the abstract idea underlying the Strang splitting. To begin with, formally rewrite the problem as an "ODE in function space" with a right-hand side, which is the sum of two terms:

$$\frac{\partial u}{\partial t} = g(u) + h(u) \,.$$

We examine a single timestep of length τ. Given $u^{(j-1)}$ we want to solve for $u^{(j)}$. For that purpose the problem is split into three sub-problems:

$$\frac{\partial z}{\partial t} = g(z) \,, \quad \text{on } \left]0, \ \frac{\tau}{2}\right] \qquad \text{with} \quad z(0) = u^{(j-1)} \tag{10.5.15a}$$

$$\frac{\partial w}{\partial t} = h(w) \,, \quad \text{on } \ ]0, \ \tau] \qquad \text{with} \quad w(0) = z\Big(\frac{\tau}{2}\Big) \tag{10.5.15b}$$

$$\frac{\partial y}{\partial t} = g(y) \,, \quad \text{on } \left]0, \ \frac{\tau}{2}\right] \qquad \text{with} \quad y(0) = w(\tau) \tag{10.5.15c}$$

Finally, we obtain $u^{(j)} := y(\frac{\tau}{2})$.

Now let us flesh out (10.5.15)

- (10.5.15a) amounts to solving the standard advection-diffusion IBVP

$$\begin{aligned} \frac{\partial z}{\partial t} - \Delta z + \mathbf{v}(x) \cdot \mathbf{grad}\, z = 0 \quad &\text{in} \quad \Omega \times]0, \tfrac{\tau}{2}[\,, \\ z = 0 \quad &\text{on} \quad \partial\Omega \times]0, \frac{\tau}{2}[\,, \\ z(x,0) = u^{(j-1)}(x) \quad &\forall x \in \Omega \,. \end{aligned} \tag{10.5.16}$$

- (10.5.15b) amounts to solving the *purely local* evolution problem

$$\begin{aligned} \frac{\partial w}{\partial t} + c(x)\, w = 0 \quad &\text{in} \quad \Omega \times]0, \tau[\,, \\ w = 0 \quad &\text{on} \quad \partial\Omega \times]0, \tau[\,, \\ w(x,0) = z(x, \tfrac{\tau}{2}) \quad &\forall x \in \Omega \,. \end{aligned} \tag{10.5.17}$$

- (10.5.15c) amounts to solving another standard advection-diffusion IBVP

$$\begin{aligned} \frac{\partial y}{\partial t} - \Delta y + \mathbf{v}(x) \cdot \mathbf{grad}\, y = 0 \quad &\text{in} \quad \Omega \times]0, \tfrac{\tau}{2}[\,, \\ y = 0 \quad &\text{on} \quad \partial\Omega \times]0, \frac{\tau}{2}[\,, \\ y(x,0) = w(x,\tau) \quad &\forall x \in \Omega \,. \end{aligned} \tag{10.5.18}$$

Of course, when multiple timesteps are carried out, steps (10.5.16) and (10.5.18) can be merged into a leapfrog-style timestepping scheme.

Remark. In contrast to the method of lines, in this sub-problem we developed a timestepping scheme for an evolution problem before discretization in space. This is a valid approach and it paves the way for using different spatial discretizations in different (sub-)steps of the "single-step method in function space". Compare the discussion in [Lecture → Section 5.3.2] concerning iterative method in function space for the solution of non-linear variational problems.

SOLUTION of (10-5.k):

C++ code 10.5.21: Sub-problem (10-5.j): Implementation of `ReactionStep::step()`
➥ GitLab

```cpp
template <typename FUNCTOR>
class ReactionStep {
 public:
  ReactionStep(
      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
      FUNCTOR c)
      : fe_space_(fe_space), c_(c) {
    // Assemble matrix A_{m,c}
    LumpedMassElementMatrixProvider mass_element_matrix_provider_c(c_);
    lf::assemble::COOMatrix<double> mass_matrix_c(
        fe_space_->LocGlobMap().NumDofs(), fe_space_->LocGlobMap().NumDofs());
    lf::assemble::AssembleMatrixLocally(
        0, fe_space_->LocGlobMap(), fe_space_->LocGlobMap(),
        mass_element_matrix_provider_c, mass_matrix_c);
    mass_matrix_c_sparse_ = mass_matrix_c.makeSparse();

    // Assemble matrix A_m
    LumpedMassElementMatrixProvider mass_element_matrix_provider_1(
        [](Eigen::Vector2d /*x*/) { return 1.0; });
    lf::assemble::COOMatrix<double> mass_matrix_1(
        fe_space_->LocGlobMap().NumDofs(), fe_space_->LocGlobMap().NumDofs());
    lf::assemble::AssembleMatrixLocally(
        0, fe_space_->LocGlobMap(), fe_space_->LocGlobMap(),
        mass_element_matrix_provider_1, mass_matrix_1);
    mass_matrix_1_sparse_ = mass_matrix_1.makeSparse();

  };

  Eigen::VectorXd step(const Eigen::VectorXd& u0_vector, double tau) {
    // The explicit midpoint rule computes the solution to y' = f(y) as
    // y* = y_n + tau/2 * f(y_n)
    // y_{n+1} = y_n + tau * f(y*)

    // In the evolution problem f(y) solves the discretized variational
    problem
    // A_m * f(y) = A_{m,c} * y
    // where A_m is the lumped mass matrix
    // and A_{m,c} is the lumped mass matrix with coefficient function c.

    // initialize sparse solver for mass matrix A_m
    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
    solver.compute(mass_matrix_1_sparse_);

    // evaluate f(y_n)
    Eigen::VectorXd k1 = solver.solve(mass_matrix_c_sparse_ * u0_vector);

    // evaluate f(y*)
    Eigen::VectorXd k2 =
        solver.solve(mass_matrix_c_sparse_ * (u0_vector + 0.5 * tau * k1));

    // update to y_{n+1}
```

```
52      return u0_vector + tau * k2;
53    }
54
55   private:
56    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space_;
57    FUNCTOR c_;
58    Eigen::SparseMatrix<double> mass_matrix_c_sparse_;
59    Eigen::SparseMatrix<double> mass_matrix_1_sparse_;
60  };
```

HINT 1 for (10-5.l): Of course, you should call the functions `SemiLagrStep::step()` and `reactio`
Pay attention to an efficient “leapfrog-style” implementation of the Strang-splitting. ⌟

SOLUTION of (10-5.l):

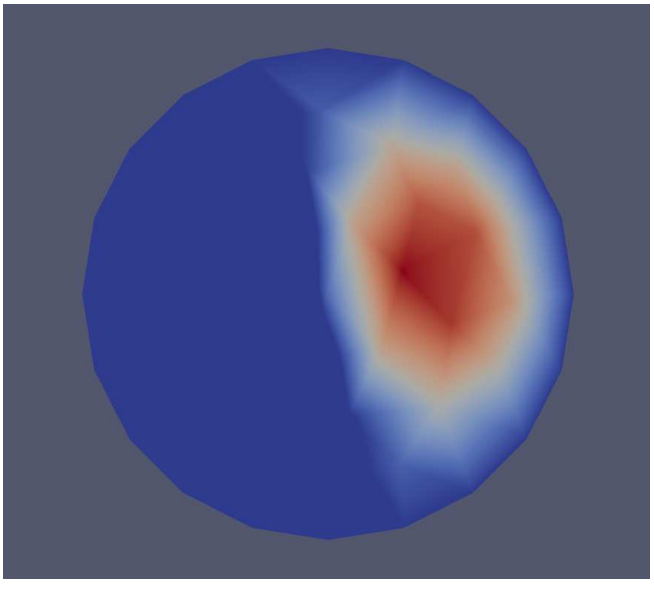
C++ code 10.5.22: Sub-problem (10-5.l): Implementation of `solvetrp` ⇢ GitLab

```
Eigen::VectorXd solvetrp(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    Eigen::VectorXd u0_vector, int N, double T) {
  // time step size
  double tau = T / N;

  // coefficient function for the semiLagr and reaction step
  auto v = [](Eigen::Vector2d x) {
    return (Eigen::Vector2d() << -x(1) + 3.0 * x(0) * x(0), x(0)).finished();
  };
  auto c = [](Eigen::Vector2d x) { return -6.0 * x(0); };

  SemiLagrStep semiLagr(fe_space, v);
  ReactionStep reaction(fe_space, c);
  // Strang splitting scheme
  //---------------
  // first SemiLagr half step:
  u0_vector = semiLagr.step(u0_vector, 0.5 * tau);

  // intermediate time steps:  Combine two semiLagr half steps to one
     step
  for (int i = 0; i < N - 1; ++i) {
    u0_vector = reaction.step(u0_vector, tau);
    u0_vector = semiLagr.step(u0_vector, tau);
  }

  // final reaction step and semiLagr half step
  u0_vector = reaction.step(u0_vector, tau);
  u0_vector = semiLagr.step(u0_vector, tau * 0.5);

  return u0_vector;

}
```

SOLUTION of (10-5.m):

C++ code 10.5.23: Sub-problem (10-5.l): Implementation of `vistrp` ↬ GitLab

```
void vistrp() {

  // The equation is solved on the test mesh circle.msh
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  lf::io::GmshReader reader(std::move(mesh_factory), "meshes/circle.msh");
  auto mesh_p = reader.mesh();

  // construct linear finite element space
  auto fe_space =
      std::make_shared<const lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);

  // initial conditions:
  const auto u0 = [](const Eigen::Vector2d& x) {
    const double r2 = x(1) * x(1) + x(0) * x(0);
    return (1 - r2) * x(0) / sqrt(r2);
  };

  const Eigen::VectorXd u0_vector = lf::fe::NodalProjection(
      *fe_space, lf::mesh::utils::MeshFunctionGlobal(u0));

  const Eigen::VectorXd sol_rot_20 =
      TranspSemiLagr::solvetrp(fe_space, u0_vector, 5, 0.2);

  const lf::fe::MeshFunctionFE mf_sol_rot_20(fe_space, sol_rot_20);

  lf::io::VtkWriter vtk_writer_rot_20(mesh_p, "./rot_20_trp.vtk");
  vtk_writer_rot_20.WritePointData("rot_20_trp", mf_sol_rot_20);

};
```



◁ Surface plot of finite-element solution at final time $T = 0.2$.

SOLUTION of (10-5.b):

Note that due to the cardinal basis property of the tent function basis $\{b_h^{\boldsymbol{p}}\}_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})\cap\Omega}$ of $\mathcal{S}^0_{1,0}(\mathcal{M})$

$$u_h^{(j)} = \sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})\cap\Omega}^{N} \mu_{\boldsymbol{p}}^{(j)} b_h^{\boldsymbol{p}} \quad \Leftrightarrow \quad \left(\vec{\boldsymbol{\mu}}^{(j)}\right)_\ell = \mu_{\boldsymbol{p}}^{(j)} \,, \quad \ell \hat{=} \text{ index of } \boldsymbol{p} \in \mathcal{N}(\mathcal{M})\cap\Omega \,.$$

Therefore, (10.5.3) has the structure (10.5.4) with

$$\begin{aligned} \mathbf{D}(\tau) &= \frac{1}{\tau}\operatorname{diag}\left(\tfrac{1}{3}|U_{\boldsymbol{p}}|\right)_{\boldsymbol{p}\in\mathcal{N}\cap\Omega} \,, \\ \left(\vec{\boldsymbol{\varphi}}(\tau, u_h^{(j-1)})\right)_{\boldsymbol{p}} &= \tfrac{1}{3}|U_{\boldsymbol{p}}|\left(\frac{1}{\tau}u_h^{(j-1)}(\boldsymbol{p}-\tau\mathbf{v}(\boldsymbol{p})) + f(\boldsymbol{p},t_j)\right) \,, \quad \boldsymbol{p}\in\mathcal{N}(\mathcal{M})\cap\Omega \,, \end{aligned} \tag{10.5.5}$$

and $\mathbf{A}$ is the $\mathcal{S}^0_{1,0}(\mathcal{M})$ finite element Galerkin matrix of $(u,v)\mapsto\int_\Omega \mathbf{grad}\, u\cdot\mathbf{grad}\, v\,\mathrm{d}\boldsymbol{x}$ using the standard tent-function basis of $\mathcal{S}^0_{1,0}(\mathcal{M})$.

Note that in (10.5.5) we used “indexing by mesh nodes”, tacitly assuming that those are numbered from 1 to N.

SOLUTION of (10-5.c):

In light of what we found in Sub-problem (10-5.b) the following data can be prepared in the constructor:

- The large sparse finite element Galerkin matrix $\mathbf{A} \in \mathbb{R}^{N,N}$. It can be built by standard assembly algorithms and stored in a *sparse matrix format*.
- The areas $|U_{\boldsymbol{p}}|$, $\boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega$, which can be stored in a vector.

Within `step()` we have to compute

- the coordinates of the points $\boldsymbol{q}(\boldsymbol{p}) := \boldsymbol{p} - \tau \mathbf{v}(\boldsymbol{p})$, $\boldsymbol{p} \in \mathcal{N}(\mathcal{M} \cap \Omega)$,
- the values $u_h^{(j-1)}(\boldsymbol{q})$ for all points $\boldsymbol{q} = \boldsymbol{q}(\boldsymbol{p})$,
- and the vector $\left[f(\boldsymbol{p}, t_j)\right]_{\boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega}$.

Remark. It the timestep $\tau > 0$ was assumed to be fixed, then the points $\boldsymbol{q}(\boldsymbol{p})$ could already be computed in the constructor. If f did not depend on time, this would also apply to the vector $\left[f(\boldsymbol{p}, t_j)\right]_{\boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega}$.

SOLUTION of (10-5.d):

- `Eigen::SparseMatrix<double> A_` of size $N \times N$ to store the finite-element Galerkin matrix $\mathbf{A}$
- `Eigen::VectorXd Up_`, a vector of length N, to store the values $|U_{\boldsymbol{p}}|$, $\boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega$.
- `Eigen::MatrixXd vp_`, a $2 \times N$-matrix, to store the velocity vectors $\mathbf{v}(\boldsymbol{p})$, $\boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega$.
- `fe_space_` containing a shared pointer to a **lf::uscalfe::FeSpaceLagrangeO1** object.

SOLUTION of (10-5.e):

(I) The assembly of the *full* $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix $\mathbf{A}$ in LEHRFEM++ is standard and details how to obtain a corresponding **lf::assemble::COOMatrix** object can be seen from [Lecture → Code 2.7.6.20 Use `lf::assemble::AssembleMatrixLocally()` and **lf::uscalfe::LinearFELaplaceElem**

In a second step we have to take care of the zero Dirichlet boundary conditions. This can be done by invoking the LEHRFEM++ function `FixFlaggedSolutionComponents()`, see [Lecture → § 2.7.6.13]. You have to use a "dummy right-hand side vector" and a helper lambda function in the process. This will yield the sparse Galerkin matrix in COO-format with all rows/columns corresponding to boundary nodes set to zero, with the exception of the diagonal entries, which are set to 1, see [Lecture → Eq. (2.7.6.15)].

Finally, one creates a sparse matrix in CCS-format by calling the `makeSparse()` method of the **lf::assemble::COOMatrix** object.

(II) In order to compute the values $|U_p$, $p \in \mathcal{N}(\mathcal{M}) \cap \Omega$, you can relay on `lf:assemble::Assembl` supplied with a **ENTITY_VECTOR_PROVIDER**, which returns a 3×3 identity matrix weighted with $\frac{1}{3}$ the area of the triangle.

C++ code 10.5.6: Sub-problem (10-5.d): Constructor of SemiLagrStep ↬ GitLab

```
2   SemiLagrStep(
3       std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
4       FUNCTOR v)
5       : fe_space_(fe_space),
6         A_(fe_space->LocGlobMap().NumDofs(), fe_space->LocGlobMap().NumDofs()),
7         Atmp_(fe_space->LocGlobMap().NumDofs(),
8               fe_space->LocGlobMap().NumDofs()),
9         Up_(fe_space->LocGlobMap().NumDofs()),
10        vp_(2, fe_space->LocGlobMap().NumDofs()),
11        N_dofs_(fe_space->LocGlobMap().NumDofs())
12  {
13    // precalculate the finite element Galerkin matrix A for the
        bilinear
14    // form (u,v) ↦ ∫_Ω gradugradvdx associated with
15    // −Δ.  Use EIGEN's built-in facilities for the computation of the
16    // corresponding element matrices.
17    lf::assemble::COOMatrix<double> A_COO(N_dofs_, N_dofs_);
18    lf::uscalfe::ReactionDiffusionElementMatrixProvider
19        galerkin_element_matrix_provider(
20            fe_space_, lf::mesh::utils::MeshFunctionConstant(1.0),
21            lf::mesh::utils::MeshFunctionConstant(0.0));
22    lf::assemble::AssembleMatrixLocally(
23        0, fe_space->LocGlobMap(), fe_space->LocGlobMap(),
24        galerkin_element_matrix_provider, A_COO);
25    Eigen::VectorXd b_dummy = Eigen::VectorXd::Zero(N_dofs_);
26    enforce_zero_boundary_conditions(fe_space_, A_COO, b_dummy);
27
28    A_ = A_COO.makeSparse();
29    // NOLINTNEXTLINE(cppcoreguidelines-prefer-member-initializer)
30    Atmp_ = A_;
31
32    // Precalculate the quadrature weight contributions Up_.
33    // Note that mass_matrix will be a diagonal matrix
```

```
    // because the eval function of LumpedMassElementMatrixProcider always
    // returns diagonal matrices.  Could be implemented also without the detaour
    // via a Eigen::SparseMatrix.
    LumpedMassElementMatrixProvider lumped_mass_element_matrix_provider(
        [](const Eigen::Vector2d& /*x*/) { return 1.; });
    lf::assemble::COOMatrix<double> mass_matrix(N_dofs_, N_dofs_);
    lf::assemble::AssembleMatrixLocally(
        0, fe_space->LocGlobMap(), fe_space->LocGlobMap(),
        lumped_mass_element_matrix_provider, mass_matrix);
    Eigen::SparseMatrix<double> mass_sparse = mass_matrix.makeSparse();
    Up_ = mass_sparse.diagonal();

    // precalculate vp_, v evaluated at each node in the mesh
    for (const auto& node : fe_space->Mesh()->Entities(2)) {
      const lf::geometry::Geometry* geo_ptr = node->Geometry();
      Eigen::Vector2d position = lf::geometry::Corners(*geo_ptr);
      const lf::assemble::size_type idx = fe_space->Mesh()->Index(*node);
      vp_.col(idx) = v(position);
    }
  };
```

HINT 1 for (10-5.f): Exploit the structure (10.5.4) of the linear system to be solved: You need only change the diagonal of the system matrix in every step. ⌟

HINT 2 for (10-5.f): The challenging part is to compute the values $u_h^{(j-1)}(\boldsymbol{p} - \tau \mathbf{v}(\boldsymbol{p}))$ (efficiently). A sub-optimal solution is

1. to compute the barycentric coordinates of $\boldsymbol{q}$ with respect to the triangle K with vertices $\boldsymbol{a}_K^1 = \begin{bmatrix} a_1^1 \\ a_2^1 \end{bmatrix}$, $\boldsymbol{a}_K^2 = \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix}$, and $\boldsymbol{a}_K^3 = \begin{bmatrix} a_1^3 \\ a_2^3 \end{bmatrix}$ by solving the small linear system of equations

$$\begin{bmatrix} a_1^1 & a_1^2 & a_1^3 \\ a_2^1 & a_2^2 & a_2^3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} q_1 \\ q_2 \\ 1 \end{bmatrix}, \quad \boldsymbol{q} = \begin{bmatrix} q_1 \\ q_2 \end{bmatrix}.$$

   If $\lambda_i > 0$ for all $i = 1, 2, 3$, then the point $\boldsymbol{q}$ is located in the interior of the triangle K.

2. Run through all triangles to find the one containing the point $\boldsymbol{q}$. In case $\boldsymbol{p} - \tau \mathbf{v}(\boldsymbol{p}) \notin \Omega$, use $u_h^{(j-1)}(\boldsymbol{p} - \tau \mathbf{v}(\boldsymbol{p})) = 0$.

⌟

SOLUTION of (10-5.f):

Notice that only the diagonal of the system matrix of (10.5.4) and the right-hand side vector will change from timestep to timestep.

So it might be a good idea to introduce another data member `Atmp_`, initialize it with a copy of `A_` in the constructor and then use EIGEN's `coeffRef()` entry access method to modify the diagonal entries in each call to `step()`.

Note that when solving the linear systems of equations with coefficient matrix `Atmp_` the entries of the right-hand side vector corresponding to nodes on $\partial\Omega$ have to be set to zero to enforce the correct Dirichlet boundary conditions.

C++ code 10.5.7: Sub-problem (10-5.f): `step()` member function of SemiLagrStep ➙ GitLab

```
2   Eigen::VectorXd step(const Eigen::VectorXd& u0_vector, double tau) {
3     // add the scaled diagonal values of Up_ to the matrix
4     for (int i = 0; i < N_dofs_; i++) {
5       Atmp_.coeffRef(i, i) = A_.coeff(i, i) + Up_(i) / tau;
6     }
7     // Warp u0 into a mesh function (required by the Vector provider) &
8     // assemble rhs.
9     auto u0_mf = lf::fe::MeshFunctionFE(fe_space_, u0_vector);
10    UpwindLagrangianElementVectorProvider vector_provider(
11        vp_, tau, fe_space_->Mesh(), u0_mf);
12
13    Eigen::VectorXd b = Eigen::VectorXd::Zero(N_dofs_);
14    lf::assemble::AssembleVectorLocally(0, fe_space_->LocGlobMap(),
15                                        vector_provider, b);
16    // set rhs vector to zero for all boundary nodes to ensure
17    // that the Dirichlet boundary conditions are enforced correctly
18    lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
19        lf::mesh::utils::flagEntitiesOnBoundary(fe_space_->Mesh(), 2)};
20    auto entities = fe_space_->Mesh()->Entities(2);
21    for (int i = 0; i < b.size(); i++) {
22      if (bd_flags(*entities[i])) {
23        b(i) = 0.;
24      }
25    }
26    // solve LSE
27    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
28    solver.compute(Atmp_);
29    return solver.solve(b);
30  }
```

SOLUTION of (10-5.g):

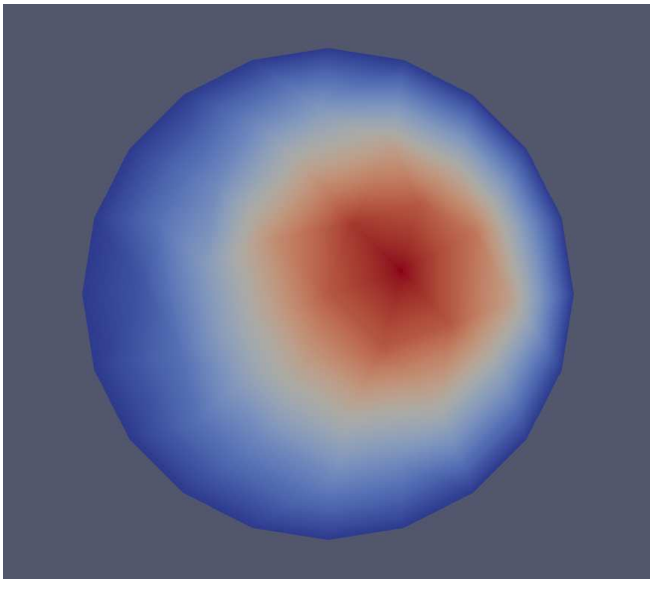
C++ code 10.5.9: Sub-problem (10-5.g): Implementation of `solverot` ⇢ GitLab

```
Eigen::VectorXd solverot(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    Eigen::VectorXd u0_vector, int N, double T) {
  // time step size
  double tau = T / N;

  // velocity field of the model problem
  auto v = [](Eigen::Vector2d x) {
    return (Eigen::Vector2d() << -x(1) + 3.0 * x(0) * x(0), x(0)).finished();
  };

  SemiLagrStep semiLagr(fe_space, v);
  // approximate solution based on N uniform time steps.
  for (int i = 0; i < N; ++i) {
    u0_vector = semiLagr.step(u0_vector, tau);
  }

  return u0_vector;
}
```

SOLUTION of (10-5.h):

C++ code 10.5.11: Sub-problem (10-5.g): Implementation of `visSLSolution()` ⇝ GitLab

```
void visSLSolution() {

  // The equation is solved on the test mesh circle.msh
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  lf::io::GmshReader reader(std::move(mesh_factory), "meshes/circle.msh");
  auto mesh_p = reader.mesh();

  // construct linear finite element space
  auto fe_space =
      std::make_shared<const lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);

  // initial conditions:
  const auto u0 = [](const Eigen::Vector2d& x) {
    const double r2 = x(1) * x(1) + x(0) * x(0);
    return (1 - r2) * x(0) / sqrt(r2);
  };

  const Eigen::VectorXd u0_vector = lf::fe::NodalProjection(
      *fe_space, lf::mesh::utils::MeshFunctionGlobal(u0));

  const Eigen::VectorXd sol_rot_20 =
      TranspSemiLagr::solverot(fe_space, u0_vector, 5, 0.2);

  const lf::fe::MeshFunctionFE mf_sol_rot_20(fe_space, sol_rot_20);

  lf::io::VtkWriter vtk_writer_rot_20(mesh_p, "./rot_20_sl.vtk");
  vtk_writer_rot_20.WritePointData("rot_20_sl", mf_sol_rot_20);

};
```



◁ Visualization of finite-element solution at final time.

HINT 1 for (10-5.i): Look up [Lecture → Lemma 1.5.2.1].

SOLUTION of (10-5.i):

The product rule of [Lecture → Lemma 1.5.2.1] gives us

$$\operatorname{div}(\mathbf{v}\,u)(\boldsymbol{x}) = \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u(\boldsymbol{x}) + \operatorname{div}\mathbf{v}(\boldsymbol{x})\, u(\boldsymbol{x}) \quad \text{for} \quad \mathbf{v} \in (C^1(\Omega))^2\,, \quad u \in C^1(\Omega)\,.$$

Hence we have $c(\boldsymbol{x}) = -\operatorname{div}(\mathbf{v}(\boldsymbol{x}))$.

HINT 1 for (10-6.a): The streamlines for the velocity field $\boldsymbol{x} \mapsto \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}$ are concentric circles. ⌟

SOLUTION of (10-6.a):

We apply the method of characteristics from [Lecture → § 10.2.1.7]. The given velocity field is that of a rigid rotation around $\mathbf{0}$. Hence the streamlines are concentric circles with center $\mathbf{0}$. As $f \equiv 0$ the boundary values are just transported along the streamlines. Note that only the boundary data on the inflow boundary, i.e. $\{\boldsymbol{x} \in \partial\Omega\colon \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\}$ gets transported into the domain. In this case, the inflow boundary consists of $\Gamma_1 = \{\boldsymbol{x} \in \partial\Omega\colon x_1 = 1\}$ and $\Gamma_2 = \{\boldsymbol{x} \in \partial\Omega\colon x_2 = 0\}$. The concentric circles eminating from Γ_2, will yield the following solution for $\|\boldsymbol{x}\| < 1$

$$u(\boldsymbol{x}) = \sin^2(\pi\|\boldsymbol{x}\|)\,, \quad \boldsymbol{x} \in \{\boldsymbol{x} \in \Omega; \|\boldsymbol{x}\| < 1\}\,.$$

The concentric circles eminating from Γ_1, will yield the following solution for $\|\boldsymbol{x}\| \geq 1$

$$u(\boldsymbol{x}) = 0\,, \quad \boldsymbol{x} \in \{\boldsymbol{x} \in \Omega; \|\boldsymbol{x}\| \geq 1\}\,.$$

We conclude that the final solution is

$$u(\boldsymbol{x}) = \begin{cases} \sin^2(\pi\|\boldsymbol{x}\|), & \|\boldsymbol{x}\| < 1\,, \\ 0 & \|\boldsymbol{x}\| \geq 1\,. \end{cases} \tag{10.6.4}$$

SOLUTION of (10-6.b):

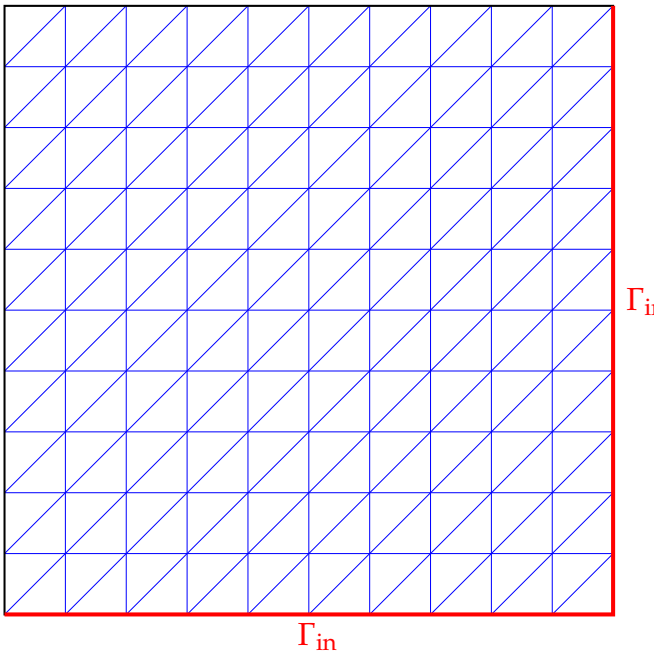
For the finite element space $\mathcal{S}_2^0(\mathcal{M})$ there is exactly one global shape function associated with every node of $\mathcal{M}$, and exactly one with every edge. This yields, by a simple counting argument,

$$\dim \mathcal{S}_2^0(\mathcal{M}) = (N+1)^2 + 2(N+1)N + N^2 = (2N+1)^2 \,.$$

For the vector field $\boldsymbol{x} \mapsto \left[\begin{smallmatrix} -x_2 \\ x_1 \end{smallmatrix}\right]$ and $\Omega =]0,1[^2$ we find

$$\Gamma_{\text{in}} = [0,1] \times \{0\} \cup \{1\} \times [0,1] \,,$$

marked with red in the figure beside. ▷



However, global shape functions on the (the closure of) the inflow boundary part Γ_{in} do not contribute unknowns, because their values are fixed by the Dirichlet data.

We count that

$$2N + 1 + 2N = 4N + 1$$

global shape functions belong to mesh entities lying on $\overline{\Gamma}_{\text{in}}$.

Subtracting that number from $\dim \mathcal{S}_2^0(\mathcal{M})$ we find

$$\text{No. of FE unknowns} = (2N+1)^2 - 4N - 1 = \boxed{4N^2} \,.$$

Remark. Why do we regards degrees of freedom associated with mesh entities contained in the *closure* of Γ_{in} as known? This question is relevant for the treatment of the global shape function sitting on the corner node in $\left[\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right]$. The reason is that, obviously, no upstream cell exists for that node and, thus, there is no information fixing the value of u in that node. The only source of information is the Dirichlet data.

SOLUTION of (10-6.c):

From [Lecture → Def. 2.7.4.5] we learn that

$$\begin{aligned}(\mathbf{A}_K)_{i,j} &= \int_K \mathbf{v}(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})^\top \mathbf{grad}\, b_K^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_K^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ &= \int_K \left(\mathbf{v}(\boldsymbol{x})^\top \mathbf{grad}\, b_K^j(\boldsymbol{x})\right) \cdot \left(\mathbf{v}(\boldsymbol{x})^\top \mathbf{grad}\, b_K^i(\boldsymbol{x})\right) \mathrm{d}\boldsymbol{x}\,, \quad i,j \in \{1,\ldots,6\}\,,\end{aligned}$$

where $b_K^1,\ldots,b_K^6$ are the local shape functions on $K \in \mathcal{M}$. Since $\mathcal{S}_2^0(\mathcal{M})$ is a parametric finite element space, we have according to [Lecture → Def. 2.8.3.1]

$$\widehat{b}^j = \boldsymbol{\Phi}_K^* b_K^j\,, \quad \boldsymbol{\Phi}_K^* \;\widehat{=}\; \text{pullback [Lecture → Def. 2.8.1.2]}\,.$$

Then we follow the considerations of [Lecture → Section 2.8.3] and transform the integral to $\widehat{K}$ using the formula from [Lecture → Lemma 2.8.3.10]:

$$\begin{aligned}(\mathbf{A}_K)_{i,j} = \int_{\widehat{K}} \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))^\top \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\boldsymbol{x}}) \cdot \\ \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})^{-\top} \mathbf{grad}\, \widehat{b}^i(\widehat{\boldsymbol{x}})\,|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})|\mathrm{d}\widehat{\boldsymbol{x}}\,.\end{aligned}$$

Finally, we apply the quadrature rule:

$$\begin{aligned}(\mathbf{A}_K)_{i,j} = \sum_{\ell=1}^{6} \omega_\ell \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))^\top \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell) \cdot \\ \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)^{-\top} \mathbf{grad}\, \widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell)\,|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)|\,.\end{aligned} \tag{10.6.8}$$

HINT 1 for (10-6.d): You must not forget that for the approximation of all integrals over cells of the mesh we use the quadrature rule supplied by the LEHRFEM++ function `lf::quad::QuadRule make_TriaQR_P6O4()`.

SOLUTION of (10-6.d):

The derivation is the same as in the solution of Sub-problem (10-6.c) and we end up with

$$(\mathbf{B}_K)_{i,j} = \sum_{\ell=1}^{6} \omega_\ell \mathbf{v}(\mathbf{\Phi}_K(\widehat{\zeta}_\ell)) \cdot \mathrm{D}\mathbf{\Phi}_K(\widehat{\zeta}_\ell)^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\zeta}_\ell)\, \widehat{b}^i(\widehat{\zeta}_\ell)\, |\det \mathrm{D}\mathbf{\Phi}_K(\widehat{\zeta}_\ell)| \,. \tag{10.6.10}$$

HINT 1 for (10-6.e): The code [Lecture → Code 2.8.3.31] may give you some ideas. ⌟

HINT 2 for (10-6.e): The flag `use_delta_` should be ignored. ⌟

HINT 3 for (10-6.e): You may use the variables `val_ref_lsf_` and `grad_ref_lsf_` defined in the class body. ⌟

SOLUTION of (10-6.e):

We pick $K \in \mathcal{M}$, use the notations of Sub-problem (10-6.c) and write $\mathbf{M}_K$ for the element matrix to be returned by `Eval()`. From the formulas (10.6.11), (**??**), and (10.6.9) we see

$$\mathbf{M}_K = \delta_K \mathbf{A}_K + \mathbf{B}_K\,, \quad \delta_K := \delta|_K \quad \text{from (10.6.6)}\,,$$

which means

$$\begin{aligned}(\mathbf{M}_K)_{i,j} = \sum_{\ell=1}^{6} \omega_\ell \Big(&\delta_K \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))^\top \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)^{-\top} \mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell) \cdot \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)^{-\top} \mathbf{grad}\,\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell) \\ &+ \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)) \cdot \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)^{-\top} \mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)\,\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell) \Big) |\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)|\,,\end{aligned} \tag{10.6.13}$$

$i,j \in \{1,\ldots,6\}$. Abbreviating

$$\mathbf{v}_\ell := \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\,, \quad \mathbf{g}_\ell^j := \mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)\,, \quad \beta_\ell^i := b^i(\widehat{\boldsymbol{\zeta}}_\ell)\,, \quad \mathbf{J}_\ell := \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)^{-\top}\,,$$

we arrive at the more compact formula

$$(\mathbf{M})_{i,j} = \sum_{\ell=1}^{6} \omega_\ell \Big(\delta_K \mathbf{v}_\ell \mathbf{v}_\ell^\top \mathbf{J}_\ell \mathbf{g}_\ell^j \cdot \mathbf{J}_\ell \mathbf{g}_\ell^i + \mathbf{v}_\ell \cdot \mathbf{J}_\ell \mathbf{g}_\ell^j\, \beta_\ell^i \Big) |\det \mathbf{J}_\ell| \tag{10.6.14}$$

Quantities related to local shape functions on $\widehat{K}$ are accessible through **lf::uscalfe::FeLagrangeO2Tria<do** and the corresponding member functions are called in the constructor, see Code 10.6.12. Therefore, we can extract (C++ indexing now!)

$$\begin{aligned}\beta_\ell^i &\leftrightarrow \texttt{val_ref_lsf_}(i,\ell)\,,\\ \mathbf{g}_\ell^j &\leftrightarrow \texttt{grad_ref_lsf_.block}(j,2*\ell,1,2)\texttt{.transpose()}\,,\end{aligned}$$

$i,j,\ell \in \{0,\ldots,5\}$.

Information about $\mathbf{J}_\ell$ and $|\det \mathbf{J}_\ell|$ is provided by member functions of the **lf::geometry::Geometry** object associated with K, see [Lecture → § 2.8.3.14]. Set

```
dets  = cell.Geometry()->IntegrationElement(qr_.Points());
JinvT = cell.Geometry()->JacobianInverseGramian(qr_.Points());
```

Then we have (C++ indexing !)

$$\begin{aligned}\mathbf{J}_\ell &\leftrightarrow \texttt{JinvT.block}(0,2\ell,2,2)\\ |\det \mathbf{j}_\ell| &\leftrightarrow \texttt{dets}[\ell]\,,\end{aligned} \quad l \in \{0,\ldots,5\}\,, \tag{10.6.15}$$

and the transformed gradients can be computed as in [Lecture → Code 2.8.3.31]:

$$\mathbf{J}_\ell \left[\mathbf{g}_\ell^0,\ldots,\mathbf{g}_\ell^5\right] \leftrightarrow \texttt{JinvT.block}(0,2\ell,2,2) * \texttt{grad_ref_lsf_.block}(0,2*\ell,6,2)\texttt{.transpose()}\,,$$

$\ell \in \{0,\ldots,5\}$. Setting $\mathbf{t}_\ell^j := \mathbf{J}_\ell \mathbf{g}_\ell^j$, we get the most compact formulas (C++ indexing, $i,j \in \{0,\ldots,5\}$)

$$(\mathbf{M})_{i,j} = \sum_{\ell=0}^{5} \omega_\ell |\det \mathbf{J}_\ell| \Big(\delta_K \Big(\mathbf{v}_\ell^\top \mathbf{t}_\ell^j \Big) \Big(\mathbf{v}_\ell^\top \mathbf{t}_\ell^i \Big) + \mathbf{v}_\ell \cdot \mathbf{t}_\ell^j\, \beta_\ell^i \Big)\,, \tag{10.6.16}$$

$$\mathbf{M} = \sum_{\ell=0}^{5} \omega_\ell |\det \mathbf{J}_\ell| \left(\delta_K \mathbf{m}_\ell + \left[\beta_\ell^i \right]_{i=0}^{5} \right) \mathbf{m}_\ell^\top , \quad \mathbf{m}_\ell := \left[\mathbf{v}_\ell^\top \mathbf{t}_\ell^i \right]_{i=0}^{5} \in \mathbb{R}^6 . \tag{10.6.17}$$

The following code implements (10.6.16).

C++ code 10.6.18: Implementation of `Eval()` member function ↪ GitLab

```
template <class MESHFUNCTION_V>
typename SUAdvectionElemMatrixProvider<MESHFUNCTION_V>::ElemMat
SUAdvectionElemMatrixProvider<MESHFUNCTION_V>::Eval(
    const lf::mesh::Entity &cell) {
  LF_ASSERT_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Only implemented for triangles");
  // Obtain geometry information
  const lf::geometry::Geometry *geo_ptr = cell.Geometry();
  LF_ASSERT_MSG(geo_ptr->DimGlobal() == 2,
                "Only implemented for planar triangles");
  // Gram determinant at quadrature points
  const Eigen::VectorXd dets(geo_ptr->IntegrationElement(qr_.Points()));
  LF_ASSERT_MSG(dets.size() == qr_.NumPoints(),
                "Mismatch " << dets.size() << " <-> " << qr_.NumPoints());
  // Fetch the transformation matrices for the gradients
  const Eigen::MatrixXd JinvT(geo_ptr->JacobianInverseGramian(qr_.Points()));
  LF_ASSERT_MSG(JinvT.cols() == 2 * qr_.NumPoints(),
                "Mismatch " << JinvT.cols() << " <-> " << 2 * qr_.NumPoints());
  // Obtain values of velocity at the 6 quadrature points
  std::vector<Eigen::Vector2d> v_vals = v_(cell, qr_.Points());
  // 6 x 6 element matrix (There are six local shape functions.)
  ElemMat mat(6, 6);
  mat.setZero();
  // Compute the factor δ_K
  const Eigen::MatrixXd vertices{lf::geometry::Corners(*geo_ptr)};
  // Size of triangle
  const double hK = std::max({(vertices.col(1) - vertices.col(0)).norm(),
                              (vertices.col(2) - vertices.col(1)).norm(),
                              (vertices.col(0) - vertices.col(2)).norm()});
  // Maximal modulus of velocity in quadrature nodes
  const double vmax =
      std::max_element(v_vals.begin(), v_vals.end(),
                       [](Eigen::Vector2d &a, Eigen::Vector2d &b) -> bool {
                         return (a.norm() < b.norm());
                       })
          ->norm();
  const double delta_K = use_delta_ ? (hK / (2.0 * vmax)) : 1.0;
  // Outer summation loop over quadrature points
  for (unsigned int l = 0; l < 6; ++l) {
    // Metric factor |det DΦ_K(ζ̂_ℓ)| scaled with
    // quadrature weight ω_ℓ
    const double fac = qr_.Weights()[l] * dets[l];
    // Compute transformed gradients t_ℓ^i of all local shape
    // functions and collect them in the columns of a 2 x 6-matrix
    const Eigen::Matrix<double, 2, 6> TrfG =
        JinvT.block(0, 2 * l, 2, 2) *
        (grad_ref_lsf_.block(0, 2 * l, 6, 2).transpose());
    // Compute the inner products of the transformed gradients with the
    // velocity vector at the current quadrature point.
    const Eigen::Matrix<double, 1, 6> mvec = v_vals[l].transpose() * TrfG;
    // Loop over columns of element matrix
```

```
53      for (unsigned int j = 0; j < 6; ++j) {
54        // Loop over rows of element matrix
55        for (unsigned int i = 0; i < 6; ++i) {
56          mat(i, j) +=
57              fac * (delta_K * mvec[i] * mvec[j] + mvec[j] * val_ref_lsf_(i, l));
58        }
59      }
60    }
61    return mat;
62  }
```

Remark. The flag `use_delta` makes it possible to switch off the scaling bny δ_K. This is useful for testing purposes.

HINT 1 for (10-6.f): The treatment of the non-zero Dirichlet boundary conditions requires particular care. You may call `lf::assemble::FixFlaggedSolutionCompAlt()` with a **SELECTOR** argument that picks basis functions associated with entities on Γ_{in}. To determined, if a mesh entity belongs to Γ_{in} you may simply rely on testing

$$(\boldsymbol{m})_1 = 1 \quad \text{or} \quad (\boldsymbol{m})_2 = 0$$

for the center of gravity (midpoint) $\boldsymbol{m}$ of a node or an edge. For the sake of efficiency you can use an auxiliary object of type **lf::mesh::utils::AllCodimMeshDataSet<bool>** that flags mesh entities on Γ_{in}. ⌟

SOLUTION of (10-6.f):

In Sub-problem (10-6.e) we have already implemented the **ENTITY_MATRIX_PROVIDER** helper class that can take care of computing element matrices for triangular cells. We use it as an argument to a suitable instance of the function `lf::assemble::AssembleMatrixLocally()` to build the Galerkin matrix for the streamline upwind method, see [Lecture → Code 2.7.4.40] for a demonstration code.

C++ code 10.6.19: Implementation the function `solveRotSUPG()` ↬ GitLab

```
2  Eigen::VectorXd solveRotSUPG(
3      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fe_space) {
4    // Velocity field
5    auto vf = [](Eigen::Vector2d x) -> Eigen::Vector2d { return {-x[1], x[0]}; };
6    lf::mesh::utils::MeshFunctionGlobal mf_v(vf);
7    // Initialize the Matrix provider
8    SUAdvectionElemMatrixProvider elmat_builder(mf_v);
9    // Build the lhs Matrix
10   lf::assemble::COOMatrix<double> A(fe_space->LocGlobMap().NumDofs(),
11                                     fe_space->LocGlobMap().NumDofs());
12   lf::assemble::AssembleMatrixLocally(0, fe_space->LocGlobMap(),
13                                       fe_space->LocGlobMap(), elmat_builder, A);
14   Eigen::VectorXd b(fe_space->LocGlobMap().NumDofs());
15   b.setZero();
16   // Modify linear system to account for (essential) Dirichlet boundasry
17   // conditions
18   enforce_boundary_conditions(fe_space, A, b);
19   // Solve the linear system of equations
20   const Eigen::SparseMatrix<double> A_sparse = A.makeSparse();
21   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
22   solver.compute(A_sparse);
23   return solver.solve(b);
24 };
```

C++ code 10.6.20: Implementation the function `enforce_boundary_conditions()` ↬ GitLab

```
2  void enforce_boundary_conditions(
3      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fe_space,
4      lf::assemble::COOMatrix<double>& A, Eigen::VectorXd& b) {
5    const lf::mesh::utils::MeshFunctionGlobal mf_g_Gamma_in{
6        [](const Eigen::Vector2d& x) { return std::pow(sin(M_PI * x(0)), 2); }};
7
8    lf::mesh::utils::AllCodimMeshDataSet<bool> bd_flags(fe_space->Mesh(), false);
9    // Loop over all edges
10   for (const auto& edge : fe_space->Mesh()->Entities(1)) {
11     LF_ASSERT_MSG(edge->RefEl() == lf::base::RefEl::kSegment(),
12                   "Entity should be an edge");
13     const lf::geometry::Geometry* geo_ptr = edge->Geometry();
14     const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_ptr);
15     // Check if the edge lies on Γ_in (geometric test)
16     if ((corners(0, 0) + corners(0, 1)) / 2. > 1. - 1e-5 ||
17         (corners(1, 0) + corners(1, 1)) / 2. < 1e-5) {
18       // Add the edge to the flagged entities
19       bd_flags(*edge) = true;
```

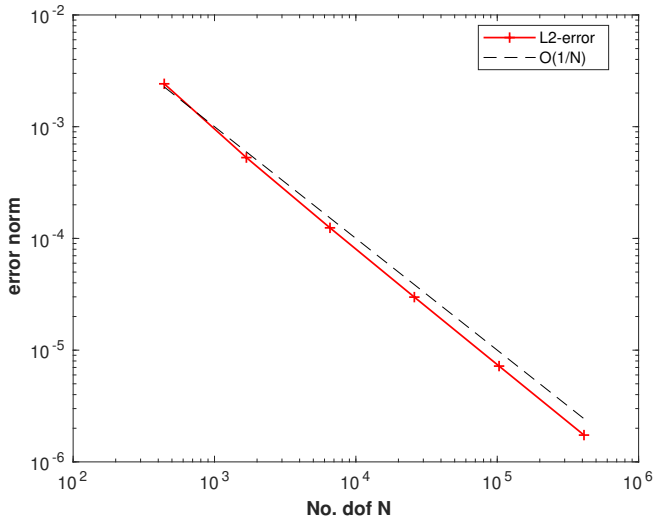
```
20      }
21    }
22    // Loop over all Points
23    for (const auto& point : fe_space->Mesh()->Entities(2)) {
24      LF_ASSERT_MSG(point->RefEl() == lf::base::RefEl::kPoint(),
25                    "Entity should be an edge");
26      const lf::geometry::Geometry* geo_ptr = point->Geometry();
27      const Eigen::VectorXd coords = lf::geometry::Corners(*geo_ptr);
28      // Check if the node lies on Γ_in (geometric test)
29      if (coords(0) > 1. - 1e-5 || coords(1) < 1e-5) {
30        // Add the edge to the flagged entities
31        bd_flags(*point) = true;
32      }
33    }
34    auto flag_values{lf::fe::InitEssentialConditionFromFunction(
35        *fe_space, bd_flags, mf_g_Gamma_in)};
36
37    lf::assemble::FixFlaggedSolutionCompAlt<double>(
38        [&flag_values](lf::assemble::glb_idx_t dof_idx) {
39          return flag_values[dof_idx];
40        },
41        A, b);
42  };
```

HINT 1 for (10-6.g): Study [Lecture → Ex. 3.2.3.4] to learn how to compute $\|u - u_h\|_{L^2(\Omega)}$ in LEHRFEM+

SOLUTION of (10-6.g):

C++ code 10.6.21: Implementation the function `cvgL2SUPG()` ↬ GitLab

```
void cvgL2SUPG() {
  // Generate triangular mesh of the unit square
  lf::mesh::utils::TPTriagMeshBuilder builder(
      std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2));
  builder.setBottomLeftCorner(Eigen::Vector2d{0., 0.})
      .setTopRightCorner(Eigen::Vector2d{1., 1.});

  // Initialize a vector that will hold all the approximated solutions
  std::vector<Eigen::VectorXd> u_h(8);
  // Initialize a vector that will hold all the approximate solutions
  std::vector<double> err(8);
  // Initialize a mesh function that will represent the analytical
     solution
  const lf::mesh::utils::MeshFunctionGlobal mf_u{
      [](const Eigen::Vector2d& x) -> double {
        if (x.norm() > 1.) return 0.;
        return std::pow(sin(M_PI * x.norm()), 2);
      }};

  std::cout << "N" << std::setw(8) << "|"
            << "L2 Error" << std::endl;

  // Compute error on 6 meshes of dyadically increasing resolution
  for (int i = 0; i < 6; i++) {
    // Refine mesh
    builder.setNumXCells(10 * pow(2, i)).setNumYCells(10 * pow(2, i));
    // Get the mesh
    auto mesh_p = builder.Build();
    // Set up finite element space
    auto fe_space =
        std::make_shared<lf::uscalfe::FeSpaceLagrangeO2<double>>(mesh_p);
    // Compute the solution
    u_h[i] = AdvectionSUPG::solveRotSUPG(fe_space);

    const lf::fe::MeshFunctionFE mf_sol(fe_space, u_h[i]);
    // Compute the error
    err[i] = std::sqrt(lf::fe::IntegrateMeshFunction(
        *mesh_p, lf::mesh::utils::squaredNorm(mf_sol - mf_u), 4));
    // Print the error in console
    std::cout << fe_space->LocGlobMap().NumDofs() << std::setw(20) << "|"
              << err[i] << std::endl;
    if (i == 5) {
      visSolution(fe_space, u_h[i]);
    }
  }
};
```



◁ Doubly logarithmic plot of the $L^2(\Omega)$-norm of the discretization error versus the dimension N of the finite-element space

The error points nicely lie on a line with slope $O(N^{-1})$.

We conclude algebraic convergence in $\frac{1}{N}$ with rate 1. Since for the tensor-product triangular mesh of Fig. 160 we obviously have $h^2 \sim N^{-1}$, this amounts to an asymptotic algebraic convergence $O(h^2)$ for meshwidth $h \to 0$.

HINT 1 for (10-7.a): The streamlines for the velocity field $\boldsymbol{x} \mapsto \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}$ are concentric circles. ⌟

SOLUTION of (10-7.a):

We apply the method of characteristics from [Lecture → § 10.2.1.7]. The given velocity field is that of a rigid rotation around $\mathbf{0}$. Hence the streamlines are concentric circles with center $\mathbf{0}$. As $f \equiv 0$ the boundary values are just transported along the streamlines. Note that only the boundary data on the inflow boundary, i.e. $\{\boldsymbol{x} \in \partial\Omega\colon \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\}$ gets transported into the domain. In this case, the inflow boundary consists of $\Gamma_1 = \{\boldsymbol{x} \in \partial\Omega\colon x_1 = 1\}$ and $\Gamma_2 = \{\boldsymbol{x} \in \partial\Omega\colon x_2 = 0\}$. The concentric circles emanating from Γ_2, will yield the following solution for $\|\boldsymbol{x}\| < 1$

$$u(\boldsymbol{x}) = \|\boldsymbol{x}\|(1 - \|\boldsymbol{x}\|)\,, \quad \boldsymbol{x} \in \{\boldsymbol{x} \in \Omega; \|\boldsymbol{x}\| < 1\}\,.$$

The concentric circles emanating from Γ_1, will yield the following solution for $\|\boldsymbol{x}\| \geq 1$

$$u(\boldsymbol{x}) = 0\,, \quad \boldsymbol{x} \in \{\boldsymbol{x} \in \Omega; \|\boldsymbol{x}\| \geq 1\}\,.$$

We conclude that the final solution is

$$u(\boldsymbol{x}) = \begin{cases} \|\boldsymbol{x}\|(1 - \|\boldsymbol{x}\|), & \|\boldsymbol{x}\| < 1\,, \\ 0 & \|\boldsymbol{x}\| \geq 1\,. \end{cases} \tag{10.7.5}$$

HINT 1 for (10-7.b): The class **AdvectionElementMatrixProvider** is adapted from the code of **lf::uscalfe** whose `Eval()` method is listed in [Lecture → Code 2.8.3.31]. The highlighted part of that listing has to be replaced by proper code specific for the bilinear form b. ⌟

SOLUTION of (10-7.b):

We follow the considerations of [Lecture → § 2.8.3.6] and pick a mesh cell K. We write $\left\{b_K^1,\ldots,b_K^Q\right\}$, $\left\{\widehat{b}^1,\ldots,\widehat{b}^Q\right\}$, $Q\in\mathbb{N}$, for the set of local shape functions on K and the reference element $\widehat{K}$, respectively. Then the element matrix $\mathbf{B}\in\mathbb{R}^{Q,Q}$ for a mesh cell is defined as

$$(\mathbf{B}_K)_{i,j}=\int_K \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\, b_K^j(\boldsymbol{x})\, b_K^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,,\quad 1\le i,j\le Q\,. \tag{10.7.13}$$

Writing $\boldsymbol{\Phi}_K:\widehat{K}\to K$ for the mapping onto the actual cell K, we transform back to the reference element and get

$$(\mathbf{B}_K)_{i,j}=\int_{\widehat{K}} \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\cdot \mathrm{D}\boldsymbol{\Phi}_K^{-\top}(\widehat{\boldsymbol{x}})\,\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{x}})\,\widehat{b}^i(\widehat{\boldsymbol{x}})|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})|\,\mathrm{d}\widehat{\boldsymbol{x}}\,,\quad 1\le i,j\le Q\,. \tag{10.7.14}$$

We approximate the integral using a P-point quadrature formula on the reference element

$$\int_{\widehat{K}} f(\widehat{\boldsymbol{x}})\,\mathrm{d}\widehat{\boldsymbol{x}}\approx\sum_{\ell=1}^P \widehat{\omega}_\ell f(\widehat{\boldsymbol{\zeta}}_\ell)\,, \qquad \text{([Lecture → Eq. (2.7.5.23)])}$$

which means that we implement

$$(\mathbf{B}_K)_{i,j}=\sum_{\ell=1}^P \widehat{\omega}_\ell \mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\cdot \mathrm{D}\boldsymbol{\Phi}_K^{-\top}(\widehat{\boldsymbol{\zeta}}_\ell)\,\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)\,\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell)|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)|\,,\quad 1\le i,j\le Q\,. \tag{10.7.15}$$

We introduce the abbreviations

$$\begin{aligned} g_\ell^j &:= (\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))^{-\top}\,\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)\,,\quad \ell\in\{1,\ldots,P\}\,,\quad j\in\{1,\ldots,Q\}\,,\\ \mathbf{G} &:= \left[g_\ell^1,\ldots g_\ell^Q\right]\in\mathbb{R}^{2,Q}\,, \end{aligned}$$

which paves the way for rewriting (10.7.15) in a more compact form:

$$\mathbf{B}_K=\sum_{\ell=1}^P \widehat{\omega}_\ell|\det \mathrm{D}\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell)|\begin{bmatrix}\widehat{b}^1(\widehat{\boldsymbol{\zeta}}_\ell)\\ \vdots\\ \widehat{b}^Q(\widehat{\boldsymbol{\zeta}}_\ell)\end{bmatrix}\mathbf{v}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))^\top\mathbf{G}\,. \tag{10.7.16}$$

C++ code 10.7.17: Implementation of `Eval()` method, which computes the element matrices $\mathbf{B}_K$

```
template <typename VELOCITY>
typename SUFEM::AdvectionElementMatrixProvider<VELOCITY>::ElemMat
AdvectionElementMatrixProvider<VELOCITY>::Eval(const lf::mesh::Entity &cell) {
  // Topological type of the cell
  const lf::base::RefEl ref_el{cell.RefEl()};
  // Obtain precomputed information about values of local shape
     functions
  // and their gradients at quadrature points.
  lf::uscalfe::PrecomputedScalarReferenceFiniteElement<double> &pfe =
      fe_precomp_[ref_el.Id()];
```

```
  if (!pfe.isInitialized()) {
    // Accident:  cell is of a type not covered by finite element
    // specifications or there is no quadrature rule available for this
    // reference element type
    std::stringstream temp;
    temp << "No local shape function information or no quadrature rule for "
            "reference element type "
         << ref_el;
    throw lf::base::LfException(temp.str());
  }

  // Query the shape of the cell
  const lf::geometry::Geometry *geo_ptr = cell.Geometry();
  LF_ASSERT_MSG(geo_ptr != nullptr, "Invalid geometry!");
  LF_ASSERT_MSG((geo_ptr->DimLocal() == 2),
                "Only 2D implementation available!");

  // Physical dimension of the cell (must be 2)
  const lf::base::dim_t world_dim = geo_ptr->DimGlobal();
  LF_ASSERT_MSG_CONSTEXPR(world_dim == 2, "Only available for flat domains");
  // Gram determinant at quadrature points
  const Eigen::VectorXd determinants(
      geo_ptr->IntegrationElement(pfe.Qr().Points()));
  LF_ASSERT_MSG(
      determinants.size() == pfe.Qr().NumPoints(),
      "Mismatch " << determinants.size() << " <-> " << pfe.Qr().NumPoints());
  // Fetch the transformation matrices for the gradients
  const Eigen::MatrixXd JinvT(
      geo_ptr->JacobianInverseGramian(pfe.Qr().Points()));
  LF_ASSERT_MSG(
      JinvT.cols() == 2 * pfe.Qr().NumPoints(),
      "Mismatch " << JinvT.cols() << " <-> " << 2 * pfe.Qr().NumPoints());
  LF_ASSERT_MSG(JinvT.rows() == world_dim,
                "Mismatch " << JinvT.rows() << " <-> " << world_dim);

  // Get the velocity vectors at quadrature points in the cell
  auto veloval = velo_(cell, pfe.Qr().Points());

  // Element matrix
  ElemMat mat(pfe.NumRefShapeFunctions(), pfe.NumRefShapeFunctions());
  mat.setZero();

  // Loop over quadrature points
  for (lf::base::size_type k = 0; k < pfe.Qr().NumPoints(); ++k) {
    const double w = pfe.Qr().Weights()[k] * determinants[k];
    // Matrix G whose columns contain transformed gradients of all local
    // shape functions
    const auto trf_grad(
        JinvT.block(0, 2 * static_cast<Eigen::Index>(k), world_dim, 2) *
        pfe.PrecompGradientsReferenceShapeFunctions()
            .block(0, 2 * k, mat.rows(), 2)
            .transpose());
    // Row vector of transformed gradients multiplied with velocity
    vectors
#if SOLUTION
    const auto velo_times_trf_grad(veloval[k].transpose() * trf_grad);
    mat +=
        w * (pfe.PrecompReferenceShapeFunctions().col(k) * velo_times_trf_grad);
```

```
68 #else
69     /*******************************************************
70     Your implementation to fill in 'mat' goes here
71     *******************************************************/
72 #endif
73   }
74   return mat;
75 }
```

SOLUTION of (10-7.c):

The LEHRFEM++ class **lf::fe::DiffusionElementMatrixProvider** computes element matrices for Lagrangia finite element spaces and the bilinear form

$$(u,w) \mapsto \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \text{on} \quad H^1(\Omega)\times H^1(\Omega)\,. \tag{10.7.20}$$

Its constructor requires the diffusion tensor $\boldsymbol{\alpha}$ encoded by a **MeshFunction**. Recalling that

$$a_{SU}(u,w) = \underbrace{\int_\Omega \delta(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})\mathbf{v}^\top(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{\text{diffusive part}} + \underbrace{\int_\Omega \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\,u(\boldsymbol{x})\,w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{\text{advective part}}\,, \tag{10.7.8}$$

we see that its diffusive part agrees with (10.7.20) for

$$\boldsymbol{\alpha}(\boldsymbol{x}) = \delta(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})\mathbf{v}(\boldsymbol{x})^\top \in \mathbb{R}^{2,2}\,.$$

This function has to be implemented using the definition (10.7.10) of $\delta(\boldsymbol{x})$.

C++ code 10.7.21: Evaluation operator for MeshFunctionDiffTensor

```
2  template <typename VELOCITY>
3  std::vector<Eigen::Matrix2d> MeshFunctionDiffTensor<VELOCITY>::operator()(
4      const lf::mesh::Entity &e, const Eigen::MatrixXd &local) const {
5    std::vector<Eigen::Matrix2d> ret;  // Element matrix
6    const lf::geometry::Geometry &geo{*e.Geometry()};
7    const double area = lf::geometry::Volume(geo);
8    const Eigen::MatrixXd &corners_refc(e.RefEl().NodeCoords());
9    const auto velo_cvals(velo_(e, corners_refc));  // v at cell corners
10   const auto velovals(velo_(e, local));           // v in given points
11
12   // Formula (10.7.10)
13 #if SOLUTION
14   double max_v = 0.0;
15   for (auto &velovec : velo_cvals) {
16     max_v = std::max(max_v, velovec.norm());
17   }
18   const double delta = std::min(1.0, std::sqrt(area) / max_v);
19   for (auto &velovec : velovals) {
20     // δ v(ζ_ℓ) v(ζ_ℓ)^⊤
21     ret.push_back(delta * velovec * velovec.transpose());
22   }
23 #else
24   /*************************************************
25   Adjust the following loop to fill in 'ret' correctly
26   *************************************************/
27   for (auto &velovec : velovals) {
28     ret.push_back(Eigen::Matrix2d());
29   }
30 #endif
31   return ret;
32 }
```

HINT 1 for (10-7.d): The file `sufem.h` also provides a function

```
template <typename VELOCITY>
lf::mesh::utils::CodimMeshDataSet<bool> flagNodesOnInflowBoundary(
    const std::shared_ptr<const lf::mesh::Mesh> &mesh_p, VELOCITY
      velo);
```

that returns a flag array for the nodes of the mesh, in which nodes on Γ_{in} are flagged with **true**. ⌟

SOLUTION of (10-7.d):

We follow the approach of [Lecture → § 2.7.6.13] and use the LEHRFEM++ built-in function `lf::assembl` as in [Lecture → Code 2.7.6.20].

To that end we have create a "selector" object that returns **true** for nodes $\boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Gamma_{\text{in}}$ and also the value $g(\boldsymbol{p})$.

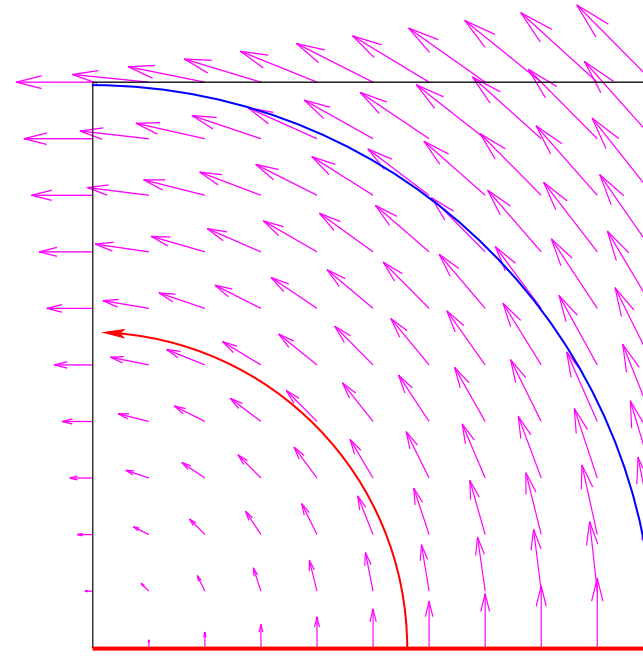
C++ code 10.7.22: Full code for `solveAdvectionDirichlet()`

```
2  template <typename VELOCITY, typename DIRICHLET_DATA>
3  Eigen::VectorXd solveAdvectionDirichlet(
4      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
5      VELOCITY velo, DIRICHLET_DATA mf_g) {
6    static_assert(lf::mesh::utils::MeshFunction<VELOCITY>);
7    static_assert(lf::mesh::utils::MeshFunction<DIRICHLET_DATA>);
8    // Obtain full Galerkin matrix in COO format
9    lf::assemble::COOMatrix<double> A_COO =
10       SUFEM::buildSUGalerkinMatrix(fe_space, velo);
11   // Zero right-hand side vector
12   const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
13   const lf::base::size_type N_dofs = dofh.NumDofs();
14   Eigen::VectorXd phi = Eigen::VectorXd::Zero(N_dofs);
15   // ** Set boundary conditions **
16   // Interpolate Dirichlet data
17   Eigen::VectorXd g_coeffs = lf::fe::NodalProjection(*fe_space, mf_g);
18   // Determine nodes on the inflow boundary part
19   auto inflow_nodes{SUFEM::flagNodesOnInflowBoundary(fe_space->Mesh(), velo)};
20 #if SOLUTION
21   lf::assemble::FixFlaggedSolutionCompAlt<double>(
22       [&inflow_nodes, &g_coeffs,
23        &dofh](lf::assemble::glb_idx_t dof_idx) -> std::pair<bool, double> {
24         const lf::mesh::Entity &dof_node{dofh.Entity(dof_idx)};
25         LF_ASSERT_MSG(dof_node.RefEl() == lf::base::RefEl::kPoint(),
26                       "All dofs must be associated with points");
27         return {inflow_nodes(dof_node), g_coeffs[dof_idx]};
28       },
29       A_COO, phi);
30 #else
31   /*************************************************************
32   Your implementation to adjust A_COO and phi to enforce Dirichlet BCs goes here
33   *************************************************************/
34 #endif
35   // Solve linear system using sparse elimination
36   Eigen::SparseMatrix<double> A(A_COO.makeSparse());
37   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
38   solver.compute(A);
39   if (solver.info() != Eigen::Success) {
40     throw std::runtime_error("Could not factorize A");
41   }
42   return solver.solve(phi);
43 }
```

SOLUTION of (10-7.e):

SOLUTION of (10-8.a):

Boundary layers for $\epsilon \ll 1$ will emerge where the Dirichlet data, when transported with the flow induced by $\mathbf{v}$ do not match the Dirichlet data imposed on $\partial\Omega$.



The non-zero boundary data from the bottom side $[0,1] \times \{0\}$ are carried to the left side $\{0\} \times [0,1]$, where zero Dirichlet boundary conditions are imposed. There we have a mismatch of the transported and imposed boundary values.

The zero boundary condition from the right side $\{1\} \times [0,1]$ is carried to the top side $[0,1] \times \{1\}$, where it meets another boundary zero condition: mismatch, no boundasry layer.

Boundary layers will arise next to the side $\{0\} \times [0,1]$

SOLUTION of (10-8.b):

Looking at Fig. 164 we see that (closure of) the inflow boundary is the union of the bottom and the left side of Ω,

$$\Gamma_{\rm in} = ([0,1] \times \{0\}) \cup (\{1\} \times [0,1]) \, .$$

$$\left\{ i \in \{1,\ldots,13\} : \ \boldsymbol{p}^i \in \overline{\Gamma}_{\rm in} \right\} = \{1,2,3,6,9,13\} \, .$$

SOLUTION of (10-8.c):

As can be seen from Fig. 167, the negative velocity vector $-\mathbf{v}(\boldsymbol{p}^7)$ points into cell $\sharp 10$.

$$\Big\{ i : \ K_i \in \mathcal{K}_{\rm uw}(\boldsymbol{p}^7) \Big\} = \{10\} \ .$$

SOLUTION of (10-8.d):

The entries of the Galerkin matrix are computed by plugging the nodal basis functions b_h^i, $i = 1, \ldots, 13$, into the bilinear form $\mathsf{a}_{h,\mathrm{uw}}$:

$$\mathbf{B} = \left[\mathsf{a}_{h,\mathrm{uw}}(b_h^j, b_h^i)\right]_{i,j=1}^{13} \in \mathbb{R}^{13,13} .$$

The 7-th row of the Galerkin matrix arises from testing with the tent function associated with node $\sharp 7$:

$$(\mathbf{B})_{7,:} = \left[\left(\tfrac{1}{3}\sum_{K\in\mathcal{U}_{p^7}} |K|\right)\cdot \mathbf{v}(\boldsymbol{p}^7)\cdot \left.\mathbf{grad}\, b_h^j\right|_{K_{10}}(\boldsymbol{p}^7)\right]_{j=1}^{13} .$$

Here we have used

1. the cardinal basis property of the tent functions with respect to the nodes of the mesh: $b_h^i(\boldsymbol{p}^k) = \delta_{i,k}$, and
2. the result of Sub-problem (10-8.c): $\mathcal{K}_{\mathrm{uw}}(\boldsymbol{p}^7) = \{K_{10}\}$.

So we have to determine those tent functions, whose gradient do not vanish on K_{10}. These are the tent functions, whose supports cover K_{10},

$$\left\{i \in \{1,\ldots,13\} : \ (\mathbf{B})_{7,i} \neq 0\right\} = \{7,8,10\} .$$

Remark. Entries of $\mathbf{B}$ corresponding to nodes on the inflow boundary are not really well-defined, because for those nodes $\mathcal{K}_{\mathrm{uw}} = \emptyset$.

SOLUTION of (10-8.e):

As we have seen in Sub-problem (10-8.b) the following nodes are not located on $\Gamma_{\rm in}$:

$$\left\{p^5, p^4, p^8, p^{10}, p^7, p^{11}, p^{12}\right\} .$$

We apply the considerations of Sub-problem (10-8.d) to all nodes $\in \mathcal{V}_+(\mathcal{M})$ based on

$$\mathcal{K}_{\rm uw}(p^4) = \{K_6\} , \quad \mathcal{K}_{\rm uw}(p^5) = \{K_3\} , \quad \mathcal{K}_{\rm uw}(p^8) = \{K_5\} , \quad \mathcal{K}_{\rm uw}(p^7) = \{K_{10}\} ,$$
$$\mathcal{K}_{\rm uw}(p^{10}) = \{K_{11}\} , \quad \mathcal{K}_{\rm uw}(p^{11}) = \{K_{12}\} , \quad \mathcal{K}_{\rm uw}(p^{12}) = \{K_{14}\} .$$

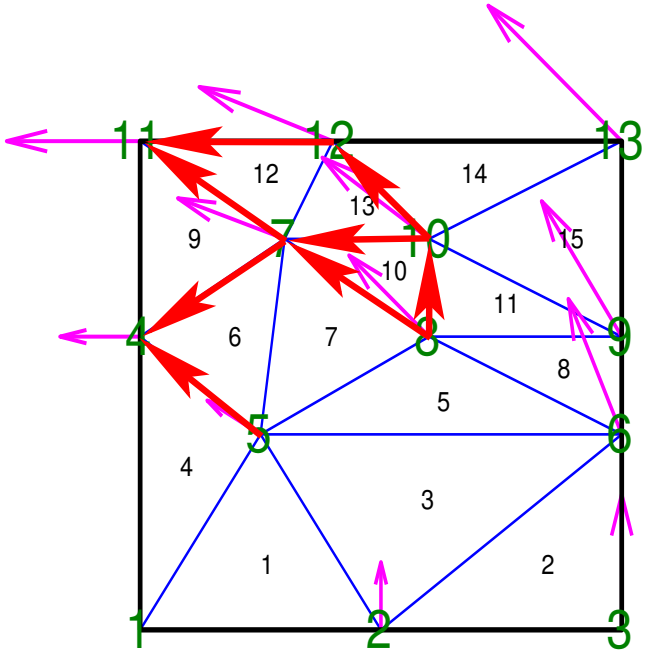
The reasoning elaborated in Sub-problem (10-8.d) yields the general formula

$$\left\{j : \ (\mathbf{B})_{i,j} \neq 0\right\} \subset \left\{j : \ p^j \in \overline{K}, \ K \in \mathcal{K}_{\rm uw}(p^i)\right\} . \tag{10.8.7}$$

Thus, using the numbering given in Fig. 167, the possible non-zero entries of $\mathbf{B}_+$ are at the following positions

$$\mathbf{B}_+ = \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 \\ 0 & * & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 \\ 0 & * & 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & 0 & 0 \\ 0 & 0 & * & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & * & 0 & * \end{bmatrix} , \qquad \begin{pmatrix} 4 \\ 5 \\ 7 \\ 8 \\ 10 \\ 11 \\ 12 \end{pmatrix} ,$$

where the "vertical sequence" shows the node numbers corresponding to the rows/columns of $\mathbf{B}_+$.



A good way to motivate the ordering is to draw a **dependence graph** of the nodes in $\mathcal{V}_+(\mathcal{M})$, where a node i is regarded as dependent of node j, if $(\mathbf{B})_{i,j} \neq 0$.

In the figure beside a red arrow $\rightarrow$ pointing from node j to node i means that node i depends on node j.

Then the ordering rule is:

> If a node i depends on node j, $i \neq j$, it must occur before j in the ordering.

One of many eligible orderings is:

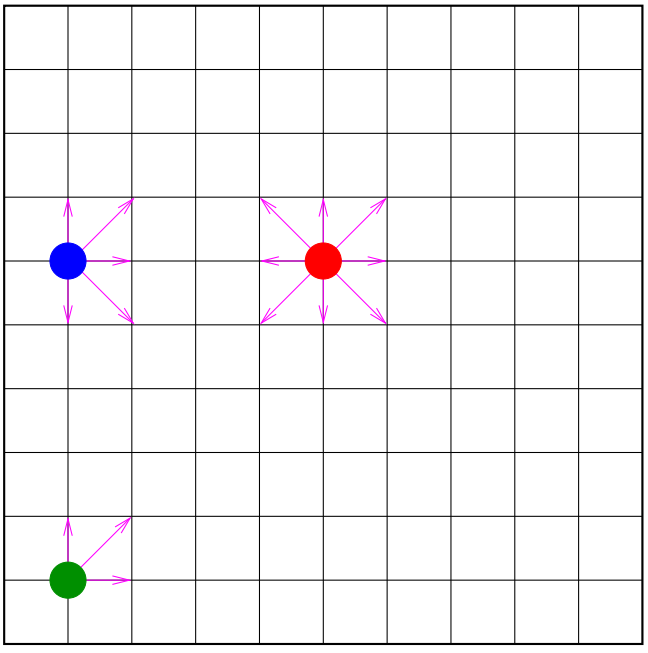
$$(11, 12, 4, 7, 10, 8, 5) .$$

Remark. In general it is possible to sort the basis functions in a way that renders the Galerkin matrix resulting from upwind quadrature discretization of the pure transport problem a triangular matrix. It goes without saying that this greatly facilitates the solution of relates LSEs by elimination techniques.

SOLUTION of (10-9.a):

Since u is known on $\partial\Omega$, the expansion coefficients have to be determined for global shape functions associated with all *interior* nodes, of which there are M^2. Hence $\mathbf{A}$ must be a $M^2 \times M^2$ -matrix, and $\vec{\boldsymbol{\varphi}}$ a vector with M^2 components.

SOLUTION of (10-9.b):



Since the element matrices (10.9.3) are fully populated, every node is “connected” (in the sense of a non-zero entry of the matrix $\mathbf{A}$) to all other nodes with which it shares a cell.

Hence, an interior node is connected to 8 other nodes (and itself), whereas a node next to a corner is connected to only three other nodes (and itself). ▷

Thus the number of non-zero entries of rows of $\mathbf{A}$ is between 4 and 9 .

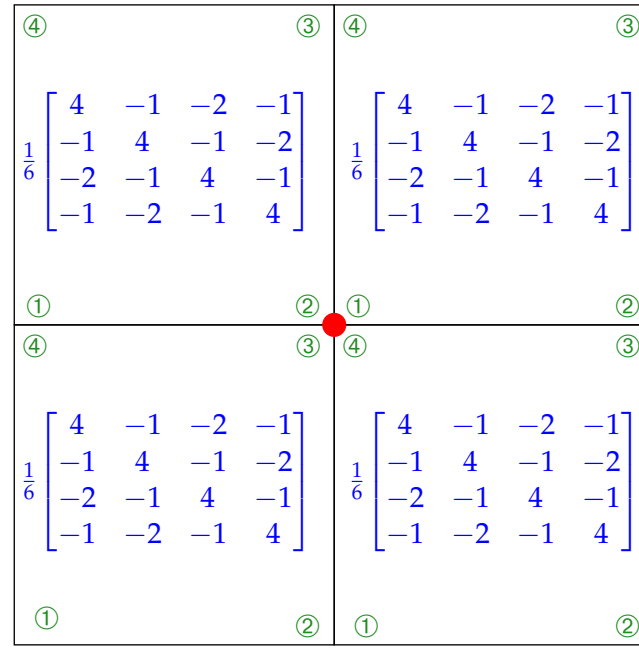
SOLUTION of (10-9.c):

A stencil represents a single row of the matrix $\mathbf{A}$ and visualizes the connectivity of a single degree of freedom (global shape function) with those associated with neighboring nodes.

To determine the entries of the •-row of $\mathbf{A}$ we follow the "assembly on paper" approach of [Lecture → § 4.1.3.3].

- The diagonal entry corresponding to the central entry of the stencil is obtained by adding four diagonal entries of the element matrices, which yields 16
- The vertical and horizontal connection weights arise from adding two entries of the element matrix belonging to a pair of adjacent vertices, which gives -2.
- A diagonal connection has the weight -2 occurring as entry of the element matrix for a pair of opposite vertices.

All these values have to be scaled by the factor $\frac{1}{6}$.



The final stencil is

$$\frac{1}{6} \cdot \begin{bmatrix} -2 & -2 & -2 \\ -2 & 16 & -2 \\ -2 & -2 & -2 \end{bmatrix}_h \quad . \tag{10.9.4}$$

It is symmetric and its entries add up to zero, which reflects the symmetry of Δ and the fact that constants are in the kernel of Δ.

HINT 1 for (10-9.d):

- The member function `insert()` of **Eigen::SparseMatrix** simply sets an entry of the sparse matrix.
- The method `reserve()` of a sparse-matrix object expects an integer vector argument whose components tell the maximal number of non-zero entries per column.

⌟

SOLUTION of (10-9.d):

The following code performs a column-wise initialization. A similar row-wise initialization is also possible and can be done by simply swapping the index arguments when inserting matrix entries.

C++ code 10.9.6: Initialization of sparse Galerkin matrix $\mathbf{A}$

```
Eigen::SparseMatrix<double> initializeA(unsigned int M) {
  // For the sake of efficiency the use of Eigen's sparse matrix data type is
  // essential.  The matrix is stored in CCS format.
  Eigen::SparseMatrix<double> A(M * M, M * M);
  // We already know that the matrix has at most 9 non-zero entries per row
  // and column.  This information is passed to Eigen via the reserve() member
  // function.
  A.reserve(Eigen::VectorXi::Constant(M * M, 9));
  // Iterate over all interior nodes of the mesh and apply the stencil and
  // initialize the matrix in column-wise order, from top to bottom in every
  // column, which is most efficient for the CCS storage format.
  for (int i = 0; i < M; ++i) {    // "vertical" loop
    for (int j = 0; j < M; ++j) {  // "horizontal" loop
      // Index of the current node
      const int k = i * M + j;
      // Self-interaction weight
      A.insert(k, k) = 16.0 / 6;
      // Interaction term with the node below to the left
      if (i > 0 && j > 0) {
        A.insert(k - M - 1, k) = -2.0 / 6;
      }
      // Interaction term with the node below
      if (i > 0) {
        A.insert(k - M, k) = -2.0 / 6;
      }
      // Interaction term with the node below to the right
      if (i > 0 && j < M - 1) {
        A.insert(k - M + 1, k) = -2.0 / 6;
      }
      // Interaction term with the node to the left
      if (j > 0) {
        A.insert(k - 1, k) = -2.0 / 6;
      }
      // Interaction term with the node to the right
      if (j < M - 1) {
        A.insert(k + 1, k) = -2.0 / 6;
      }
      // Interaction term with the node above to the left
      if (i < M - 1 && j > 0) {
        A.insert(k + M - 1, k) = -2.0 / 6;
      }
      // Interaction term with the node above
      if (i < M - 1) {
        A.insert(k + M, k) = -2.0 / 6;
      }
      // Interaction term with the node above to th right
```

```
48          if (i < M - 1 && j < M - 1) {
49            A.insert(k + M + 1, k) = -2.0 / 6;
50          }
51        }
52      }
53      A.makeCompressed();
54      return A;
55  }
```

SOLUTION of (10-9.e):

From [Lecture → Section 2.7.6] remember the treatment of essential (Dirichlet) boundary conditions by means of the offset function technique. Let $g_h \in \mathcal{S}_1^0(\mathcal{M})$ be a function in the extended (unconstrained) finite element space $\mathcal{S}_1^0(\mathcal{M})$ exclusively spanned by global shape functions belonging to nodes on $\partial\Omega$:

$$g_h = \sum_{k=1}^{L} g(\boldsymbol{p}_k) b_\partial^k \,, L = 4M + 4 \,,$$

with $\{b_\partial^1, \ldots, b_\partial^L\}$ the global shape functions "on $\partial\Omega$", $\boldsymbol{p}_k$, $k = 1, \ldots, L$ the nodes on $\partial\Omega$.

Then the right-hand-side vector is

$$\vec{\boldsymbol{\varphi}} = \left[-\int_\Omega \mathbf{grad}\, g_h \cdot \mathbf{grad}\, b_h^i \,\mathrm{d}\boldsymbol{x} \right]_{i=1}^{N} \,, \quad N := M^2 = \dim \mathcal{S}_{1,0}^0(\mathcal{M}) \,, \tag{10.9.8}$$

where $\{b_h^1, \ldots, b_h^N\}$ is the nodal basis of $\mathcal{S}_{1,0}^0(\mathcal{M})$. Therefore

$$(\vec{\boldsymbol{\varphi}})_i = \sum_{k=1}^{L} g(\boldsymbol{p}_k) \cdot \underbrace{\int_\Omega \mathbf{grad}\, b_\partial^k \cdot \mathbf{grad}\, b_h^i \,\mathrm{d}\boldsymbol{x}}_{= \text{off-center weight of stencil (10.9.4)}} \,.$$

We infer, that the component i of the right-hand-side vector is a sum of point values of g in nodes on $\partial\Omega$ that share a mesh cell with node i multiplied with off-center stencil weights.

Hence only components of $\vec{\boldsymbol{\varphi}}$ whose associated mesh nodes abut the boundary will be non-zero. More precisely, these are the components (C++ indexing!)

- $(\vec{\boldsymbol{\varphi}})_0$, $(\vec{\boldsymbol{\varphi}})_{M-1}$, $(\vec{\boldsymbol{\varphi}})_{M(M-1)}$, and $(\vec{\boldsymbol{\varphi}})_{M^2-1}$ for the interior nodes next to corners: contributions from 5 nodes on the boundary
- $(\vec{\boldsymbol{\varphi}})_k$, $k = 1, \ldots, M-2$, for the bottom row of nodes,
- $(\vec{\boldsymbol{\varphi}})_{M(M-1)+k}$, $k = 1, \ldots, M-2$ for the top row of nodes,
- $(\vec{\boldsymbol{\varphi}})_{Mk}$, $k = 1, \ldots, M-2$, for the leftmost column of interior nodes
- $(\vec{\boldsymbol{\varphi}})_{M(k+1)-1}$, $k = 1, \ldots, M-2$, for the rightmost column of interior nodes.

C++ code 10.9.9: Initialization of right-hand-side vector $\vec{\boldsymbol{\varphi}}$

```
2  Eigen::VectorXd initializeRHSVector(
3      const std::function<double(double, double)> &g, unsigned int M) {
4    // Mesh width
5    const double h = 1.0 / (M + 1);
6    // Off-center entry of stencil
7    const double w = 2.0 / 6;
8    Eigen::VectorXd phi = Eigen::VectorXd::Zero(M * M);
9    // Four corner points
10   phi[0] = w * (g(0.0, 0.0) + g(0.0, h) + g(h, 0.0) + g(2 * h, 0.0) +
11                 g(0.0, 2 * h));  // Bottom left corner
12   phi[M - 1] =
13       w * (g(1.0, 0.0) + g(1.0 - h, 0.0) + g(1.0, h) + g(1.0 - 2 * h, 0.0) +
14            g(1.0, 2 * h));  // Bottom right corner
```

```
  phi[M * (M - 1)] =
      w * (g(0.0, 1.0) + g(0.0, 1.0 - h) + g(h, 1.0) + g(2 * h, 1.0) +
           g(0.0, 1.0 - 2 * h));  // Top left corner
  phi[M * M - 1] =
      w * (g(1.0, 1.0) + g(1.0, 1.0 - h) + g(1.0 - h, 1.0) +
           g(1.0, 1.0 - 2 * h) + g(1.0 - 2 * h, 1.0));  // Top right corner
  for (unsigned int l = 1; l < M - 1; ++l) {
    phi[l] = w * (g(h * l, 0.0) + g((l + 1) * h, 0.0) +
                  g((l + 2) * h, 0.0));  // bottom
    phi[M * l] = w * (g(0.0, l * h) + g(0.0, (l + 1) * h) +
                      g(0.0, (l + 2) * h));  // left
    phi[M * (l + 1) - 1] = w * (g(1.0, l * h) + g(1.0, (l + 1) * h) +
                                g(1.0, (l + 2) * h));  // right
    phi[M * (M - 1) + l] =
        w * (g(h * l, 1.0) + g((l + 1) * h, 1.0) + g((l + 2) * h, 1.0));  // top
  }
  return phi;
}
```

SOLUTION of (10-9.f):

Set $\psi(t) := u(\mathbf{y}(t))$. Applying the chain rule we find

$$\frac{d\psi}{dt}(t) = \mathbf{grad}\, u(\mathbf{y}(t)) \cdot \dot{\mathbf{y}}(t) \overset{(10.9.10)}{=} \|\mathbf{grad}\, u(\mathbf{y}(t))\|^2 \geq 0 \,.$$

Hence we conclude that u is non-decreasing along streamlines.

Remark. This means that streamlines that do not collapse to a point cannot be closed in Ω. This is why the flow is called vortex-free.

SOLUTION of (11-1.a):

In both cases the flux function depends on the state only. Matching with [Lecture → Eq. (11.2.1.6)]:

$$\text{for (11.1.1)} \qquad f(u) = \tfrac{1}{2}u^2 \ ,$$

$$\text{for (11.1.1)} \qquad f(u) = \tfrac{2}{3}u^{3/2} \ .$$

HINT 1 for (11-1.j): The solution is constant along the characteristics. In case characteristics intersect, there will be a shock that moves at variable speed, see [Lecture → Eq. (11.2.4.2)]. Get some ideas from [Lecture → Ex. 11.2.6.4].

SOLUTION of (11-1.j):

We have $u_0(x) \geq 0$ and $f'(u) = \sqrt{u}$, which is non-negative and increasing. Therefore all information propagates to the right.

Since the flux function $f(u) =$ is convex, we expect to see a shock eventually developing from characteristics to the right of $x = \frac{1}{2}$, where the initial condition (11.1.5) is decreasing. The shock will emerge at some finite time $t_0 > 0$ at a position $0 < x \leq 1$ and will move to the right with increasing speed.

Since w_0 is increasing in $[0, \frac{1}{2}]$, a rarefaction fan will develop in that zone. Its left edge will stay at $x = 0$ and its right edge will finally catch up with the shock.

[Lecture $\rightarrow$ Fig. 505] qualitatively describes the situation.

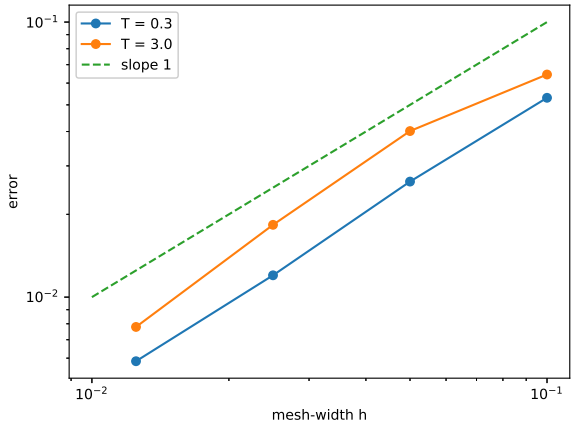
SOLUTION of (11-1.k):

C++11 code 11.1.9: Sub-problem (11-1.k), function `numexpBurgersGodunov()`, ⇝ GitLab

```
Eigen::VectorXd reduce(const Eigen::VectorXd &mu, unsigned int N) {
  Eigen::VectorXd mu_sub(N + 1);
  int fraction = mu.size() / N;
  for (int j = 0; j < N + 1; ++j) {
    mu_sub(j) = mu(j * fraction);
  }
  return mu_sub;
}

Eigen::Matrix<double, 3, 4> numexpBurgersGodunov() {
  const unsigned int N_large = 3200;
  Eigen::Vector2d T{0.3, 3.0};
  Eigen::Vector4i N{5 * 10, 5 * 20, 5 * 40, 5 * 80};
  Eigen::Vector4d h;
  for (int i = 0; i < 4; ++i) h(i) = 5.0 / N(i);

  Eigen::Matrix<double, 3, 4> result;
  result.row(0) = h.transpose();

  for (int k = 0; k < 2; ++k) {
    Eigen::VectorXd mu_ref = solveBurgersGodunov(T(k), N_large);
    Eigen::Vector4d error;
    for (int i = 0; i < 4; ++i) {
      Eigen::VectorXd mu = solveBurgersGodunov(T(k), N(i));
      Eigen::VectorXd mu_ref_sub = reduce(mu_ref, N(i));
      error(i) = h(i) * (mu - mu_ref_sub).lpNorm<1>();
    }
    result.row(k + 1) = error.transpose();
  }

  return result;
}
```

C++11 code 11.1.10: Sub-problem (11-1.k), in function `main()`, ⇝ GitLab

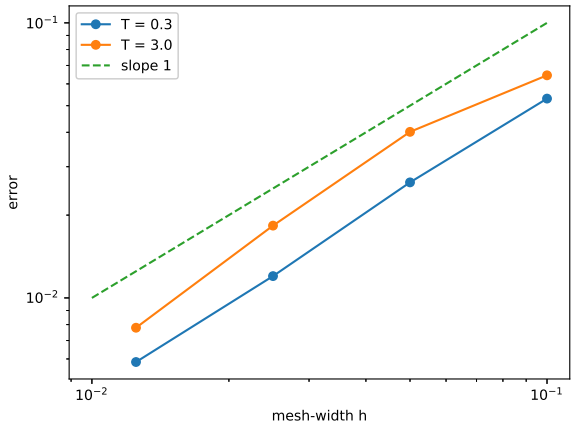
```
  Eigen::Matrix<double, 3, 4> result = BurgersEquation::numexpBurgersGodunov();

  // Write the result to a file that can be used for plotting.
  std::ofstream error_file;
  error_file.open("error.csv");
  error_file << result.format(BurgersEquation::CSVFormat) << std::endl;
  error_file.close();
  std::cout << "Generated error.csv" << std::endl;
  systemcall::execute("python3 ms_scripts/plot_error.py error.csv error.eps");
```



h	0.1	0.05	0.025	0.0125
$e(0.3,h)$	0.053	0.026	0.012	0.006
$e(3,h)$	0.064	0.040	0.018	0.007

◁ Plot of space-time L^1-error versus the spatial meshwidth = timestep.

SOLUTION of (11-1.l):



We follow [Lecture → § 3.2.2.5]. The error curves are almost straight lines in a doubly logarithmic plot, which indicates algebraic convergence,
From Fig. 181 we read off slope 1, which means convergence rate 1 for both final times.
Alternatively, from the table created in Subproblem (11-1.f) we see that halving meshwidth and timestep leads to a reduction of the error by roughly a factor of 2.

HINT 1 for (11-1.b): By monotonicity [Lecture $\rightarrow$ Thm. 11.2.7.1], $u(x,t) > 0$ for all $(x,t) \in \mathbb{R}\times]0,T[$. Then the proof boils down to clever use of the chain rule. ⌟

SOLUTION of (11-1.b):

Assume u solves (11.1.1), is positive, and is C^1-*smooth*. Then, by the chain rule,

$$\frac{\partial w}{\partial t} + \frac{\partial}{\partial x}\left(\frac{2}{3}w^{3/2}\right) = 2u\frac{\partial u}{\partial t} + 2u^2\frac{\partial u}{\partial x} = 2u\left(\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right)\right) = 0\,.$$

HINT 1 for (11-1.c): The flux function is the f in [Lecture $\rightarrow$ Eq. (11.2.2.1)]. It is convex, if $f''(u) > 0$ everywhere. ⌟

SOLUTION of (11-1.c):

The flux function is $f(w) = \frac{2}{3}w^{3/2}$, with $f''(w) = \frac{1}{2\sqrt{w}} > 0$ for $w > 0$. By monotonicity, we will never have $w = 0$ if $w_0 > 0$ everywhere, and of course $w < 0$ is impossible by definition.

HINT 1 for (11-1.d): You will need the Rankine-Hugoniot jump condition [Lecture → Eq. (11.2.4.2)] and may use [Lecture → Lemma 11.2.5.4]. ⌟

SOLUTION of (11-1.d):

We rely on the discussion of [Lecture → Section 11.2.5] and, in particular, [Lecture → Lemma 11.2.5.4]:

(I) For the case of the Burgers equation (11.1.1), we have $f'(u) = u$ and there are characteristics emanating from the negative x-axis with a velocity of $f'(2) = 2$, and characteristics emanating from the positive x-axis with a velocity of $f'(1) = 1$. Hence we will have a shock emanating from the origin with a velocity (the Rankine-Hugoniot velocity [Lecture → Eq. (11.2.5.3)]) of

$$\dot{s} = \frac{f(2) - f(1)}{2 - 1} = \frac{2^2}{2} - \frac{1^2}{2} = \frac{3}{2},$$

so the solution is

$$u(x,t) = \begin{cases} 1, & x/t > 3/2 \\ 2 & x/t < 3/2. \end{cases}$$

(I) The second case of (11.1.2) is similar. We have $f'(w) = \sqrt{w}$ and the characteristics emanating from the negative and positive x-axes will intersect. This time the shock moves with a velocity of

$$\dot{2} = \frac{f(4) - f(1)}{4 - 1} = \frac{2 \cdot 4^{3/2}/3 - 2 \cdot 1^{3/2}/3}{4 - 1} = \frac{14}{9},$$

so

$$w(x,t) = \begin{cases} 1, & x/t > 14/9 \\ 4 & x/t < 14/9. \end{cases}$$

HINT 1 for (11-1.e): Recall [Lecture → Lemma 11.2.5.10] about rarefaction solutions. ⌟

SOLUTION of (11-1.e):

Here, the characteristics diverge, so we get a rarefaction solution in the wedge between them. For (11.1.1), $f'(u) = u$, so using [Lecture $\to$ Lemma 11.2.5.10], we get

$$u(x,t) = \begin{cases} 2, & x/t > 2, \\ x/t & 1 < x/t < 2, \\ 1 & x/t < 1, \end{cases}$$

and for (11.1.2), $f'(w) = \sqrt{w}$, giving $g(x/t) = (x/t)^2$, so

$$w(x,t) = \begin{cases} 4, & x/t > 2 \\ (x/t)^2 & 1 < x/t < 2 \\ 1 & x/t < 1. \end{cases}$$

SOLUTION of (11-1.f):

The relationship $w = u^2$ is obviously violated for the shock solution and holds only for the rarefaction solution. However, this is not a logical dilemma, because a shock solution obviously fails to satisfy the C^1-condition of Sub-problem (11-1.b). Thus the transformation by the change rule cannot be applied when adopting the concept of weak solutions.

The problem sends a warning that manipulations of conservation laws using the formulas of classical calculus can be dangerous.

SOLUTION of (11-1.g):

The Godunov numerical flux is a two-point flux function such that the associated semi-discrete evolution for (11.1.2) in conservation form reads, see [Lecture → Eq. (11.3.2.8)],

$$\dot{\mu}_j(t) = -\frac{1}{h}\big[F(\mu_j, \mu_{j+1}) - F(\mu_{j-1}, \mu_j)\big] \ , \tag{11.1.3}$$

where F in this case is defined as $F(v,w) = f(u^{\downarrow}(v,w))$, where $u^{\downarrow}$ is defined as in [Lecture → Eq. (11.3.4.31)], because the flux function $f(u) = \frac{2}{3}u^{3/2}$ is convex.

We use the numbering of the different cases from [Lecture → Eq. (11.3.4.31)], [Lecture → Fig. 537]–[Lecture → Fig. 541]. Note that we will never encounter $f'(v) < 0$ or $f'(w) < 0$, so two of five cases (❷ and ❺) need not be taken into account. Moreover, the Rankine-Hugoniot speed $\dot{s}$ will always be positive, because the flux function is increasing. This eliminates case ❶. Finally we have

$$u^{\downarrow}(v,w) = v \quad \text{for all} \quad v, w > 0 \ .$$

This means that (11.1.3) becomes

$$\dot{\mu}_j(t) = -\frac{1}{h}\big[f(\mu_j) - f(\mu_{j-1})\big] = -\frac{2}{3h}\Big[\mu_j^{3/2} - \mu_{j-1}^{3/2}\Big] \ . \tag{11.1.4}$$

SOLUTION of (11-1.h):

The general formula for the fully discrete evolution operator for a 2-point numerical flux function in combination wit explicit Euler timestepping is given in [Lecture → Eq. (11.4.1.10)]. The particular 2-point numerical flux function has been derived in Sub-problem (11-1.g) as $F(v,w) = v$.

Alternative you may start from (11.1.4) and just apply the explicit Euler method.

C++11 code 11.1.6: Implementation of `solveBurgersGodunov()` for Sub-problem (11-1.h)
➥ GitLab

```
2  constexpr double PI = 3.14159265358979323846;
3
4  double Square(double x) { return x * x; }
5
6  double f(double x) { return 2.0 / 3.0 * std::sqrt(x * x * x); }
7
8  Eigen::VectorXd solveBurgersGodunov(double T, unsigned int N) {
9    double h = 5.0 / N;            // meshwidth
10   double tau = h;                // timestep = meshwidth by CFL condition
11   int m = std::round(T / tau);   // no. of timesteps
12
13   // initialize vector with initial nodal values
14   Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N + 1, -1.0, 4.0);
15   Eigen::VectorXd mu =
16       x.unaryExpr([](double x) {
17          return 0.0 <= x && x <= 1.0 ? Square(std::sin(PI * x)) : 0.0;
18        }).eval();
19
20   for (int i = 0; i < m; ++i) {
21     for (int j = N; 0 < j; --j) {
22       // Standard fully discrete evolution based on explicit Euler timestepping
23       mu(j) = mu(j) - tau / h * (f(mu(j)) - f(mu(j - 1)));
24     }
25     // truncation to a finite vector. Only required on one side, because all
26     // information flows from left to right.
27     mu(0) = 0.0;  // Value of u0 to the left of x=0
28   }
29
30   return mu;
31 }
```

SOLUTION of (11-1.i):

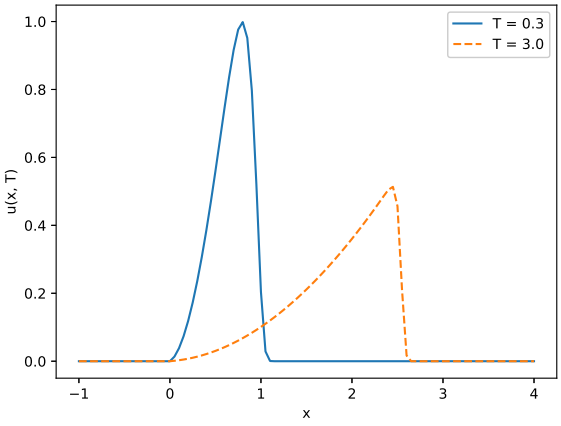
There are many ways to create a plot. One is to generate formatted output from the C++ code and then leave the actual creation of the plot to a PYTHON script ⇢ GitLab.

C++11 code 11.1.7: Sub-problem (11-1.i), excerpt of `main()` ⇢ GitLab

```
2   const unsigned int N = 100;
3   Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N + 1, -1.0, 4.0);
4   Eigen::VectorXd mu03 = BurgersEquation::solveBurgersGodunov(0.3, N);
5   Eigen::VectorXd mu30 = BurgersEquation::solveBurgersGodunov(3.0, N);
6
7   // Write the solutions to a file that can be used for plotting.
8   std::ofstream solution_file;
9   solution_file.open("solution.csv");
10  solution_file << x.transpose().format(BurgersEquation::CSVFormat)
11                << std::endl;
12  solution_file << mu03.transpose().format(BurgersEquation::CSVFormat)
13                << std::endl;
14  solution_file << mu30.transpose().format(BurgersEquation::CSVFormat)
15                << std::endl;
16  solution_file.close();
17  std::cout << "Generated solution.csv" << std::endl;
18  systemcall::execute(
19      "python3 ms_scripts/plot_solution.py solution.csv "
20      "solution.eps");
```

PYTHON script 11.1.8: Sub-problem (11-1.i): Helper script for visualization ⇢ GitLab

```
1  from matplotlib.pyplot import figure, legend, plot, savefig, xlabel, ylabel
2  from numpy import genfromtxt
3  from sys import argv
4
5  input_file = str(argv[1])
6  output_file = str(argv[2])
7
8  data = genfromtxt(input_file, delimiter=',')
9  x = data[0]
10 solution_short = data[1]
11 solution_long = data[2]
12
13 fig = figure()
14 plot(x, solution_short, '-', label='T = 0.3')
15 plot(x, solution_long, '--', label='T = 3.0')
16 xlabel('x')
17 ylabel('u(x, T)')
18 legend()
19 savefig(output_file)
20
21 print('Generated ' + output_file)
```



◁ Finite-volume solution of (11.1.2) at different times, computed on a mesh with $N = 100$ cells.

SOLUTION of (11-10.a):

We start from the integral form [Lecture → Eq. (11.2.3.3)] for the 1D scalar conservation law (11.10.1):

$$\int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t_{k+1})\,\mathrm{d}x - \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t_k)\,\mathrm{d}x = \int_{t_k}^{t_{k+1}} f(u(x_{j-\frac{1}{2}},t))\,\mathrm{d}t - \int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t\,. \tag{11.10.4}$$

Taking into account the definition of $\rho_j^{(k)}$ this means

$$h\big(\rho_j^{(k+1)} - \rho_j^{(k)}\big) = \int_{t_k}^{t_{k+1}} f(u(x_{j-\frac{1}{2}},t))\,\mathrm{d}t - \int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t\,,$$

and we get

$$\rho_j^{(k+1)} = \rho_j^{(k)} - \frac{\tau}{h}\left(\frac{1}{\tau}\int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t - \frac{1}{\tau}\int_{t_k}^{t_{k+1}} f(u(x_{j-\frac{1}{2}},t))\,\mathrm{d}t\right),$$

which means that (11.10.3) holds, if we set

$$\widetilde{f}_{\ell+\frac{1}{2}} := \frac{1}{\tau}\int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t\,, \quad \ell = j-1, j\,.$$

Hence, only the last option is correct.

SOLUTION of (11-10.b):

Using a 2-point numerical flux the spatially semi-discrete conservation form for (11.10.1) reads [Lecture → Eq. (11.3.2.8)]

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big) \;, \quad j \in \mathbb{Z} \;.$$

As discussed in [Lecture → Ex. 11.4.1.9], we apply explicit Euler timestepping to this "infinite-dimensional" ODE and get [Lecture → Eq. (11.4.1.10)]

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\big(F(\mu_j^{(k)},\mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)},\mu_j^{(k)})\big) \;,$$

which agrees with (11.10.5) when setting

$$f_{\ell+\frac{1}{2}} := F(\mu_\ell^{(k)},\mu_{\ell+1}^{(k)}) \;, \quad \ell \in \mathbb{Z} \;.$$

SOLUTION of (11-10.c):

For the scalar linear advection equation with advection velocity $\beta > 0$ the Godunov numerical flux boils down to the simple upwind numerical flux:

$$F_{\mathrm{GD}}(v,w) = \beta v\,, \quad v,w \in \mathbb{R}\,, \tag{11.10.8}$$

The Rankine-Hugoniot speed $\dot{s}$ agrees with β: $\dot{s} = \beta > 0$, which means that in (11.10.6d)

$$\theta = \frac{\mu_\ell^{(k)} - \mu_{\ell-1}^{(k)}}{\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}}\,.$$

Summing up, in the k-th timestep we use the flux terms

$$f_{\ell+\frac{1}{2}} = \beta\mu_\ell^{(k)} + \tfrac{1}{2}\beta(1-\gamma\beta)\varphi\Big(\frac{\mu_\ell^{(k)} - \mu_{\ell-1}^{(k)}}{\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}}\Big)\big(\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}\big)\,, \quad l \in \mathbb{Z}\,.$$

Plugging this into (11.10.5) yields

$$\begin{aligned}\mu_j^{(k+1)} = \mu_j^{(k)} &- \beta\gamma(\mu_j^{(k)} - \mu_{j-1}^{(k)}) - \\ &\tfrac{1}{2}\beta\gamma(1-\beta\gamma)\bigg(\varphi\Big(\frac{\mu_j^{(k)} - \mu_{j-1}^{(k)}}{\mu_{j+1}^{(k)} - \mu_j^{(k)}}\Big)\big(\mu_{j+1}^{(k)} - \mu_j^{(k)}\big) - \\ &\qquad \varphi\Big(\frac{\mu_{j-1}^{(k)} - \mu_{j-2}^{(k)}}{\mu_j^{(k)} - \mu_{j-1}^{(k)}}\Big)\big(\mu_j^{(k)} - \mu_{j-1}^{(k)}\big)\bigg)\,.\end{aligned} \tag{11.10.9}$$

We tacitly deal with "division by zero exceptions" is hinted above. Since $\varphi(\cdot)$ is always finite, for $\mu_{j+1}^{(k)} - \mu_j^{(k)}$ or $\mu_j^{(k)} - \mu_{j-1}^{(k)} = 0$ the ill-defined terms can just be skipped.

Remark. For $\varphi \equiv 1$ the recursion simplifies to

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \beta\gamma(\mu_j^{(k)} - \mu_{j-1}^{(k)}) - \tfrac{1}{2}\beta\gamma(1-\beta\gamma)\Big(\mu_{j+1}^{(k)} - 2\mu_j^{(k)} + \mu_{j-1}^{(k)}\Big)\,.$$

This is the well-known 2nd-order **Lax-Wendroff scheme** for linear advection.

HINT 1 for (11-10.d): To avoid division by zero when computing θ, you can use the provided function

```
inline double thetaquotient(double u, double v, double w) {
  const double denom = w-v;
  return (denom == 0.0)?1E17*(v-u):(v-u)/denom;
}
```

whose arguments stand for the three consecutive $\mu_*^{(k)}$-values entering the definition (11.10.6d) of θ. ⌟

SOLUTION of (11-10.d):

The implementation boils down to coding (11.10.9). We have to make special provision for the treatment of he terminal mesh cells:

- For $j = 0$ set $\mu_{-2}^{(k)} = \mu_{-1}^{(k)} = \mu_0^{(k)}$.
- For $j = 1$ set $\mu_{-1}^{(k)} = \mu_0^{(k)}$.
- For $j = N - 1$ set $\mu_N^{(k)} = \mu_{N-1}^{(k)}$.

C++ code 11.10.10: code for `fluxlimAdvection` ↪ GitLab

```
template <typename FLUXLIM = std::function<double(double)>>
Eigen::VectorXd fluxlimAdvection(
    double beta, const Eigen::VectorXd &mu0, double h, double tau,
    unsigned int nb_timesteps,
    FLUXLIM &&phi = [](double /*theta*/) { return 1.0; }) {
  if (beta < 0) {
    throw std::domain_error("fluxlimAdvection: negative beta!");
  }
  Eigen::VectorXd mu;  // return vector

  // The length of the vector passing the initial conditions also
     determines teh
  // number of spatial (dual) cells
  int N = mu0.size();
  // Ratio of spatial and temporal timestep
  double gamma = tau / h;
  // Set initial conditions
  mu = mu0;
  // Constant continuation index mapping tool.  This is only one of many
     ways how
  // to implement the "endpoint conditions"
  auto idx_map = [N](int idx) {
    if (idx < 0) {
      return 0;
    }
    if (idx > N - 1) {
      return N - 1;
    }
    return idx;
  };

  Eigen::VectorXd mu_next(N);
  for (int k = 0; k < nb_timesteps; k++) {
    // Computing forward step
    for (int j = 0; j < N; j++) {
      // Implementation of (11.10.9).
      const double backward_diff = mu(idx_map(j)) - mu(idx_map(j - 1));
      const double shifted_backward_diff =
          mu(idx_map(j - 1)) - mu(idx_map(j - 2));
      const double forward_diff = mu(idx_map(j + 1)) - mu(idx_map(j));
      // Values of flux limiter
      double phi_1, phi_2;
      if (forward_diff == 0.0) {
        phi_1 = 0.0;
```

```
      } else {
        phi_1 = phi(backward_diff / forward_diff);
      }
      if (backward_diff == 0.0) {
        phi_2 = 0.0;
      } else {
        phi_2 = phi(shifted_backward_diff / backward_diff);
      }
      // Main recursion
      mu_next(j) = mu(idx_map(j)) - beta * gamma * backward_diff -
                   0.5 * beta * gamma * (1 - beta * gamma) *
                       (phi_1 * forward_diff - phi_2 * backward_diff);
    }
    mu = mu_next;  // step forward (loop update)
  }
  return mu;
};
```

SOLUTION of (11-10.e):

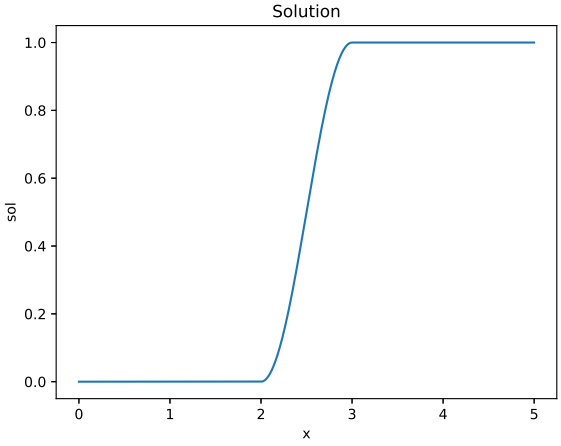
The quotients of successive error norms in the columns of the table evaluate to values that do not vary much between the rows. Thus, for both columns the data indicate **Algebraic convergence** of the error norms as $M \to \infty / \tau \to 0$.

Middle column: Algebraic convergence $O(\tau^{\alpha})$ with rate $\alpha < 1$
Right column: Algebraic convergence $O(\tau^2)$.

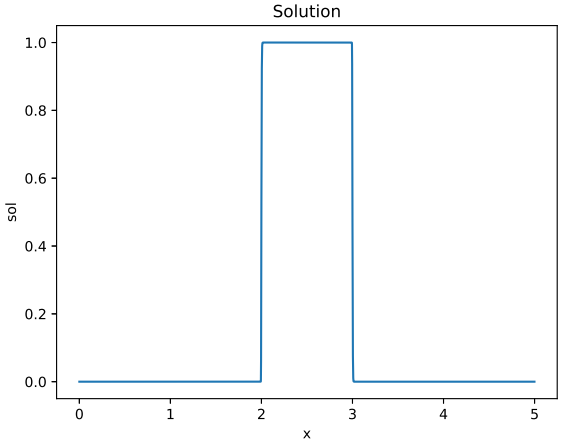
Convergence is much faster for the right column, which can only be possible for a smooth solution. Such a smooth solution can arise only from smooth u_0. Therefore we conclude

Middle column: u_0^B

Right column: u_0^A



FV-solution, $M = 1280$, $u_0 = u_0^A$



FV-solution, $M = 1280$, $u_0 = u_0^B$

SOLUTION of (11-10.f):

To evaluate $F_{\mathrm{GD}}(v,w)$ we have to distinguish several cases as in [Lecture → § 11.3.4.32] and examine the solutions of Riemann problems with left state v and right state w. The Burgers flux function is strictly convex.

❶ If $v > w$ the Riemann problem will have a shock solution and we get

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) = \frac{1}{2}v^2 & \text{, if } \frac{f(w)-f(v)}{w-v} = \frac{1}{2}(v+w) > 0\,, \\ f(w) = \frac{1}{2}w^2 & \text{, if } \frac{f(w)-f(v)}{w-v} = \frac{1}{2}(v+w) < 0\,. \end{cases} \tag{11.10.12}$$

❷ If $v < w$ the solution of the Riemann problem will be a rarefaction wave and we arrive at

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) = \frac{1}{2}v^2 & \text{, if } f'(v) = v > 0\,, \\ 0 & \text{, if } f'(v) = v < 0 < f'(w) = w\,, \\ f(w) = \frac{1}{2}w^2 & \text{, if } f'(w) = w < 0\,. \end{cases} \tag{11.10.13}$$

All these different cases are taken into account in the following lambda function where `f()` stands for the Burgers flux function $f(u) = \frac{1}{2}u^2$:

```
auto godnfnburgers = [f](double v, double w) -> double {
  if (v > w) {
    if (v + w > 0) return f(v); else  return f(w);
  } else {
    if (v > 0)  return f(v);
    else if (0 < w) return 0.0; else return f(w);
  }
};
```

SOLUTION of (11-10.g):

We implement (11.10.5) based on (11.10.6).

C++ code 11.10.14: code for `fluxlimBurgers` ↪ GitLab

```
template <typename FLUXLIM = std::function<double(double)>>
Eigen::VectorXd fluxlimBurgers(
    const Eigen::VectorXd &mu0, double h, double tau, unsigned int nb_timesteps,
    FLUXLIM &&phi = [](double /*theta*/) { return 1.0; }) {
  Eigen::VectorXd mu;  // return vector
  int N = mu0.size();  // Number of sptial dual cells
  double gamma = tau / h;
  // Set initial conditions
  mu = mu0;
  // Flux function for Burgers equation
  auto f = [](double u) { return 0.5 * u * u; };
  // Godunov numerical flux
  auto godnfnburgers = [f](double v, double w) -> double {
    if (v > w) {
      if (v + w > 0) {
        return f(v);
      }
      return f(w);
    }
    if (v > 0) {
      return f(v);
    }
    if (0 < w) {
      return 0.0;
    }
    return f(w);

  };

  // Constant continuation index mapping tool
  auto idx_map = [N](int idx) {
    if (idx < 0) {
      return 0;
    }
    if (idx > N - 1) {
      return N - 1;
    }
    return idx;
  };
  // Rankine-Hugoniot speed
  auto s_dot = [f](double v, double w) {
    if (v == w) {
      return 0.0;
    }
    return (f(w) - f(v)) / (w - v);
  };
  // The quantity θ from (11.10.6d)
  // Implementation avoid division by zero, see thetaquotient()
  auto theta = [s_dot](double x, double v, double w, double y) {
    if (s_dot(v, w) < 0) {
```

```
52        if (y == w) {
53          return 1.0e17 * (w - v);
54        }
55        return (w - v) / (y - w);
56      }
57      if (v == w) {
58        return 1.0e17 * (v - x);
59      }
60      return (v - x) / (w - v);
61      ;
62    };
63    // See (11.10.6b)
64    auto f_D = [&](int i) -> double {
65      double diff = mu(idx_map(i + 1)) - mu(idx_map(i));
66      double s_dot_abs = std::abs(s_dot(mu(idx_map(i)), mu(idx_map(i + 1))));
67      return 0.5 * s_dot_abs * (1 - gamma * s_dot_abs) * diff *
68             phi(theta(mu(idx_map(i - 1)), mu(idx_map(i)), mu(idx_map(i + 1)),
69                       mu(idx_map(i + 2))));
70    };
71    // Auxility arrays:  must be declared outside the main timestepping
       loop
72    Eigen::VectorXd mu_next(N);
73    Eigen::VectorXd fluxdiff(N);
74    for (int k = 0; k < nb_timesteps; k++) {
75      // Computing modified flux difference vector, see (11.10.6a)
76      for (int i = 0; i < N; i++) {
77        fluxdiff(i) = godnfnburgers(mu(idx_map(i)), mu(idx_map(i + 1))) + f_D(i) -
78                      godnfnburgers(mu(idx_map(i - 1)), mu(idx_map(i))) -
79                      f_D(i - 1);
80      }
81      // Computing forward Euler step (11.10.5)
82      for (int j = 0; j < N; j++) {
83        mu_next(j) = mu(j) - gamma * fluxdiff(j);
84      }
85      mu = mu_next;  // step forward (loop update)
86    }
87    return mu;
88  }
```

SOLUTION of (11-11.a):

Write $r : \mathbb{R} \to \mathbb{R}$ for the inverse function of ρ: $r(\rho(\xi)) = \rho(r(\xi)) = \xi$ for all $\xi \in \mathbb{R}$. Then $z = r(u)$ and we have

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{on} \quad \mathbb{R} \times]0, T[\tag{11.11.3}$$

$$\text{with} \qquad f(u) = g(r(u)) \ . \tag{11.11.4}$$

SOLUTION of (11-11.b):

- (A) is **true**. By [Lecture → Thm. 11.2.7.1] for the solution u of the transformed equation (11.11.3) we have with $u_0 := \rho \circ z_0$

$$u^- \leq u_0(x) \leq u^+ \quad \Rightarrow \quad u^- \leq u(x,t) \leq u^+ \quad \forall (x,t) \ .$$

  Since ρ is increasing, this carries over to z.

- (B) is **false**. Though by integration by parts we know that

$$\frac{d}{dt} \int_{-\infty}^{\infty} u(x,t) \, \mathrm{d}x = 0 \ ,$$

  this does not imply the same property for z, because

$$\frac{d}{dt} \int_{-\infty}^{\infty} z(x,t) \, \mathrm{d}x = \int_{\infty}^{\infty} r'(u) \frac{\partial u}{\partial t}(x,t) \, \mathrm{d}x \ ,$$

  and we cannot draw any conclusion.

- (C) is **false**. From [Lecture → Thm. 11.2.7.3] we know that

$$\int_{-\infty}^{\infty} |u(x,t)| \, \mathrm{d}x \leq \int_{-\infty}^{\infty} |u_0(x)| \, \mathrm{d}x \quad \forall t \in]0, T[$$

  for the solution $u := \rho(z)$ of (11.11.3). As before, this does not imply the same property for z.

- (D) is **true**. In [Lecture → Section 11.2.7] we learned that the number of local extrema of the solution u of (11.11.3) is decreasing. By the monotonicity of ρ this remains true for z.

- (E) is **false**. The weak solution u of (11.11.3) can develop discontinuities (shocks) in finite time even for smooth initial data, and this is also true for z.

SOLUTION of (11-11.c):

Obviously u (solution of (11.11.3)) and z will attain the constant values $\rho(z^*)/z^*$ outside the same interval. So it is sufficient to study (11.11.3) and find I for that Cauchy problem.

We want to appeal to [Lecture $\rightarrow$ Thm. 11.2.7.3], [Lecture $\rightarrow$ Cor. 11.2.7.5] and have to determine the maximal speed of propagation

$$\dot{s} := \max\left\{ |f'(\xi)| : \inf_{x\in\mathbb{R}} u_0(x) \le \xi \le \sup_{x\in\mathbb{R}} u_0(x) \right\} .$$

By the chain rule $f'(\xi) = g'(r(\xi))r'(\xi)$, $\xi \in \mathbb{R}$, and the derivative of the inverse function gives $r'(u) = 1/$
Therefore,

$$\dot{s} = \max\left\{ \left|\frac{g'(\xi)}{\rho'(\xi)}\right| : \inf_{x\in\mathbb{R}} z_0(x) \le \xi \le \sup_{x\in\mathbb{R}} z_0(x) \right\} .$$

Thus we have found the first possible solution

$$I(t) =]-1-\dot{s}\,t, 1+\dot{s}\,t[.$$

A cruder second possible solution can be derived from it: From (11.11.2) we learn that $\dot{s} \le \frac{\sigma}{\gamma}$, which gives us

$$I(t) =]-1-\tfrac{\sigma}{\gamma}\,t, 1+\tfrac{\sigma}{\gamma}\,t[,$$

$$\sigma := \max\left\{ |g'(\xi)| : \inf_{x\in\mathbb{R}} z_0(x) \le \xi \le \sup_{x\in\mathbb{R}} z_0(x) \right\} ,$$

because by the comparison principle $x \mapsto z(x,t)$ will remain within the extremal values of z_0 for all times.

It does not matter whether the interval $I(t)$ is given as an open or closed interval.

SOLUTION of (11-11.d):

We convert $\rho(z) = u$ into a problem of finding zeros:

$$\rho(z) = u \quad \Leftrightarrow \quad F(z) \stackrel{!}{=} 0 \quad \text{for} \quad F(z) := \rho(z) - u \,.$$

The **Newton iteration** for $F(z) = 0$ reads

$$z^{(k+1)} = z^{(k)} - \frac{F(z^{(k)})}{F'(z^{(k)})} = z^{(k)} - s\,, \quad s := \frac{\rho(z^{(k)}) - u}{\rho'(z^{(k)})}\,.$$

The iteration should be stop, when

$$|s| \leq \texttt{rtol}|z^{(k+1)}| \quad \text{or} \quad |s| \leq \texttt{atol}\,.$$

C++ code 11.11.6: Implementation of `rhoInverse()` ↬ GitLab

```
template <typename RHOFUNCTOR, typename DRHOFUNCTOR>
double rhoInverse(double u, double z0, RHOFUNCTOR &&rho, DRHOFUNCTOR &&drho,
                  double atol = 1.0E-10, double rtol = 1.0E-5) {
  // Reduce to problem of finding zeros:  phi(z)=0 <==> rho(z)=u
  auto F = [rho, u](double z) -> double { return rho(z) - u; };
  auto dF = [drho](double z) -> double { return drho(z); };

  // Newton iteration for finding a zero of F
  double s;
  do {
    s = F(z0) / dF(z0);
    z0 = z0 - s;
  } while (std::abs(s) > rtol * std::abs(z0) && std::abs(s) > atol);
  return z0;
}
```

SOLUTION of (11-11.e):

Let $[a,b]$ be the interval returned by the `domain()` member function of `prb` and N be the number of mesh cells, which agrees with the lengths of the vectors `mu` and `zeta`. Then we have $h := \frac{b-a}{N}$ for the mesh width.

According to [Lecture → Eq. (11.3.2.8)] in the context of a spatial conservative finite-volume discretization the approximate cell averages $\mu_j = \mu_j(t)$ of $u = u(x,t)$ satisfy

$$\dot{\mu}_j(t) = -\frac{1}{h}\big(F_R(\mu_j(t),\mu_{j+1}(t)) - F_R(\mu_{j-1}(t),\mu_j(t))\big) \;, \quad j = 1,\ldots,N \;.$$

Constant continuation of states means that we set $\mu_0 := \mu_1$ and $\mu_{N+1} := \mu_N$.

We know that

$$\mu_j(t) = \rho(\zeta_j(t)) \quad , \quad f(\mu) = g(\zeta) \quad , \quad f'(\mu) = \frac{g'(\zeta)}{\rho'(\zeta)} \;, \quad \zeta = \rho^{-1}(\mu) \;.$$

This means that we can evaluate the simplified Rusanov numerical flux (11.11.5) as follows:

$$F_R(\mu_j,\mu_{j+1}) = \tfrac{1}{2}\big(g(\zeta_j) + g(\zeta_{j+1})\big) - \tfrac{1}{2}(\mu_{j+1}-\mu_j)\cdot\max\big\{|\Phi(\zeta_j)|,|\Phi(\zeta_{j+1})|\big\} \;,$$

with $\Phi(\zeta) := f'(\mu) = \frac{g'}{\rho'}(\zeta)$. Hence, for the evaluation of the simplified Rusanov numerical flux we need not invert ρ.

C++ code 11.11.8: Implementation of `semiDiscreteRhs()` ↪ GitLab

```
template <class CAUCHYPROBLEM>
Eigen::VectorXd semiDiscreteRhs(const Eigen::VectorXd &mu,
                                const Eigen::VectorXd &zeta,
                                CAUCHYPROBLEM prb) {
  int N = mu.size();
  Eigen::VectorXd rhs(N);

  // Define the (simplified) Rusanov 2-point numerical flux
  auto F = [&mu, &zeta, &prb](int i, int j) {
    double v = mu[i];
    double w = mu[j];
    double fv = prb.g(zeta[i]);
    double fw = prb.g(zeta[j]);
    double dfv = prb.dg(zeta[i]) / prb.drho(zeta[i]);
    double dfw = prb.dg(zeta[j]) / prb.drho(zeta[j]);
    double max_abs_df = std::max(std::abs(dfv), std::abs(dfw));
    return 0.5 * (fv + fw) - 0.5 * (w - v) * max_abs_df;
  };

  // Compute (-h) * [right-hand side]
  rhs[0] = F(0, 1) - F(0, 0);
  for (int j = 1; j < N - 1; ++j) {
    rhs[j] = F(j, j + 1) - F(j - 1, j);
  }
  rhs[N - 1] = F(N - 1, N - 1) - F(N - 2, N - 1);

  // Compute cell spacing h
  std::pair<double, double> limits = prb.domain();
```

```
30    double h = (limits.second - limits.first) / (N - 1);
31
32    // Include factor -1.0 / h
33    rhs = (-1.0 / h) * rhs;
34
35    return rhs;
36  }
```

SOLUTION of (11-11.f):

We write

$$\dot{\vec{\mu}}(t) = \mathbf{f}(\vec{\mu}(t)) \,, \quad \mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N \,, \tag{11.11.10}$$

for the ODE (11.11.7) arising from the conservative finite-volume spatial semi-discretization of $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u)$ One step (timestep size $\tau > 0$) of the explicit 3-stage Runge-Kutta single-step method with Butcher scheme (11.11.9) applied to (11.11.10) reads [Lecture → Def. 6.4.0.9]

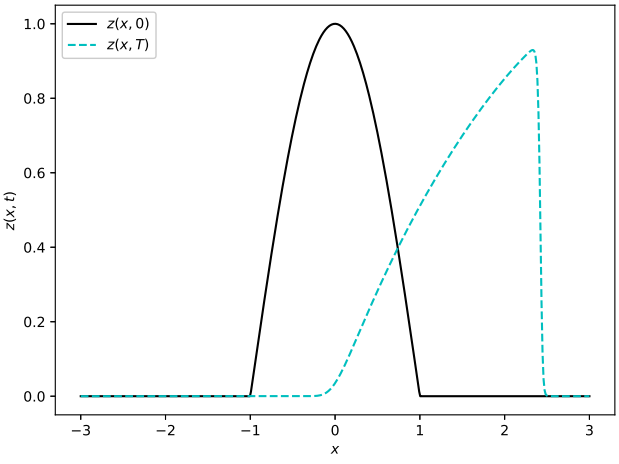
$$\begin{aligned}
\kappa_1 &:= \mathbf{f}(\vec{\mu}^{(k)}) \,, \\
\kappa_2 &:= \mathbf{f}(\vec{\mu}^{(k)} + \tau\tfrac{1}{3}\kappa_1) \,, \\
\kappa_3 &:= \mathbf{f}(\vec{\mu}^{(k)} + \tau\tfrac{2}{3}\kappa_2) \,, \\
\vec{\mu}^{(k+1)} &:= \vec{\mu}^{(k)} + \tau\tfrac{1}{4}\kappa_1 + \tau\tfrac{3}{4}\kappa_3 \,.
\end{aligned}$$

The evaluation of $\mathbf{f}$ can be accomplished through calling `semiDiscreteRhs()`. Once $\vec{\mu}^{(k+1)}$ is available, invoke `rhoInverse()` on each of its components to get $\vec{\zeta}^{(k+1)}$, using the corresponding component of $\vec{\zeta}^{(k)}$ as initial guess.

C++ code 11.11.11: Implementation of `solveCauchyPrb()` ➙ GitLab

```
template <class CAUCHYPROBLEM, typename RECORDER = std::function<
                                  void(double, const Eigen::VectorXd &)>>
Eigen::VectorXd solveCauchyPrb(
    unsigned int M, unsigned int N, CAUCHYPROBLEM prb,
    RECORDER &&rec = [](double /*time*/, const Eigen::VectorXd &
                        /*zstate*/) -> void {}) {
  // Get inital data for zeta
  std::pair<double, double> limits = prb.domain();
  Eigen::VectorXd x =
      Eigen::VectorXd::LinSpaced(N, limits.first, limits.second);
  auto z0 = [&prb](double y) { return prb.z0(y); };
  Eigen::VectorXd zeta = x.unaryExpr(z0);

  // Compute time step
  double T = prb.T();
  double dt = T / M;

  // Wrap the involved functions for simpler use below
  auto rho = [&prb](double v) { return prb.rho(v); };
  auto drho = [&prb](double v) { return prb.drho(v); };
  auto r = [rho, drho](double u, double z) {
    return rhoInverse(u, z, rho, drho);
  };

  // Get inital data for mu and get the time grid
  Eigen::VectorXd mu = zeta.unaryExpr(rho);
  Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(M + 1, 0.0, T);

  // record solution at initial time
  rec(t[0], zeta);

  for (int k = 0; k < M; ++k) {
```

```
    // Compute increment k1
    Eigen::VectorXd k1 = semiDiscreteRhs(mu, zeta, prb);

    // Compute incerment k2
    Eigen::VectorXd mu_temp = mu + dt / 3.0 * k1;
    Eigen::VectorXd zeta_temp = mu_temp.binaryExpr(zeta, r);
    Eigen::VectorXd k2 = semiDiscreteRhs(mu_temp, zeta_temp, prb);

    // Compute increment k3
    mu_temp = mu + dt * 2.0 / 3.0 * k2;
    zeta_temp = mu_temp.binaryExpr(zeta, r);
    Eigen::VectorXd k3 = semiDiscreteRhs(mu_temp, zeta_temp, prb);

    // Update mu and zeta according to the Butcher tableau
    mu = mu + dt * (0.25 * k1 + 0.0 * k2 + 0.75 * k3);
    zeta = mu.binaryExpr(zeta, r);

    // record solution after current time step
    rec(t[k + 1], zeta);
  }

  return zeta;
}
```



Solution computed by the conservative finite volume method discussed in this problem.

- $z_0(x) = \begin{cases} \cos(\frac{\pi}{2}x) & \text{for } |x| < 1\,, \\ 0 & \text{elsewhere,} \end{cases}$
- $\rho(z) = \exp(z)$, $g(z) = \frac{1}{2}\exp(2z)$,
- $M = 500$, $N = 500$, $T = 1.0$,
- computational domain $[-3, 3]$.

◁ Plot of solution at initial and final time

For this particular choice we have $f(u) = \frac{1}{2}u^2$ so that the transformed conservation law boils down to Burgers' equation [Lecture → § 11.1.3.3]. The behavior of its solution has extensively been discussed in the lecture, *cf.* [Lecture → Exp. 11.4.4.1]. Compare those results with Fig. 200.

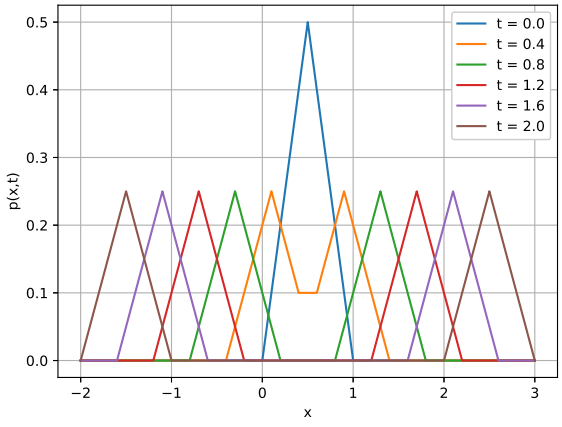
SOLUTION of (11-12.a):

Since $[t] = s$ and $[x] = m$, differentiation with respect to time multiplies the physical units of a quantity with s^{-1}m, while differentiation with respect to space has be be taken into account through a multiplication with m^{-1}. Thus we find the following physical units for the partial derivatives of p:

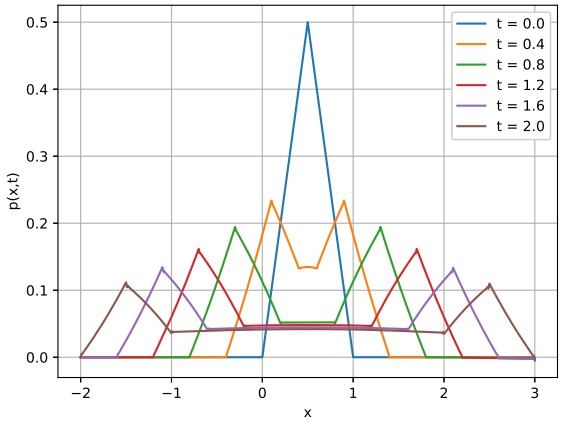
- $[\frac{\partial^2 p}{\partial t^2}] = 1\mathrm{Nm}^{-2}\mathrm{s}^{-2}$,
- $[\frac{\partial p}{\partial t}] = 1\mathrm{Nm}^{-2}\mathrm{s}^{-1}$,
- $[\frac{\partial^2 p}{\partial x^2}] = 1\mathrm{Nm}^{-4}$.

All the terms in (11.12.1) should have equal dimension, so we conclude that $[r] = s^{-1}$ and $[c] = ms^{-1}$. Note that c has the physical units of a velocity.

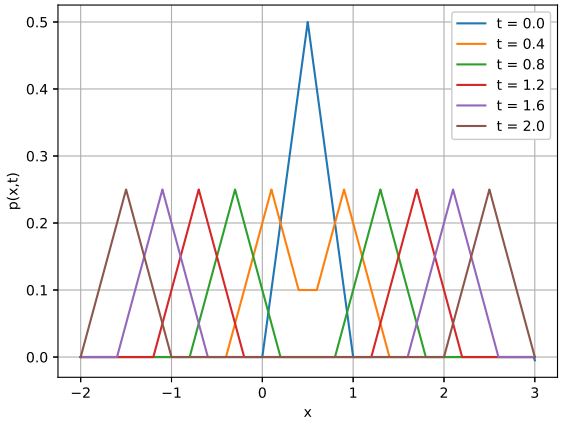
SOLUTION of (11-12.j):



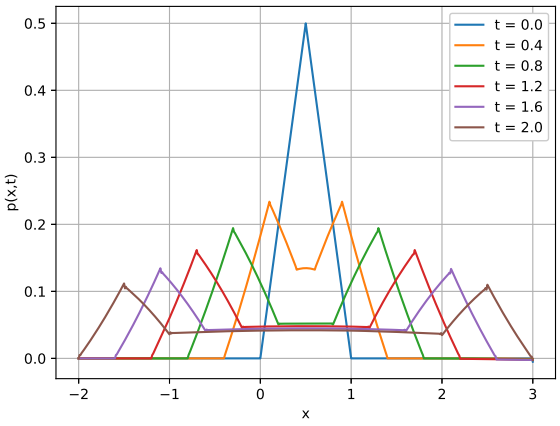
◁ Undamped solution using Lax-Friedrich flux.



◁ Damped solution using Lax-Friedrich flux.



◁ Undamped solution using Upwind flux.



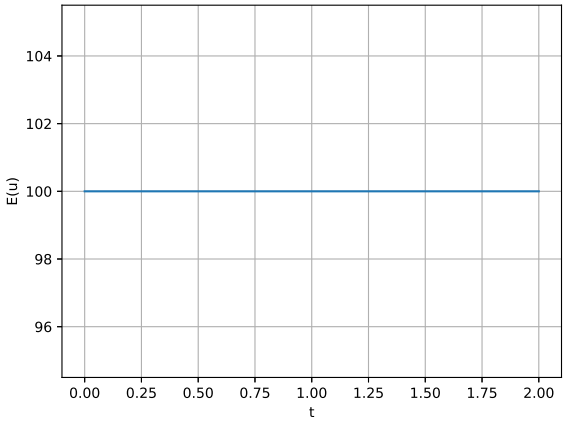
◁ Damped solution using Upwind flux.

C++ code 11.12.13: Code for `visWavSol()` ⇢ GitLab

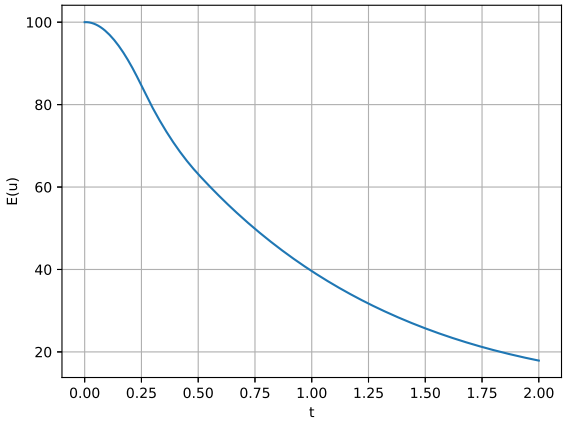
```
void visWavSol(double c, double r, double T, unsigned int N,
               Fluxtype nf_selector) {
  // Store the solution in this matrix, every column is a timestep
  Eigen::MatrixXd p;
  // Define the initial condition
  auto u0 = [](double x) -> Eigen::Vector2d {
    Eigen::Vector2d out;
    out(0) = 0.;
    if (0 <= x && x <= .5)
      out(1) = 1.;
    else if (0.5 <= x && x <= 1)
      out(1) = -1.;
    else
      out(1) = 0.;
    return out;
  };

  // Some parameters of the numerical method
  double maxspeed = c;
  double a = -T * maxspeed;
  double b = 1 + T * maxspeed;
  double h = (b - a) / N;

  // Recorder that automatically constructs p from the state variable
     for dp/dx
  auto record = [&p, h](Eigen::MatrixXd mu) -> void {
    p.conservativeResize(p.rows() + 1, mu.cols() + 1);
    Eigen::VectorXd dpdx = h * mu.row(1);
    p(p.rows() - 1, 0) = 0.;
    std::partial_sum(dpdx.begin(), dpdx.end(), p.row(p.rows() - 1).begin() + 1,
                     std::plus<double>());
    return;
  };

  // Execute the solver
  ev1DampWave(c, r, T, N, u0, nf_selector, record);

  // Write the solution to a txt-file.
  const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
                                         Eigen::DontAlignCols, ", ", "\n");
```

```
41    std::ofstream file;
42    file.open("solution.csv");
43    for (int j = 0; j < p.rows(); ++j)
44      file << p.row(j).format(CSVFormat) << std::endl;
45    file.close();
46
47    // Return
48    return;
49  }
```

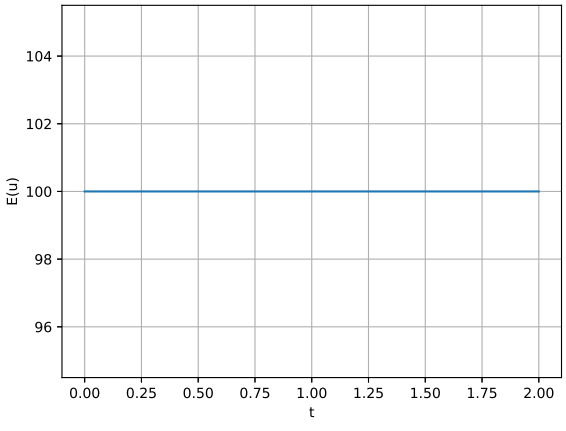
SOLUTION of (11-12.k):



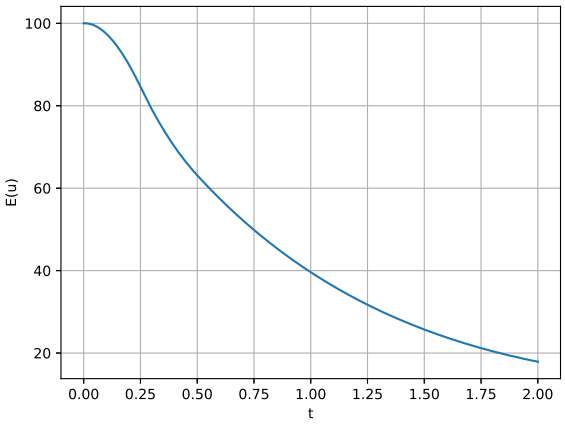
◁ Energy undamped solution using Lax-Friedrich flux.



◁ Energy damped solution using Lax-Friedrich flux.



◁ Energy undamped solution using Upwind flux.



◁ Energy damped solution using Upwind flux.

C++ code 11.12.14: Implementation of `trackEnergysWavSol()` ⇝ GitLab

```
Eigen::Matrix<double, Eigen::Dynamic, 2> trackEnergysWavSol(
    double c, double r, double T, unsigned int N, Fluxtype nf_selector) {
  // Store energies in the following matrix
  Eigen::Matrix<double, Eigen::Dynamic, 2> energies;

  // Define the initial condition
  auto u0 = [](double x) -> Eigen::Vector2d {
    Eigen::Vector2d out;
    out(0) = 0.;
    if (0 <= x && x <= .5)
      out(1) = 1.;
    else if (0.5 <= x && x <= 1)
      out(1) = -1.;
    else
      out(1) = 0.;
    return out;
  };

  // Some parameters of the numerical method
  double maxspeed = c;
  double a = -T * maxspeed;
  double b = 1 + T * maxspeed;
  double h = (b - a) / N;
  double dt = h / maxspeed;  //*(b - a) / ((N * maxspeed));

  // Recorder that computes energy
  auto record = [&energies, c, dt](Eigen::MatrixXd mu) -> void {
    unsigned int i = energies.rows();
    energies.conservativeResize(energies.rows() + 1, 2);
    energies(i, 0) = dt * i;
    energies(i, 1) =
        mu.row(0).dot(mu.row(0)) + c * c * mu.row(1).dot(mu.row(1));
    return;
  };

  // Execute the solver
  ev1DampWave(c, r, T, N, u0, nf_selector, record);

  // Write to txt-file
  const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
```

```
                                        Eigen::DontAlignCols, ", ", "\n");
  std::ofstream file;
  file.open("energies.csv");
  file << energies.col(0).transpose().format(CSVFormat) << std::endl;
  file << energies.col(1).transpose().format(CSVFormat) << std::endl;
  file.close();

  // Return
  return energies;
}
```

HINT 1 for (11-12.b): Start with the **separation-of-variables** trial expression $p(x,t) = q(t)\sin(\omega x)$ and find $q : \mathbb{R}^+ \to \mathbb{R}$ by solving an initial value problem for an ordinary differential equation. ⌟

SOLUTION of (11-12.b):

You learned the separation-of-variables approach in your analysis class: Plugging $p(x,t) = q(t)\sin(\omega x)$ into (11.12.1a), we obtain the 2nd-order *linear* ODE for $t \mapsto q(t)$

$$\frac{d^2q}{dt^2}(t) + r\frac{dq}{dt}(t) - c^2\omega^2 q(t) = 0\,. \tag{11.12.2}$$

We also know the initial conditions

$$q(0) = 1 \quad , \quad \frac{dq}{dt}(0) = 0\,. \tag{11.12.3}$$

The characteristic polynomial of (**??**) is $\xi \mapsto \xi^2 + r\xi - c^2\omega^2$ with zeros $\xi_{1/2} = \frac{1}{2}(-r \pm \sqrt{r^2 - 4c^2\omega^2})$. According to the sign of the discriminant we have to distinguish two cases:

(I) $r \geq 2c\omega$: In this case we have the fundamental system of exponentially decaying solutions

$$\left\{t \mapsto \exp(\tfrac{1}{2}(-r + \sqrt{r^2 - 4c^2\omega^2})t), t \mapsto \exp(\tfrac{1}{2}(-r - \sqrt{r^2 - 4c^2\omega^2})t)\right\}\,.$$

Use (11.12.3) to find the right linear combination of fundamental solutions yielding the unique solution for q:

$$q(t) = A\exp(\xi_1 t) + B\exp(xi_2 t) \quad \blacktriangleright \quad \begin{array}{r} A + B = 1\,, \\ \xi_1 A + \xi_2 B = 0\,. \end{array}$$

$$\blacktriangleright \quad B = \frac{\xi_1}{\xi_2 - \xi_1} \quad , \quad A = \frac{\xi_2}{\xi_1 - \xi_2}\,.$$

(II) $r < 2c\omega$: The fundamental system comprises exponentially decaying oscillating solutions

$$\left\{t \mapsto \exp(-\tfrac{1}{2}r)\,\exp(\imath\sqrt{4c^2\omega^2 - r}\,t), t \mapsto \exp(-\tfrac{1}{2}r)\,\exp(-\imath\sqrt{4c^2\omega^2 - r}\,t)\right\}\,,$$

which can equivalently be written as

$$\left\{t \mapsto \exp(-\tfrac{1}{2}rt)\,\sin(\kappa t), t \mapsto \exp(-\tfrac{1}{2}rt)\,\cos(\kappa t)\right\}\,, \quad \kappa := \sqrt{4c^2\omega^2 - r}\,.$$

Taking into account the initial conditions (11.12.3) we get

$$q(t) = \exp(-\tfrac{1}{2}rt)\,\sin(\kappa t)\,, \quad \kappa := \sqrt{4c^2\omega^2 - r}\,.$$

SOLUTION of (11-12.c):

Fro details refer to [Lecture $\rightarrow$ Ex. 11.6.1.5]. We get the standard form of a linear system of conservation laws with

- $\mathbf{u}(x,t) = \begin{bmatrix} v(x,t) \\ w(x,t) \end{bmatrix}$,
- $\mathbf{A} = \begin{bmatrix} 0 & -c^2 \\ -1 & 0 \end{bmatrix}$,
- $\mathbf{g}(\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}) = \begin{bmatrix} -rv_1 \\ 0 \end{bmatrix}$., $\begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \in \mathbb{R}^2$.

SOLUTION of (11-12.d):

Note that, thanks to $\mathbf{D}$ being symmetric, by the product rule and the PDE (11.12.4) we arrive at

$$\frac{dE(t)}{dt} = \int_{\mathbb{R}} \mathbf{u}(x,t)^\top \mathbf{D} \frac{\partial \mathbf{u}(x,t)}{\partial t} \, \mathrm{d}x = \int_{\mathbb{R}} \mathbf{u}(x,t)^\top \mathbf{D} \left(-\mathbf{A} \frac{\partial \mathbf{u}}{\partial x}(x,t) + \mathbf{g}(\mathbf{u}(x,t)) \right) \mathrm{d}x \, .$$

For the first term, we can apply integration by parts

$$\int_{\mathbb{R}} \mathbf{u}(x,t)^\top \mathbf{D}\mathbf{A} \frac{\partial \mathbf{u}}{\partial x}(x,t) \, \mathrm{d}x = \frac{1}{2} \int_{\mathbb{R}} \mathbf{u}(x,t)^\top \mathbf{D}\mathbf{A} \frac{\partial \mathbf{u}}{\partial x} \, \mathrm{d}x - \frac{1}{2} \int_{\mathbb{R}} \left(\frac{\partial \mathbf{u}}{\partial x}(x,t) \right)^\top \mathbf{D}\mathbf{A}\mathbf{u}(x,t) \, \mathrm{d}x = 0,$$

where the boundary terms vanish, since the initial data are assumed to be constant outside a compact interval. Also we have that $\mathbf{DA} = (\mathbf{DA})^\top$, so the two terms cancel to zero. For the second term, we have

$$\int_{\mathbb{R}} \mathbf{u}(x,t)^\top \mathbf{D}\mathbf{g}(\mathbf{u}(x,t)) \, \mathrm{d}x = -r \int_{\mathbb{R}} |(\mathbf{u}(x,t))_1|^2 \, \mathrm{d}x < 0 \, , \tag{11.12.6}$$

which gives us the final result $\frac{dE}{dt}(t) \leq 0$ for all $t \geq 0$.

HINT 1 for (11-12.e): Refer to the explanations given in [Lecture → § 11.6.1.27].

SOLUTION of (11-12.e):

The eigenvalues of $\mathbf{A}$ are $\lambda_1 = c$ and $\lambda_2 = -c$ and they supply the characteristic speeds of propagation. We thus obtain that the domain of influence is

$$\left\{(x,t) \in \mathbb{R} \times \mathbb{R}^+ : a - tc \leq x \leq b + tc\right\} .$$

HINT 1 for (11-12.f): Of course, the time-dependent state variable of the ODE is the sequence of approximations of dual cell averages $\left(\boldsymbol{\mu}_j\right)_{j\in\mathbb{Z}}$ of $\mathbf{u}(x,t)$, *cf.* [Lecture $\rightarrow$ Eq. (11.6.3.3)]. ⌟

SOLUTION of (11-12.f):

We start from the local conservation property for dual cell averages:

$$\frac{d}{dt}\int_{\mathbf{x}_i-\frac{h}{2}}^{\mathbf{x}_i+\frac{h}{2}} \mathbf{u}(x,t)\,\mathrm{d}x = -\mathbf{A}\left(\mathbf{u}(\mathbf{x}_i+\frac{h}{2},t)-\mathbf{u}(\mathbf{x}_i-\frac{h}{2},t)\right)+\begin{bmatrix}-r & 0\\ 0 & 0\end{bmatrix}\int_{\mathbf{x}_i-\frac{h}{2}}^{\mathbf{x}_i+\frac{h}{2}} \mathbf{u}(x,t)\,\mathrm{d}x\,. \quad (11.12.7)$$

From it we obtain the **method-of-lines ODE**

$$\frac{d\boldsymbol{\mu}_i}{dt} = -\frac{1}{h}\left(\widehat{\mathbf{F}}(\boldsymbol{\mu}_{i+1},\boldsymbol{\mu}_i)-\widehat{\mathbf{F}}(\boldsymbol{\mu}_i,\boldsymbol{\mu}_{i-1})\right)+\begin{bmatrix}-r & 0\\ 0 & 0\end{bmatrix}\boldsymbol{\mu}_i\,. \quad (11.12.8)$$

SOLUTION of (11-12.g):

The implemetation closely follows [Lecture → Code 11.3.2.9]. Refer to the commenst for further explanations.

C++ code 11.12.9: Implementation of `linsysoderhs()` ↪ GitLab

```
2  template <typename FFunctor, typename gFunctor>
3  Eigen::MatrixXd linsysoderhs(const Eigen::MatrixXd &mu, double h,
4                               FFunctor &&numflux, gFunctor &&g) {
5    unsigned n = mu.cols();                              // length of state vector
6    Eigen::MatrixXd fd = Eigen::MatrixXd::Zero(2, n);    // flux difference vector
7    Eigen::MatrixXd rhs = Eigen::MatrixXd::Zero(2, n);   // return vector
8
9    // constant continuation of data
10   fd.col(0) = numflux(mu.col(0), mu.col(1)) - numflux(mu.col(0), mu.col(0));
11   // Fluxes on the interior
12   for (unsigned j = 1; j < n - 1; ++j) {
13     fd.col(j) = numflux(mu.col(j), mu.col(j + 1)) -
14                 numflux(mu.col(j - 1), mu.col(j));  // see (??)
15   }
16   // constant continuation of data!
17   fd.col(n - 1) = numflux(mu.col(n - 1), mu.col(n - 1)) -
18                   numflux(mu.col(n - 2), mu.col(n - 1));
19
20   // Compute the rhs
21   for (unsigned j = 0; j < n; ++j) {
22     rhs.col(j) = -fd.col(j) / h + g(mu.col(j));
23   }
24
25   // Efficient thanks to return value optimization (RVO)
26   return rhs;
27 }
```

SOLUTION of (11-12.h):

The explicit Euler method for (11.12.8) with timestep $\tau > 0$ reads

$$\boldsymbol{\mu}_i^{(k+1)} - \boldsymbol{\mu}_i^{(k)} = -\frac{\tau}{h}\left(\widehat{\mathbf{F}}(\boldsymbol{\mu}_{i+1}^{(k)}, \boldsymbol{\mu}_i^{(k)}) - \widehat{\mathbf{F}}(\boldsymbol{\mu}_i^{(k)}, \boldsymbol{\mu}_{i-1}^{(k)})\right) + \tau \begin{bmatrix} -r & 0 \\ 0 & 0 \end{bmatrix} \boldsymbol{\mu}_i^{(k)}$$

C++ code 11.12.10: Implementation of `fvEulLinSys()` ➙ GitLab

```
2  template <typename FFunctor, typename gFunctor, typename u0Functor,
3            typename RECORDER = std::function<void(const Eigen::MatrixXd &)>>
4  Eigen::MatrixXd fvEulLinSys(
5      double a, double b, double T, unsigned int N, unsigned int M,
6      u0Functor &&u0, FFunctor &numflux, gFunctor &&f,
7      RECORDER &&recorder = [](const Eigen::MatrixXd &) -> void {}) {
8    // Determine constants of scheme
9    double dt = T / M;
10   double h = (b - a) / N;
11   Eigen::MatrixXd mu(2, N);  // return vector
12
13   // Set initial conditions
14   for (unsigned j = 0; j < N - 1; ++j) {
15     mu.col(j) = u0(a + h / 2. + j * h);
16   }
17
18   // Solve the ODE
19   recorder(mu);
20   for (double t = 0; t < T - dt / 2.; t += dt) {
21     Eigen::MatrixXd rhs = linsysoderhs(mu, h, numflux, f);
22     mu += dt * rhs;
23     recorder(mu);
24   }
25
26   // Return
27   return mu;
28 }
```

SOLUTION of (11-12.i):

Recall that we have

$$\mathbf{A} = \begin{bmatrix} 0 & -c^2 \\ -1 & 0 \end{bmatrix} \tag{11.12.11}$$

The eigenvalues are $\lambda_1 = c$ and $\lambda_2 = -c$ with eigenvectors $\mathbf{r}_1 = [c, -1]^\top$ and $\mathbf{r}_2 = [c, 1]^\top$, respectively. We thus conclude that we have

- $\mathbf{R} = \begin{bmatrix} c & c \\ -1 & 1 \end{bmatrix}$
- $\mathbf{R}^{-1} = \frac{1}{2c} \begin{bmatrix} 1 & -c \\ 1 & c \end{bmatrix}$
- $\mathbf{D} = \begin{bmatrix} c & 0 \\ 0 & -c \end{bmatrix}$

such that $\mathbf{AR} = \mathbf{RD}$. We also have

- $\mathbf{D}^+ = \begin{bmatrix} c & 0 \\ 0 & 0 \end{bmatrix}$
- $\mathbf{D}^- = \begin{bmatrix} 0 & 0 \\ 0 & -c \end{bmatrix}$
- $\mathbf{A}^+ = \mathbf{R}\mathbf{D}^+\mathbf{R}^{-1} = \frac{1}{2} \begin{bmatrix} c & -c^2 \\ -1 & c \end{bmatrix}$
- $\mathbf{A}^- = \mathbf{R}\mathbf{D}^-\mathbf{R}^{-1} = \frac{1}{2} \begin{bmatrix} -c & -c^2 \\ -1 & -c \end{bmatrix}$

From [Lecture → Eq. (11.6.3.17)] and [Lecture → Eq. (11.6.3.21)] recall that the Godunov/upwinf flux $\widehat{\mathbf{F}}_{\text{uw}}$ and the Lax-Friedrichs flux $\widehat{\mathbf{F}}_{\text{LF}}$ for constant-coefficient linear systems of conservation laws are given as

- $\widehat{\mathbf{F}}_{\text{uw}}(\mathbf{v}, \mathbf{w}) = \mathbf{A}^+\mathbf{v} + \mathbf{A}^-\mathbf{w}$
- $\widehat{\mathbf{F}}_{\text{lf}}(\mathbf{v}, \mathbf{w}) = \mathbf{A}(\mathbf{v} + \mathbf{w}) + \frac{1}{2}(\mathbf{A}^+ - \mathbf{A}^-)(\mathbf{w} - \mathbf{v})$

We end up with the following code

C++ code 11.12.12: Code for `evlDampWave()` ⇝ GitLab

```
2  template <typename u0Functor,
3            typename RECORDER = std::function<void(const Eigen::MatrixXd &)>>
4  Eigen::Matrix<double, 2, Eigen::Dynamic> ev1DampWave(
5      double c, double r, double T, unsigned int N, u0Functor &&u0,
6      Fluxtype nf_selector,
7      RECORDER &&recorder = [](const Eigen::MatrixXd &) -> void {}) {
8    // Some parameters for the numerical scheme
9    double maxspeed = c;
10   double a = -T * maxspeed;
11   double b = 1 + T * maxspeed;
12   double h = (b - a) / N;
13   double dt = h / maxspeed;  //*(b - a) / ((N * maxspeed));
```

```
  unsigned int M = std::ceil(T / dt);

  // Define numerical flux
  auto numflux = [c, nf_selector](Eigen::VectorXd mu1,
                                  Eigen::VectorXd mu2) -> Eigen::VectorXd {
    Eigen::Vector2d out;
    if (nf_selector == UW) {  // Upwind
      out(0) = .5 * (c * mu1(0) - c * c * mu1(1)) +
               .5 * (-c * mu2(0) - c * c * mu2(1));
      out(1) = .5 * (-1 * mu1(0) + c * mu1(1)) + .5 * (-mu2(0) - c * mu2(1));
    } else {  // Lax-Friedrichs
      out(0) = -.5 * c * c * (mu1(1) + mu2(1)) - .5 * c * (mu2(0) - mu1(0));
      out(1) = -.5 * (mu1(0) + mu2(0)) - .5 * c * (mu2(1) - mu1(1));
    }
    return out;
  };

  // The RHS
  auto g = [r](Eigen::VectorXd u) -> Eigen::VectorXd {
    Eigen::Vector2d out;
    out(0) = -r * u(0);
    out(1) = 0.;
    return out;
  };

  // Solve the system
  Eigen::MatrixXd sol = fvEulLinSys(-T * maxspeed, 1 + T * maxspeed, T, N, M,
                                    u0, numflux, g, recorder);
  // Return
  return sol;
}
```

SOLUTION of (11-13.a):

Obviously the p-system (11.13.1) is of the form

$$\frac{\partial \mathbf{u}}{\partial t}(x,t) + \frac{\partial \mathbf{F}(\mathbf{u})}{\partial x}(x,t) = 0 \quad \text{on} \quad D\,, \qquad \text{([Lecture} \rightarrow \text{Eq. (11.6.2.3)])}$$

for the following choices of state vector and **flux function**,

$$\mathbf{u}(x,t) = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} := \begin{bmatrix} v(x,t) \\ w(x,t) \end{bmatrix} \quad , \quad \mathbf{F}(\boldsymbol{u}) := \begin{bmatrix} -u_2 \\ p(u_1) \end{bmatrix} . \qquad (11.13.2)$$

SOLUTION of (11-13.j):

Note that a reasonable state space for $p(v) = -e^v$ is $\mathbb{R}^2$, because this pressure function is decreasing on all of $\mathbb{R}$.

Recall from Sub-problem (11-13.i) that we have the following eigenvalues for $\mathbf{A}_R(\mathbf{v}, \mathbf{w})$

$$\widehat{\lambda}_{1,2} = \mp\sqrt{-\frac{p(w) - p(v)}{w - v}} \quad , w, v \in \mathbb{R},\ v \neq w\ . \tag{11.13.17}$$

The corresponding eigenvectors can be computed:

$$\mathbf{r}_{1,2} = \begin{bmatrix} 1 \\ \pm\sqrt{-\frac{p(w)-p(v)}{w-v}} \end{bmatrix} \quad , w, v \in \mathbb{R},\ v \neq w\ . \tag{11.13.18}$$

This allows us to define the "eigenbasis" matrix $\mathbf{R}$,

$$\mathbf{R} = \begin{bmatrix} 1 & 1 \\ \sqrt{-\frac{p(w)-p(v)}{w-v}} & -\sqrt{-\frac{p(w)-p(v)}{w-v}} \end{bmatrix} , w, v \in \mathbb{R},\ v \neq w\ . \tag{11.13.19}$$

with inverse

$$\mathbf{R}^{-1} = \frac{1}{-\sqrt{-\frac{p(w)-p(v)}{w-v}}} \begin{bmatrix} -\sqrt{-\frac{p(w)-p(v)}{w-v}} & -1 \\ -\sqrt{-\frac{p(w)-p(v)}{w-v}} & 1 \end{bmatrix} . \tag{11.13.20}$$

This allows us to write the Roe matrix as

$$\mathbf{A}_R(\mathbf{v}, \mathbf{w}) = \mathbf{R}\,\mathrm{diag}\left(-\sqrt{-\frac{p(w) - p(v)}{w - v}}, \sqrt{-\frac{p(w) - p(v)}{w - v}} \right) \mathbf{R}^{-1} . \tag{11.13.21}$$

and we find

$$\begin{aligned} |\mathbf{A}_R(\mathbf{v}, \mathbf{w})| &= \mathbf{R}\,\mathrm{diag}\left(\sqrt{-\frac{p(w) - p(v)}{w - v}}, \sqrt{-\frac{p(w) - p(v)}{w - v}} \right) \mathbf{R}^{-1} \\ &= \begin{bmatrix} -2\sqrt{-\frac{p(w)-p(v)}{w-v}} & 0 \\ 0 & 2\sqrt{-\frac{p(w)-p(v)}{w-v}} \end{bmatrix} \end{aligned}$$

We conclude that the Lax-Friedrichs flux is

$$\widehat{\mathbf{F}}_{\mathrm{LF}}(\mathbf{v}, \mathbf{w}) = \begin{bmatrix} -\frac{1}{2}(v_2 + w_2) + \sqrt{-\frac{p(w) - p(v)}{w - v}}(w_1 - v_1) \\ \frac{1}{2}(p(v_1) + p(w_1)) - \sqrt{-\frac{p(w) - p(v)}{w - v}}(w_2 - v_2) \end{bmatrix} . \tag{11.13.22}$$

SOLUTION of (11-13.k):

Recall that we have the eigenvalues of $D\mathbf{f}(\mathbf{v})$

$$\lambda_{1,2}(\mathbf{v}) = \mp\sqrt{-p'(v_1)} \tag{11.13.23}$$

and the eigenvalues of the Roe matrix

$$\widehat{\lambda}_{1,2} = \mp\sqrt{-\frac{p(w)-p(v)}{w-v}} \tag{11.13.24}$$

Thus, according to [Lecture → Eq. (11.6.3.35)], have for a three-state solution $\breve{\mathbf{u}}$ with $\mathbf{v}$ on the left and Vw on the right the edge/shock speeds

$$\dot{s}^- = \min(-\sqrt{-p'(v_1)}, -\sqrt{-\frac{p(w_1)-p(v_1)}{w_1-v_1}}) \tag{11.13.25}$$

and

$$\dot{s}^+ = \min(\sqrt{-p'(w_1)}, \sqrt{-\frac{p(w_1)-p(v_1)}{w_1-v_1}}) \,. \tag{11.13.26}$$

From them we can computer the middle state $\mathbf{u}^*$ by the straightforward formula [Lecture → Eq. (11.6.3.37)], which allows us to compute the HLLE numerical flux directly from [Lecture → Eq. (11.6.3.38)].

We observe that for all pairs of states $\mathbf{v}, \mathbf{w}$ we have $\dot{s}^- < 0 < \dot{s}^+$ in this case, which means that $\widehat{\mathbf{F}}_{\text{HLLE}} = \mathbf{F}$ throughout in this case.

$$\widehat{\mathbf{F}}_{\text{HLLE}} = \mathbf{F}\left(\frac{\mathbf{F}(\mathbf{w}) - \mathbf{F}(\mathbf{v}) - \dot{s}^+\mathbf{w} + \dot{s}^-\mathbf{v}}{\dot{s}^- - \dot{s}^+}\right) \,. \tag{11.13.27}$$

SOLUTION of (11-13.l):

The method-of-lines ODE for a finite-volume methods with numerical flux function $\widehat{\mathbf{F}} = \widehat{\mathbf{F}}(\mathbf{v}, \mathbf{w})$ is

$$\frac{d\boldsymbol{\mu}_j}{dt}(t) = -\frac{1}{h}\left(\widehat{\mathbf{F}}(\boldsymbol{\mu}_j(t), \boldsymbol{\mu}_{j+1}(t)) - \widehat{\mathbf{F}}(\boldsymbol{\mu}_{j-1}(t), \boldsymbol{\mu}_j(t))\right), \quad \text{for a suitable range of } j\,.$$

[Lecture → Eq. (11.6.3.13)]

The fully discrete evolution resulting from explicit Euler timestepping with timestep $\tau > 0$ reads

$$\boldsymbol{\mu}_j^{(k+1)} = \vec{\boldsymbol{\mu}}_j^{(k)} - \frac{\tau}{h}\left(\widehat{\mathbf{F}}(\boldsymbol{\mu}^{(k)}{}_j(t), \boldsymbol{\mu}^{(k)}{}_{j+1}(t)) - \widehat{\mathbf{F}}(\boldsymbol{\mu}_{j-1}^{(k)}(t), \boldsymbol{\mu}_j^{(k)}(t))\right), \quad k \in \mathbb{N}_0\,. \tag{11.13.28}$$

A C++ implementation of the right-hand side of [Lecture → Eq. (11.6.3.13)] in the scalar case is demonstrated in [Lecture → Code 11.3.2.9]. The extension to vector-valued stated should be straightforward.

The following code implements (11.13.28) with uniform timestep $\tau := T/M$. The HLLE numerical flux is realized through a lambda function.

C++ code 11.13.29: Sub-problem (11-13.l): Implementation of function `evlExpPSystem()`
→ GitLab

```
2  template <typename u0Functor,
3            typename RECORDER = std::function<void(const Eigen::MatrixXd &)>>
4  Eigen::MatrixXd ev1ExpPSystem(
5      double a, double b, double T, unsigned int N, unsigned int M,
6      u0Functor &&u0,
7      RECORDER recorder = [](const Eigen::MatrixXd &) -> void {}) {
8    // Determine constants of scheme
9    const double dt = T / M;       // timestep size
10   const double h = (b - a) / N;  // meshwidth
11   Eigen::MatrixXd mu(2, N);      // return vector
12
13   // Set initial conditions
14   for (unsigned j = 0; j < N; ++j) {
15     mu.col(j) = u0(a + h / 2. + j * h);
16   }
17
18   // Define the function p, note that p' = p for this special case
19   auto p = [](double u) -> double { return -std::exp(u); };
20
21   // Define the continuous flux function F, see (11.13.2)
22   auto F = [p](Eigen::Vector2d v) -> Eigen::Vector2d {
23     return {-v[1], p(v[0])};
24   };
25
26   // Define the HLLE flux [Lecture → Eq. (11.6.3.38)]
27   auto numflux = [p, F](Eigen::MatrixXd v,
28                         Eigen::MatrixXd w) -> Eigen::VectorXd {
29     double smin = std::min(-std::sqrt(-p(v(0))),
30                            -std::sqrt(-(p(w(0)) - p(v(0))) / (w(0) - v(0))));
31     double smax = std::max(std::sqrt(-p(w(0))),
32                            std::sqrt(-(p(w(0)) - p(v(0))) / (w(0) - v(0))));
33     if (smin > 0) return F(v);
34     if (smax > 0) {
35       const Eigen::Vector2d ustar =
```

```
36            1.0 / (smin - smax) * (F(w) - F(v) - smax * w + smin * v);
37        return F(ustar);
38      }
39      return F(w);
40    };
41
42    // Solve the MOL ODE using explicit Euler timestepping
43    recorder(mu);
44    for (double t = 0; t < T - dt / 2.; t += dt) {
45      Eigen::MatrixXd fd(2, mu.cols());  // Flux difference
46
47      // constant continuation of data
48      fd.col(0) = numflux(mu.col(0), mu.col(1)) - numflux(mu.col(0), mu.col(0));
49      // Fluxes on the interior
50      for (unsigned j = 1; j < N - 1; ++j) {
51        fd.col(j) = numflux(mu.col(j), mu.col(j + 1)) -
52                    numflux(mu.col(j - 1), mu.col(j));  // see (??)
53      }
54      // constant continuation of data!
55      fd.col(N - 1) = numflux(mu.col(N - 1), mu.col(N - 1)) -
56                      numflux(mu.col(N - 2), mu.col(N - 1));
57
58      mu -= dt / h * fd;
59      recorder(mu);
60    }
61
62    // Return
63    return mu;
64  }
```

SOLUTION of (11-13.m):

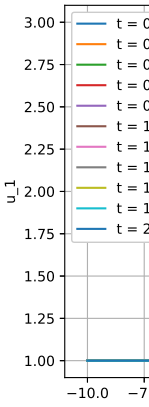
We have the following code

C++ code 11.13.30: Sub-problem (11-13.m): driver code for dynamic visualization of the solution of a Riemann problem ⇝ GitLab

```
int main(int /*argc*/, char** /*argv*/) {
  std::cout << "NumPDE homework problem FVPsystem by Wouter Tonnon\n";

  // Parameters of the simulation
  double a = -10;
  double b = 10;
  double T = 2;
  unsigned int N = 1000;
  unsigned int M = 1000;

  // We store the solution in this matrix, columns are timesteps
  Eigen::MatrixXd u1, u2;

  // Define the initial condition
  auto u0 = [](double x) -> Eigen::Vector2d {
    Eigen::Vector2d out(2);
    if (x <= 0) {
      out(0) = 1;
      out(1) = 1;
    } else {
      out(0) = 3;
      out(1) = 4;
    }
    return out;
  };

  // Recorder stores both elements of the vector solution in different
     matrices
  auto record = [&u1, &u2](Eigen::MatrixXd mu) -> void {
    u1.conservativeResize(u1.rows() + 1, mu.cols());
    u2.conservativeResize(u2.rows() + 1, mu.cols());
    // std::cout« mu.rows() « mu.cols() « u1.rows()«u1.cols()«std::endl;
    u1.row(u1.rows() - 1) = mu.row(0);
    u2.row(u2.rows() - 1) = mu.row(1);
    return;
  };

  // Solve the system
  auto mu = FVPsystem::ev1ExpPSystem(a, b, T, N, M, u0, record);

  // Write the solution to a txt-file.
  const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
                                         Eigen::DontAlignCols, ", ", "\n");
  std::ofstream file;
  file.open("solution_u1.csv");
  for (int j = 0; j < u1.rows(); ++j)
    file << u1.row(j).format(CSVFormat) << std::endl;
  file.close();
  file.open("solution_u2.csv");
  for (int j = 0; j < u2.rows(); ++j)
```

```
51      file << u2.row(j).format(CSVFormat) << std::endl;
52    file.close();
53
54    // Plot the solution using a python script
55    systemcall::execute(
56        "python3 ms_scripts/vis_solution.py solution_u1.csv solution_u1.eps u_1");
57
58    systemcall::execute(
59        "python3 ms_scripts/vis_solution.py solution_u2.csv solution_u2.eps u_2");
60
61    return 0;
62  }
```

We obtain the analytical solution to the 2-schock Riemann problem that we computed earlier.



We observe that the solution of the Riemann problem consists of

- one left-moving shock and
- a right-moving rarefaction wave.

SOLUTION of (11-13.b):

The Jacobian of the flux function $\mathbf{F} = \mathbf{F}(\mathbf{u})$ from (11.13.2) is

$$\mathrm{D}\mathbf{F}(\mathbf{u}) = \begin{bmatrix} 0 & -1 \\ p'(u_1) & 0 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}, \quad p' \triangleq \text{derivative of } p\,. \tag{11.13.4}$$

The characteristic polynomial of that matrix is the quadratic polynomial

$$\chi_{\mathrm{D}\mathbf{F}(\mathbf{u})}(\xi) = \xi^2 + p'(u_1)\,, \quad \xi \in \mathbb{R}\,.$$

Its zeros are easy to find and give the set of eigenvalues $\sigma(\mathrm{D}\mathbf{F}(\mathbf{u}))$ (spectrum) of $\mathrm{D}\mathbf{F}(\mathbf{u})$:

$$\sigma(\mathrm{D}\mathbf{F}(\mathbf{u})) = \left\{ +\sqrt{-p'(u_1)}, -\sqrt{-p'(u_1)} \right\}.$$

Obviously, $\sigma(\mathrm{D}\mathbf{F}(\mathbf{u}))$ contains two real elements, if and only if $p'(u_1) < 0$. Otherwise, $\sigma(\mathrm{D}\mathbf{F}(\mathbf{u})) = \{0\}$ or it contains two complex conjugate purely imaginary numbers.

Strict hyperbolicity of p-system

If and only if $p'(u_1) < 0$ for all admissible state components u_1 is the p-system strictly hyperbolic.

SOLUTION of (11-13.c):

The Rankine-Hugoniot jump condition for the p-system reads

$$\dot{s}(\mathbf{u}_l - \mathbf{u}_r) = \begin{bmatrix} \dot{s}(v_l - v_r) \\ \dot{s}(w_l - w_r) \end{bmatrix} = \mathbf{F}(\mathbf{u}_l) - \mathbf{F}(\mathbf{u}_r) = \begin{bmatrix} -w_l + w_r \\ p(v_l) - p(v_r) \end{bmatrix} .$$

From the first component we get $\dot{s} = (w_r - w_l)/(v_l - v_r)$ and plug this into the second component, which yields

$$-\frac{(w_l - w_r)^2}{v_l - v_r} = p(v_l) - p(v_r) \qquad \blacktriangleright \qquad (11.13.6) .$$

Remark. If $p'(v) < 0$ for all v, that is p is strictly decreasing, the argument of the square root in (11.13.6) is positive.

SOLUTION of (11-13.d):

The shock speed $\dot{s}$ can directly be read off the Rankine-Hugoniot jump conditions

$$\boxed{\dot{s}(\mathbf{u}_l - \mathbf{u}_r) = \mathbf{F}(\mathbf{u}_l) - \mathbf{F}(\mathbf{u}_r)} \quad , \quad \dot{s} := \frac{d\gamma}{d\tau} \quad \text{“shock speed” .} \qquad [\text{Lecture} \rightarrow \text{Eq. (11.6.2.20)}]$$

From the manipulations of the Rankine-Hugoniot jump condition in Sub-problem (11-13.c), we immediately deduce

$$\dot{s} = \pm\sqrt{-\frac{p(v_l) - p(v_r)}{v_l - v_r}} \quad \text{for} \quad v_l \neq v_r \; . \tag{11.13.7}$$

The eigenvalues of $\mathrm{D}\mathbf{F}(\mathbf{u})$ are $\pm\sqrt{p'(v)}$. So for $\mathbf{u}_l \rightarrow \mathbf{u}_r$ the shock speeds tend to the speeds given by the eigenvalues of $\mathrm{D}\mathbf{F}(\mathbf{u}_{l/r})$.

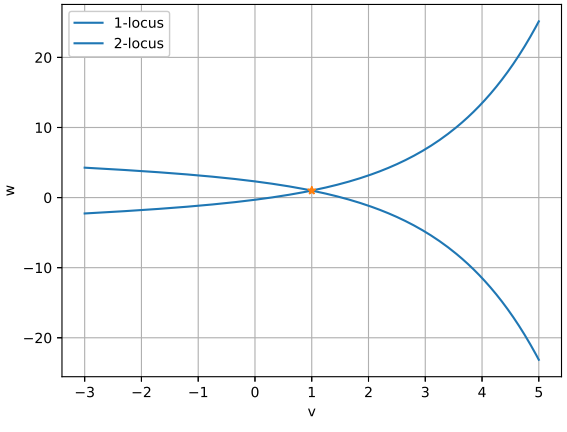
SOLUTION of (11-13.e):

Note that the Hugoniot loci are curves in the state-space/v/w-plane. In the case of the p-system these curves can be obtain as function graphs from (11.13.6).

PYTHON script 11.13.8: Sub-problem (11-13.e): Pyhton script for plotting Rankine-Hugoniot loci ⇝ GitLab

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel, legend, grid,
    gca,xlim, title
from numpy import genfromtxt, linspace, exp, sqrt
from sys import argv
from scipy.optimize import fsolve

# Define the argument of the Hugoniot Locus
wstar = 1
vstar = 1

# Define the parameter space
vr = linspace(-3,5,161)

# There are the 1- and 2-locus
wra = wstar + sqrt(-(vstar-vr)*(-exp(vstar)--exp(vr)))
wrb = wstar - sqrt(-(vstar-vr)*(-exp(vstar)--exp(vr)))

# Plot the 1- and 2-locus
figure()
ax = gca()
color = next(ax._get_lines.prop_cycler)['color']
plot(vr,wra,color=color,label="1-locus")
plot(vr,wrb,color=color,label="2-locus")
xlabel('v')
ylabel('w')
legend()
grid()
savefig("HugoniotLocus.eps")
```

We obtain the following result.



◁ Hugoniot locus for state $\left[\begin{smallmatrix}1\\2\end{smallmatrix}\right] \leftrightarrow \bullet$

The eigenvectors $\mathbf{r}_1$ and $\mathbf{r}_2$ are tangent to the two curves forming the Hugoniot locus. The curve whose tangent in $\bullet$ is $\mathbf{r}_i$, $i = 1,2$, is therefore called the i-locus, see also [Lecture → Fig. 609] and [Lecture → Fig. 610].

HINT 1 for (11-13.f): You may modify your PYTHON script from Sub-problem (11-13.e) to also plot the Hugoniot locus for the state $\mathbf{u}_r = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$. The intersection points of the two Hugoniot loci give estimates for the intermediate state $\mathbf{u}^*$ of the two-shock solution, *cf.* [Lecture $\rightarrow$ **??**]. ⌟

HINT 2 for (11-13.f): The two possible intermediate states are approximately

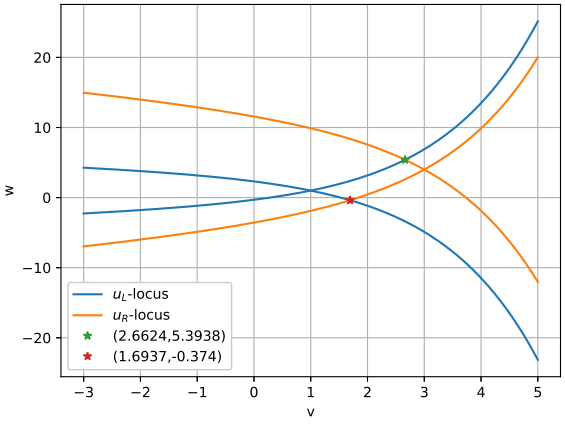
$$\mathbf{u}^* = \begin{bmatrix} 2.6624 \\ 5.3938 \end{bmatrix} \quad \text{and} \quad \mathbf{u}^* = \begin{bmatrix} 1.6937 \\ -0.374 \end{bmatrix}.$$

SOLUTION of (11-13.f):

A two-shock solution of the Riemnann problem comprises three zones of constant states, the left state $\mathbf{u}_l$, the **intermediate state** $\mathbf{u}^*$, and the right state $\mathbf{u}_r$. These zones are separated by discontinuities moving with constant speeds.

In Sub-problem (11-13.c) we already found conditions on pairs of states separated by a shock. The conditions must hold for the pairs of states $(\mathbf{u}_l, \mathbf{u}^*)$ and $(\mathbf{u}^*, \mathbf{u}_r)$, which gives a 2×2 non-linear system of equations for the unknown intermediate state $\mathbf{u}^*$:

$$w_l - w^* \pm \sqrt{-(p(v_l) - p(v^*))(v_l - v^*)} = 0\,,$$
$$w_r - w^* \pm \sqrt{-(p(v_r) - p(v^*))(v_r - v^*)} = 0\,.$$



Plots as in [Lecture → Ex. 11.6.2.25].

◁ Plot of the Hugoniot loci $\mathcal{H}(\mathbf{u}_l)$ and $\mathcal{H}(\mathbf{u}_r)$ for both (left and right) states involved in the Riemann problem.

Note that they intersect in two points. Apparently these are the only two intersection points.

We conclude that there are two possible intermediate states $\mathbf{u}^*$ for an all-shock Riemann solution.

We ran the following code to produce the figure.

PYTHON script 11.13.9: Sub-problem (11-13.f): Pyhton script for plotting graphically determining intersection of Hugoniot loci. ⇝ GitLab

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel, legend, grid,
    gca,xlim, title
from numpy import genfromtxt, linspace, exp, sqrt
from sys import argv
from scipy.optimize import fsolve

# Define the arguments of the Hugoniot Loci
vl = 1
wl = 1
vr = 3
wr = 4

# Define the parameter space
vstar = linspace(-3,5,161)

# We have the 1- and 2-locus of both Hugoniot Locus
wstara = wl + sqrt(-(vl-vstar)*(exp(vstar)-exp(vl)))
wstarb = wl - sqrt(-(vl-vstar)*(exp(vstar)-exp(vl)))
wstarc = wr + sqrt(-(vr-vstar)*(exp(vstar)-exp(vr)))
wstard = wr - sqrt(-(vr-vstar)*(exp(vstar)-exp(vr)))
```

```
20
21 # Plot the Hugoniot Loci in different colors.  Also plot their
       intersection points.
22 figure()
23 ax = gca()
24 color = next(ax._get_lines.prop_cycler)['color']
25 plot(vstar,wstara,color=color,label="$u_L$-locus")
26 plot(vstar,wstarb,color=color,label="_locus")
27 color = next(ax._get_lines.prop_cycler)['color']
28 plot(vstar,wstarc,color=color,label="$u_R$-locus")
29 plot(vstar,wstard,color=color,label="_4-locus")
30 color = next(ax._get_lines.prop_cycler)['color']
31 plot([2.6624],[5.3938],'*',color=color,label="(2.6624,5.3938)")
32 color = next(ax._get_lines.prop_cycler)['color']
33 plot([1.6937],[-0.374],'*',color=color,label="(1.6937,-0.374)")
34 xlabel('v')
35 ylabel('w')
36 legend()
37 grid()
38 savefig("HugoniotLocusIntermediateState.eps")
39
40 # Next we numerically find the solution
41 # There are two points where the Hugoniot Loci cross.  The points solve
       the following sets of equations as defined in f1 and f2
42 def f1(ustar):
43     vstar = ustar[0]
44     wstar = ustar[1]
45
46     return [wl + sqrt(-(vl-vstar)*(exp(vstar)-exp(vl))) - wstar, \
47             wr + sqrt(-(vr-vstar)*(exp(vstar)-exp(vr))) - wstar]
48
49 def f2(ustar):
50     vstar = ustar[0]
51     wstar = ustar[1]
52
53     return [wl - sqrt(-(vl-vstar)*(exp(vstar)-exp(vl))) - wstar, \
54             wr - sqrt(-(vr-vstar)*(exp(vstar)-exp(vr))) - wstar]
55
56 # Solve for the intersections
57 x1 = fsolve(f1,[0,0])
58 x2 = fsolve(f2,[0,0])
59
60 # Print the solution
61 print(x1)
62 print(x2)
```

SOLUTION of (11-13.g):

The two possible intermediate solutions are approximately

$$\mathbf{u}_a^* = \begin{bmatrix} 2.6624 \\ 5.3938 \end{bmatrix} \quad \text{and} \quad \mathbf{u}_b^* = \begin{bmatrix} 1.6937 \\ -0.374 \end{bmatrix}.$$

We write the following code to graphically check the Lax entropy condition. See [Lecture → Ex. 11.6.2.31] for details about the underlying reasoning.

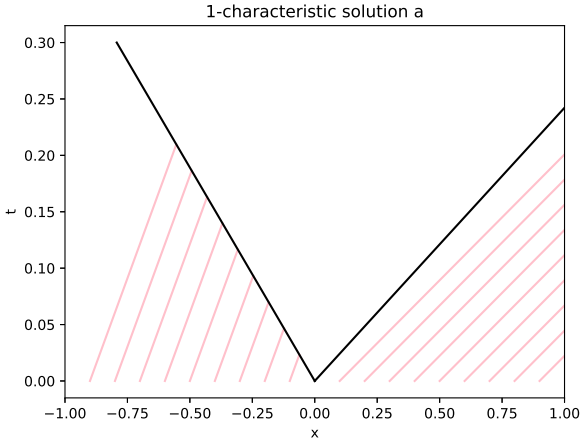
PYTHON script 11.13.10: Sub-problem (11-13.g): Python script for graphically investigating the satisfaction of the Lax entropy condition ➙ GitLab

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel, legend, grid,
    gca,xlim, title
from numpy import genfromtxt, linspace, exp, sqrt
from sys import argv
from scipy.optimize import fsolve

# First we defined the left-, right-, and intermediate states
uL = [1,1]
ustara = [2.6624, 5.3938]
ustarb = [1.6937, -0.374]
uR = [3,4]

# We compute the shock-speed velocities for all different possible
    solutions
sLa = (ustara[1]-uL[1])/(uL[0]-ustara[0])
sLb = (ustarb[1]-uL[1])/(uL[0]-ustarb[0])
sRa = (uR[1]-ustara[1])/(ustara[0]-uR[0])
sRb = (uR[1]-ustarb[1])/(ustarb[0]-uR[0])

# We compute the eigenvalues for all possible solutions
lambda1uL = sqrt(exp(1))
lambda1uR = sqrt(exp(3))
lambda2uL = -sqrt(exp(1))
lambda2uR = -sqrt(exp(3))
lambda1ustara = sqrt(exp(ustara[0]))
lambda2ustara = -sqrt(exp(ustara[0]))
lambda1ustarb = sqrt(exp(ustarb[0]))
lambda2ustarb = -sqrt(exp(ustarb[0]))

# We plot only up to t=0.3
tmax = 0.3

# We plot the 1-characteristics for solution a
# We plot multiple parallel lines to show impinging of the
    characteristics
figure()
for i in range(0,10):
    t = linspace(0,min(tmax,-0.1*i/(sLa-lambda1uL)),101)
    plot((lambda1uL*t-.1*i),t,color="pink")
    t = linspace(0,min(tmax,-.1/(sRa-lambda1uR)*i),101)
    plot((lambda1uR*t+.1*i),t,color="pink")

# We plot the shock speed
t = linspace(0,tmax,101)
```

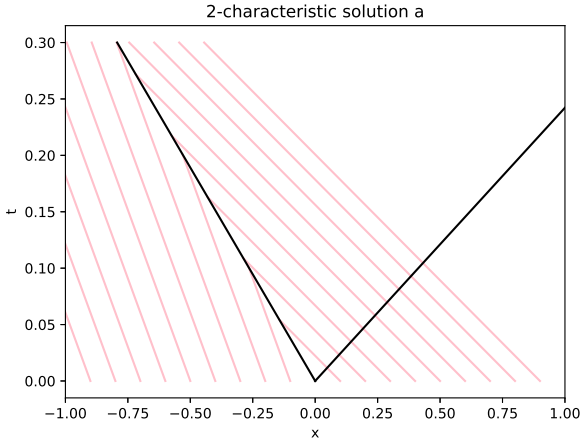
```
plot(sLa*t,t,color="black")
plot(sRa*t,t,color="black")

# We make the figure fancy and save it
xlim([-1,1])
title('1-characteristic solution a')
xlabel('x')
ylabel('t')
savefig("solution_a_lambda1.eps")

# We plot the 2-characteristics for solution a
# We plot multiple parallel lines to show impinging of the
    characteristics
figure()
for i in range(0,10):
    t = linspace(0,min(tmax,-0.1*i/(sLa-lambda2uL)),101)
    plot((lambda2uL*t-.1*i),t,color="pink")
    t = linspace(0,min(tmax,.1/(sLa-lambda2uR)*i),101)
    plot((lambda2uR*t+.1*i),t,color="pink")

# We plot the shock speed
t = linspace(0,tmax,101)
plot(sLa*t,t,color="black")
plot(sRa*t,t,color="black")

# We make the figure fancy and save it
title('2-characteristic solution a')
xlabel('x')
ylabel('t')
xlim([-1,1])
savefig("solution_a_lambda2.eps")

# We plot the 1-characteristics for solution b
# We plot multiple parallel lines to show impinging of the
    characteristics
figure()
for i in range(0,10):
    t = linspace(0,min(tmax,-.1*i/(sRb-lambda1uL)),101)
    plot((lambda1uL*t-.1*i),t,color="pink")
    t = linspace(0,tmax,101)
    plot((lambda1uR*t+.1*i),t,color="pink")

# We plot the shock speed
t = linspace(0,tmax,101)
plot(sLb*t,t,color="black")
plot(sRb*t,t,color="black")

# We make the figure fancy and save it
xlim([-1,1])
title('1-characteristic solution b')
xlabel('x')
ylabel('t')
savefig("solution_b_lambda1.eps")

# We plot the 2-characteristics for solution b
# We plot multiple parallel lines to show impinging of the
    characteristics
figure()
for i in range(0,10):
```

```
    t = linspace(0,min(tmax,-0.1*i/(sRb-lambda2uL)),101)
    plot((lambda2uL*t-.1*i),t,color="pink")
    t = linspace(0,min(tmax,.1/(sRb-lambda2uR)*i),101)
    plot((lambda2uR*t+.1*i),t,color="pink")

# We plot the shock speed
t = linspace(0,tmax,101)
plot(sLb*t,t,color="black")
plot(sRb*t,t,color="black")

# We make the figure fancy and save it
title('2-characteristic solution b')
xlabel('x')
ylabel('t')
xlim([-1,1])
savefig("solution_b_lambda2.eps")
```

Neither solution satisfies the Lax entropy condition, which we can see from the following plots.



1-characteristics for solution a

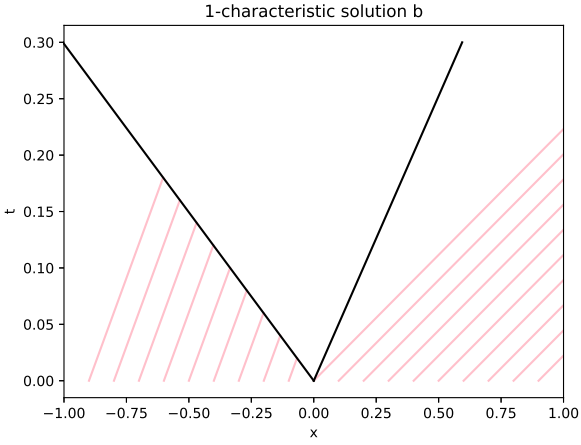


2-characteristics for solution a

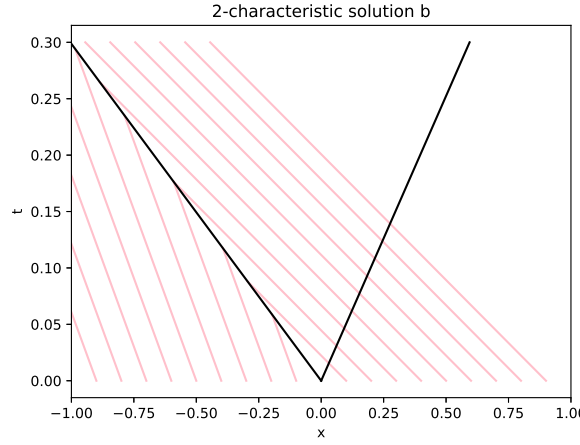
We observe

- that the 1-characteristics cross the slow shock from left to right and that the 2-characteristics converge in the slow shock. Hence the slow shock is “physical”.
- that the 1-characteristics emanate from the fast shock. Though the 2-characteristics cross is, it violates the Lax entropy condition.

The all-shock solution with intermediate state $\mathbf{u}_a^*$ does not comply with the Lax entropy condition.



1-characteristics for solution b



2-characteristics for solution b

From the plots we infer

- that the 1-characteristics cross the slow shock, while the 2-charateristics converge in it. The slow shock satisfies the Lax entropy condtion.
- that the 1-characteristics emanate from the fest shock, which, therefore, fails to meet the Lax entropy condition.

The all-shock solution with intermediate state $\mathbf{u}_b^*$ is “non-physical”.

SOLUTION of (11-13.h):

We write $\mathbf{u} = \left[\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix}\right]$. As we have seen in Sub-problem (11-13.b), we have the eigenvalues $\lambda_{1,2} = \pm\sqrt{-p'($
with corresponding eigenvectors

$$\mathbf{r}_{1,2} = \begin{bmatrix} 1 \\ \mp\sqrt{-p'(u_1)} \end{bmatrix}, \tag{11.13.12}$$

We obtain

$$\nabla\lambda_{1,2} \cdot \mathbf{r}_{1,2} = \frac{-p''(u_1)}{2\sqrt{-p'(u_1)}} \tag{11.13.13}$$

Recall that $p'(u_1) < 0$ by assumption. We thus conclude that we must have the condition

$$p''(u_1) \neq 0 \quad \forall\text{admissible } u_1 \,. \tag{11.13.14}$$

If the admissible range for u_1 is connected, then (11.13.14) means that p should have uniform curvature.

SOLUTION of (11-13.i):

We need to check the following

- Since $\dfrac{p(u_1+\delta)-p(u_1)}{u_1+\delta-u_1} \to p'(u)$, we have that $\mathbf{A}_R(\mathbf{u},\mathbf{u}) = D\mathbf{F}(\mathbf{u})$.
- $\mathbf{A}_R(\mathbf{v},\mathbf{w})(\mathbf{w}-\mathbf{v}) = \begin{bmatrix} v_2 - w_2 \\ \frac{p(w_1)-p(v_1)}{w_1-v_1}(w_1-v_1) \end{bmatrix} = \mathbf{F}(\mathbf{w}) - \mathbf{F}(\mathbf{v})$
- Since $p'(u_1) < 0$, we have that $\frac{p(v_{r,1})-p(v_{l,1})}{v_{r,1}-v_{l,1}} < 0$. We conclude that the eigenvalues are real and given as

$$\widehat{\lambda}_{1,2} = \mp\sqrt{-\frac{p(v_{r,1})-p(v_{l,1})}{v_{r,1}-v_{l,1}}}\,, \tag{11.13.16}$$

We conclude that the matrix is in fact a Roe matrix.

SOLUTION of (11-14.a):

We start from the formulas given in [Lecture → Ex. 11.6.2.13]:

$$\begin{aligned} \lambda_1 &= v - \sqrt{h} \quad \leftrightarrow \quad \mathbf{r}_1 = \begin{bmatrix} 1 \\ v - \sqrt{h} \end{bmatrix} \\ \lambda_2 &= v + \sqrt{h} \quad \leftrightarrow \quad \mathbf{r}_2 = \begin{bmatrix} 1 \\ v + \sqrt{h} \end{bmatrix} . \end{aligned}$$

[Lecture → Eq. (11.6.2.16)]

We have to express the eigenvalues and eigenvectors in terms of the conserved variables, the "$\mathbf{u}$-variables":

$$\lambda_1(\mathbf{u}) = \frac{u_2}{u_1} - \sqrt{u_1} \quad , \quad \mathbf{r}_1(\mathbf{u}) = \begin{bmatrix} 1 \\ \frac{u_2}{u_1} - \sqrt{u_1} \end{bmatrix} , \tag{11.14.3a}$$

$$\lambda_2(\mathbf{u}) = \frac{u_2}{u_1} + \sqrt{u_1} \quad , \quad \mathbf{r}_1(\mathbf{u}) = \begin{bmatrix} 1 \\ \frac{u_2}{u_1} + \sqrt{u_1} \end{bmatrix} . \tag{11.14.3b}$$

This implies by straightforward partial differentiation

$$\mathbf{grad}_{\mathbf{u}}\, \lambda_{1/2}(\mathbf{u}) = \begin{bmatrix} -\frac{u_2}{u_1^2} \mp \frac{1}{2}\frac{1}{\sqrt{u_1}} \\ \frac{1}{u_1} \end{bmatrix} = -\frac{1}{u_1} \begin{bmatrix} \frac{u_2}{u_1} \pm \frac{1}{2}\sqrt{u_1} \\ -1 \end{bmatrix} .$$

$$\blacktriangleright \quad \mathbf{grad}_{\mathbf{u}}\, \lambda_{1/2}(\mathbf{u}) \cdot \mathbf{r}_{1/2}(\mathbf{u}) = -\frac{1}{u_1} \begin{bmatrix} \frac{u_2}{u_1} \pm \frac{1}{2}\sqrt{u_1} \\ -1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ \frac{u_2}{u_1} \mp \sqrt{u_1} \end{bmatrix} = \mp \frac{3}{2\sqrt{u_1}} .$$

$$\blacktriangleright \quad \mathbf{grad}_{\mathbf{u}}\, \lambda_i(\mathbf{u}) \cdot \mathbf{r}_i(\mathbf{u}) = \begin{cases} -\dfrac{3}{2\sqrt{u_1}} & \text{for} \quad i = 1 , \\ +\dfrac{3}{2\sqrt{u_1}} & \text{for} \quad i = 2 . \end{cases}$$

This could also have been established by means of the following PYTHON code:

```
import numpy as np
# ---------------------
# define function and compute its derivatives
import sympy
x, y = sympy.symbols("x, y")
lam_mat = sympy.Matrix([y/x - sympy.sqrt(x), y/x + sympy.sqrt(x)])
r_mat = sympy.Matrix(
    [[1, 1],
     [y/x - sympy.sqrt(x), y/x + sympy.sqrt(x)]]
    )
Jlam_mat = lam_mat.jacobian(sympy.Matrix([x, y]))
res1 = sympy.simplify(Jlam_mat[0,:] * r_mat[:,0])[0]
res2 = sympy.simplify(Jlam_mat[1,:] * r_mat[:,1])[0]
sympy.pprint(res1)
sympy.pprint(res2)
```

Remark. Since $u_1 > 0$ we see that $\mathbf{grad}_{\mathbf{u}}\, \lambda_{1/2}(\mathbf{u}) \cdot \mathbf{r}_{1/2}(\mathbf{u}) \neq 0$ for any admissible state. The shallow water equations are genuinely non-linear.

HINT 1 for (11-14.b): Be aware that usually testing numerical computed floating-point results for equality is pointless. ⌟

SOLUTION of (11-14.b):

We have to do four tests.

1. We have to check whether the states are admissible
$$\mathbf{u}_l, \mathbf{u}_s, \mathbf{u}_r \in U := \left\{ \mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \in \mathbb{R}^2 : \ u_1 > 0 \right\} .$$
2. if $\mathbf{u}_l = \mathbf{u}^*$ exclude $(\mathbf{u}_l, \mathbf{u}^*)$ from testing, if $\mathbf{u}^* = \mathbf{u}_r$ do not further inspect the pair of states $(\mathbf{u}^*, \mathbf{u}_r)$.
3. Then we have to check, whether the pairs of states $(\mathbf{u}_l, \mathbf{u}^*)$ and $(\mathbf{u}^*, \mathbf{u}_r)$ satisfy the **Rankine-Hugoniot jump conditions** of [Lecture → Thm. 11.6.2.19],
$$\begin{aligned} \exists \dot{s}_1 \in \mathbb{R}: \quad & \dot{s}_1(\mathbf{u}_l - \mathbf{u}^*) = \mathbf{F}(\mathbf{u}_l) - \mathbf{F}(\mathbf{u}^*) , \\ \exists \dot{s}_2 \in \mathbb{R}: \quad & \dot{s}_2(\mathbf{u}_r - \mathbf{u}^*) = \mathbf{F}(\mathbf{u}_r) - \mathbf{F}(\mathbf{u}^*) , \end{aligned}$$
with the additional constraint $\dot{s}_1 \leq \dot{s}_2$.

   To test the Rankine-Hugoniot jump condition for two states $\mathbf{u}_1, \mathbf{u}_2 \in \mathbb{R}^2$ we investigate whether the null space of a particular matrix is non-trivial:
$$\mathcal{N}\left(\begin{bmatrix} \mathbf{u}_2 - \mathbf{u}_1 & \mathbf{F}(\mathbf{u}_1) - \mathbf{F}(\mathbf{u}_2) \end{bmatrix} \right) \neq \{\mathbf{0}\} . \tag{11.14.5}$$

   The simplest way to carry out this test in a numerically stable way in EIGEN, is to inquire about the rank of the matrix:

```
M.jacobiSvd().setThreshold(10 *
   std::numeric_limits<double>::epsilon).rank()) < 2
```

4. If a pair of states passes the second test, we have to verify the **Lax entropy condition**

Definition [Lecture → Def. 11.6.2.29]. Lax entropy condition for systems of conservation laws

A discontinuity separating states $\mathbf{u}_l$ and $\mathbf{u}_r$ and propagating at speed $\dot{s}$ satisfies the Lax entropy condition (LEC), if

(i) $\exists k \in \{1, \ldots, m\}$: $\lambda_k(\mathbf{u}_l) > \dot{s} > \lambda_k(\mathbf{u}_r)$
(ii) $\forall j < k$: $\lambda_j(\mathbf{u}_l), \lambda_j(\mathbf{u}_r) < \dot{s}$
(iii) $\forall j > k$: $\lambda_j(\mathbf{u}_l), \lambda_j(\mathbf{u}_r) > \dot{s}$

In the current situation this amounts to the following tests:

- $\exists k \in \{1,2\}$: $\lambda_k(\mathbf{u}_l) > \dot{s}_1 > \lambda_k(\mathbf{u}^*)$
- $\forall j < k$: $\lambda_j(\mathbf{u}_l), \lambda_j(\mathbf{u}^*) < \dot{s}_1$
- $\forall j > k$: $\lambda_j(\mathbf{u}_l), \lambda_j(\mathbf{u}^*) > \dot{s}_1$
- $\exists k \in \{1,2\}$: $\lambda_k(\mathbf{u}^*) > \dot{s}_2 > \lambda_k(\mathbf{u}_r)$
- $\forall j < k$: $\lambda_j(\mathbf{u}^*), \lambda_j(\mathbf{u}_r) < \dot{s}_2$
- $\forall j > k$: $\lambda_j(\mathbf{u}^*), \lambda_j(\mathbf{u}_r) > \dot{s}_2$

In fact, it is advisable to implement two helper functions

```
bool checkSWERHJC(Eigen::Vector2d ul, Eigen::Vector2d ur, double
  *speed);
bool checkSWEPhysicalShock(Eigen::Vector2d ul, Eigen::Vector2d ur);
```

that determine whether a shock with the two passed constant edge states satisfies the Rankine-Hugoniot jump conditions and the Lax entropy conditions, respectively.

C++ code 11.14.6: Check Rankine-Hugoniot Jump Condition

```
2  bool checkSWERHJC(Eigen::Vector2d ul, Eigen::Vector2d ur, double *speed) {
3    Eigen::Matrix2d M;
4    double g = 1;
5    M << ur(0) - ul(0), ul(1) - ur(1), ur(1) - ul(1),
6        ul(1) * ul(1) / ul(0) + .5 * g * ul(0) * ul(0) -
7            (ur(1) * ur(1) / ur(0) + .5 * g * ur(0) * ur(0));
8    auto rank = M.jacobiSvd()
9                    .setThreshold(10. * std::numeric_limits<double>::epsilon())
10                   .rank();
11   if (rank < 2) {
12     Eigen::FullPivLU<Eigen::MatrixXd> lu(M);
13     Eigen::MatrixXd M_null_space = lu.kernel();
14     *speed = M_null_space(0) / M_null_space(1);
15     return true;
16   }
17   return false;
18 }
```

C++ code 11.14.7: Check Lax Entropy Condition

```
2  bool checkSWEPhysicalShock(Eigen::Vector2d ul, Eigen::Vector2d ur) {
3    double s;
4    auto lambda_1 = [](Eigen::Vector2d u) -> double {
5      return u(1) / u(0) - std::sqrt(u(0));
6    };
7    auto lambda_2 = [](Eigen::Vector2d u) -> double {
8      return u(1) / u(0) + std::sqrt(u(0));
9    };
10   if (checkSWERHJC(ul, ur, &s)) {
11     // find k
12     if (lambda_1(ul) > s && s > lambda_1(ur)) {
13       if (lambda_2(ul) > s && lambda_2(ur) > s) return true;
14     } else if (lambda_2(ul) > s && s > lambda_2(ur)) {
15       if (lambda_1(ul) < s && lambda_1(ur) < s) return true;
16     }
17   }
18   return false;
19 }
```

C++ code 11.14.8: Check if the given 2-shock solution is physical

```
2  bool isPhysicalTwoShockSolution(Eigen::Vector2d ul, Eigen::Vector2d us,
3                                  Eigen::Vector2d ur) {
4    // Check if the states are admissible
```

```
5     if (ul(0) <= 0 || us(0) <= 0 || ur(0) <= 0) return false;
6
7     // Check whether the states are equal
8     bool ul_eq_us =
9         (ul - us).norm() < 10. * std::numeric_limits<double>::epsilon();
10    bool us_eq_ur =
11        (us - ur).norm() < 10. * std::numeric_limits<double>::epsilon();
12
13    // We check the Rankine-Hugoniot Jump Conditions
14    double s1, s2;
15    if (!ul_eq_us && !checkSWERHJC(ul, us, &s1)) return false;
16    if (!us_eq_ur && !checkSWERHJC(us, ur, &s2)) return false;
17    if (!ul_eq_us && !us_eq_ur && s1 > s2) return false;
18
19    // We check the Lax Entropy Condition
20    if (!checkSWEPhysicalShock(ul, us)) return false;
21    if (!checkSWEPhysicalShock(us, ur)) return false;
22
23      // If all tests passed, we have a physical shock
24    return true;
25  }
```

SOLUTION of (11-14.c):

The Lax-Friedrich numerical flux (function) is a 2-point numerical flux defined as

$$\widehat{\mathbf{F}}_{\mathrm{LF}}(\mathbf{v},\mathbf{w}) = \frac{1}{2}(\mathbf{F}(\mathbf{v}) + \mathbf{F}(\mathbf{w})) - \frac{1}{2}|\mathbf{A}_R(\mathbf{v},\mathbf{w})|(\mathbf{w}-\mathbf{v}) \quad , \quad \mathbf{v},\mathbf{w} \in U \ , \ [\text{Lecture} \rightarrow \text{Eq. (11.6.3.48)}]$$

where $\mathbf{A}_R(\mathbf{v},\mathbf{w})$ is a **Roe matrix** for the shallow water equations, and

$$|\mathbf{A}_R(\mathbf{v},\mathbf{w})| := \mathbf{R}\,\mathrm{diag}(|\widehat{\lambda}_1|,\ldots,|\widehat{\lambda}_m|)\mathbf{R}^{-1} \ , \quad \mathbf{A}_R(\mathbf{v},\mathbf{w}) = \mathbf{R}\,\mathrm{diag}(\widehat{\lambda}_1,\ldots,\widehat{\lambda}_m)\mathbf{R}^{-1} \ . \tag{11.14.9}$$

Note that the matrix $\mathbf{R}$ in (11.14.9) is the matrix whose columns are the eigenvectors of $\mathbf{A}_R(\mathbf{v},\mathbf{w})$.

From [Lecture → Eq. (11.6.3.32)], [Lecture → Eq. (11.6.3.32)], we already know how a Roe matrix for the shallow water equations can be defined:

$$\mathbf{A}_R(\mathbf{v},\mathbf{w}) = \mathrm{D}\mathbf{F}\left(\begin{bmatrix}\overline{h}\\ \overline{h}\widehat{v}\end{bmatrix}\right) \ , \quad \begin{aligned}\overline{h} &:= \tfrac{1}{2}(h_v + h_w) \ ,\\ \widehat{v} &:= \frac{\sqrt{h_v}v_v + \sqrt{h_w}v_w}{\sqrt{h_v}+\sqrt{h_w}} \ ,\end{aligned} \quad \mathbf{v} = \begin{bmatrix}h_v\\ h_v v_v\end{bmatrix}, \ \mathbf{w} = \begin{bmatrix}h_w\\ h_w v_w\end{bmatrix} \ .$$

Hence, in this case the Roe matrix $\mathbf{A}_R$ is $\mathrm{D}\mathbf{F}$ evaluated at the **Roe average state** $\begin{bmatrix}\overline{h}\\ \overline{h}\widehat{v}\end{bmatrix}$. Unfortunately, this state is given in "physical variables", whereas we consistently work with conserved variables. So we have to transform back and forth:

$$\mathbf{u} = \begin{bmatrix}u_1\\ u_2\end{bmatrix} := \begin{bmatrix}h\\ hv\end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix}h\\ v\end{bmatrix} = \begin{bmatrix}u_1\\ u_2/u_1\end{bmatrix} \ , \quad u_1 = h > 0 \ . \tag{11.14.10}$$

Thus the Roe average state $\overline{\mathbf{u}} = \overline{\mathbf{u}}(\mathbf{v},\mathbf{w}) \in U$, expressed in conserved variables is

$$\begin{aligned}\overline{u}_1 &= \tfrac{1}{2}(v_1 + w_1) \ ,\\ \overline{u}_2 &= \overline{u}_1 \frac{\sqrt{v_1}v_2/v_1 + \sqrt{w_1}w_2/w_1}{\sqrt{v_1}+\sqrt{w_1}} \ ,\end{aligned} \quad \mathbf{v} = \begin{bmatrix}v_1\\ v_2\end{bmatrix} \ , \quad \mathbf{w} = \begin{bmatrix}w_1\\ w_2\end{bmatrix} \ . \tag{11.14.11}$$

To obtain $|\mathbf{A}_R|$ we have to diagonalize the Roe matrix. Since $\mathbf{A}_R(\mathbf{v},\mathbf{w}) = \mathrm{D}\mathbf{F}(\overline{\mathbf{u}}(\mathbf{v},\mathbf{w}))$, we can do this based on (11.14.3), that is, we have

$$\mathbf{R} = \mathbf{R}(\mathbf{v},\mathbf{w}) = \mathbf{R}(\overline{\mathbf{u}}) = \begin{bmatrix}1 & 1\\ \frac{\overline{u}_2}{\overline{u}_1} - \sqrt{\overline{u}_1} & \frac{\overline{u}_2}{\overline{u}_1} + \sqrt{\overline{u}_1}\end{bmatrix} \ , \quad \begin{aligned}\widehat{\lambda}_1(\overline{\mathbf{u}}) &= \frac{\overline{u}_2}{\overline{u}_1} - \sqrt{\overline{u}_1} \ ,\\ \widehat{\lambda}_2(\overline{\mathbf{u}}) &= \frac{\overline{u}_2}{\overline{u}_1} + \sqrt{\overline{u}_1} \ ,\end{aligned}$$

$$\mathbf{R}^{-1} = \frac{2}{\sqrt{\overline{u}_1}}\begin{bmatrix}\frac{\overline{u}_2}{\overline{u}_1} + \sqrt{\overline{u}_1} & -1\\ -\frac{\overline{u}_2}{\overline{u}_1} + \sqrt{\overline{u}_1} & 1\end{bmatrix} \ .$$

The formula $|\mathbf{A}_R(\mathbf{v},\mathbf{w})| = \mathbf{R}\,\mathrm{diag}(|\widehat{\lambda}_1(\overline{\mathbf{u}})|,|\widehat{\lambda}_2(\overline{\mathbf{u}})|)\mathbf{R}^{-1}$ is implemented in Code 11.14.13. There $\mathbf{R}^{-1}$ is computed by means of EIGEN's built-in function `inverse()`.

Alternatively, one can simply use EIGEN's facilities for the diagonalization of matrices to compute $|\mathbf{A}_R|$.

C++ code 11.14.12: EIGEN-based computation of $|\mathbf{M}|$ for $\mathbf{M} \in \mathbb{R}^{2,2}$

```
Eigen::Matrix2d absMat(Eigen::Matrix2d M) {
  Eigen::EigenSolver<Eigen::Matrix2d> eigensolver(M);
  const Eigen::Vector2d ref_ev = eigensolver.eigenvalues().cwiseAbs();
  const auto ref_EV = eigensolver.eigenvectors();
  return (ref_EV * ref_ev.asDiagonal() * ref_EV.inverse()).real();
}
```

Once we have $|\mathbf{A}_R|$, the computation of $\widehat{\mathbf{F}}_{\mathrm{LF}}$ is straightforward:

C++ code 11.14.13: Code for `numfluxLFSWE()`

```
Eigen::Vector2d numfluxLFSWE(Eigen::Vector2d v, Eigen::Vector2d w) {
  assert((v[0] > 0.0) && "v-height must be positive!");
  assert((w[0] > 0.0) && "w-height must be positive!");
  Eigen::Vector2d nfLF{};
  const auto &F = &sweFluxFunction;
  // Compute Row average and asscoiated characteristic speeds
  const auto [u_bar, l1, l2] = RoeAvgSWE(v, w);
  // Compute "modulus" of Roe matrix
  const Eigen::Matrix2d EV_bar = sweEV(u_bar);
  const Eigen::Matrix2d D_abs =
      (Eigen::Matrix2d() << std::abs(l1), 0.0, 0.0, std::abs(l2)).finished();
  const Eigen::Matrix2d AR_abs = (EV_bar * D_abs) * EV_bar.inverse();
  // Implment [Lecture → Eq. (11.6.3.48)]
  nfLF = 0.5 * (F(v) + F(w)) - 0.5 * AR_abs * (w - v);
  return nfLF;
}
```

The following auxiliary functions are used. They can easily be incorporated into the body of `numfluxLFSW`

C++ code 11.14.14: Computation of Roe average state and associated characteristic velocities.

```
std::tuple<Eigen::Vector2d, double, double> RoeAvgSWE(Eigen::Vector2d v,
                                                      Eigen::Vector2d w) {
  assert((v[0] > 0.0) && "v-height must be positive!");
  assert((w[0] > 0.0) && "w-height must be positive!");
  // Roe average state for SWE
  Eigen::Vector2d u_bar;
  // Implements (11.14.11) to compute ū
  u_bar[0] = 0.5 * (v[0] + w[0]);  // Arithmetic average of first component
  const double sv0 = std::sqrt(v[0]);
  const double sw0 = std::sqrt(w[0]);
  // Row average in physical velocity
  const double v_avg = (sv0 * v[1] / v[0] + sw0 * w[1] / w[0]) / (sv0 + sw0);
  u_bar[1] = v_avg * u_bar[0];
  const std::pair<double, double> lambdas{sweLambdas(u_bar)};
  return {u_bar, lambdas.first, lambdas.second};
}
```

C++ code 11.14.15: Computes eigenvalues $\lambda_1(\mathbf{u})$, $\lambda_2(\mathbf{u})$ according to (11.14.3)**.**

```
inline std::pair<double, double> sweLambdas(Eigen::Vector2d u) {
  assert((u[0] > 0.0) && "Height must be positive!");
  const double sqrtu1 = std::sqrt(u[0]);
  const double quot = u[1] / u[0];
  return {quot - sqrtu1, quot + sqrtu1};
}
```

C++ code 11.14.16: Compute $\mathbf{R} := [\mathbf{r}_1(\mathbf{u}), \mathbf{r}_2(\mathbf{u})]$ based on (11.14.3)**.**

```
inline Eigen::Matrix2d sweEV(Eigen::Vector2d u) {
  assert((u[0] > 0.0) && "Height must be positive!");
  Eigen::Matrix2d EVs;
  EVs(0, 0) = 1;
  EVs(1, 0) = u[1] / u[0] - std::sqrt(u[0]);
  EVs(0, 1) = 1;
  EVs(1, 1) = u[1] / u[0] + std::sqrt(u[0]);
  return EVs;
}
```

SOLUTION of (11-14.d):

We recall the definition of the HLLE numerical flux for the shallow water equations:

$$\widehat{\mathbf{F}}_{\mathrm{HLLE}}(\mathbf{v},\mathbf{w}) = \begin{cases} \mathbf{F}(\mathbf{v}) & \text{, if } \quad \dot{s}^- > 0 \\ \mathbf{F}(\mathbf{u}^*) & \text{, if } \quad \dot{s}^- < 0 < \dot{s}^+ \\ \mathbf{F}(\mathbf{w}) & \text{, if } \quad \dot{s}^+ < 0 \end{cases} \begin{array}{l} \text{, } \dot{s}^- \text{ from [Lecture } \rightarrow \text{ Eq. (11.6.3.35)] ,} \\ \text{, } \mathbf{u}^* \text{ as in [Lecture } \rightarrow \text{ Eq. (11.6.3.37)] ,} \\ \text{, } \dot{s}^+ \text{ according to [Lecture } \rightarrow \text{ Eq. (11.6.3.35)] ,} \\ \qquad\qquad \text{[Lecture } \rightarrow \text{ Eq. (11.6.3.38)]} \end{array}$$

where

$$\mathbf{u}^* := \frac{\mathbf{F}(\mathbf{w}) - \mathbf{F}(\mathbf{v}) - \dot{s}^+\mathbf{w} + \dot{s}^-\mathbf{v}}{\dot{s}^- - \dot{s}^+} \,, \qquad \text{[Lecture } \rightarrow \text{ Eq. (11.6.3.37)]}$$

$$\dot{s}^- := \min\{\widehat{\lambda}_1, \lambda_1(\mathbf{v})\} \,,$$

$$\dot{s}^+ := \max\{\widehat{\lambda}_m, \lambda_m(\mathbf{w})\} \,, \qquad \text{[Lecture } \rightarrow \text{ Eq. (11.6.3.35)]}$$

and $\widehat{\lambda}_i$, $i = 1, \ldots, m$, are the (sorted) eigenvalues of a Roe matrix $\mathbf{A}_R(\mathbf{v},\mathbf{w})$. In the solution of Sub-problem (11-14.c) we already discussed how to obtain the Roe matrix $\mathbf{A}_R(\mathbf{v},\mathbf{w})$ as the Jacobian $D\mathbf{F}$ evaluated at the Roe average state $\overline{\mathbf{u}}(\mathbf{v},\mathbf{w})$. We reuse quite a few of the auxiliary functions devised for Sub-problem (11-14.c).

C++ code 11.14.17: Implementation of `numfluxHLLESWE()`

```
2  Eigen::Vector2d numfluxHLLESWE(Eigen::Vector2d v, Eigen::Vector2d w) {
3    assert((v[0] > 0.0) && "v-height must be positive!");
4    assert((w[0] > 0.0) && "w-height must be positive!");
5    Eigen::Vector2d nfHLLE{};
6    const auto &F = &sweFluxFunction;
7    // Compute characteristic speeds for edge states
8    const auto [l1v, l2v] = sweLambdas(v);
9    const auto [l1w, l2w] = sweLambdas(w);
10   // Compute Row average and associated characteristic speeds
11   auto [u_bar, l1, l2] = RoeAvgSWE(v, w);
12   // HLLE edge shock speeds according ro [Lecture → Eq. (11.6.3.35)]
13   const double s_minus = std::min(l1, l1v);
14   const double s_plus = std::max(l2, l2w);
15   // Compute middle state u* according to [Lecture → Eq. (11.6.3.37)]
16   const Eigen::Vector2d Fv{F(v)};
17   const Eigen::Vector2d Fw{F(w)};
18   if (s_minus > 0) {
19     nfHLLE = Fv;
20   } else if (s_plus < 0) {
21     nfHLLE = Fw;
22   } else {
23     const Eigen::Vector2d us =
24         (Fw - Fv - s_plus * w + s_minus * v) / (s_minus - s_plus);
25     nfHLLE = F(us);
26   }
27   return nfHLLE;
28 }
```

SOLUTION of (11-14.e):

We use the auxiliary function that computes the eigenvalues $\lambda_1(\mathbf{u})$ and $\lambda_2(\mathbf{u})$ of $\mathrm{D}\mathbf{F}(\mathbf{u})$, relying on the formulas (11.14.3).

C++ code 11.14.20: Computation of eigenvalues of the Jacobian $\mathrm{D}\mathbf{F}(\mathbf{u})$

```
2  inline std::pair<double, double> sweLambdas(Eigen::Vector2d u) {
3    assert((u[0] > 0.0) && "Height must be positive!");
4    const double sqrtu1 = std::sqrt(u[0]);
5    const double quot = u[1] / u[0];
6    return {quot - sqrtu1, quot + sqrtu1};
7  }
```

The timestep control respecting the CFL-condition with maximum speed of propagation according to (11.14.19) boils down to running over all current states to find the maximum characteristic speed.

C++ code 11.14.21: FVM for solving the shallow water equations.

```
2  template <typename U0FUNCTOR, typename NUMFLUX>
3  std::vector<Eigen::Vector2d> solveSWE(double T, unsigned int N, NUMFLUX &&NF,
4                                        U0FUNCTOR &&u0,
5                                        std::ostream *out = nullptr) {
6    // Initialize initial data for N cell values
7    std::vector<Eigen::Vector2d> u0_vec{N};
8    // Sample initial values on equidistant mesh
9    const double h = 2.0 / N;
10   double x = -1.0 + h / 2;
11   for (int k = 0; k < N; ++k, x += h) {
12     u0_vec[k] = u0(x);
13   }
14   // Functor for timestep control
15   auto ts_ctrl = [](double h,
16                     const std::vector<Eigen::Vector2d> &u_vec) -> double {
17     double s_max = 0.0;
18     double l1;
19     double l2;
20     // Determine maximal characteristic speed
21     for (const Eigen::Vector2d &u : u_vec) {
22       std::tie(l1, l2) = sweLambdas(u);
23       s_max = std::max({s_max, std::abs(l1), std::abs(l2)});
24     }
25     return h / s_max;
26   };
27   std::vector<std::vector<Eigen::Vector2d>> data;
28   std::vector<double> times;
29   FVMEvlGeneric(
30       -1.0, 1.0, T, ts_ctrl, u0_vec, NF,
31       [&data, &times](double t, const std::vector<Eigen::Vector2d> &u) -> void {
32         times.push_back(t);
33         data.push_back(u);
34       });
35   if (out) {
36     (*out) << "times = [";
37     for (auto t : times) {
```

```
      (*out) << t << ' ';
    }
    (*out) << "];\n u1 = [ ...\n";
    for (auto u_vec : data) {
      for (const Eigen::Vector2d &u : u_vec) {
        (*out) << u[0] << ' ';
      }
      (*out) << "; ...\n";
    }
    (*out) << "]; \n u2 = [ ...\n";
    for (auto u_vec : data) {
      for (const Eigen::Vector2d &u : u_vec) {
        (*out) << u[1] << ' ';
      }
      (*out) << "; ...\n";
    }
    (*out) << "];\n";
  }
  return data.back();
}
```

SOLUTION of (11-15.a):

shallow water equations read

$$\frac{\partial h}{\partial t} + \frac{\partial}{\partial x}(vh) = 0 \qquad \text{on} \quad \mathbb{R}\times]0,T[\,, \qquad [\text{Lecture} \rightarrow \text{Eq. (11.6.2.14a)}]$$

$$\frac{\partial}{\partial t}(hv) + \frac{\partial}{\partial x}(hv^2 + \tfrac{1}{2}gh^2) = 0 \qquad \text{on} \quad \mathbb{R}\times]0,T[\,, \qquad [\text{Lecture} \rightarrow \text{Eq. (11.6.2.14b)}]$$

Obviously, this is the same as (11.15.1) for $\rho \leftrightarrow h$, and $p(\rho) = \frac{1}{2}g\rho^2$.

SOLUTION of (11-15.b):

The transformation is the same as for the 1D shallow water equations [Lecture → Ex. 11.6.2.13].

$$\mathbf{u}(x,t) = \begin{bmatrix} u_1(x,t) \\ u_2(x,t) \end{bmatrix} = \begin{bmatrix} \rho(x,t) \\ \rho(x,t)v(x,t) \end{bmatrix}, \qquad (x,t) \in \mathbb{R} \times \mathbb{R}^+ ,$$

$$\mathbf{F}(\mathbf{u}) = \begin{bmatrix} u_2 \\ u_2^2/u_1 + p(u_1) \end{bmatrix}, \qquad u \in U := \mathbb{R}^+ \times \mathbb{R} ,$$

because the pressure function is defined for positive arguments only.

SOLUTION of (11-15.c):

According to

Assumption [Lecture → Ass. 11.6.2.6]. Strict hyperbolicity

The Jacobian $\mathrm{D}\mathbf{F}(\mathbf{u})$ of the flux function $\mathbf{F} : \mathbb{R}^m \to \mathbb{R}^m$ must have m *distinct real eigenvalues* for any $\mathbf{u} \in U$.

we have to compute the Jacobian $\mathrm{D}\mathbf{F}(\mathbf{u})$ and verify that it has two distinct real eigenvalues for all $\mathbf{u} \in U$. By straightforward differentiation we get

$$\mathrm{D}\mathbf{F}(\mathbf{u}) = \begin{bmatrix} 0 & 1 \\ -\frac{u_2^2}{u_1^2} + p'(u_1) & 2\frac{u_2}{u_1} \end{bmatrix}, \quad \mathbf{u} := \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \in \mathbb{R}^+ \times \mathbb{R}, \tag{11.15.4}$$

with characteristic polynomial

$$\xi \in \mathbb{R} \mapsto \xi(\xi - 2\frac{u_2}{u_1}) + \frac{u_2^2}{u_1^2} - p'(u_1) .$$

Its root, the **eigenvalues** of $\mathrm{D}\mathbf{F}(\mathbf{u})$, is given by

$$\lambda_1(\mathbf{u}) = \frac{u_2}{u_1} - \sqrt{p'(u_1)} \quad , \quad \lambda_2(\mathbf{u}) = \frac{u_2}{u_1} + \sqrt{p'(u_1)} \tag{11.15.5}$$

Since we have assumed $p'(\rho) > 0$ for all $\rho > 0$ and admissible states are characterized by $u_1 > 0$, *cf.* (11.15.2a), the two eigenvalues are *real and different* for all admissible states.

HINT 1 for (11-15.d): What you found in Sub-problem (11-15.a) can give you key hint. ⌟

SOLUTION of (11-15.d):

In Sub-problem (11-15.a) we learned that the isentropic Euler equations are essentially "generalized" shallow water equations. For the latter [Lecture → Ex. 11.6.3.31] has presented a suitable Roe matrix $\mathbf{A}_R^{\mathrm{SW}}(\mathbf{v},\mathbf{w})$ based on the so-called Roe average of the involved states.

$$\mathbf{A}_R^{\mathrm{SW}}(\mathbf{v},\mathbf{w}) = \mathrm{D}\mathbf{F}\left(\begin{bmatrix}\overline{h}\\ \overline{h}\widehat{v}\end{bmatrix}\right), \quad \begin{aligned}\overline{h} &:= \tfrac{1}{2}(h_v + h_w)\,,\\ \widehat{v} &:= \frac{\sqrt{h_v}v_v + \sqrt{h_w}v_w}{\sqrt{h_v}+\sqrt{h_w}}\,,\end{aligned} \quad \mathbf{v} = \begin{bmatrix}h_v\\ h_v v_v\end{bmatrix},\ \mathbf{w} = \begin{bmatrix}h_w\\ h_w v_w\end{bmatrix}.$$

Hence, in this case the Roe matrix $\mathbf{A}_R^{\mathrm{SW}}$ is $\mathrm{D}\mathbf{F}^{\mathrm{SW}}$ evaluated at the **Roe average state** $\begin{bmatrix}\overline{h}\\ \overline{h}\widehat{v}\end{bmatrix}$. Unfortunately, this state is given in the "physical variables" h, v for the shallow water equations, whereas we consistently work with conserved "$\mathbf{u}$-variables". So we have to transform back and forth.

$$\mathbf{u} = \begin{bmatrix}u_1\\ u_2\end{bmatrix} := \begin{bmatrix}h\\ hv\end{bmatrix} \quad \Leftrightarrow \quad \begin{bmatrix}h\\ v\end{bmatrix} = \begin{bmatrix}u_1\\ u_2/u_1\end{bmatrix}, \quad u_1 = h > 0\,. \tag{11.15.7}$$

This transformation is the same for the shallow water equations as for the isentropic Euler equations.

Thus, the Roe average state $\overline{\mathbf{u}} = \overline{\mathbf{u}}(\mathbf{v},\mathbf{w}) \in U$, expressed in conserved variables is

$$\begin{aligned}\overline{u}_1 &= \tfrac{1}{2}(v_1 + w_1)\,,\\ \overline{u}_2 &= \overline{u}_1 \frac{\sqrt{v_1}v_2/v_1 + \sqrt{w_1}w_2/w_1}{\sqrt{v_1}+\sqrt{w_1}}\,,\end{aligned} \qquad \mathbf{v} = \begin{bmatrix}v_1\\ v_2\end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix}w_1\\ w_2\end{bmatrix}. \tag{11.15.8}$$

We can slightly rewrite this as

$$\overline{u}_1 = \tfrac{1}{2}(v_1 + w_1) \quad , \quad \overline{u}_2 = \overline{u}_1 \left(\frac{\sqrt{w_1}v_2 + \sqrt{v_1}w_2}{\sqrt{w_1}v_1 + \sqrt{v_1}w_1}\right). \tag{11.15.9}$$

Recall from (11.15.4) that

$$\mathrm{D}\mathbf{F}(\mathbf{u}) = \begin{bmatrix}0 & 1\\ -\frac{u_2^2}{u_1^2} + p'(u_1) & 2\frac{u_2}{u_1}\end{bmatrix}, \quad \mathbf{u} := \begin{bmatrix}u_1\\ u_2\end{bmatrix} \in \mathbb{R}^+ \times \mathbb{R}\,, \tag{11.15.4}$$

In light of

$$\frac{\overline{u}_2}{\overline{u}_1} = \frac{\sqrt{w_1}v_2 + \sqrt{v_1}w_2}{\sqrt{w_1}v_1 + \sqrt{v_1}w_1} \tag{11.15.10}$$

it is easy to see that the matrix from (11.15.6b) is just $\mathrm{D}\mathbf{F}$ evaluated at the Roe average state $\overline{\mathbf{u}}(\mathbf{v},\mathbf{w})$. This matrix will be a valid Roe matrix for the isentropic Euler equations, fulfilling all criteria from [Lecture → Def. 11.6.3.30]. The other matrices fail to satisfy at least one of the criteria.

SOLUTION of (11-15.e):

From [Lecture → § 11.6.3.33] we recall the definition of the HLLE numerical flux for a non-linear system of conservation laws:

$$\widehat{\mathbf{F}}_{\mathrm{HLLE}}(\mathbf{v},\mathbf{w}) = \begin{cases} \mathbf{F}(\mathbf{v}) & \text{, if } \quad \dot{s}^- > 0 \\ \mathbf{F}(\mathbf{u}^*) & \text{, if } \quad \dot{s}^- < 0 < \dot{s}^+ \\ \mathbf{F}(\mathbf{w}) & \text{, if } \quad \dot{s}^+ < 0 \end{cases} \begin{array}{l} \text{, } \dot{s}^- \text{ from [Lecture → Eq. (11.6.3.35)] ,} \\ \text{, } \mathbf{u}^* \text{ as in [Lecture → Eq. (11.6.3.37)] ,} \\ \text{, } \dot{s}^+ \text{ according to [Lecture → Eq. (11.6.3.35)] ,} \\ \qquad\qquad \text{[Lecture → Eq. (11.6.3.38)]} \end{array}$$

where

$$\mathbf{u}^* := \frac{\mathbf{F}(\mathbf{w}) - \mathbf{F}(\mathbf{v}) - \dot{s}^+\mathbf{w} + \dot{s}^-\mathbf{v}}{\dot{s}^- - \dot{s}^+} , \qquad \text{[Lecture → Eq. (11.6.3.37)]}$$

$$\begin{aligned} \dot{s}^- &:= \min\{\widehat{\lambda}_1, \lambda_1(\mathbf{v})\} , \\ \dot{s}^+ &:= \max\{\widehat{\lambda}_m, \lambda_m(\mathbf{w})\} , \end{aligned} \qquad \text{[Lecture → Eq. (11.6.3.35)]}$$

and $\widehat{\lambda}_i$, $i = 1, \ldots, m$, are the (sorted) eigenvalues of a Roe matrix $\mathbf{A}_R(\mathbf{v},\mathbf{w})$.

In Sub-problem (11-15.d) we found that for the isentropic Euler equations a Roe matrix is

$$\mathbf{A}_R(\mathbf{v},\mathbf{w}) = \mathrm{D}\mathbf{F}(\overline{u}(\mathbf{v},\mathbf{w})) , \quad \overline{\mathbf{u}}(\mathbf{v},\mathbf{w}) = \begin{bmatrix} \overline{u}_1 \\ \overline{u}_2 \end{bmatrix} = \begin{bmatrix} \frac{1}{2}(v_1 + w_1) \\ \overline{u}_1 \left(\frac{\sqrt{w_1}v_2 + \sqrt{v_1}w_2}{\sqrt{w_1}v_1 + \sqrt{v_1}w_1} \right) \end{bmatrix} . \tag{11.15.11}$$

From (11.15.5) recall the formula for the eigenvalues of $\mathrm{D}\mathbf{F}(\mathbf{u})$:

$$\lambda_1(\mathbf{u}) = \frac{u_2}{u_1} - \sqrt{p'(u_1)} \quad , \quad \lambda_2(\mathbf{u}) = \frac{u_2}{u_1} + \sqrt{p'(u_1)} . \tag{11.15.5}$$

Hence, the eigenvalues of the Roe matrix $\mathbf{A}_R(\mathbf{v},\mathbf{w})$ are

$$\lambda_1(\mathbf{u}) = \frac{\overline{u}_2}{\overline{u}_1} - \sqrt{p'(\overline{u}_1)} \quad , \quad \lambda_2(\mathbf{u}) = \frac{\overline{u}_2}{\overline{u}_1} + \sqrt{p'(\overline{u}_1)} . \tag{11.15.12}$$

All these formulas can be combined in a rather straightforward way.

C++ code 11.15.13: Code for `numfluxHLLEIseEul()` based on (11.15.5), (11.15.12), (11.15.11)

```
template <typename PFUNCTOR>
Eigen::Vector2d numfluxHLLEIseEul(PFUNCTOR &&p, PFUNCTOR &&pd,
                                  Eigen::Vector2d v, Eigen::Vector2d w) {
  assert((v[0] > 0.0) && "v-density must be positive!");
  assert((w[0] > 0.0) && "w-density must be positive!");
  Eigen::Vector2d nfHLLE{};
  // Compute the characteristic speeds for the two edge states
  const double lambda_1_v = v[1] / v[0] - std::sqrt(pd(v[0]));
  const double lambda_2_w = w[1] / w[0] + std::sqrt(pd(w[0]));
  // Compute Roe average according to (11.15.9)
  Eigen::Vector2d u_bar{};
  u_bar[0] = 0.5 * (v[0] + w[0]);  // Arithmetic average of first component
  const double sv0 = std::sqrt(v[0]);
```

```
15    const double sw0 = std::sqrt(w[0]);
16    const double v_bar = (sw0 * v[1] + sv0 * w[1]) / (sw0 * v[0] + sv0 * w[0]);
17    u_bar[1] = u_bar[0] * v_bar;
18    // Compute the eigenvalues of the Roe matrix A_R(v,w)
19    const double spd = std::sqrt(pd(u_bar[0]));
20    const double lambda_1 = v_bar - spd;
21    const double lambda_2 = v_bar + spd;
22    // HLLE edge shock speeds according ro [Lecture -> Eq. (11.6.3.35)]
23    const double s_minus = std::min(lambda_1, lambda_1_v);
24    const double s_plus = std::max(lambda_2, lambda_2_w);
25    // Flux function
26    auto F = [&p](Eigen::Vector2d u) -> Eigen::Vector2d {
27      assert((u[0] > 0.0) && "u-density must be positive!");
28      return {u[1], u[1] * u[1] / u[0] + p(u[0])};
29    };
30    const Eigen::Vector2d Fv{F(v)};
31    const Eigen::Vector2d Fw{F(w)};
32    if (s_minus > 0) {
33      nfHLLE = Fv;
34    } else if (s_plus < 0) {
35      nfHLLE = Fw;
36    } else {
37      const Eigen::Vector2d us =
38          (Fw - Fv - s_plus * w + s_minus * v) / (s_minus - s_plus);
39      nfHLLE = F(us);
40    }
41    return nfHLLE;
42  }
```

SOLUTION of (11-15.f):

- The number of vector-valued states is equal to the number of columns of the matrix `mu`.
- Operations on scalar states have to be transformed to operations on the columns of a matrix.
- Operations on the state vector become operations on matrices.

C++ code 11.15.15: Code for `slopelimfluxdiff_sys()`, the system counterpart of `slopelimfluxdiff_scalar()`

```
template <typename FFUNCTOR, typename SLOPEFUNCTOR>
Eigen::MatrixXd slopelimfluxdiff_sys(const Eigen::MatrixXd &mu, FFUNCTOR &&F,
                                     SLOPEFUNCTOR &&slopes) {
  Eigen::Index n = mu.cols();  // Number of active dual grid cells
  // All slopes for all components of cell states
  Eigen::MatrixXd sigma = Eigen::MatrixXd::Zero(mu.rows(), mu.cols());
  Eigen::MatrixXd fd = Eigen::MatrixXd::Zero(mu.rows(), mu.cols());

  // Computation of slopes σ_j, uses μ_0 = μ_1,
  // μ_{N+1} = μ_N, which amounts to constant extension of states
  // beyond domain of influence [a,b] of non-constant intial data.  Same
  // technique has been applied in [Lecture → Code 11.3.2.9]
  sigma.col(0) = slopes(mu.col(0), mu.col(0), mu.col(1));
  for (Eigen::Index j = 1; j < n - 1; ++j) {
    sigma.col(j) = slopes(mu.col(j - 1), mu.col(j), mu.col(j + 1));
  }
  sigma.col(n - 1) = slopes(mu.col(n - 2), mu.col(n - 1), mu.col(n - 1));

  // Compute linear reconstruction at endpoints of dual cells [Lecture →
     Eq. (11.5.1.2)]
  Eigen::MatrixXd nup = mu + 0.5 * sigma;
  Eigen::MatrixXd num = mu - 0.5 * sigma;

  // As in [Lecture → Code 11.3.2.10]:  constant continuation of data
     outside
  // [a,b]!
  fd.col(0) = F(nup.col(0), num.col(1)) - F(mu.col(0), num.col(0));
  for (Eigen::Index j = 1; j < n - 1; ++j) {
    // see [Lecture → Eq. (11.3.2.8)]
    fd.col(j) = F(nup.col(j), num.col(j + 1)) - F(nup.col(j - 1), num.col(j));
  }
  fd.col(n - 1) =
      F(nup.col(n - 1), mu.col(n - 1)) - F(nup.col(n - 2), num.col(n - 1));
  return fd;
}
```

SOLUTION of (11-15.g):

The first task is to implement the formula for the slope σ_j from

Definition [Lecture → Def. 11.5.2.9]. Minmod reconstruction

The minmod reconstruction R_{mm} is a piecewise linear reconstruction (→ [Lecture → Def. 11.5.1.3]) defined by

$$(R_{mm}\vec{\mu})(x) = \mu_j + \sigma_j(x - x_j) \quad \text{for} \quad x_{j-\frac{1}{2}} < x < x_{j+\frac{1}{2}}\,, \quad j \in \mathbb{Z}\,,$$

$$\sigma_j := \operatorname{minmod}\left(\frac{\mu_{j+1}-\mu_j}{x_{j+1}-x_j}, \frac{\mu_j-\mu_{j-1}}{x_j-x_{j-1}}\right),$$

$$\operatorname{minmod}(v,w) := \begin{cases} v & , vw > 0, |v| < |w|\,, \\ w & , vw > 0, |w| < |v|\,, \\ 0 & , vw \le 0\,. \end{cases}$$

component-wise for the three state vectors passed to the lambda function.

The second task is a straightforward implementation of

$$\vec{\kappa} := \vec{\mu}^{(k)} + \tau\mathcal{L}_h(\vec{\mu}^{(k)})\,,$$
$$\vec{\mu}^{(k+1)} := \vec{\mu}^{(k)} + \tfrac{1}{2}\tau\Big(\mathcal{L}_h(\vec{\mu}^{(k)}) + \mathcal{L}_h(\vec{\kappa})\Big)\,,$$

[Lecture → Eq. (11.5.3.2)]

where the evaluation of $\mathcal{L}_h$ is supplied by `slopelimfluxdiff_sys()`.

C++ code 11.15.17: Code for `musclIseEul()`, based omn `slopelimfluxdiff_sys()`

```
template <typename PFUNCTOR, typename RECORDER = std::function<
                                 void(double, const Eigen::MatrixXd &)> >
Eigen::MatrixXd musclIseEul(
    double a, double b, double T, const Eigen::MatrixXd &mu0_mat, PFUNCTOR &&p,
    PFUNCTOR &&dp,
    RECORDER &&rec = [](double /*t*/,
                        const Eigen::MatrixXd & /*mu_mat*/) -> void {}) {
  // Matrix whose columns store the states
  Eigen::MatrixXd mu(mu0_mat.rows(), mu0_mat.cols());
  mu << mu0_mat;
  double h = (b - a) / mu0_mat.cols();
  double t = 0;
  rec(t, mu);
  // Main timestepping loop
  while (t < T) {
    double smax = 0;
    // Adaptive timestep size selection according to (1.2.16)
    for (int j = 0; j < mu.cols(); ++j) {
      double lambda1 = mu(1, j) / mu(0, j) - std::sqrt(dp(mu(0, j)));
      double lambda2 = mu(1, j) / mu(0, j) + std::sqrt(dp(mu(0, j)));
      smax = std::max({smax, std::abs(lambda1), std::abs(lambda2)});
    }
    double tau = std::min(h / (2. * smax), T - t);
```

```
    // Lambda function providing HLLE numerical flux
    std::function<Eigen::Vector2d(Eigen::Vector2d, Eigen::Vector2d)> F =
        [p, dp](Eigen::Vector2d u, Eigen::Vector2d v) -> Eigen::Vector2d {
      return numfluxHLLEIseEul(p, dp, u, v);
    };
    // Lambda function for local minmod-limited slope recostruction
    std::function<Eigen::Vector2d(Eigen::Vector2d, Eigen::Vector2d,
                                  Eigen::Vector2d)>
        slopes = [](Eigen::Vector2d u1, Eigen::Vector2d u2,
                    Eigen::Vector2d u3) -> Eigen::Vector2d {
      Eigen::Vector2d out;
      for (int j = 0; j < 2; ++j) {
        if (u1(j) > u2(j) && u2(j) > u3(j))
          out(j) = std::max(u2(j) - u1(j), u3(j) - u2(j));
        else if (u1(j) < u2(j) && u2(j) < u3(j))
          out(j) = std::min(u2(j) - u1(j), u3(j) - u2(j));
        else
          out(j) = 0;
      }
      return out;
    };
    // Second-order explicit Heun timestepping based on
       slopelimfluxdiff_sys()
    Eigen::MatrixXd Lmu = -1. / h * slopelimfluxdiff_sys(mu, F, slopes);
    Eigen::MatrixXd k = mu + tau * Lmu;
    Eigen::MatrixXd Lk = -1. / h * slopelimfluxdiff_sys(k, F, slopes);
    mu = mu + .5 * tau * (Lmu + Lk);
    t += tau;
    rec(t, mu);
  }
  return mu;
}
```

SOLUTION of (11-2.a):

Definition [Lecture → Def. 11.3.3.5]. Consistent numerical flux function

A numerical flux function $F : \mathbb{R}^{m_l+m_r} \mapsto \mathbb{R}$ is **consistent** with the flux function $f : \mathbb{R} \mapsto \mathbb{R}$, if

$$F(u,\ldots,u) = f(u) \quad \forall u \in \mathbb{R} .$$

We have

$$F_{\mathrm{EO}}(v,v) = \frac{1}{2}(f(v)+f(v)) - \frac{1}{2}\int_v^v |f'(\xi)|\,\mathrm{d}\xi = f(v) ,$$

so the flux is consistent.

HINT 1 for (11-2.b): Use the criterion of [Lecture → Cor. 11.3.5.6].

SOLUTION of (11-2.b):

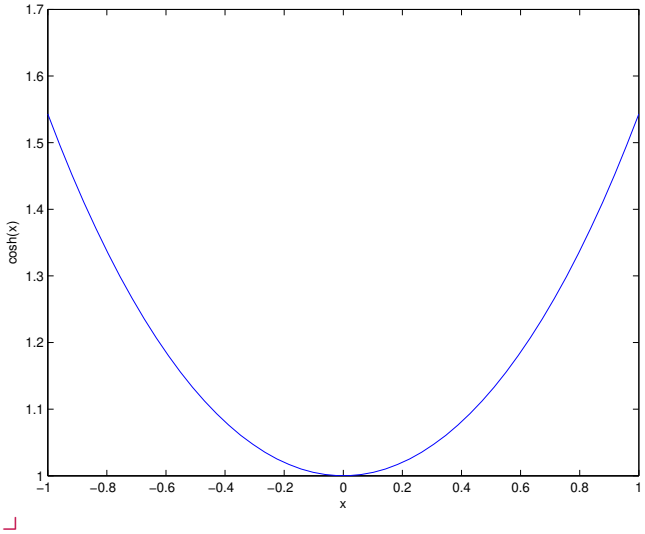
By the fundamental theorem of calculus we find

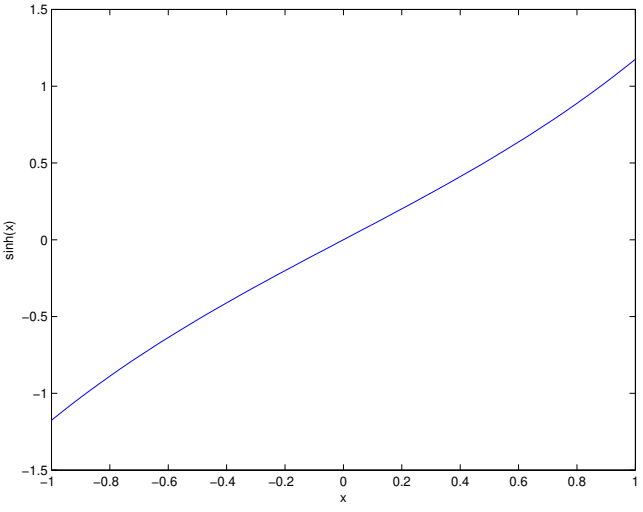
$$\frac{\partial F_{\mathrm{EO}}}{\partial v}(v,w) = \tfrac{1}{2}f'(v) + \tfrac{1}{2}|f'(v)| = \begin{cases} f'(v) & \text{, if } f'(v) \geq 0\,, \\ 0 & \text{, if } f'(v) < 0 \end{cases} \geq 0\,,$$

$$\frac{\partial F_{\mathrm{EO}}}{\partial w}(v,w) = \tfrac{1}{2}f'(w) - \tfrac{1}{2}|f'(w)| = \begin{cases} 0 & \text{, if } f'(w) \geq 0\,, \\ f'(w) & \text{, if } f'(w) < 0 \end{cases} \leq 0\,.$$

Hence, from [Lecture → Cor. 11.3.5.6] we conclude that F_{EO} is monotone.

HINT 1 for (11-2.d): The flux function $f(u) = \cosh(u)$ and its dervative $\sinh$ are plotted in Fig. 182 and Fig. 183. What is the “maximal speed of propagation”?





∟

SOLUTION of (11-2.d):

The characteristic emanating from $]-1,0[$ can move to the left with maximal speed $\sinh(-1)$, and the characteristic emanating from $]1,0[$ can move to the right with maximal speed $\sinh(1)$. Thus, the support of $u(\cdot,t)$ will at least be contained in the interval

$$[0+t\sinh(-1),1+t\sinh(1)] = [-t\sinh(1),1+t\sinh(1)] \ .$$

The value of $\sinh(1)$ is about 1.1752.

SOLUTION of (11-2.e):

To simplify the integration, we can split the integral in two. One in the interval $[\min(v,0), \min(w,0)]$ and one in the interval $[\max(v,0), \max(w,0)]$. The first is guaranteed to be contained in $(-\infty, 0]$, where $|f'(x)| = -\sinh(x)$ and the second is contained in $[0,\infty)$, where $|f'(x)| = \sinh(x)$. The integration can then be performed analytically:

$$F_{\text{EO}}(v,w) = \tfrac{1}{2}(f(v)+f(w)) - \frac{1}{2}\cdot\begin{cases} f(w)-f(v) & \text{, if } v,w \geq 0\,, \\ f(v)+f(w)-2f(0) & \text{, if } v<0,\ w\geq 0\,, \\ 2f(0)-f(v)-f(w) & \text{, if } w<0,\ v\geq 0\,, \\ f(v)-f(w) & \text{, if } v,w<0\,. \end{cases}$$

$$= \begin{cases} f(v) & \text{, if } v,w \geq 0\,, \\ f(0) & \text{, if } v<0,\ w\geq 0\,, \\ f(v)+f(w)-f(0) & \text{, if } w<0,\ v\geq 0\,, \\ f(w) & \text{, if } v,w<0\,. \end{cases}$$

The following is a rather obfuscated implementation of this formula. Can you see, why it works?

C++11 code 11.2.3: Sub-problem (11-2.e): function `EngquistOsherNumFlux()` → GitLab

```
2 double EngquistOsherNumFlux(double v, double w) {
3   double result;
4   auto Fv = [](double v) { return 0 <= v ? std::cosh(v) - 0.5 : 0.5; };
5   result = Fv(v) + Fv(-w);
6   return result;
7 }
```

SOLUTION of (11-2.g):

By the comparison principle from [Lecture → Thm. 11.2.7.1] we have

$$A \le u(x,t) \le B \quad \forall (x,t) \in \mathbb{R} \times \mathbb{R}_0^+ \ .$$

The maximal analytic domain of depdence follows from the formulas in [Lecture → § 11.4.2.4]:

$$D^-(\overline{x},\overline{t}) := \{(x,t) \in \mathbb{R} \times [0,\overline{t}] : \dot{s}_{\min}(\overline{t}-t) \le x - \overline{x} \le \dot{s}_{\max}(\overline{t}-t)\} \ . \tag{11.2.4}$$

with maximal speeds of propagation in either direction

$$\dot{s}_{\min} := \min\{f'(\xi) : \inf_{x\in\mathbb{R}} u_0(x) \le \xi \le \sup_{x\in\mathbb{R}} u_0(x)\} \ , \tag{11.2.5}$$

$$\dot{s}_{\max} := \max\{f'(\xi) : \inf_{x\in\mathbb{R}} u_0(x) \le \xi \le \sup_{x\in\mathbb{R}} u_0(x)\} \ . \tag{11.2.6}$$

Since $f'(u) = \sinh(u)$, which is increasing, we find:

$$\dot{s}_{\min} = \sinh(A) \quad \text{and} \quad \dot{s}_{\max} = \sinh(B) \ . \tag{11.2.7}$$

Hence, the maximal speed of propagation is

$$\dot{s} = \max\{|\sinh A|, |\sinh B|\} \ .$$

The Engquist-Osher numerical flux is a 2-point numerical flux and the explicit Euler method is a 1-stage explicit Runge-Kutta single-step method. Thus the resulting space-time stencil has width $m = 1$, see [Lecture → § 11.4.2.1] and [Lecture → Fig. 552]. According to [Lecture → Def. 11.4.2.5] the numerical domain of dependence is therefore

$$D_h^-(x_j,t_k) := \{(x_n,t_l) \in \mathbb{R} \times [0,t_k] : j-(k-l) \le n \le j+m(k-l)\} \ , \quad x_j = hj \ , \quad t_k = \tau k \ . \tag{11.2.8}$$

Thus we have by [Lecture → Eq. (11.4.2.12)]

$$\tau \le \frac{h}{\dot{s}} \quad \Longrightarrow \quad D^-(x_j,t_k) \subset \operatorname{convex}(D_h^-(x_j,t_k)) \ .$$

This means that the CFL-condition implies the timestep constraint

$$\tau \le \frac{h}{\max\{|\sinh A|, |\sinh B|\}} \ .$$

SOLUTION of (11-2.h):

We have to implement the formula [Lecture → Eq. (11.4.1.10)]

$$\mu^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\left(F_{\mathrm{EO}}(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F_{\mathrm{EO}}(\mu_{j-1}^{(k)}, \mu_j^{(k)})\right) ,$$

plus truncation of the computational domain assuming constant entries of the vector of cell values for indices outside of a finite range, see [Lecture → Code 11.3.2.9].

C++11 code 11.2.9: Sub-problem (11-2.h): function `solveCP()` ⇝ GitLab

```
2  Eigen::VectorXd solveCP(double a, double b, Eigen::VectorXd u0, double T) {
3    // Find the maximal speed of propagation
4    double A = u0.minCoeff();
5    double B = u0.maxCoeff();
6    double K = std::max(std::abs(std::sinh(A)), std::abs(std::sinh(B)));
7    // Set uniform timestep according to CFL condition
8    int N = u0.size();
9    double h = (b - a) / N;
10   double tau_max = h / K;
11   double timesteps = std::ceil(T / tau_max);
12   double tau = T / timesteps;
13
14   // Main timestepping loop
15   Eigen::VectorXd mu(N);
16   for (int i = 0; i < timesteps; ++i) {
17     mu.swap(u0);
18     double F_minus;
19     double F_plus = EngquistOsherNumFlux(mu(0), mu(0));
20     for (int j = 0; j < N - 1; ++j) {
21       F_minus = F_plus;
22       F_plus = EngquistOsherNumFlux(mu(j), mu(j + 1));
23       u0(j) = mu(j) - tau / h * (F_plus - F_minus);
24     }
25     F_minus = F_plus;
26     F_plus = EngquistOsherNumFlux(mu(N - 1), mu(N - 1));
27     u0(N - 1) = mu(N - 1) - tau / h * (F_plus - F_minus);
28   }
29
30   return u0;
31 }
```

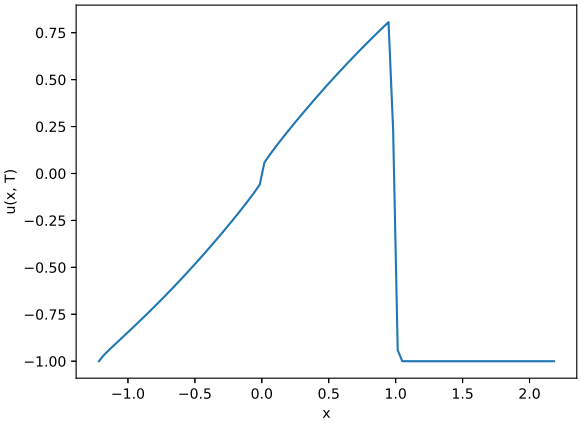
SOLUTION of (11-2.i):

C++11 code 11.2.10: Sub-problem (11-2.h): function `solveCP()` ⇝ GitLab

```cpp
const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
                                       Eigen::DontAlignCols, ", ", "\n");

double u0(double x) { return (0.0 < x && x <= 1.0) ? 1.0 : -1.0; }

int main() {
  unsigned int N = 100;
  double a = -1.2;
  double b = 2.2;
  double T = 1.0;
  double h = (b - a) / N;

  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, a - 0.5 * h, b - 0.5 * h);
  Eigen::VectorXd uinitial = x.unaryExpr(std::ref(u0));
  Eigen::VectorXd ufinal =
      EngquistOsherNumericalFlux::solveCP(a, b, uinitial, T);

  std::ofstream file;
  file.open("ufinal.csv");
  file << x.transpose().format(CSVFormat) << std::endl;
  file << ufinal.transpose().format(CSVFormat) << std::endl;
  file.close();

  std::cout << "Generated ufinal.csv" << std::endl;
  systemcall::execute("python3 scripts/plot_solution.py ufinal.csv ufinal.eps");

  return 0;
}
```

PYTHON script 11.2.11: Sub-problem (11-2.h): PYTHON script for plotting solution ⇝ GitLab

```python
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel
from numpy import genfromtxt
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
x = data[0]
y = data[1]

figure()
plot(x, y)
xlabel('x')
ylabel('u(x,1)')
savefig(output_file)

print('Generated ' + output_file)
```



◁ Solution of Riemann problems at $T = 1$.

Note the remnants of a spurious expansion shock at state $u = 0$ for which $f'(u) = 0$ (“entry glitch”).
See also [Lecture → Exp. 11.3.4.36] for a similar observation.

SOLUTION of (11-3.a):

Our considerations are based on [Lecture → Section 11.2.6], where we studied the traffic flow equations as a specimen of a scalar conservation law with concave flux.

We see that $f(u) = \sin \pi u$, which is concave on $[0,1]$. Thus, the condition (i) should lead to a shock, and condition (ii) should lead to a rarefaction wave. The situation is the same as with the traffic flow conservation law discussed in [Lecture → Section 11.2.5].

For (i), we can apply [Lecture → Lemma 11.2.5.4] and use the Rankine-Hugoniot jump condition to determine the shock speed.

$$\dot{s} = \frac{\sin 0 - \sin \pi}{0 - 1} = 0,$$

and so we find that the shock doesn't move (stationary shock).

$$u(x,t) = \begin{cases} 0, & x < 0, \\ 1, & x > 0. \end{cases}$$

For the rarefaction wave in the case of (ii), we invoke [Lecture → Lemma 11.2.5.10] with $g(x) = (f')^{-1}(x) = \frac{1}{\pi} \arccos(x/\pi)$, hence we obtain the similarity solution

$$u(x,t) = \begin{cases} 1 & \text{for} \quad x < -\pi t\,, \\ \frac{1}{\pi} \arccos\left(\frac{x}{\pi t}\right) & \text{for} \quad -\pi < x/t < \pi\,, \\ 0 & \text{for} \quad x > \pi t\,. \end{cases}$$

HINT 1 for (11-3.j): Recall the finite volume interpretation of the coefficients of the vector argument `mu`.

⌟

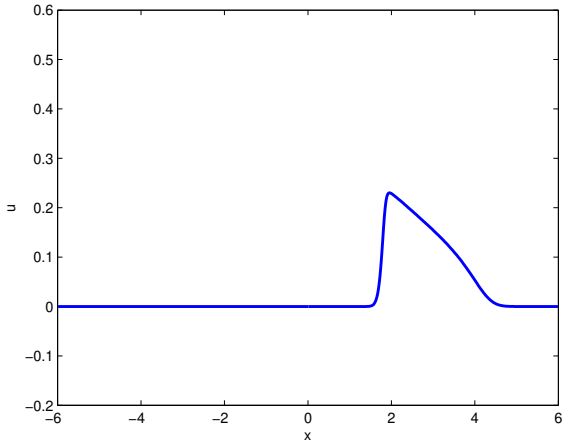
SOLUTION of (11-3.j):

The modification of the right-hand-side of the method-of-lines ODE just amounts to subtracting c times the vector $\vec{\mu}$ (suitably truncated).

C++ code 11.3.13: Sub-problem (11-3.i): function `sineClawReactionRhs()` ↪ GitLab

```
2 Eigen::VectorXd sineClawReactionRhs(const Eigen::VectorXd &mu, double c) {
3   Eigen::VectorXd rhs(mu.size());
4   rhs = sineClawRhs(mu) - c * mu;
5   return rhs;
6 }
```

SOLUTION of (11-3.k):



◁ Solution of the Cauchy problem for (11.3.12) at $T = 1$

The extra reaction term causes substantial damping of the solution.
To understand why consider the Cauchy problem for the pure reaction equation $\frac{\partial u}{\partial t} = -cu$, whose solution is $u(x,t) = \exp(-ct)u_0(x)$.

SOLUTION of (11-3.b):

By [Lecture → Thm. 11.2.7.1] we know that $0 \leq u(x,t) \leq 1$ for all (x,t). Furthermore, for the flux function $f(u) = \sin(\pi u)$,

$$-\pi \leq f'(u) \leq \pi\,, \quad 0 \leq u \leq 1\,.$$

Hence the maximal speed of propagation is $-\pi$ to the left, and π to the right. Hence, by [Lecture → Cor. 11.2.7.5], the maximal support is

$$\{(x,t) : -\pi t \leq x \leq 1 + \pi t,\ t \geq 0\}\,.$$

HINT 1 for (11-3.c): Note that the flux function $f(u) = \sin(\pi u)$ is concave on $[0, 1]$. ⌟

SOLUTION of (11-3.c):

Note that the discussion in the beginning of [Lecture → Section 11.3.4.4] considers only convex flux functions. What you can use is the general formula [Lecture → Eq. (11.3.4.33)]. Alternatively, you may just flip the conditions $v < w$ and $v > w$ in the formulas for convex f.

Under the assumption that $0 \le v, w \le 1$, we end up with the following formula for the intermediate state:

$$u^{\downarrow}(v,w) = \begin{cases} v & \text{, if } v < w \text{ and } \dot{s} > 0, \\ v & \text{, if } v \ge w \text{ and } \cos \pi v > 0, \\ w & \text{, if } v < w \text{ and } \dot{s} < 0, \\ w & \text{, if } v \ge w \text{ and } \cos \pi w < 0, \\ \frac{1}{2} & \text{, if } v \ge w \text{ and } \cos \pi v \le 0 \le \cos \pi w\,, \end{cases} \qquad \dot{s} := \frac{\sin(\pi w) = \sin(\pi v)}{w - v}\,.$$

No simplifications are possible, as all conditions can occur.

C++11 code 11.3.3: Sub-problem (11-3.c): function `sineGodFlux()` ↬ GitLab

```
2  constexpr double PI = 3.14159265358979323846;
3
4  double f(double x) { return std::sin(PI * x); }
5
6  double sineGodFlux(double v, double w) {
7    double result;
8    // Rankine-Hugoniot speed.  No safeguards against cancellation are
       taken.  For v
9    // close to w the result will severely be affected by amplified
       round-off
10   // error.  However, this does not do any harm, because we are
       interested in the
11   // the sign alone.
12   double s = (v != w) ? (f(w) - f(v)) / (w - v) : PI * std::cos(PI * v);
13   // Treat all different cases separately
14   if (((v < w) && (s > 0.0)) || ((v >= w) && (std::cos(PI * v) > 0.0))) {
15     result = f(v);
16   } else if (((v < w) && (s < 0.0)) || ((v >= w) && (std::cos(PI * w) < 0.0))) {
17     result = f(w);
18   } else
19     result = f(0.5);
20
21   return result;
22 }
```

SOLUTION of (11-3.d):

C++11 code 11.3.4: Sub-problem (11-3.d): function `sineClawRhs()` ⇝ GitLab

```
2  Eigen::VectorXd sineClawRhs(const Eigen::VectorXd &mu) {
3    int N = mu.size();
4    double h = 12.0 / N;
5    Eigen::VectorXd result(N);
6
7    double F_minus;
8    double F_plus = sineGodFlux(0.0, mu(0));
9    for (int j = 0; j < N - 1; ++j) {
10     F_minus = F_plus;
11     F_plus = sineGodFlux(mu(j), mu(j + 1));
12     result(j) = -1.0 / h * (F_plus - F_minus);
13   }
14   F_minus = F_plus;
15   F_plus = sineGodFlux(mu(N - 1), 0.0);
16   result(N - 1) = -1.0 / h * (F_plus - F_minus);
17
18   return result;
19 }
```

SOLUTION of (11-3.e):

The fully discrete evolution operator based of a 2-point flux and the explicity trapezoidal rule is given in [Lecture → Eq. (11.4.1.11)]:

$$\vec{\kappa} = \boldsymbol{\mu}^{(k)} + \tau \boldsymbol{\mathcal{L}}_h(\vec{\boldsymbol{\mu}}^{(k)}) \quad , \quad \vec{\boldsymbol{\mu}}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \frac{\tau}{2}\Big(\boldsymbol{\mathcal{L}}_h(\boldsymbol{\mu}^{(k)}) + \boldsymbol{\mathcal{L}}_h(\vec{\kappa})\Big) .$$

$$\begin{cases} \kappa_j := (\vec{\kappa})_j = \mu_j - \dfrac{\tau}{h}\big(F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big) , \\ \big(\boldsymbol{\mathcal{H}}_h(\vec{\boldsymbol{\mu}})\big)_j = \mu_j - \dfrac{\tau}{2h}\big(F(\kappa_j,\kappa_{j+1}) - F(\kappa_{j-1},\kappa_j) + F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big) . \end{cases} \tag{11.3.6}$$

Given this formula the implementation should be straightforward. Take care to take into account the truncation of the state vector, see [Lecture → Code 11.3.2.9].

C++11 code 11.3.7: Sub-problem (11-3.e): function `explTrpzTimestepping()` ↪ GitLab

```
2  template <typename RHSFUNCTOR>
3  Eigen::VectorXd explTrpzTimestepping(RHSFUNCTOR &&g, const Eigen::VectorXd &y0,
4                                       double T, unsigned int M) {
5    double tau = T / M;
6    Eigen::VectorXd y = y0;
7
8    for (int j = 0; j < M; ++j) {
9      Eigen::VectorXd k0 = g(y);
10     Eigen::VectorXd k1 = g(y + tau * k0);
11     y = y + tau * 0.5 * (k0 + k1);
12   }
13
14   return y;
15 }
```

SOLUTION of (11-3.f):

C++11 code 11.3.8: Sub-problem (11-3.f): function `solveSineConsLaw()` ⇝ GitLab

```
2  template <typename RHSFUNCTOR>
3  Eigen::VectorXd solveSineConsLaw(RHSFUNCTOR &&g, unsigned int N,
4                                   unsigned int M) {
5    const double h = 12.0 / N;
6    const double T = 1.0;
7    Eigen::VectorXd result(N);
8
9    Eigen::VectorXd x =
10       Eigen::VectorXd::LinSpaced(N, -6.0 + 0.5 * h, 6.0 - 0.5 * h);
11   Eigen::VectorXd u0 =
12       x.unaryExpr([](double x) { return 0.0 <= x && x < 1.0 ? 1.0 : 0.0; });
13   result = explTrpzTimestepping(std::forward<RHSFUNCTOR>(g), u0, T, M);
14
15   return result;
16 }
```

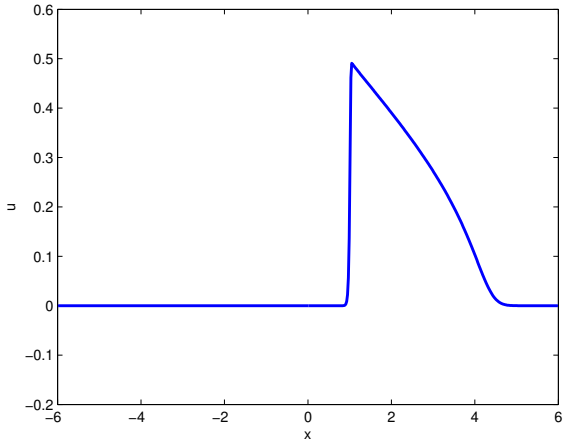
SOLUTION of (11-3.g):

C++11 code 11.3.9: Sub-problem (11-3.g): excerpt from `main()` with the code requested in the sub-problem ⇝ GitLab

```
  unsigned int N = 600;
  unsigned int M = 200;
  double h = 12.0 / N;
  Eigen::VectorXd x =
      Eigen::VectorXd::LinSpaced(N, -6.0 + 0.5 * h, 6.0 - 0.5 * h);

  std::ofstream file;

  // without reaction term
  Eigen::VectorXd ufinal = solveSineConsLaw(&sineClawRhs, N, M);
  file.open("ufinal.csv");
  file << x.transpose().format(CSVFormat) << std::endl;
  file << ufinal.transpose().format(CSVFormat) << std::endl;
  file.close();

  std::cout << "Generated ufinal.csv" << std::endl;
  systemcall::execute("python3 scripts/plot.py ufinal.csv ufinal.eps");
```

PYTHON script 11.3.10: PYTHON script for visualization ⇝ GitLab

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel
from numpy import genfromtxt
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
x = data[0]
ufinal = data[1]

figure()
plot(x, ufinal, '-')
xlabel('x')
ylabel('u(x, T)')
savefig(output_file)

print('Generated ' + output_file)
```



The solution at final time $T = 1$.

SOLUTION of (11-3.h):

Numerically, the solution $\mu_j(t_k)$ depends on $\mu_{j-2}(t_{k-1})$ through $\mu_{j+2}(t_{k-1})$ (because of the double layered timestepping method). In the parlance of [Lecture → § 11.4.2.1] we deal with a stencil of with $m = 2$ due to the "stencil spread" induced by the 2-stage explicit Runge-Kutta timestepping employed. The numerical domain of dependence is thus a wedge with slope $2h/\tau$.

Analytically, the maximal speed of a characteristic is π.

We then need

$$\frac{2h}{\tau} \geq \pi \Longrightarrow \tau \leq \frac{2h}{\pi}.$$

HINT 1 for (11-3.i): You have to implement a bisection type approach based on monitoring a possible blow-up of solutions. To detect a blow-up, watch the maximal (in modulus) nodal value of the numerical solution at final time. Classify the run as affected by blow-up, in case the maximum is ≥ 2, ⌟

SOLUTION of (11-3.i):

One idea is to find the smallest timestep that keeps the values of the final solution bounded by within $[0,2]$ (try out a few experiments with many timesteps to see why). The algorithm works thus: we start with two values M_L and M_R which bound the critical value of M (thus, M_R is too many timesteps and M_L too few). We then investigate the midpoint $M_{\text{mid}} = (M_L + M_R)/2$. If this value of M causes no (or negligible) blow-up, we know it is safe, and otherwise it is not. In the first case, set $M_R \leftarrow M_{\text{mid}}$, and in the second, set $M_L \leftarrow M_{\text{mid}}$. Iterate until $M_L = M_R$.

C++ code 11.3.11: Sub-problem (11-3.i): function `findTimestep()` ⇝ GitLab

```
2  bool blowup(const Eigen::VectorXd &mu) {
3    return mu.minCoeff() < 0.0 || mu.maxCoeff() > 2.0;
4  }
5
6  unsigned int findTimesteps() {
7    const unsigned int N = 600;
8    unsigned int ML = 100;
9    unsigned int MR = 200;
10
11   for (int i = 0; i < 100; ++i) {
12     if (ML == MR) {
13       return MR;
14     }
15     unsigned int M = (unsigned int)(0.5 * (ML + MR));
16     if (blowup(solveSineConsLaw(&sineClawRhs, N, M))) {
17       ML = M + 1;
18     } else {
19       MR = M;
20     }
21   }
22
23   return MR;
24 }
```

SOLUTION of (11-4.a):

The flow of information induced by (11.4.1) is described by the stencil of [Lecture → Fig. 552]. We face the situation of [Lecture → Fig. 556] and a stencil of width $m = 1$.

The maximal speed of propagation is the maximal modulus of $f'(u)$ taken over all possible states:

$$\dot{s} := \max\{f'(u) : \min_x u_0(x) \leq u \leq \max_{x\in\mathbb{R}} u_0(x)\} .$$

Thus, [Lecture → Eq. (11.4.2.12)] tell us that we have to demand the CFL-condition

$$\frac{\tau}{h} \leq \frac{1}{\dot{s}} .$$

SOLUTION of (11-4.j):

We implement the standard formula [Lecture → Eq. (11.4.1.10)]

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\left(F(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)}, \mu_j^{(k)})\right) , \tag{11.4.16}$$

for the numerical flux function F found in Sub-problem (11-4.i).

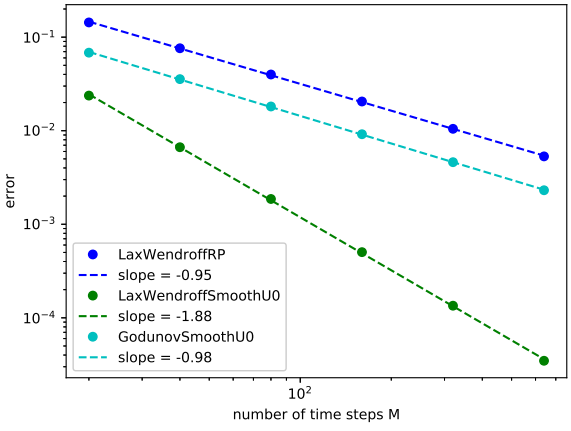
C++ code 11.4.17: Sub-problem (11-4.j): function `solveGodunov()` ➪ GITHUB

```
2  Eigen::VectorXd solveGodunov(const Eigen::VectorXd &u0, double T,
3                               unsigned int M) {
4    double tau = T / M;
5    double h = Constant::e * tau;
6    unsigned int N = u0.size();
7    Eigen::VectorXd mu = u0;
8
9    for (int i = 0; i < M; ++i) {
10     for (int j = N - 1; 0 < j; --j) {
11       mu(j) = mu(j) - tau / h * (std::exp(mu(j)) - std::exp(mu(j - 1)));
12     }
13   }
14   return mu;
15 }
```

SOLUTION of (11-4.k):

C++ code 11.4.18: Sub-problem (11-4.k): function `numexpGodunovSmoothU0()`
↪ GITHUB

```
Eigen::VectorXd numexpGodunovSmoothU0(const Eigen::VectorXi &M) {
  const double T = 1.0;
  const int M_size = M.size();
  Eigen::VectorXd error(M_size);

  for (int i = 0; i < M_size; ++i) {
    Eigen::VectorXd x = getXValues(T, M(i));
    Eigen::VectorXd u0 = x.unaryExpr(&smoothU0);
    Eigen::VectorXd uT = solveGodunov(u0, T, M(i));

    double tau = T / M(i);
    double h = Constant::e * tau;
    error(i) = h * (referenceSolution(x) - uT).lpNorm<1>();
  }
  return error;
}
```



◁ The double logarithmic error plot indicates algebraic convergence with rate 1 for the lowest-order finite volume method based on the monotone Godunov flux.

This matches the statement made in [Lecture → Section 11.4.4] that conservative finite volume methods based on monotone fluxes can achieve only first order convergence (“order barrier”).

Conversely, the Lax-Wendroff scheme achieves *second-order algebraic convergence*.

SOLUTION of (11-4.b):

For this case we have $f(u) = vu$, giving $f'(u) = v$ and $f'(u)^2 = v^2$:

$$\mu_j^{(k)} = \mu_j^{(k-1)} + \frac{\gamma v}{2}\left(\mu_{j+1}^{(k-1)} - \mu_{j-1}^{(k-1)}\right) + \frac{\gamma^2 v^2}{2}\left(\mu_{j+1}^{(k-1)} - 2\mu_j^{(k-1)} + \mu_{j-1}^{(k-1)}\right) . \tag{11.4.3}$$

Writing the recursion this way reveals that the Lax-Wendroff scheme pursues an idea similar to that motivating the Lax-Friedrichs/Rusanov flux [Lecture → § 11.3.4.7]; one adds artificial diffusionfor the sake of stabilization. Here the term $\mu_{j+1}^{(k-1)} - 2\mu_j^{(k-1)} + \mu_{j-1}^{(k-1)}$ is an h-weighted discrete diffusion operator.

SOLUTION of (11-4.c):

We can rewrite (11.4.3) as

$$\vec{\mu}^{(k)} = \boldsymbol{\mathcal{H}}(\vec{\mu}^{(k-1)}) \,,$$

$$\text{with} \qquad (\boldsymbol{\mathcal{H}}(\vec{\mu}))_j := \mu_j + \frac{\gamma v}{2}(\mu_{j+1} - \mu_{j-1}) + \frac{\gamma^2 v^2}{2}(\mu_{j+1} - 2\mu_j + \mu_{j-1}) \,.$$

Then

$$\begin{aligned} \boldsymbol{\mathcal{H}}(\vec{\boldsymbol{\psi}}_\xi) &= \left\{1 - \frac{\gamma v}{2}\left(e^{\imath\xi} - e^{-\imath\xi}\right) + \frac{\gamma^2 v^2}{2}\left(e^{\imath\xi} - 2 + e^{-\imath\xi}\right)\right\} \cdot \vec{\boldsymbol{\psi}}_\xi \\ &= \underbrace{\left\{1 - \gamma v \imath \sin(\xi) + (\gamma v)^2(\cos(\xi) - 1)\right\}}_{\hat{=}\text{ “eigenvalue”}} \cdot \vec{\boldsymbol{\psi}}_\xi \,. \end{aligned}$$

Thus, the necessary and sufficient condition for unconditional stability is

$$\begin{aligned} &\left|1 - \gamma v \imath \sin(\xi) + (\gamma v)^2(\cos(\xi) - 1)\right| \le 1 \quad \forall -\pi < \xi \le \pi \,, \\ \Leftrightarrow &\left(1 + (\gamma v)^2(\cos(\xi) - 1)\right)^2 + (\gamma v)^2 \sin(\xi)^2 \le 1 \quad \forall -\pi < \xi \le \pi \,, \\ \Leftrightarrow &\left(1 + (\gamma v)^2(t - 1)\right)^2 + (\gamma v)^2(1 - t^2) \le 1 \quad \forall -1 \le t \le 1 \,, \end{aligned} \tag{11.4.5}$$

We have to understand the behavior of the function

$$f_\alpha(t) := \left(1 + (\gamma v)^2(t - 1)\right)^2 + (\gamma v)^2(1 - t^2) \,, \quad -1 \le t \le 1 \,.$$

and have to find the values for $\alpha := \gamma v$ for which it stays below 1 on $[-1, 1]$. First we compute

$$f_\alpha(1) = 1 \quad \text{and} \quad f_\alpha(-1) = (1 - 2\alpha^2)^2 \,,$$

and find that there is no extremum in $]-1, 1[$. Thus we have to determine

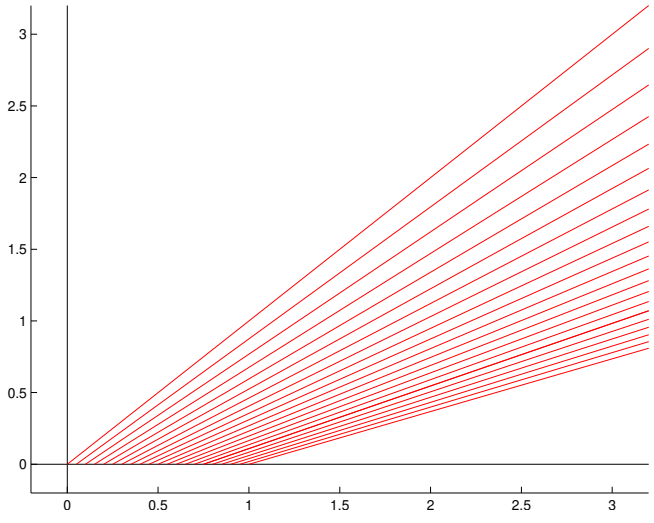
$$\alpha > 0\text{:} \quad (1 - 2\alpha^2)^2 \le 1 \quad \Leftrightarrow \quad \alpha^2 \le 1 \,.$$

Hence we, we have finally found the necessary and sufficient CFL-like condition for stability of the Lax-Wendroff full discrete discrete evolution (11.4.3)

$$\frac{\tau}{h} v \le 1 \quad \Leftrightarrow \quad \tau \le \frac{h}{v} \,.$$

This agrees with the geometric DFL-condition found in Sub-problem (11-4.a), because for $f(u) = vu$ the maximal speed of propagation is $s = v$, of course.

SOLUTION of (11-4.d):



There are characteristics emanating from the area $x \leq 0$ with slope $\gamma'(t) = e^A$ (where $u(x,t) = A$, and from the area $x \geq 1$ with slope $\gamma'(t) = e^B$ (where $u(x,t) = B$). On the wedge inbetween, we have $A \leq u(x,t) \leq B$, and indeed, this is the maximal set where the inequalities could be strict:

$$\{(x,t) \in \mathbb{R} \times [0,1] : \ x/t > e^A, \ (x-1)/t < e^B\} \ .$$

SOLUTION of (11-4.e):

The implementation boils down to coding (11.4.1) for $f(u) = f'(u) = e^u$. Note that the actual length of the spatial interval is needed only implicitly.

C++ code 11.4.7: Sub-problem (11-4.e): function `solveLaxWendroff()` ⇝ GITHUB

```
Eigen::VectorXd LaxWendroffRhs(const Eigen::VectorXd &mu, double gamma) {
  int N = mu.size();
  Eigen::VectorXd result(N);

  auto f = [](double x) { return std::exp(x); };
  auto df2 = [](double x) { return Square(std::exp(x)); };

  // Lax-Wendroff fully discrete evolution (11.4.1) for
  // f(u) = e^u.  Store the values of f(x) and (f'(x))^2 from the
  // previous iteration:

  // We extend mu to the left by mu(0):
  double mu_left = mu(0);

  // f'(1/2(mu_j + mu_{j-1}))^2:
  double df2_old = df2(0.5 * (mu(0) + mu_left));

  // f'(1/2(mu_{j+1} + mu_j))^2:
  double df2_new = df2(0.5 * (mu(1) + mu(0)));

  double f_old = f(mu_left);  // f(mu_{j-1})
  double f_mid = f(mu(0));    // f(mu_j)
  double f_new = f(mu(1));    // f(mu_{j+1})

  result(0) = mu(0) - 0.5 * gamma * (f_new - f_old) +
              0.5 * Square(gamma) *
                  (df2_new * (mu(1) - mu(0)) - df2_old * (mu(0) - mu_left));

  for (int j = 1; j < N - 1; ++j) {
    df2_old = df2_new;
    df2_new = df2(0.5 * (mu(j + 1) + mu(j)));
    f_old = f_mid;
    f_mid = f_new;
    f_new = f(mu(j + 1));
    result(j) =
        mu(j) - 0.5 * gamma * (f_new - f_old) +
        0.5 * Square(gamma) *
            (df2_new * (mu(j + 1) - mu(j)) - df2_old * (mu(j) - mu(j - 1)));
  }

  // We extend mu to the right by mu(N-1):
  double mu_right = mu(N - 1);

  df2_old = df2_new;
  df2_new = df2(0.5 * (mu_right + mu(N - 1)));

  f_old = f_mid;
  f_new = f(mu_right);
```

```
50
51    result(N - 1) = mu(N - 1) - 0.5 * gamma * (f_new - f_old) +
52                    0.5 * Square(gamma) *
53                        (df2_new * (mu_right - mu(N - 1)) -
54                         df2_old * (mu(N - 1) - mu(N - 2)));
55
56    return result;
57  }
58
59  Eigen::VectorXd solveLaxWendroff(const Eigen::VectorXd &u0, double T,
60                                   unsigned int M) {
61    double gamma = 1.0 / Constant::e;
62    Eigen::VectorXd mu = u0;
63    // Main timestepping loop
64    for (int j = 0; j < M; ++j) mu = LaxWendroffRhs(mu, gamma);
65    return mu;
66  }
```

HINT 1 for (11-4.f): We deal with a 1D scalar conservation law with a *convex* flux function $f(u) = e^u$ that is, with increasing $f'(u) = e^u$. Thus, initial data with $u_l < u_r$ will lead to a rarefaction entropy solution according to [Lecture $\rightarrow$ Lemma 11.2.5.10]. ⌟

SOLUTION of (11-4.f):

We rely on the formulas given in [Lecture → Lemma 11.2.5.10] with $u_l = 0$, $u_r = 1$, $f'(u_l) = 1$, $f'(u_r) = e$ $f'(u) = e^u$, $g(\xi) = \log(\xi)$, $\xi > 0$. This gives the concrete rarefaction solution of the Riemann problem:

$$u(x,t) = \begin{cases} 0 & \text{, if } x \leq t \text{ ,} \\ \log(x/t) & \text{, if } t < x < te \text{ ,} \\ 1 & \text{, if } x \geq et \text{ ,} \end{cases} \qquad x \in \mathbb{R} \text{ ,} \quad t \geq 0 \text{ .} \tag{11.4.9}$$

SOLUTION of (11-4.g):

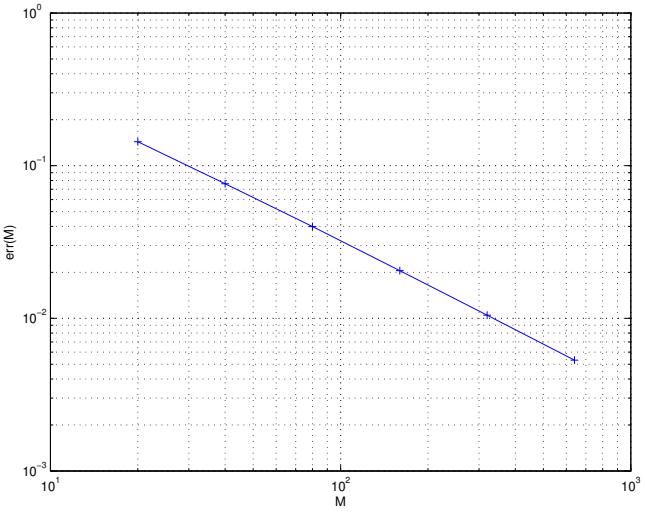
C++ code 11.4.11: Sub-problem (11-4.g): function `numexpLaxWendroffRP()` ⇝ GITHUB

```cpp
// Build spatial grid
Eigen::VectorXd getXValues(double T, unsigned int M) {
  double tau = T / M;
  double h = Constant::e * tau;
  int j_max = (int)(std::ceil((3.0 * T + 1.0) / h) + 0.5);
  int j_min = (int)(std::floor(-3.0 * T / h) - 0.5);
  unsigned int N = j_max - j_min + 1;
  return Eigen::VectorXd::LinSpaced(N, j_min * h, j_max * h);
}
Eigen::VectorXd numexpLaxWendroffRP(const Eigen::VectorXi &M) {
  const double T = 1.0;
  const int M_size = M.size();
  Eigen::VectorXd error(M_size);
  // Initial values for the Riemann problem
  auto u_initial = [](double x) { return 0.0 <= x ? 1.0 : 0.0; };
  // Exact solution (??) at time T = 1.0
  auto u_exact = [](double x) {
    return (x <= 1.0) ? 0.0 : ((Constant::e <= x) ? 1.0 : std::log(x));
  };
  for (int i = 0; i < M_size; ++i) {
    Eigen::VectorXd x = getXValues(T, M(i));
    Eigen::VectorXd u0 = x.unaryExpr(u_initial);
    Eigen::VectorXd uT = solveLaxWendroff(u0, T, M(i));

    double tau = T / M(i);
    double h = Constant::e * tau;
    error(i) = h * (x.unaryExpr(u_exact) - uT).lpNorm<1>();
  }
  return error;
}
```

PYTHON script 11.4.12: Sub-problem (11-4.g): PYTHON script for plotting ⇝ GITHUB

```python
from matplotlib.pyplot import figure, legend, loglog, savefig, xlabel, ylabel
from numpy import exp, genfromtxt, log, polyfit
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
M = data[0]

name = ['LaxWendroffRP', 'LaxWendroffSmoothU0', 'GodunovSmoothU0']
color = ['b', 'g', 'c']
n = len(name)

error = []
p = []
for i in range(n):
    error.append(data[i + 1])
```

```
19      p.append(polyfit(log(M), log(error[i]), 1))
20
21  fig = figure()
22  for i in range(n):
23      loglog(M, error[i], color[i] + 'o', label = name[i])
24      loglog(M, exp(p[i][0] * log(M) + p[i][1]), color[i] + '--', label='slope = ' +
            str(round(p[i][0], 2)))
25  xlabel('number of time steps M')
26  ylabel('error')
27  legend()
28  savefig(output_file)
29
30  print('Generated ' + output_file)
```



◁ Neatly aligned error points in a doubly logarithmic plot hint at *algebraic convergence*.

From the slope of the line we read off a rate of convergence of ≈ 1.

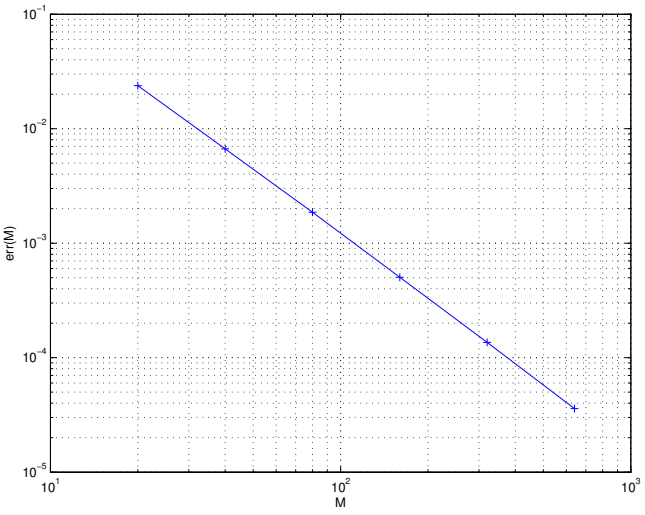
SOLUTION of (11-4.h):

C++ code 11.4.14: Sub-problem (11-4.h): Computation of reference solutuion ⇝ GITHUB

```
double smoothU0(double x) {
  return (x < 0.0)
             ? 0.0
             : ((1.0 < x) ? 1.0 : Square(std::sin(0.5 * Constant::pi * x)));
}
Eigen::VectorXd referenceSolution(const Eigen::VectorXd &x) {
  double T = 1.0;
  // Reference solution on a very fine mesh
  unsigned int M = 3200;

  Eigen::VectorXd y = getXValues(T, M);
  Eigen::VectorXd u0 = y.unaryExpr(&smoothU0);
  Eigen::VectorXd u = solveLaxWendroff(u0, T, M);
  int N = x.size();
  Eigen::VectorXd u_ref(N);
  // The vector u is larger than u_ref. Use eval() from above the "evaluate" u
  // at the positions x(i) and thus obtain the reference solution u_ref.
  for (int i = 0; i < N; ++i) {
    u_ref(i) = eval(u, y, x(i));
  }
  return u_ref;
}
```

C++ code 11.4.15: Sub-problem (11-4.h): function `numexpLaxWendroffSmoothU0()` ⇝ GITHUB

```
Eigen::VectorXd numexpLaxWendroffSmoothU0(const Eigen::VectorXi &M) {
  const double T = 1.0;
  const int M_size = M.size();
  Eigen::VectorXd error(M_size);

  for (int i = 0; i < M_size; ++i) {
    Eigen::VectorXd x = getXValues(T, M(i));
    Eigen::VectorXd u0 = x.unaryExpr(&smoothU0);
    Eigen::VectorXd uT = solveLaxWendroff(u0, T, M(i));

    double tau = T / M(i);
    double h = Constant::e * tau;
    error(i) = h * (referenceSolution(x) - uT).lpNorm<1>();
  }
  return error;
}
```



Again we observe algebraic convergence with a slightly better rate of almost 2.

SOLUTION of (11-4.i):

The Godunov numerical flux function is a two-point flux function defined as

$$F(v,w) = f(u^{\downarrow}(v,w)) = e^{u^{\downarrow}(v,w)},$$

where $u^{\downarrow}$ is defined as in [Lecture $\rightarrow$ Eq. (11.3.4.31)]. There are five cases. However, the second and fifth can never occur, because f' is always positive in our case, and the first can never occur because $\dot{s} \geq 0$ always. So our Godunov flux function is quite simply

$$F(v,w) = e^{v}.$$

SOLUTION of (11-5.a):

$\dim V_h = \infty$, because it is defined on an infinite mesh!

SOLUTION of (11-5.j):

The modulus in the integral makes this problem a little awkward. In general, $f'(u) = 1 - 2u$, which changes sign at $u = 1/2$. Then, we can split the integral,

$$\int_v^w |f'(\xi)| \mathrm{d}\xi = \int_v^{1/2} |f'(\xi)| \mathrm{d}\xi - \int_w^{1/2} |f'(\xi)| \mathrm{d}\xi \,,$$

where we are now ensured that $f'(\xi)$ has constant sign in each of the terms on the right. Finally, we have

$$\int_s^{1/2} |f'(\xi)| \mathrm{d}\xi = \begin{cases} \frac{1}{4} - s(1-s), & \text{, if } s < \frac{1}{2} \,, \\ s(1-s) - \frac{1}{4}, & \text{, if } s \geq \frac{1}{2} \,, \end{cases}$$

or, expressed differently,

$$\int_s^{1/2} |f'(\xi)| \, \mathrm{d}\xi = \left(s(1-s) - \frac{1}{4} \right) \operatorname{sgn}\left(s - \frac{1}{2} \right) \,,$$

where sgn is the sign function.

C++ code 11.5.15: Sub-problem (11-5.j): function `Feo()` ⇝ GitLab

```
2  double Feo(double v, double w) {
3    double result;
4    auto f = [](double u) { return u * (1.0 - u); };
5    auto I = [](double s) {
6      double y = 0.25 - s * (1.0 - s);
7      return s < 0.5 ? y : -y;
8    };
9    result = 0.5 * (f(v) + f(w)) - 0.5 * (I(v) - I(w));
10   return result;
11 }
```

SOLUTION of (11-5.k):

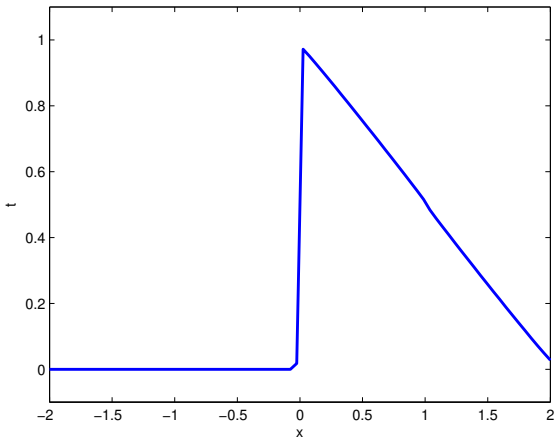
C++ code 11.5.17: Sub-problem (11-5.k): function `solveTrafficFlow()` ⇝ GitLab

```
Solution solveTrafficFlow() {
  int Ml = 40;
  int Mr = 40;
  int N_half = Mr + Ml + 1;
  int N = 2 * N_half;

  double h = 0.05;
  double tau = h / 3;
  double T = 1.0;
  unsigned int m = (unsigned int)(T / tau);

  // Spatial mesh
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N_half, -2.0, 2.0);
  // Initial condition according to (11.5.16). Note that we just
  // initialize the cell averages; a more refined implementation would
     also
  // sample the local first moments of u_0.
  Eigen::VectorXd mu0 = Eigen::VectorXd::Zero(N);
  auto u0 = [](double x) { return 0.0 <= x && x <= 1.0 ? 1.0 : 0.0; };
  for (int i = 0; i < N_half; ++i) {
    mu0(2 * i) = u0(x(i));
  }
  // Flux function
  auto f = [](double u) { return u * (1.0 - u); };
  // Perform fully discrete evolution
  Eigen::VectorXd mu = dgcl(mu0, f, Feo, T, Ml, Mr, h, m);
  // Retrieve cell averages
  Eigen::VectorXd u(N_half);
  for (int i = 0; i < N_half; ++i) {
    u(i) = mu(2 * i);
  }

  return Solution(std::move(x), std::move(u));
}
```

PYTHON script 11.5.18: Sub-problem (11-5.k): Pyhton script for plotting solution at final time ⇝ GitLab

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel
from numpy import genfromtxt
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
x = data[0]
y = data[1]

figure()
plot(x, y)
```

```
14 xlabel('x')
15 ylabel('u(x,1)')
16 savefig(output_file)
17
18 print('Generated ' + output_file)
```



◁ Solution at final time $T = 1$.
The solution features a leading rarefaction fan and a trailing stationary shock.

HINT 1 for (11-5.b): Perform the usual steps in the derivation of a spatial variational formulation: test, integrate, and integrate by parts (in one spatial dimension). ⌟

SOLUTION of (11-5.b):

The important part is the integration by parts in the flux term.

$$\int_{x_{j-1}}^{x_j} v_h(x)\frac{\partial}{\partial x}\big(f(u(x,t))\mathrm{d}x = [f(u(x,t))v_h(x)]_{x_{j-1}}^{x_j} - \int_{x_{j-1}}^{x_j} f(u(x,t))\frac{\mathrm{d}v_h}{\mathrm{d}x}(x)\mathrm{d}x$$

$$= f(u(x_j,t))v_h(x_j-) - f(u(x_{j-1},t))v_h(x_{j-1}+) - \int_{x_{j-1}}^{x_j} f(u(x,t))\frac{\mathrm{d}v_h}{\mathrm{d}x}(x)(x)\mathrm{d}x\ .$$

SOLUTION of (11-5.c):

The *maximal speed of propagation* in either direction, and at any time, is c. Thus, at time $T = 1$, u is supported in $[-c, 1+c]$. We want $x_{-M_l-1} = (-M_l - 1)h \leq -c$, which implies $M_l + 1 \geq c/h$. Similarly, $M_r \geq (1+c)/h$. Thus

$$M_l = \left\lceil \frac{c}{h} \right\rceil - 1, \qquad M_r = \left\lceil \frac{1+c}{h} \right\rceil.$$

SOLUTION of (11-5.d):

The matrix $\mathbf{B}$ is actually just the mass matrix. We can see that two basis functions b_k^i and b_l^j have disjoint support if $i \neq j$, and since b_2^j has average value zero, we also see that b_1^i and b_2^i are orthogonal. This makes $\mathbf{B}$ fully diagonal. The entries corresponding to the constant basis functions are h, and the others are (using the $[0,h]$ interval to compute)

$$\int_0^h \left(x - \frac{h}{2}\right)^2 \mathrm{d}x = \frac{1}{3}\left(h - \frac{h}{2}\right)^3 - \frac{1}{3}\left(0 - \frac{h}{2}\right)^3 = \frac{h^3}{12}.$$

Thus,

$$\mathbf{B} = \begin{pmatrix} h & & & & \\ & \frac{h^3}{12} & & & \\ & & h & & \\ & & & \frac{h^3}{12} & \\ & & & & \ddots \end{pmatrix}.$$

SOLUTION of (11-5.e):

Pay attention to the efficient initialization of the sparse matrix in EIGEN. Since we know in advance that every row will have exactly one non-zero entry, we can reserve that space.

C++ code 11.5.9: Sub-problem (11-5.e): function `compBmat()` ↪ GitLab

```
2  Eigen::SparseMatrix<double> compBmat(int Ml, int Mr, double h) {
3    const int N = 2 * (Ml + Mr + 1);
4    Eigen::SparseMatrix<double> A(N, N);
5    // Tell Eigen that the matrix is diagonal, which allows efficient
6    // initialization
7    A.reserve(Eigen::VectorXi::Ones(N));
8    for (int i = 0; i < N; i += 2) {
9      A.insert(i, i) = h;
10     A.insert(i + 1, i + 1) = h * h * h / 12.0;
11   }
12   return A;
13 }
```

HINT 1 for (11-5.f): If it helps, you can use the notation $\mu_{i,1}$ and $\mu_{i,2}$ to reference the elements in a vector $\vec{\boldsymbol{\mu}}$ corresponding to basis functions b_1^i and b_2^i respectively (e.g. as if $\boldsymbol{\mu}$ were a matrix). ⌟

SOLUTION of (11-5.f):

We need to evaluate the expression

$$-\sum_{j=\infty}^{\infty}\int_{x_{j-1}}^{x_j} f(u_h(x))\frac{\mathrm{d}v_h}{\mathrm{d}x}(x)\mathrm{d}x + \sum_{j=-\infty}^{\infty} F(u_h(x_j-),u_h(x_j+))(v_h(x_j-)-v_h(x_j+))$$

for $v_h = b_l^i$, $l = 1,2$.

For $l = 1$ (the constant basis functions), we see that the derivative is zero, and that $b_1^i(x_{i-1}+) = b_1^i(x_i-) = 1$, with all other grid-point evaluations being zero. The expression then collapses to

$$G(\vec{\boldsymbol{\mu}})_{i,1} = F(u_h(x_i-),u_h(x_i+)) - F(u_h(x_{i-1}-),u_h(x_{i-1}+)) \ .$$

For $l = 2$ (the linear basis function with vanishing mean), the derivative is constant 1, and $b_2^i(x_{i-1}+) = -h/2$ and $b_2^i(x_i-) = h/2$. Thus, the integral term does *not* vanish, the numerical flux terms will be multiplied by $h/2$ and there is one sign change. We get

$$G(\vec{\boldsymbol{\mu}})_{i,2} = -\int_{x_{i-1}}^{x_i} f(u_h(x))\mathrm{d}x + \frac{h}{2}[F(u_h(x_i-),u_h(x_i+)) + F(u_h(x_{i-1}-),u_h(x_{i-1}+))] \ .$$

Further, we have the formulas for u_h:

$$u_h(x_i-) = \mu_{i,1} + \frac{h}{2}\mu_{i,2}, \qquad u_h(x_{i-1}+) = \mu_{i,1} - \frac{h}{2}\mu_{i,2} \ .$$

HINT 1 for (11-5.g): Do not forget the rescaling of quadrature weights when using (11.5.10) on an interval of length h. ⌟

SOLUTION of (11-5.g):

The quadrature points relative to $[x_{i-1}, x_i]$ are

$$x_{i-1} + \frac{h}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right) .$$

And u_h evaluated at those points is (we benefit from nice cancellations)

$$u_h(x_{i-1} + \frac{h}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right)) = \mu_{i,1} + \mu_{i,2}\left[x_{i-1} + \frac{h}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right) - \frac{1}{2}(2x_{i-1} + h)\right] = \mu_{i,1} \pm \frac{h}{2\sqrt{3}}\mu_{i,2} .$$

Our approximation for the integral in question is thus

$$\int_{x_{i-1}}^{x_i} f(u_h(x))\mathrm{d}x \approx \frac{h}{2}\left[f\left(\mu_{i,1} - \frac{h}{2\sqrt{3}}\mu_{i,2}\right) + f\left(\mu_{i,1} + \frac{h}{2\sqrt{3}}\mu_{i,2}\right)\right] ,$$

giving

$$\begin{aligned} G(\vec{\boldsymbol{\mu}})_{i,2} = &- \frac{h}{2}\left[f\left(\mu_{i,1} - \frac{h}{2\sqrt{3}}\mu_{i,2}\right) + f\left(\mu_{i,1} + \frac{h}{2\sqrt{3}}\mu_{i,2}\right)\right] + \\ &+ \frac{h}{2}[F(u_h(x_i-), u_h(x_i+)) + F(u_h(x_{i-1}-), u_h(x_{i-1}+))] . \end{aligned}$$

The formula for $G(\vec{\boldsymbol{\mu}})_{i,1}$ remains the same.

SOLUTION of (11-5.h):

C++ code 11.5.11: Sub-problem (11-5.h): function G() ⇝ GitLab

```
template <typename FUNCTOR, typename NUMFLUX>
Eigen::VectorXd G(const Eigen::VectorXd &mu, FUNCTOR &&f, NUMFLUX &&F, int Ml,
                  int Mr, double h) {
  const int N_half = (Ml + Mr + 1);
  const int N = 2 * N_half;
  Eigen::VectorXd Gvec(N);
  double uN_xminus = 0.0;  // since we extend mu to the left by zero
  double uN_xplus = mu(0) - 0.5 * h * mu(1);
  double F_old;
  double F_new = F(uN_xminus, uN_xplus);
  const double w = h / (2.0 * std::sqrt(3.0));

  for (int i = 0; i < N_half - 1; ++i) {
    uN_xminus = mu(2 * i) + 0.5 * h * mu(2 * i + 1);
    uN_xplus = mu(2 * (i + 1)) - 0.5 * h * mu(2 * (i + 1) + 1);
    F_old = F_new;
    F_new = F(uN_xminus, uN_xplus);
    Gvec(2 * i) = F_new - F_old;

    double x_minus = mu(2 * i) - w * mu(2 * i + 1);
    double x_plus = mu(2 * i) + w * mu(2 * i + 1);
    double I = 0.5 * h * (f(x_minus) + f(x_plus));
    Gvec(2 * i + 1) = 0.5 * h * (F_new + F_old) - I;
  }

  uN_xminus = mu(2 * (N_half - 1)) + 0.5 * h * mu(2 * (N_half - 1) + 1);
  uN_xplus = 0.0;  // since we extend mu to the right by zero
  F_old = F_new;
  F_new = F(uN_xminus, uN_xplus);
  Gvec(2 * (N_half - 1)) = F_new - F_old;

  double x_minus = mu(2 * (N_half - 1)) - w * mu(2 * (N_half - 1) + 1);
  double x_plus = mu(2 * (N_half - 1)) + w * mu(2 * (N_half - 1) + 1);
  double I = 0.5 * h * (f(x_minus) + f(x_plus));
  Gvec(2 * (N_half - 1) + 1) = 0.5 * h * (F_new + F_old) - I;
  return Gvec;
}
```

SOLUTION of (11-5.i):

Applying the 2-stage explicit Runge-Kutta method to (11.5.8) entails the computation of the increment vector

$$\vec{\kappa} := \vec{\mu}^{(k)} - \tfrac{1}{2}\tau \mathbf{B}^{-1} G(\vec{\mu}^{(k)})$$

and the update

$$\vec{\mu}^{(k+1)} := \vec{\mu}^{(k)} + \tau \mathbf{B}^{-1} G(\vec{\kappa}) \,.$$

C++ code 11.5.13: Sub-problem (11-5.i): function `dgcl()` ↪ GitLab

```
2  template <typename FUNCTOR, typename NUMFLUX>
3  Eigen::VectorXd dgcl(Eigen::VectorXd mu0, FUNCTOR &&f, NUMFLUX &&F, double T,
4                       int Ml, int Mr, double h, unsigned int m) {
5    Eigen::SparseMatrix<double> B = compBmat(Ml, Mr, h);
6    Eigen::SparseMatrix<double> Binv = B.cwiseInverse();
7
8    auto G_bound = [&f, &F, T, Ml, Mr, h](const Eigen::VectorXd &mu) {
9      return G(mu, std::forward<FUNCTOR>(f), std::forward<NUMFLUX>(F), Ml, Mr, h);
10   };
11   // Timestepping based on explicit midpoint rule, a 2-stage explicit
       Rune-Kutta
12   // single-step method
13   double tau = T / m;
14   for (int i = 0; i < m; ++i) {
15     // First compute the increment and then update the state vector, see
16     // [Lecture → Eq. (11.4.1.12)]
17     Eigen::VectorXd k = -Binv * G_bound(mu0);
18     mu0 = mu0 - tau * Binv * G_bound(mu0 + 0.5 * tau * k);
19   }
20   return mu0;
21 }
```

SOLUTION of (11-6.a):

From [Lecture → Thm. 11.2.7.1] we know that

$$-1 \leq \min_{x \in \mathbb{R}} u_0(x) \leq u(x,t) \leq \max_{x \in \mathbb{R}} u_0(x) \leq 1 \ .$$

Hence, all we need to know is $f|_{[-1,1]}$, that is, the spline model needs to cover only $[-1,1]$: $a = -1$, $b = 1$.

HINT 1 for (11-6.b): A function $f \in C^2([a,b])$ is strictly convex, if and only if its *second derivative* is strictly positive. ⌟

SOLUTION of (11-6.b):

Note that for a cubic spline f in the representation (11.6.4) its second derivative $s := f''$ is a piecewise linear and continuous function with $s(\xi_j) = M_j$, see (11.6.5). Obviously, such a function is strictly positive, if and only if all its function values at the knots ξ_j are positive

$$s(x) > 0 \quad \forall x \in [a,b] \quad \Leftrightarrow \quad s(\xi_j) > 0 \quad \forall j \in \{0,\ldots,n\} \; .$$

Hence, f is strictly convex, if and only if $M_j > 0$ for all $j \in \{0,\ldots,n\}$.

SOLUTION of (11-6.c):

Recall [Lecture → Thm. 11.2.7.3] and the accompanying discussion.

First we have to determine the maximal speeds of propagation in both directions:

$$\dot{s}_{\min} := \min\{f'(\xi) : \min_{x\in\mathbb{R}} u_0(x) \le \xi \le \max_{x\in\mathbb{R}} u_0(x)\} .$$

In light of (11.6.2) and the strict convexity of f, which implies that f' is *strictly increasing*, we conclude

$$\dot{s}_{\min} = f'(-1) \quad , \quad \dot{s}_{\max} = f'(1) . \tag{11.6.6}$$

Hence, if we know that $\operatorname{supp} u_0 \subset [A,B]$ (from `initsupp`), then

$$\operatorname{supp} u(\cdot,t) \subset [A + f'(-1)t, B + f'(1)t] , \tag{11.6.7}$$

see [Lecture → Fig. 512]. Thanks to the strict convexity of f we know $f'(-1) < f'(1)$ so that (11.6.7) always defines a valid interval. The numbers $f'(-1)$ and $f'(1)$ can be obtained by calling the `derivative` method of **UniformCubicSpline**.

We can easily convert (11.6.7) into a code.

C++ code 11.6.8: Implementation of `findSupport()` ⇝ GitLab

```
2 Eigen::Vector2d findSupport(const UniformCubicSpline &f,
3                             Eigen::Vector2d initsupp, double t) {
4   Eigen::Vector2d result;
5   Eigen::Vector2d speed = {f.derivative(-1.0), f.derivative(1.0)};
6   result = initsupp + t * speed;
7   return result;
8 }
```

HINT 1 for (11-6.d): You may call the function `findSupport()` from Sub-problem (11-6.c). ⌟

SOLUTION of (11-6.d):

Note that the support of $\{x \mapsto u(x,t)\}$ may not necessarily contain the support of u_0! In fact, $\bigcup_{0\le t\le T} \operatorname{supp} u$ is the convex hull of $[-1,1]$ and $\operatorname{supp} u(\cdot,T)$. Thus, using (11.6.7) we find

$$\bigcup_{0\le t\le T} \operatorname{supp} u(\cdot,t) = [\min\{-1,-1+f'(-1)T\}, \max\{1,1+f'(1)T\}]\,,$$

which gives

$$m^- = \left\lfloor \frac{\min\{-1,-1+f'(-1)T\}}{h} \right\rfloor \quad , \quad m^+ = \left\lceil \frac{\max\{1,1+f'(1)T\}}{h} \right\rceil .$$

The two numbers $-1+f'(-1)T$ and $1+f'(1)T$ are returned by `findSupport()`.

C++ code 11.6.9: Implementation of `computeInitVec()` ⇝ GitLab

```
template <typename FUNCTOR>
Eigen::VectorXd computeInitVec(const UniformCubicSpline &f, FUNCTOR &&u0,
                               double h, double T) {
  Eigen::VectorXd mu0;
  const double A = -1.0;  // left bound of support
  const double B = 1.0;   // right bound of support
  // Here one could also call findSupport()
  const double AT = std::min(A, A + f.derivative(-1.0) * T);
  const double BT = std::max(B, B + f.derivative(1.0) * T);

  const int m_minus = (int)std::floor(AT / h);
  const int m_plus = (int)std::ceil(BT / h);
  const int N = m_plus - m_minus + 1;

  const Eigen::VectorXd x =
      Eigen::VectorXd::LinSpaced(N, m_minus * h, m_plus * h);
  mu0 = x.unaryExpr(std::forward<FUNCTOR>(u0));
  return mu0;
}
```

SOLUTION of (11-6.e):

As we have seen in [Lecture → Section 11.2.6], since f is strictly convex the entropy solution of a Riemann problem for (11.6.1) in the case of a downward jump $u_l \to u_r$, $u_l > u_r$, in the initial data u_0 is a **shock** moving with the Rankin-Hugoniot speed

$$\dot{s} := \frac{f(u_l) - f(u_r)}{u_l - u_r} .$$

Thus we find the solution

$$u(x,t) = \begin{cases} 1 & \text{for } x < \dot{s}\, t\,, \\ -1 & \text{for } x > \dot{s}\, t\,, \end{cases} \quad , \quad \dot{s} := \frac{f(u_l) - f(u_r)}{u_l - u_r} .$$

HINT 1 for (11-6.f): A simple bisection-type root finding algorithm is available in the function (also in `clempiricflux.cc`)

```
template <typename FUNCTOR>
double findRoots(double v, double w, FUNCTOR &&g,
                 double tol = 1.0E-6);
```

The functor `g` must pass a function $g : [v,w] \to \mathbb{R}$ with signature **std**`::function<`**double**`(`**double**`)>`, which changes sign in $[v,w]$. Th function returns an approximation of $\zeta \in [v,w]$ with $g(\zeta) = 0$. ⌟

SOLUTION of (11-6.f):

We write $F_{\mathrm{GD}} = F_{\mathrm{GD}}(v,w)$ for the Godunov numerical flux. The discussion follows [Lecture → § 11.3.4.32]

❶: If $v \geq w$, the local Riemann problem will have as an entropy solution a shock moving with the Rankine-Hugoniot speed

$$\dot{s} := \frac{f(u_l) - f(u_r)}{u_l - u_r} .$$

Hence,

$$F(v,w) = \begin{cases} f(v) & \text{, if } \dot{s} \geq 0 \,, \\ f(w) & \text{, if } \dot{s} < 0 \,. \end{cases} \tag{11.6.10}$$

❷: If $v < w$ we have to deal with a local rarefaction solution. The left edge of the rarefaction fan will move with speed $f'(v)$, the right edge with speed $f'(w)$.

$$F(v,w) = \begin{cases} f(v) & \text{, if } f'(v) > 0 \,, \\ f(z) & \text{, if } f'(v) < 0 < f'(w) \,, \\ f(w) & \text{, if } f'(w) < 0 \,. \end{cases} \tag{11.6.11}$$

where z is the unique zero of f' in $[v,w]$: $f'(z) = 0$. `findRoots()` can be used to determine z.

Computing the Godunov flux for a general flux function entails computing the zeros of a function ("root finding"). To find this zero, you may apply simple root-finding algorithms like the regula falsi or bisection merely require point evaluations of f', which are available through the `derivative()` method of **UniformCubicSpline**.

C++ code 11.6.12: Implementation of `findRoots()` ⇝ GitLab

```
2  template <typename FUNCTOR>
3  double findRoots(double v, double w, FUNCTOR &&g, double tol = 1.0E-6) {
4    double x = v;  // approximate root
5    const double len = w - v;
6    constexpr static const int maxN = 1000;
7    double gv = g(v), gw = g(w);
8    // Ensure that function changes sign
9    assert(gv * gw <= 0);
10   for (int N = 0; (std::abs(w - v) > tol * len) && N < maxN; N++) {
11     x = (v + w) / 2.0;
12     const double gx = g(x);
13     if (gv * gx < 0.0) {
14       // Sign change in left half of [v,w]
15       w = x;
16       gw = gx;
17     } else {
18       // Sign change in right half of [v,w]
19       v = x;
20       gv = gx;
21     }
22   }
23   return x;
24 }
```

C++ code 11.6.13: Evaluation operator for GodunovFlux ⇝ GitLab

```
double GodunovFlux::operator()(double v, double w) const {
  double result;
  if (v >= w)
    result = std::max(_f(v), _f(w));
  else {
    if (_f.derivative(v) > 0.0)
      result = _f(v);
    else if (_f.derivative(w) < 0.0)
      result = _f(w);
    else {
      auto df = [this](double x) { return _f.derivative(x); };
      double z = findRoots(v, w, df);
      result = _f(z);
    }
  }
  return result;
}
```

SOLUTION of (11-6.g):

Of course, we have to use [Lecture → Def. 7.3.3.1] and the conventions linking it with the Butcher scheme [Lecture → Eq. (7.3.3.3)]. From the (11.6.14) we get the coefficients

$$a_{11} = 0\,, \quad a_{12} = 0\,, \quad a_{21} = \frac{2}{3}\,, \quad a_{22} = 0\,,$$
$$b_1 = \frac{1}{4} \quad , \quad b_2 = \frac{3}{4}\,.$$

The coefficients c_i are not needed, because we deal with an autonomous ODE. For one step of the Ralston scheme starting from the state $\mathbf{y}_0$ and yielding the state $\mathbf{y}_1$ we obtain the increment equations

$$\mathbf{k}_1 := \mathbf{f}(\mathbf{y}_0) \quad , \quad \mathbf{k}_2 := \mathbf{f}(\mathbf{y}_0 + \frac{2}{3}\tau\mathbf{k}_1)\,,$$

and the update equation

$$\Psi^\tau\mathbf{y}_0 = \mathbf{y}_1 := \mathbf{y}_0 + \tau(\frac{1}{4}\mathbf{k}_1 + \frac{3}{4}\mathbf{k}_2)\,.$$

The implementation of these formulas is straightforward, because the Ralston scheme is an **explicit** Runge-Kutta single-step method.

C++ code 11.6.15: Implementation of `RalstonODESolver()` ⇝ GitLab

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd RalstonODESolver(FUNCTOR &&rhs, Eigen::VectorXd mu0, double tau,
4                                   int n) {
5    for (int i = 0; i < n; ++i) {
6      Eigen::VectorXd k1 = rhs(mu0);
7      Eigen::VectorXd k2 = rhs(mu0 + tau * 2.0 / 3.0 * k1);
8      mu0 = mu0 + 0.25 * tau * (k1 + 3.0 * k2);
9    }
10   return mu0;
11 }
```

SOLUTION of (11-6.h):

- First, we initialize an object of type **GodunovFlux** that will provide the numerical flux for our method.
- Then the right-hand-side of the ODE arising after spatial semi-discretization can be represented by a λ-function.
- The fully discrete scheme is then implemented using a formula similar to [Lecture → Eq. (11.5.3.2)] with coefficients given by the Butcher scheme (11.6.14).

  This 2-stage timestepping method leads to a 5-point spatial stencil in the sense of [Lecture → § 11.4.2.1]. Thus the geometric CFL-condition according to [Lecture → Def. 11.4.2.11] stipulates the timestep

$$\tau \leq \min\left\{ \frac{2h}{|f'(-1)|}, \frac{2h}{|f'(1)|} \right\}. \tag{11.6.16}$$

  However, depending on the Runge-Kutta single-step method employed, this condition may not ensure stability. Thus a safer option is the choice

$$\tau := \min\left\{ \frac{h}{|f'(-1)|}, \frac{h}{|f'(1)|} \right\}. \tag{11.6.17}$$

In the following implementation the right-hand side of the method-of-lines ODE is provided through a separate function, in order to be able to reuse `RalstonODESolver()`.

C++ code 11.6.18: Implementation of `solveCauchyProblem()` → GitLab

```
2  Eigen::VectorXd solveCauchyProblem(const UniformCubicSpline &f,
3                                     const Eigen::VectorXd &mu0, double h,
4                                     double T) {
5    Eigen::VectorXd muT(mu0.size());
6    double tau = std::min(h / std::abs(f.derivative(-1.0)),
7                          h / std::abs(f.derivative(1.0)));
8    double n = (int)std::floor(T / tau);
9    GodunovFlux godunovFlux(f);
10   auto rhs = [h, &godunovFlux](const Eigen::VectorXd &mu) {
11     return semiDiscreteRhs(mu, h, godunovFlux);
12   };
13   muT = RalstonODESolver(rhs, mu0, tau, n);
14   return muT;
15 }
```

The following function provides the implementation of the right-hand-side function $\mathbf{f}$ of the method-of-lines ODE arising from the conservative finite volume method outlined above. The code has been adapted from [Lecture → Code 11.3.2.9]. Refer to that code for documentation.

C++ code 11.6.19: Implementation of right-hand side of the method-of-lines ODE → GitLab

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd semiDiscreteRhs(const Eigen::VectorXd &mu0, double h,
4                                  FUNCTOR &&numFlux) {
5    int m = mu0.size();
6    Eigen::VectorXd mu1(m);
```

```
  mu1(0) = -1.0 / h * (numFlux(mu0(0), mu0(1)) - numFlux(mu0(0), mu0(0)));
  for (int j = 1; j < m - 1; ++j) {
    mu1(j) =
        -1.0 / h * (numFlux(mu0(j), mu0(j + 1)) - numFlux(mu0(j - 1), mu0(j)));
  }
  mu1(m - 1) =
      -1.0 / h *
      (numFlux(mu0(m - 1), mu0(m - 1)) - numFlux(mu0(m - 2), mu0(m - 1)));
  return mu1;
}
```

SOLUTION of (11-7.a):

The flux function is defined for $u > 0$ and can be continuously extended to $u = 0$ through $f(0) := 0$. It is smooth on $]0, \infty[$. To check its curvature we examine the sign of the second derivative: By the product rule

$$f'(u) = \log u \quad , \quad f''(u) = u^{-1} > 0 \quad \text{for} \quad u > 0 \, .$$

SOLUTION of (11-7.j):

We observe that halving the timestep leads to a decrease of the error by a factor of 8, which is clear evidence of algebraic convergence with rate 3 : $\text{error} = O(\tau^3)$. The single-step method probably is of order 3.

SOLUTION of (11-7.k):

C++11 code 11.7.16: Code for `solveClaw()` ⇝ GitLab

```
template <typename U0_FUNCTOR>
Eigen::VectorXd solveClaw(U0_FUNCTOR &&u0, double T, unsigned int n) {
  // Set up spacial mesh and inital data.
  double a = 0.0;
  double b = 1.0;
  double h = (b - a) / n;
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(n, a + 0.5 * h, b - 0.5 * h);

  // Approximate dual cell averages at t=0
  Eigen::VectorXd mu = x.unaryExpr(u0);

  double alpha = mu.minCoeff();  // lower bound for initial data
  double beta = mu.maxCoeff();   // upper bound for initial data
  assert(alpha > 0.0 && beta > 0.0);

  // Set timestep tau according to the CFL-condition
  double tau =
      h / std::max(std::abs(std::log(alpha)), std::abs(std::log(beta)));
  // Semi-discretization (discretization in space)
  double h_inv = 1.0 / h;
  // Define right-hand-side for the SSP ODE solver
  auto semi_discrete_rhs =
      [h_inv](const Eigen::VectorXd &mu) -> Eigen::VectorXd {
    return -h_inv * slopelimfluxdiffper(mu, &logGodunovFlux, &limiterMC);
  };
  // Timestepping:  Solve the semi-discrete ODE
  int N = (int)(T / tau + 0.5);
  for (int i = 0; i < N; ++i) mu = sspEvolop(semi_discrete_rhs, mu, tau);

  return mu;
}
```

Note that the return type of the lambda function `semi_discrete_rhs` has to be specified explicitly. Otherwise a temporary object of the some EIGEN internal type will be returned, which causes problems later.

SOLUTION of (11-7.l):

For every point y_k we determine the corresponding cell of $\mathcal{M}$ and then perform local linear interpolation.

C++11 code 11.7.18: Code for `interpolate()` ↪ GitLab

```
2  template <typename VECSOURCE, typename VECDEST>
3  void interpolate(const VECSOURCE &s, VECDEST &d) {
4    // Determine number of cells
5    const std::size_t n = s.size();
6    const std::size_t N = d.size();
7    // Mesh widths/cell sizes
8    const double H = 1.0 / n;
9    const double h = 1.0 / N;
10   // Loop over nodes of destination mesh
11   for (int j = 0; j < N; ++j) {
12     // position of destinatioon node
13     const double xd = h * (j + 0.5);
14     // Index of the source node, potentially < 0
15     int k = static_cast<int>(std::floor(xd / H - 0.5));
16     // Position of nodes of source cell, in which xd is contained
17     const double xsl = H * (k + 0.5);  // left node
18     const double xsr = xsl + H;        // right node
19     // Values at source nodes taking into account periodic continuation
20     auto svl = s[(k < 0) ? k + n : k];
21     auto svr = s[(k + 1 >= n) ? k + 1 - n : k + 1];
22     // Linear interpolation inside source cell
23     d[j] = ((xsr - xd) * svl + (xd - xsl) * svr) / H;
24   }
25 }
```

SOLUTION of (11-7.m):

C++11 code 11.7.19: Code for `studyCvgMUSCLSolution()` ↪ GitLab

```
template <typename U0_FUNCTOR>
void studyCvgMUSCLSolution(U0_FUNCTOR &&u0, double T) {
  // For temporarily storing the number of cells and the associated
     error norms
  std::vector<std::tuple<std::size_t, double, double>> result{};
  constexpr int n_ref = 8192;
  // Compute reference solution
  std::cout << "Computing reference solution at T = " << T << " with " << n_ref
            << " cells" << std::endl;
  Eigen::VectorXd u_ref{solveClaw(std::forward<U0_FUNCTOR>(u0), T, n_ref)};
  // Compute solution on different meshes
  constexpr int lmin = 4;   // Coarsest mesh with 16 cells
  constexpr int lmax = 10;  // Finest mesh with 1024 cells
  for (int l = lmin; l <= lmax; ++l) {
    const std::size_t n = 1U << l;
    std::cout << "Computing with n = " << n << std::endl;
    // Compute solution
    Eigen::VectorXd u{solveClaw(std::forward<U0_FUNCTOR>(u0), T, n)};
    // Transfer solution to finest mesh
    Eigen::VectorXd u_prop = Eigen::VectorXd::Zero(u_ref.size());
    interpolate(u, u_prop);
    double linf_err = (u_prop - u_ref).lpNorm<Eigen::Infinity>();
    double l1_err = (u_prop - u_ref).lpNorm<1>() / n_ref;
    result.push_back({n, linf_err, l1_err});
  }
  std::cout << "n \t linf error \t l1 error" << std::endl;
  for (auto &data : result) {
    std::cout << std::get<0>(data) << " \t " << std::get<1>(data) << " \t "
              << std::get<2>(data) << std::endl;
  }
}
```

	$T = 0.2$		$T = 1.0$	
n	$L^\infty(]0,1[)$-norm	$L^1(]0,1[)$-norm	$L^\infty(]0,1[)$-norm	$L^1(]0,1[)$-norm
16	0.258554	0.0320086	0.382744	0.0279002
32	0.221642	0.0191052	0.389801	0.0131425
64	0.137047	0.00379365	0.473782	0.00757056
128	0.0792417	0.00235442	0.458947	0.00378283
256	0.0500682	0.00217349	0.47127	0.00212716
512	0.00907072	7.26122e-05	0.353872	0.000754736
1024	0.00252278	1.79136e-05	0.339504	0.000410959

As long as the solution is smooth convergence even in $L^\infty(]0,1[)$-norm can be observed. Once a shock appears, we can see only convergence in the $L^1(]0,1[)$-norm. From the data it is difficult to tell a rate of algebraic convergence. This is similar to what we saw in [Lecture → Exp. 11.4.4.1].

SOLUTION of (11-7.b):

Since the flux function f is convex, the structure of solutions of Riemann problems is the same as for the Burgers equation discussed in [Lecture → Section 11.2.6]. For $u_l > u_r$ we get a **shock solution**, a discontiunity moving with the Rankine-Hugoniot speed.

$$u(x,t) = \begin{cases} 2 = u_l & \text{for} \quad x < \dot{s}t \,, \\ 1 = u_r & \text{for} \quad x > \dot{s}t \,, \end{cases} \quad , \quad \dot{s} = \frac{f(u_l) - f(u_r)}{u_l - u_r} = 2(\log 2 - 1) + 1 \approx 0.3863 \,. \tag{11.7.2}$$

SOLUTION of (11-7.c):

The discussion for the Burgers equation in [Lecture → Section 11.2.6] reveals that in this case the entropy solution is a **rarefaction wave**. We use the formula given in [Lecture → Lemma 11.2.5.10] with $f'(u) = \log u$ and $g(\xi) = \exp \xi$, $u_l = 1$, $u_r = 2$:

$$f'(u_l) = 0 \quad , \quad f'(u_r) = log2 \approx 0.6931 > 0 \, .$$

This yields the continuous solution

$$u(x,t) = \begin{cases} 1 = u_l & \text{for} \quad x < 0 \, , \\ \exp(x/t) & \text{for} \quad 0 < x < t \log 2 \, , \\ 2 = u_r & \text{for} \quad x > t \log 2 \, . \end{cases} \tag{11.7.3}$$

SOLUTION of (11-7.d):

Of course, we have to assume admissible states $v, w > 0$. Since $u \mapsto f(u)$ is smooth and convex on $\mathbb{R}^+$, we can compute the intermediate state according to [Lecture → Eq. (11.3.4.31)] with $(f')^{-1}(0) = \exp 0 =$

$$u^{\downarrow}(v,w) = \begin{cases} w & \text{, if } v \geq w,\ \dot{s} \leq 0 \quad \text{or} \quad v < w,\ w < 1\,, \\ v & \text{, if } v > w,\ \dot{s} > 0 \quad \text{or} \quad v < w,\ v > 1\,, \\ 1 & \text{, if } v \leq 1 \leq w\,, \end{cases} \tag{11.7.4}$$

$$\text{with} \qquad \dot{s} = \begin{cases} \dfrac{v(\log v - 1) - w(\log w - 1)}{v - w} & \text{, if } v \neq w\,, \\ \log v & \text{, if } v = w\,. \end{cases} \tag{11.7.5}$$

Then $F_{\mathrm{GD}(v,w)} = f(u^{\downarrow}(v,w))$.

Remark. Note that due to cancellation (11.7.5) is prone to numerical instability in case $v \approx w$. However, we are only interested in the sign of $\dot{s}$, which will be correct even if we lose almost all the digits.

The implementation of (11.7.4) is straightforward.

C++11 code 11.7.6: Code for `logGodunovFlux()` ⇝ GitLab

```
2  double logGodunovFlux(double v, double w) {
3    double godunov_numerical_flux;
4    auto f = [](double u) { return u * (std::log(u) - 1.0); };
5    auto df = [](double u) { return std::log(u); };
6    if (v >= w)
7      godunov_numerical_flux = std::max(f(v), f(w));
8    else {
9      if (df(v) > 0.0)
10       godunov_numerical_flux = f(v);
11     else if (df(w) < 0.0)
12       godunov_numerical_flux = f(w);
13     else
14       godunov_numerical_flux = f(1.0);  // note:  df(1.0) = 0.0
15   }
16   return godunov_numerical_flux;
17 }
```

SOLUTION of (11-7.e):

The scheme (11.7.7) has a spatial stencil of $m = 1$ as explained in [Lecture → § 11.4.2.1]. The CFL-conditions enforces the timestep constraint $\tau \leq h/s$, where $s > 0$ is the maximal speed of propagation

$$s := \sup\{|f'(\xi)| : \inf_{x\in\mathbb{R}} u_0(x) \leq \xi \leq \sup_{x\in\mathbb{R}} u_0(x)\} . \tag{11.7.8}$$

For (11.7.1) we have $f'(\xi) = \log \xi$ and we infer

$$s \leq \max\{|\log \alpha|, |\log \beta|\} \quad \blacktriangleright \quad \tau \leq \frac{h}{\max\{|\log \alpha|, |\log \beta|\}} .$$

Remark. The formula for the maximal speed of propagation (11.7.8) is relevant for the CFL-condition, because [Lecture → Lemma 11.3.5.8] tells us that the Godunov numerical flux is monoton according to [Lecture → Def. 11.3.5.5]. As a consequence of the arguments in [Lecture → § 11.3.5.1] we conclude that

$$\min_{\ell\in\mathbb{Z}} \mu_\ell^{(0)} \leq \mu_j^{(k)} \leq \max_{\ell\in\mathbb{Z}} \mu_\ell^{(0)} \quad \forall j \in \mathbb{Z}, k \in \mathbb{N}_0 .$$

SOLUTION of (11-7.f):

The information “compactly supported initial data u_0” hints that u_0 vanishes outside a bounded interval, that is, we have to deal with $u_0(x) = 0$.

The flux function f remains defined for the state $u = 0$ by setting $f(0) = 0$. However, for this state we encounter an “infinite speed of propagation”. As a consequence, the CFL-condition found in Subproblem (11-7.e) will enforce a vanishing timestep; the discrete evolution will stall or the discrete evolution (11.7.7) may suffer from blow-up.

SOLUTION of (11-7.g):

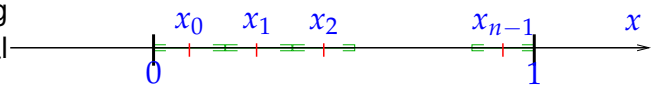
Note that the `slopes` functor yields the σ_j multiplied with the meshwidth h! In fact, the meshwidth is not even available in `slopelimfluxdiff()`.

C++11 code 11.7.11: Code for `limiterMC` ↪ GitLab

```
2  double limiterMC(double mu_left, double mu_center, double mu_right) {
3    double scaled_slope;
4
5    double sigma_l = 2.0 * (mu_center - mu_left);
6    double sigma_c = (mu_right - mu_left) / 2.0;
7    double sigma_r = 2.0 * (mu_right - mu_center);
8
9    std::initializer_list<double> sigma_list{sigma_l, sigma_c, sigma_r};
10
11   double min = std::min(sigma_list);
12   if (min > 0.0)
13     scaled_slope = min;
14   else {
15     double max = std::max(sigma_list);
16     if (max < 0.0)
17       scaled_slope = max;
18     else
19       scaled_slope = 0.0;
20   }
21
22   return scaled_slope;
23 }
```

HINT 1 for (11-7.h):

Mesh with n (dual) cells (draw in green) covering $]0,1[$. The nodes x_j are the centers of those dual cells.



SOLUTION of (11-7.h):

All it takes to adapt `slopelimfluxdiff` to the periodic setting is to replace μ_{-1} with μ_{n-1}, and μ_n with μ_0 whenever these dual-cell averages are referenced. This also applies to the values v_j^- and v_j^+ of the piecewise linear reconstruction at dual cell boundaries.

C++11 code 11.7.13: Implementation off `slopelimfluxdiffper()` ↪ GitLab

```
2  // Function parameters:
3  // double h:  meshwidth of equidistant spatial grid
4  // Vector mu:  (finite) vector μ⃗ of cell averages
5  // Functor F: 2-point numerical flux function F = F(v,w)
6  // Functor slope:  slope reconstruction function
7  // hσ_j = slopes(μ_{j-1}, μ_j, μ_{j+1})
8  //
9  // returns a vector with differences of numerical fluxes
10 //
11 // Function that realizes −h· the right hand side operator
12 // L_h for the mehtod-of-lines ODE arising from conservative finite
13 // volume semidiscretization of the Cauchy problem for a 1D scalar
       conservation
14 // law [Lecture → Eq. (11.2.2.1)] in a 1-periodic setting.
15 template <typename FunctionF, typename FunctionSlopes>
16 Eigen::VectorXd slopelimfluxdiffper(const Eigen::VectorXd &mu, FunctionF &&F,
17                                     FunctionSlopes &&slopes) {
18   int n = mu.size();  // Number of active dual grid cells
19   Eigen::VectorXd sigma = Eigen::VectorXd::Zero(n);  // Vector of slopes
20   Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);     // Flux differences
21
22   // Computation of slopes σ_j, uses μ_{-1} = μ_{n-1},
23   // μ_n = μ_0, which amounts to constant extension of states
24   // beyond domain of influence [a,b] of non-constant intial data.  Same
25   // technique has been applied in [Lecture → Code 11.3.2.9]
26   sigma[0] = slopes(mu[n - 1], mu[0], mu[1]);  // @@
27   for (int j = 1; j < n - 1; ++j)
28     sigma[j] = slopes(mu[j - 1], mu[j], mu[j + 1]);
29   sigma[n - 1] = slopes(mu[n - 2], mu[n - 1], mu[0]);  // @@
30
31   // Compute linear reconstruction at endpoints of dual cells [Lecture →
       Eq. (11.5.1.2)]
32   Eigen::VectorXd nup = mu + 0.5 * sigma;
33   Eigen::VectorXd num = mu - 0.5 * sigma;
34
35   // Rely on periodicity to compute numerical fluxes at interval ends
36   fd[0] = F(nup[0], num[1]) - F(nup[n - 1], num[0]);  // @@
37   for (int j = 1; j < n - 1; ++j)
38     // see [Lecture → Eq. (11.3.2.8)]
39     fd[j] = F(nup[j], num[j + 1]) - F(nup[j - 1], num[j]);
40   fd[n - 1] = F(nup[n - 1], num[0]) -  // @@
41               F(nup[n - 2], num[n - 1]);
42   return fd;
43 }
```

The index changes are confined to Line 26, Line 29, Line 36, Line 40.

SOLUTION of (11-7.i):

Since the single-step method (11.7.14) is explicit, the implementation of the discrete evolution operator is straightforward.

C++11 code 11.7.15: Code for `sspEvolop()` ↪ GitLab

```
2 template <typename FUNCTOR, typename State>
3 State sspEvolop(FUNCTOR &&f, State y, double tau) {
4   State y_tau;
5   State k = y + tau * f(y);
6   k = 0.75 * y + 0.25 * k + 0.25 * tau * f(k);
7   y_tau = 1.0 / 3.0 * y + 2.0 / 3.0 * k + 2.0 / 3.0 * tau * f(k);
8   return y_tau;
9 }
```

HINT 1 for (11-8.a): Integrate (11.8.1) over a cell $K \in \mathcal{M}$ and then integrate by parts using Green's formula [Lecture $\rightarrow$ Thm. 1.5.2.7]. ⌟

SOLUTION of (11-8.a):

Let us take (11.8.1) and integrate only over a cell K. Using Gauss' theorem [Lecture $\rightarrow$ Thm. 1.5.2.4], we then get

$$\int_K \frac{\partial u}{\partial t}(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} + \int_{\partial K} \boldsymbol{\beta}(\boldsymbol{x})\cdot\boldsymbol{n}_K(\boldsymbol{x})\,u(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0\,.$$

Since w_h is constant in K, we may multiply with $w_h\big|_K(\boldsymbol{x})$ and bring it under the integral. Summing over all K then yields (11.8.4).

SOLUTION of (11-8.j):

We can take the cue from [Lecture → Eq. (11.4.1.11)] and get

$$\vec{\kappa} = \boldsymbol{\mu}^{(k)} + \tau \mathbf{B} \vec{\boldsymbol{\mu}}^{(k)} \quad , \quad \vec{\boldsymbol{\mu}}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \frac{\tau}{2} \mathbf{B} \left(\boldsymbol{\mu}^{(k)} + \vec{\kappa} \right) . \tag{11.8.19}$$

SOLUTION of (11-8.k):

According to the discussion in [Lecture → § 11.4.2.10] we have to demand

$$\tau \leq m \frac{h_{\min}}{\dot{s}} ,$$

where m is the width of the "stencil", $h_{\min}$ the minimal distance of cell brycenters ($\boldsymbol{c}_j$ is the barycenter of $K_j \in \mathcal{M}$)

$$h_{\min} := \min\{ \left\| \boldsymbol{c}_i - \boldsymbol{c}_j \right\| : \ K_i, K_j \in \mathcal{M} \text{ have a common edge} \} , \tag{11.8.20}$$

and $\dot{s}$ the maximal speed of propagation. Since $\|\boldsymbol{\beta}\| \leq 1$, we have $\dot{s} \leq 1$, see Sub-problem (11-8.b). Moreover, a two-stage explicit RK-SSM for timesteping means $m = 2$, because the spatial interactions encoded in $\mathbf{B}$ are only between neighboring cells. This yields the final threshold

$$\tau \leq \frac{1}{2} \cdot \frac{2h_{\min}}{\dot{s}} ,$$

where $\frac{1}{2}$ is a "safety factor", because violation of the CFL-condition is likely to lead to blow-up.

HINT 1 for (11-8.l): The function `getAdjacentCellPointers()` comes handy. ⌟

SOLUTION of (11-8.l):

Traverse all the cells, obtain pointers to its immediate neighbors from the array returned by `getAdjacentC` compute the coordinates of the barycenters and their distance to the barycenter of the current cell. Take the minimum over all these lengths.

C++ code 11.8.21: Sub-problem (11-8.l): function `computeHmin()` ⇝ GitLab

```
2  double computeHmin(std::shared_ptr<const lf::mesh::Mesh> mesh_p) {
3    // Vector to store the distances between cells
4    std::vector<double> min_h;
5
6    // Get Adjectent Cells
7    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
8        std::array<const lf::mesh::Entity *, 4>>>
9        adjacentCells = AdvectionFV2D::getAdjacentCellPointers(mesh_p);
10
11   // Iterate over all cells
12   for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
13     const lf::geometry::Geometry *geo_p = cell->Geometry();
14     const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);
15
16     // Compute the barycenter of the cell
17     Eigen::Vector2d cur_midpoint = barycenter(corners);
18
19     // Iterate over all adjecent cells
20     for (const lf::mesh::Entity *neighbour_cell : (*adjacentCells)(*cell)) {
21       // Check that the neighbor exists
22       if (neighbour_cell != nullptr) {
23         const lf::geometry::Geometry *geo_p_neighbour =
24             neighbour_cell->Geometry();
25         const Eigen::MatrixXd neighbour_corners =
26             lf::geometry::Corners(*geo_p_neighbour);
27
28         // Compute barycenter of neighbour cell
29         Eigen::Vector2d neighbour_midpoint = barycenter(neighbour_corners);
30
31         // Compute distances between cells
32         double distance = (cur_midpoint - neighbour_midpoint).norm();
33
34         // Store value in vector
35         min_h.push_back(distance);
36       }
37     }
38   }
39
40   // Find the minimum value in the vector and return it
41   return *std::min_element(std::begin(min_h), std::end(min_h));
42 }
```

SOLUTION of (11-8.m):

The implementation of (11.8.19) is straightforward after the matrix $\mathbf{B}$ has been initialized.

C++ code 11.8.22: Sub-problem (11-8.m): function `solveAdvection2D()` ↪ GitLab

```
std::shared_ptr<
    lf::mesh::utils::CodimMeshDataSet<Eigen::Matrix<double, 2, Eigen::Dynamic>>>
computeCellNormals(std::shared_ptr<const lf::mesh::Mesh> mesh_p) {
  // Appears only in mastersolution

  // Initialize datastruture for the result
  lf::mesh::utils::CodimMeshDataSet<Eigen::Matrix<double, 2, Eigen::Dynamic>>
      result(mesh_p, 0);

  // Compute normal vectors
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    const lf::geometry::Geometry *geo_p = cell->Geometry();
    const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);

    if (corners.cols() == 3) {
      Eigen::Matrix<double, 2, Eigen::Dynamic> normal_vectors(2, 3);

      // Compute normal vectors
      Eigen::Matrix<double, 2, 3> bary_coord = gradbarycoordinates(corners);

      // Reorder computed normal vectors
      // normal_vectors[0] -> normal_vector of edge[0]
      normal_vectors.col(0) = -(bary_coord.col(2)).normalized();
      normal_vectors.col(1) = -(bary_coord.col(0)).normalized();
      normal_vectors.col(2) = -(bary_coord.col(1)).normalized();

      // Save the matrix in our datastructure
      result(*cell) = normal_vectors;
    } else {  // corners.cols() == 4
      Eigen::Matrix<double, 2, Eigen::Dynamic> normal_vectors(2, 4);

      // Split the quadrilateral into two triangles (1,2,3) and (1,3,4)
      Eigen::Matrix<double, 2, 3> tria_1;
      Eigen::Matrix<double, 2, 3> tria_2;
      tria_1 << corners.col(0), corners.col(1), corners.col(2);
      tria_2 << corners.col(0), corners.col(2), corners.col(3);

      // Compute normal vectors
      Eigen::Matrix<double, 2, 3> bary_coord_1 = gradbarycoordinates(tria_1);
      Eigen::Matrix<double, 2, 3> bary_coord_2 = gradbarycoordinates(tria_2);

      // Reorder computed normal vectors
      // normal_vectors[0] -> normal_vector of edge[0]
      normal_vectors.col(0) = -(bary_coord_1.col(2)).normalized();
      normal_vectors.col(1) = -(bary_coord_1.col(0)).normalized();
      normal_vectors.col(2) = -(bary_coord_2.col(0)).normalized();
      normal_vectors.col(3) = -(bary_coord_2.col(1)).normalized();

      // Save the matrix in our datastructure
      result(*cell) = normal_vectors;
```

```
52        }
53    }
54    return std::make_shared<lf::mesh::utils::CodimMeshDataSet<
55        Eigen::Matrix<double, 2, Eigen::Dynamic>>>(result);
56
57  }
```

SOLUTION of (11-8.n):

C++ code 11.8.26: Sub-problem (11-8.n): function `simulateAdvection()` ⇝ GitLab

```
template <typename FUNCTOR, typename VECTORFIELD>
Eigen::VectorXd simulateAdvection(
    const lf::assemble::DofHandler &dofh, VECTORFIELD &&beta, FUNCTOR &&u0,
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        std::array<const lf::mesh::Entity *, 4>>>
        adjacentCells,
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        Eigen::Matrix<double, 2, Eigen::Dynamic>>>
        normal_vectors,
    double T) {
  // Get number of dofs
  int num_dof = dofh.NumDofs();

  // Inititialize a vector for the initial condition
  Eigen::VectorXd u0_h(num_dof);

  // Iterate over all cells
  for (const lf::mesh::Entity *cell : dofh.Mesh()->Entities(0)) {
    const lf::geometry::Geometry *geo_p = cell->Geometry();
    const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);

    // Compute barycenter of the cell
    Eigen::Vector2d x = barycenter(corners);

    // Find the index of the DOF for the cell
    int idx = dofh.GlobalDofIndices(*cell)[0];

    u0_h[idx] = u0(x);
  }

  // Set up the number of steps according to the CFL condition
  int M = int((T / computeHmin(dofh.Mesh())) + 2);

  return solveAdvection2D(dofh, beta, u0_h, adjacentCells, normal_vectors, T,
                          M);
}
```

HINT 1 for (11-8.o): See [Lecture → Ex. 3.1.4.4] to learn about the creation of sequences of meshes by regular refinement in LEHRFEM++. ⌟

SOLUTION of (11-8.o):

C++ code 11.8.27: Sub-problem (11-8.o): numerical experiment and computation of error norms. ➙ GitLab

```
  // Define velocity field beta
  // Note that the problem description requires ||B|| <= 1
  auto beta = [](Eigen::Vector2d x) -> Eigen::Vector2d {
    return Eigen::Vector2d(-x[1], x[0]) / std::sqrt(2.0);
  };

  // Functor for initial bump
  auto u0 = [](Eigen::Vector2d x) -> double {
    Eigen::Vector2d x0(0.8, 0.2);
    double d = 0.2;
    double dist = (x - x0).norm();
    if (dist < d) {
      double cos = std::cos(M_PI / (2.0 * d) * dist);
      return cos * cos;
    }
    return 0.0;
  };

  double T = 1.0;
  // Generate a mesh hierarchy
  auto mesh_p = lf::mesh::test_utils::GenerateHybrid2DTestMesh(0, 1.0 / 3.0);
  auto mesh_seq_p{
      lf::refinement::GenerateMeshHierarchyByUniformRefinemnt(mesh_p, 6)};

  std::vector<int> vector_num_cells;
  std::vector<double> vector_l2error;

  // Iterate over mesh levels starting from thrid refinement
  int num_meshes = mesh_seq_p->NumLevels();
  for (int level = 3; level < num_meshes; ++level) {
    std::cout << "Computing L2Error for level: " << level << std::endl;

    // Get the current mesh
    auto cur_mesh = mesh_seq_p->getMesh(level);

    // Create a DOF Hander for the current mesh
    const lf::assemble::UniformFEDofHandler cur_dofh(
        cur_mesh, {{lf::base::RefEl::kPoint(), 0},
                   {lf::base::RefEl::kSegment(), 0},
                   {lf::base::RefEl::kTria(), 1},
                   {lf::base::RefEl::kQuad(), 1}});
    int N = cur_dofh.NumDofs();

    // Compute cell normals
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        Eigen::Matrix<double, 2, Eigen::Dynamic>>>
        normal_vectors = AdvectionFV2D::computeCellNormals(cur_dofh.Mesh());

    // Compute adjecent cells
    std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
        std::array<const lf::mesh::Entity *, 4>>>
```

```
        adjacentCells = AdvectionFV2D::getAdjacentCellPointers(cur_dofh.Mesh());

    // Get approximate solution from simulation
    Eigen::VectorXd mu_approx = AdvectionFV2D::simulateAdvection(
        cur_dofh, beta, u0, adjacentCells, normal_vectors, T);
    write_vtk(cur_dofh, mu_approx, "approx" + std::to_string(level));

    // Get exact solution at barycenters of cells
    Eigen::VectorXd mu_exact = AdvectionFV2D::refSolution(cur_dofh, u0, T);
    write_vtk(cur_dofh, mu_exact, "exact" + std::to_string(level));

    // Compute L2 error in barycenter
    double l2_error = 0;
    for (const lf::mesh::Entity *cell : cur_mesh->Entities(0)) {
      const lf::geometry::Geometry *geo_p = cell->Geometry();
      double area = lf::geometry::Volume(*geo_p);
      int idx = cur_dofh.GlobalDofIndices(*cell)[0];
      l2_error += std::pow((mu_approx[idx] - mu_exact[idx]), 2) * area;
    }
    l2_error = std::sqrt(l2_error);

    vector_num_cells.push_back(N);
    vector_l2error.push_back(l2_error);
  }

  // Write Output file of Task 8-8.o
  std::ofstream csv_file;
  csv_file.open("advectionfv2d.csv");
  for (int i = 0; i < vector_num_cells.size(); ++i) {
    std::cout << "Cells: " << vector_num_cells.at(i)
              << " | L2Error: " << vector_l2error.at(i) << std::endl;
    csv_file << vector_num_cells.at(i) << "," << vector_l2error.at(i) << "\n";
  }
  csv_file.close();

  systemcall::execute(
      "python3 scripts/advectionfv2d.py advectionfv2d.csv "
      "solution.eps");
```

◁ Doubly logarithmic plot of error norms $\|u_h(\cdot,1) - u(\cdot,1)\|_{L^2(\Omega)}$ versus the number N of mesh cells.

We observe algebraic convergence with rate 1.

SOLUTION of (11-8.p):

C++ code 11.8.28: Sub-problem (11-8.p): function `findCFLthreshold()` ↪ GitLab

```
template <typename VECTORFIELD>
int findCFLthreshold(const lf::assemble::DofHandler &dofh, VECTORFIELD &&beta,
                     double T) {
  // Set upper and lower limit
  int M_upper = int((T / computeHmin(dofh.Mesh())) + 1);
  int M_lower = 1;

  // Compute cell normals
  std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
      Eigen::Matrix<double, 2, Eigen::Dynamic>>>
      normal_vectors = AdvectionFV2D::computeCellNormals(dofh.Mesh());

  // Compute adjecent cells
  std::shared_ptr<lf::mesh::utils::CodimMeshDataSet<
      std::array<const lf::mesh::Entity *, 4>>>
      adjacentCells = AdvectionFV2D::getAdjacentCellPointers(dofh.Mesh());

  // Initialize a vector for the result and
  // randomly initialize a vector for the initial condition
  int num_dof = dofh.NumDofs();
  Eigen::VectorXd u0_h = Eigen::VectorXd::Random(num_dof);

  // Shift upper and lower bound until contition below isn't satisfied
  while ((M_upper - M_lower) > 2) {
    // Perform a simulation at M_middle
    // If it succeeds, set the upper bound to M_middle
    // Otherwise set the lower bound to M_middle
    // Repeat ...
    int M_middle = (M_upper + M_lower) / 2;
    try {
      solveAdvection2D(dofh, beta, u0_h, adjacentCells, normal_vectors, T,
                       M_middle);
      M_upper = M_middle;
    } catch (const std::exception &e) {
      M_lower = M_middle;
    }
  }

  int thres = M_upper;
  return thres;
}
```

SOLUTION of (11-8.q):

This amounts to insering a call to `findCFLthreshold()` in the loop that carries out the numerical simulation on the various refinement levels.

HINT 1 for (11-8.b): Remember that $\|\boldsymbol{\beta}(\boldsymbol{x})\| \leq 1$ for all $\boldsymbol{x}$. ⌟

HINT 2 for (11-8.b): You may quote the solution formula for the Cauchy problem for linear advection given in [Lecture → Thm. 11.1.1.8]:

$$u(\boldsymbol{x},t) = |\det(\mathrm{D}_{\boldsymbol{x}}\boldsymbol{\Phi})(\boldsymbol{x},t)|^{-1} u_0(\boldsymbol{\Phi}(\boldsymbol{x},-t)) \;, \quad (\boldsymbol{x},t) \in \mathbb{R}^d \times \mathbb{R} \;, \tag{11.8.5}$$

where $\Phi : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ is the flow induced by the velocity field $\boldsymbol{\beta}$ [Lecture → § 10.1.1.3], and $\mathrm{D}_{\boldsymbol{x}}\boldsymbol{\Phi}$ is its Jacobian. ⌟

SOLUTION of (11-8.b):

If $\boldsymbol{\Phi}$ is the flow map for $\boldsymbol{\beta}$, which satisfies

$$\frac{\partial \boldsymbol{\Phi}}{\partial t}(\boldsymbol{x},t) = \boldsymbol{\beta}(\boldsymbol{\Phi}(\boldsymbol{x},t)) \quad , \quad \boldsymbol{\Phi}(\boldsymbol{x},0) = \boldsymbol{x} \, , \quad \boldsymbol{x} \in \mathbb{R}^2 \, , \quad t \in \mathbb{R} \, , \tag{11.8.6}$$

then $t \mapsto \boldsymbol{\Phi}(\boldsymbol{x},t)$, $t \geq 0$, describes the trajectory of a "particle" with initial position $\boldsymbol{x}$. By the fundamental theorem of calculus

$$\boldsymbol{\Phi}(\boldsymbol{x},t) - \boldsymbol{x} = \int_0^t \boldsymbol{\beta}(\boldsymbol{\Phi}(\boldsymbol{x},s)) \, \mathrm{d}s \quad \overset{\|\boldsymbol{\beta}\| \leq 1}{\Longrightarrow} \quad \|\boldsymbol{\Phi}(\boldsymbol{x},t) - \boldsymbol{x}\| \leq |t| \, . \tag{11.8.7}$$

Physically speaking, this expresses that a particle with speed ≤ 1 cannot cover a distance larger than t in time t.

Thus, taking (11.8.5) for granted, it is clear that,

$$u_0(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} : \ \|\boldsymbol{x}\| > 1 \quad \Longrightarrow \quad u(\boldsymbol{x},t) = 0 \quad \forall \boldsymbol{x} : \ \|\boldsymbol{x}\| > 1 + t \, .$$

Thus, we conclude that

$$\operatorname{supp}(\{\boldsymbol{x} \mapsto u(\boldsymbol{x},t)\}) \subset \overline{B}_{1+t} := \{\boldsymbol{x} \in \mathbb{R}^2 : \ \|\boldsymbol{x}\| \leq R := 1 + t\} \, .$$

SOLUTION of (11-8.c):

If $e \subset \partial\Omega$, the second argument of F_e^K is never used in the definition (11.8.8).

SOLUTION of (11-8.d):

In (11.8.9) test with $w_h = b_h^j$, which gives, since the supports of the basis functions agree with the cells and do not overlap,

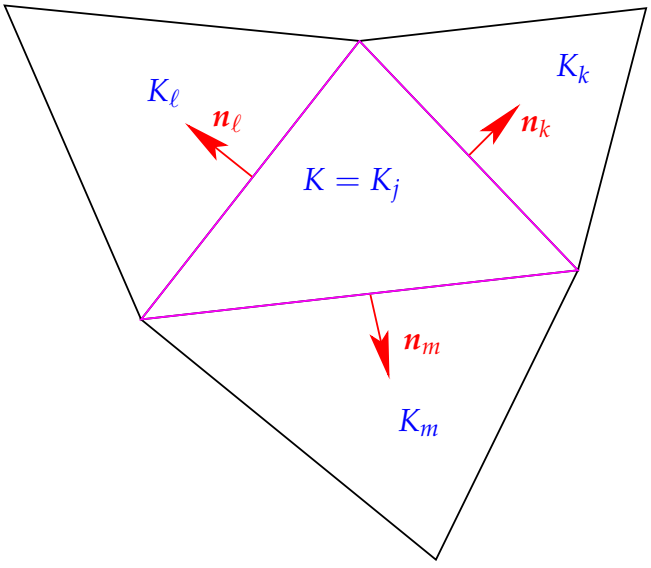
$$|K|\dot{\mu}_j(t) = -\sum_{e\in\mathcal{E},\, e\subset\partial K} |e| F_e^j(\mu_j, \mu_j^+)\,, \tag{11.8.11}$$

where F_e^j is the upwind numerical flux for cell K_j and edge e, and μ_j^+ is the value of the degree of freedom associated with the cell K^+ (which depends on e!). More explicitly, we have

$$|K|\dot{\mu}_j(t) = -\sum_{\nu\in\{\ell,m,k\}} |e_\nu| F_\nu^j(\mu_j, \mu_\nu)\,. \tag{11.8.12}$$

For the sake of compact notations we introduce the widely used symbols

$$(\xi)_+ := \begin{cases} \xi & \text{, if } \xi \geq 0\,, \\ 0 & \text{, if } \xi < 0\,, \end{cases} \quad , \quad (\xi)_- := \begin{cases} 0 & \text{, if } \xi \geq 0\,, \\ \xi & \text{, if } \xi < 0\,. \end{cases} \tag{11.8.13}$$



For $K = K_k$ we also write e_ν, $\nu \in \{\ell, m, k\}$, for the common edge of K_j and K_ν, $\boldsymbol{m}_\nu$ for the midpoint of e_ν and $\boldsymbol{n}_\nu$ for the associated unit normal vector pointing into the exterior of K.

With these notations, (11.8.12) can be converted into

$$\dot{\mu}_j(t) = -\sum_{\nu\in\{\ell,m,k\}} \frac{|e_\nu|}{|K|}\big((\boldsymbol{n}_\nu\cdot\boldsymbol{\beta}(\boldsymbol{m}_\nu))_+\mu_j + (\boldsymbol{n}_\nu\cdot\boldsymbol{\beta}(\boldsymbol{m}_\nu))_-\mu_\nu\big) = 0\,. \tag{11.8.14}$$

SOLUTION of (11-8.e):

Each cell caries a degree of freedom and can give rise to a non-zero diagonal entry of $\mathbf{B}$. From (11.8.14) we conclude that every edge contributes at most one non-zero off-diagonal entry of $\mathbf{B}$. Hence

$$\mathrm{nnz}(\mathbf{B}) \leq \sharp\mathcal{M} + \sharp\mathcal{E} \ .$$

From (11.8.14) we also learn that a row of $\mathbf{B}$ can contain at most 5 (= number of edges of a quadrilateral $+1$) non-zero entries. The same for every column of $\mathbf{B}$.

SOLUTION of (11-8.f):

There is exactly one global shape function associated with each cell.

```
lf::assemble::UniformFEDofHandler dof_handler(
      mesh_p, {{lf::base::RefEl::kPoint(), 0},
               {lf::base::RefEl::kSegment(), 0},
               {lf::base::RefEl::kTria(), 1},
               {lf::base::RefEl::kQuad(), 1}});
```

HINT 1 for (11-8.g): Writing λ_ℓ, $\ell = 1,2,3$, for the barycentric coordinates of a triangular cell K, we have the following formula for the exterior unit normals at its edges:

$$\boldsymbol{n}^j = -\frac{\mathbf{grad}\,\lambda_j}{\|\mathbf{grad}\,\lambda_j\|}\,, \tag{11.8.15}$$

see [Lecture → Fig. 76] for the numbering convention. Thus [Lecture → Code 2.4.5.11] can be used to compute the normals.

This trick can also be used for quadrilaterals after splitting them into two triangles. ⌟

SOLUTION of (11-8.g):

We implement a loop over all cells. For triangles we call `gradbarycoordinates()` from [Lecture → Code 2.4.5.11] with the vertex coordinates and normalize the columns of the matrix it gives. For quadrilaterals we do this twice for the vertices $1,2,3$ and $1,3,4$, making sure that we get the numbering right.

C++ code 11.8.16: Sub-problem (11-8.g): function `computeCellNormals()` → GitLab

```
2  std::shared_ptr<
3      lf::mesh::utils::CodimMeshDataSet<Eigen::Matrix<double, 2, Eigen::Dynamic>>>
4  computeCellNormals(std::shared_ptr<const lf::mesh::Mesh> mesh_p) {
5    // Appears only in mastersolution
6
7    // Initialize datastruture for the result
8    lf::mesh::utils::CodimMeshDataSet<Eigen::Matrix<double, 2, Eigen::Dynamic>>
9        result(mesh_p, 0);
10
11   // Compute normal vectors
12   for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
13     const lf::geometry::Geometry *geo_p = cell->Geometry();
14     const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);
15
16     if (corners.cols() == 3) {
17       Eigen::Matrix<double, 2, Eigen::Dynamic> normal_vectors(2, 3);
18
19       // Compute normal vectors
20       Eigen::Matrix<double, 2, 3> bary_coord = gradbarycoordinates(corners);
21
22       // Reorder computed normal vectors
23       // normal_vectors[0] -> normal_vector of edge[0]
24       normal_vectors.col(0) = -(bary_coord.col(2)).normalized();
25       normal_vectors.col(1) = -(bary_coord.col(0)).normalized();
26       normal_vectors.col(2) = -(bary_coord.col(1)).normalized();
27
28       // Save the matrix in our datastructure
29       result(*cell) = normal_vectors;
30     } else {  // corners.cols() == 4
31       Eigen::Matrix<double, 2, Eigen::Dynamic> normal_vectors(2, 4);
32
33       // Split the quadrilateral into two triangles (1,2,3) and (1,3,4)
34       Eigen::Matrix<double, 2, 3> tria_1;
35       Eigen::Matrix<double, 2, 3> tria_2;
36       tria_1 << corners.col(0), corners.col(1), corners.col(2);
37       tria_2 << corners.col(0), corners.col(2), corners.col(3);
38
39       // Compute normal vectors
40       Eigen::Matrix<double, 2, 3> bary_coord_1 = gradbarycoordinates(tria_1);
41       Eigen::Matrix<double, 2, 3> bary_coord_2 = gradbarycoordinates(tria_2);
42
43       // Reorder computed normal vectors
44       // normal_vectors[0] -> normal_vector of edge[0]
45       normal_vectors.col(0) = -(bary_coord_1.col(2)).normalized();
46       normal_vectors.col(1) = -(bary_coord_1.col(0)).normalized();
47       normal_vectors.col(2) = -(bary_coord_2.col(0)).normalized();
48       normal_vectors.col(3) = -(bary_coord_2.col(1)).normalized();
```

```
49
50          // Save the matrix in our datastructure
51          result(*cell) = normal_vectors;
52        }
53      }
54      return std::make_shared<lf::mesh::utils::CodimMeshDataSet<
55          Eigen::Matrix<double, 2, Eigen::Dynamic>>>(result);
56
57  }
```

The creation of a shared pointer should not be postponed to the end of the function. It could be done right when defining the `result` object.

HINT 1 for (11-8.h): You may initialize an auxiliary object of type

```
lf::mesh::utils::CodimMeshDataSet<std::array<const
   lf::mesh::Entity*,2>>
```

that provides the pointers to the one or two cells adjacent to any edge. ⌟

SOLUTION of (11-8.h):

We loop over the cells and first initialize the auxiliary array recommended in the hint. The we visit the cells once more and collect the pointers from the auxiliary array.

C++ code 11.8.17: Sub-problem (11-8.h): function `getAdjacentCellPointers()`
➥ GitLab

```
std::shared_ptr<
    If::mesh::utils::CodimMeshDataSet<std::array<const If::mesh::Entity *, 4>>>
getAdjacentCellPointers(std::shared_ptr<const If::mesh::Mesh> mesh_p) {
  // Initialize auxilary object
  If::mesh::utils::CodimMeshDataSet<std::array<const If::mesh::Entity *, 2>>
      aux_obj(mesh_p, 1, {nullptr, nullptr});

  // Iterate over every cell
  for (const If::mesh::Entity *cell : mesh_p->Entities(0)) {
    auto cell_edges = cell->SubEntities(1);

    // Iterate over every edge of the cell
    for (const If::mesh::Entity *edge : cell_edges) {
      // If aux_obj at index 0 was not set, save cell there
      // otherwise save at the second position
      // (The first position has to be set; the second might be set)
      if (aux_obj(*edge)[0] == nullptr) {
        aux_obj(*edge)[0] = cell;
      } else if (aux_obj(*edge)[1] == nullptr) {
        aux_obj(*edge)[1] = cell;
      } else {
        throw std::runtime_error("Error in aux_obj");
      }
    }
  }

  // Initialize datastructure for result
  If::mesh::utils::CodimMeshDataSet<std::array<const If::mesh::Entity *, 4>>
      result(mesh_p, 0, {nullptr, nullptr, nullptr, nullptr});

  // Collect the objects of the auxilary object
  for (const If::mesh::Entity *cell : mesh_p->Entities(0)) {
    auto cell_edges = cell->SubEntities(1);
    int counter = 0;
    for (const If::mesh::Entity *edge : cell_edges) {
      if (aux_obj(*edge)[0] != cell) {
        result(*cell)[counter] = aux_obj(*edge)[0];
      } else {
        result(*cell)[counter] = aux_obj(*edge)[1];
      }
      counter++;
    }
  }

  return std::make_shared<If::mesh::utils::CodimMeshDataSet<
      std::array<const If::mesh::Entity *, 4>>>(result);
}
```

HINT 1 for (11-8.i): Since from Sub-problem (11-8.e) we know reasonably sharp upper bounds for the maximal number of non-zero entries in the rows/columns of $\mathbf{B}$, we can directly initialize the sparse matrix after communicating a bound for $\mathrm{nnz}(\mathbf{B})$ via the `reserve()` member function ➙ documentation. ⌟

SOLUTION of (11-8.i):

We use the formula (11.8.14) inside a loop over the cells of the mesh.

SOLUTION of (11-9.a):

The unknowns for a conservative finite volume method are the averages over dual cells

$$\mu_j(t) \approx \frac{1}{h} \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t)\,\mathrm{d}x\,, \quad t \in [0,T]\,, \quad x_{j+\frac{1}{2}} := \frac{1}{2}(x_j + x_{j+1})\,.$$

This can be interpreted as the use of a *piecewise constant* approximation of $x \mapsto u(x,t)$ on a mesh dual to $\mathcal{M}$. In particular, we have $u(x_j,t) \approx \mu_j(t)$.

To derive the semi-discrete form we employ the considerations from [Lecture → Section 11.3.2], see [Lecture → Eq. (11.3.2.4)], [Lecture → Eq. (11.3.2.5)]. We integrate the PDE of (11.9.1) over a dual cell $[x_{j-1/2}, x_{j+1/2}]$, apply integration by parts, and replace the ambiguous boundary terms with the numerical flux:

$$\frac{d}{dt} \int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\,\mathrm{d}x + [f(u)]_{x_{j-1/2}}^{x_{j+1/2}} = \int_{x_{j-1/2}}^{x_{j+1/2}} s(u(x,t))\,\mathrm{d}x\,, \quad j \in \mathbb{Z}\,,$$

$$\blacktriangleright \quad h\frac{d\mu_j}{dt}(t) + F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t)) = hs(\mu_j(t))\,, \quad j \in \mathbb{Z}\,,$$

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big) + s(\mu_j(t))\,, \quad j \in \mathbb{Z}\,, \tag{11.9.2}$$

which is generalization of [Lecture → Eq. (11.3.2.8)].

SOLUTION of (11-9.b):

Starting from the code for `fluxdiff()`, all we habe to add is the extra term $s(\mu_j(t))$ in (11.9.2):

C++ code 11.9.3: Sub-problem (11-9.c): Implementation of `fluxdiffsource` ↬ GitLab

```
template <typename FunctionF, typename SourceFunction>
Eigen::VectorXd fluxdiffsource(const Eigen::VectorXd &mu, FunctionF &&F,
                               SourceFunction &&s, double h) {
  unsigned n = mu.size();
  Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);

  // constant continuation to the left by mu[0]
  fd[0] = F(mu[0], mu[1]) - F(mu[0], mu[0]) - h * s(mu[0]);
  for (unsigned j = 1; j < n - 1; ++j) {
    fd[j] = F(mu[j], mu[j + 1]) - F(mu[j - 1], mu[j]) - h * s(mu[j]);
  }
  // constant continuation to the right by mu[n - 1]
  fd[n - 1] =
      F(mu[n - 1], mu[n - 1]) - F(mu[n - 2], mu[n - 1]) - h * s(mu[n - 1]);

  return fd;
}
```

HINT 1 for (11-9.c): Adapt the idea of the proof of [Lecture → Lemma 11.3.5.9]. ⌟

SOLUTION of (11-9.c):

Since $\mu_j(0)$ are cell averages of u_0, we conclude that $\mu_j(0) \geq 0$.

Let $t_0 \in [0,T]$ be the *smallest time*, when there is a $k \in \mathbb{Z}$ such that $\mu_k(t_0) = 0$. Clearly we have $\mu_j(t_0) \geq 0$ for all $j \in \mathbb{Z}$. From the semi-discrete evolution (11.9.2) we get

$$\frac{d\mu_k}{dt}(t_0) = -\frac{1}{h}\big(F(0,\mu_{k+1}(t_0)) - F(\mu_{k-1}(t_0),0)\big) + s(0)\,. \tag{11.9.4}$$

The monotonicity of the numerical flux function combined with the fact that $\mu_{k+1}(t_0) \geq 0$ and $\mu_{k-1}(t_0) \geq 0$ implies

$$\text{[decreasing in second argument]:}\quad F(0,\underbrace{\mu_{k+1}(t_0)}_{\geq 0}) \leq F(0,0) = f(0)\,,$$

$$\text{[increasing in first argument]:}\quad F(\underbrace{\mu_{k-1}(t_0)}_{\geq 0},0) \geq F(0,0) = f(0)\,,$$

where the consistency of the numerical flux function gave us $F(0,0) = f(0)$. Hence, we infer from (11.9.4) and the assumption $s(0) = 0$ that

$$\frac{d\mu_k}{dt}(t_0) \geq -\frac{1}{h}\big(F(0,0) - F(0,0)\big) + 0 = 0\,.$$

This means that the value $\mu_k(t)$ cannot be decreasing at $t = t_0$. In particular, it cannot drop below zero.

SOLUTION of (11-9.d):

We have

$$f'(u) = \exp(u) - 1 \quad , \quad f''(u) = \exp(u) > 0 \, , \quad u \in \mathbb{R} \, .$$

Hence, the flux function f is *strictly convex*. We face a situation as in the case of Burgers equation and an upward jump of the data in the Riemann problem will give rise to a shock solution, the discontinuity moving with the Rankine-Hugoniot speed:

$$u(x,t) = \begin{cases} 1 & \text{for} \quad x < \dot{s}t \, , \\ -1 & \text{for} \quad x \geq \dot{s}t \, , \end{cases} \qquad \dot{s} = \frac{f(1) - f(-1)}{2} = \tfrac{1}{2}(e - e^{-1}) - 1 \, .$$

SOLUTION of (11-9.e):

Taking the cue from the insights gained for the Burgers equation we know that the entropy solution will be rarefaction wave. Its left/right edges will move with speeds

$$s_l := f'(-1) = e^{-1} - 1 < 0 \quad , \quad s_r := f'(1) = e - 1 > 0 \, ,$$

respectively. Hence, we face a transsonic rarefaction situation, *cf.* [Lecture $\rightarrow$ Ex. 11.3.4.22].

We have to recast the general formula of [Lecture $\rightarrow$ Lemma 11.2.5.10] for this special case. We have

$$g(v) := (f')^{-1}(v) = \log(1+v) \, , \quad v > -1 \, ,$$

so that

$$u(x,t) = \begin{cases} -1 & \text{for} \quad x < (e^{-1}-1)t \, , \\ \log(1+\frac{x}{t}) & \text{for} \quad (e^{-1}-1)t \le x \le (e-1)t \, , \\ 1 & \text{for} \quad x > (e-1)t \, , \end{cases} \qquad x \in \mathbb{R}, \; t > 0 \; .$$

SOLUTION of (11-9.f):

In [Lecture → § 11.3.4.32] we derived the following formulas for the Godunov numerical 2-point flux in the case of a *strictly convex* flux function f and $v < w$:

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) & \text{, if } f'(v) > 0\,, \\ f(z) & \text{, if } f'(v) < 0 < f'(w)\,, \\ f(w) & \text{, if } f'(w) < 0\,, \end{cases} \tag{11.9.9}$$

where z is the unique zero of f': $f'(z) = 0$.

In our concrete setting with $f'(u) = e^u - 1$, we have $z = 0$ and $f'(w) > 0 \quad \Leftrightarrow \quad w > 0$. Thus (11.9.9) becomes

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) & \text{, if } v > 0\,, \\ f(0) = 1 & \text{, if } v \le 0 \le w\,, \\ f(w) & \text{, if } w < 0\,, \end{cases} \tag{11.9.10}$$

In the case $v \ge w$ we learn from [Lecture → § 11.3.4.32]:

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) & \text{, if } \quad f(w) < f(v)\,, \\ f(w) & \text{, if } \quad f(w) \ge f(v)\,. \end{cases} \tag{11.9.11}$$

So we first test, whether $v < w$ and then implement either (11.9.10) or (11.9.11).

C++ code 11.9.12: Sub-problem (11-9.f): Implementation of `godnfn` ↬ GitLab

```
double godnfn(double v, double w) {
  auto f = [](double u) { return std::exp(u) - u; };
  if (v < w) {
    if (0.0 < v) return f(v);
    if (w < 0.0) return f(w);
    return 1.0;  // = f(0.0)
  }
  return f(w) < f(v) ? f(v) : f(w);

}
```

HINT 1 for (11-9.g):

1. As regards the CFL condition the source term is irrelevant.
2. We can expect $0 \leq u(x,t) \leq 1$ for all $(x,t) \in \mathbb{R} \times \mathbb{R}^+$.

HINT 2 for (11-9.g):

N	computed $m(3)$
10	0.00039
20	0.01353
40	0.03582
80	0.03688
160	0.04712
320	0.04696
640	0.04911
1280	0.04904

If you run the executable `ConsLawWithSource_mysolution` generated by the `make` command, it should produce the table of approximations of $m(3)$ shown beside.

⌟

SOLUTION of (11-9.g):

Since $0 \le u(x,t) \le 1$ for all $(x,t) \in \mathbb{R} \times \mathbb{R}^+$ the maximal speed of propagation in the concrete situation is

$$s_{\max} = \max\{|f'(0)|, |f'(1)|\} = e - 1 \approx 1.71828 \,.$$

Then we can apply the considerations of [Lecture → Section 11.4.2]. Since we rely on a two-point numerical flux and a standard conservative finite volume method we can apply [Lecture → Eq. (11.4.2.12)] with $m = 1$ and get the CFL timestep constraint

$$\tau \le \frac{h}{e-1} \,.$$

Since $h = 15/N$ we conclude that we can still choose the timestep $\tau = \frac{15}{N(e-1)}$. Of course, the timestep should be compatible with the duration $T = 3$ of the simulation so that we should use

$$M := \lceil 3/\tau \rceil$$

timesteps, where $\lceil \xi \rceil$ stands for the smallest integer $\ge \xi$.

C++ code 11.9.14: Sub-problem (11-9.g): Implementation of `traceMass()` ➙ GitLab

```
template <typename U0Functor>
Eigen::VectorXd traceMass(U0Functor &&u0, unsigned int N) {
  // Spacial boundaries
  double a = -5.0;
  double b = 10.0;

  // Compute μ(t = 0)
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, a, b);
  Eigen::VectorXd mu = x.unaryExpr(std::forward<U0Functor>(u0));

  // Get timestep size and number by CLF condition
  double h = (b - a) / N;
  double s_max = std::exp(1.0) - 1.0;  // since 0 ≤ u_0(x) ≤ 1
  double tau = h / s_max;              // stencil m = 1
  double T = 3.0;
  // Total number of timesteps
  int M = (int)std::ceil(T / tau);
  tau = 3.0 / M;  // Timestep size

  // Compute solution and total masses at different times
  auto totalMass = [h](const Eigen::VectorXd &mu) { return (mu * h).sum(); };
  Eigen::VectorXd m(M + 1);
  for (int i = 0; i < M; ++i) {
    m(i) = totalMass(mu);
    auto s = [](double u) { return -u; };
    mu = mu + tau * (-1.0 / h) * fluxdiffsource(mu, &godnfn, s, h);
  }
  m(M) = totalMass(mu);

  return m;
}
```

SOLUTION of (11-9.h):

Since u_0 has compact support and scalar conservation laws feature a finite speed of propagation, $u(\cdot,t)$ will have compact support for all times $t \geq 0$. Integrate the conservation law in space over $\mathbb{R}$ and notice that the contribution of the term $\frac{\partial}{\partial x} f(u)$ will vanish thanks to the fundamental theorem of calculus:

$$\int_{\mathbb{R}} \frac{\partial}{\partial x} f(u(x,t))\,\mathrm{d}x = \left[f(u(x,t))\right]_{x=-\infty}^{x=\infty} = 0\,.$$

We end up with

$$\frac{d}{dt}\int_{\mathbb{R}} u(x,t)\,\mathrm{d}x = \int_{\mathbb{R}} s(u(x,t))\,\mathrm{d}x \overset{(11.9.6)}{=} -\int_{\mathbb{R}} u(x,t)\,\mathrm{d}x\,.$$

Hence, the total mass solves the ODE

$$\frac{dm}{dt}(t) = -m(t) \quad \Rightarrow \quad m(t) = m(0)\exp(-t)\,, \quad t \geq 0\,.$$

Since $m(0) = 1$ we obtain $m(t) = e^{-t}$.

SOLUTION of (12-1.a):

Under the assumptions made above, the Stokes Pd Es reduce to

$$-v''(x_2) - p'(x_1) = 0\,, \quad 0 \le x_1, x_2 \le 1\,. \tag{12.1.2}$$

Note that the velocity field is divergence-free by its very form. By the boundary condition we known, in addition,

$$v(0) = v(1) = 0\,. \tag{12.1.3}$$

From (12.1.2) we conclude

$$\exists C \in \mathbb{R}\colon \quad -v''(x_2) = p'(x_1) = C \quad \forall 0 \le x_1, x_2 \le 1\,, \tag{12.1.4}$$

which implies, also using (12.1.3),

$$v(x_2) = \tfrac{1}{2} C\, x_2(1 - x_2) \quad , \quad p(x_1) = Cx_1 + c\,, \quad c \in \mathbb{R}\,. \tag{12.1.5}$$

This means that we have a *parabolic velocity profile* transversal to the pipe and a *linearly varying pressure* along the pipe:

$$\mathbf{v}(\boldsymbol{x}) = \tfrac{1}{2} C\, x_2(1 - x_2) \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad , \quad p(\boldsymbol{x}) = Cx_1\,, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \Omega\,, \quad C \in \mathbb{R}\,. \tag{12.1.6}$$

SOLUTION of (12-1.j):

In light of the discussion in Sub-problem (12-1.e) there are three global shape functions associated with every node (two velocity components and one for the pressure). Moreover, there are two global shape functions associated with every edge (two velocity components):

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(), 3},
  {lf::base::RefEl::kSegment(), 2},
  {lf::base::RefEl::kTria(), 0},
  {lf::base::RefEl::kQuad(), 0}});
```

SOLUTION of (12-1.k):

Apart from using **TaylorHoodElementMatrixProvider** the main job is to initialize the row and column of the Galerkin matrix introduced by the Lagrange multiplier for the zero mean constraint on the pressure.

In fact, the entries (12.1.29) can also be obtained by cell-oriented assembly using the element matrix

$$\frac{1}{3}|K|\begin{bmatrix}1 & 1 & 1\end{bmatrix} \in \mathbb{R}^{1,3} \tag{12.1.30}$$

for a triangle K. The problem is that the **lf::assemble::DofHandler** object is not aware of the extra unknown introduced by the Lagrange multiplier, which excludes using `AssembleMatrixLocally()` of LEHRFEM++. The cell-oriented assembly has to be done in dedicated loop over the cells.

C++ code 12.1.31: Sub-problem (12-1.k): Code for `buildTaylorHoodGalerkinMatrix()` ➙ GitLab

```
lf::assemble::COOMatrix<double> buildTaylorHoodGalerkinMatrix(
    const lf::assemble::DofHandler& dofh) {
  // Total number of FE d.o.f.s without Lagrangian multiplier
  lf::assemble::size_type n = dofh.NumDofs();
  // Full Galerkin matrix in triplet format taking into account the zero mean
  // constraint on the pressure.
  lf::assemble::COOMatrix<double> A(n + 1, n + 1);
  // Set up computation of element matrix
  TaylorHoodElementMatrixProvider themp{};
  // Assemble full Galerkin matrix for Taylor-Hood FEM
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, themp, A);

// Add bottom row and right column corresponding to Lagrange multiplier
// You cannot use AssembleMatrixLocally() because the DofHandler does
// not know about this extra unknown.
// Do cell-oriented assembly "manually"
  for (const lf::mesh::Entity* cell : dofh.Mesh()->Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Only implemented for triangles");
    // Obtain area of triangle
    const double area = lf::geometry::Volume(*cell->Geometry());
    // The pressure GSFs are associated with the nodes
    const std::span<const lf::mesh::Entity* const> nodes{cell->SubEntities(2)};
    // Loop over nodes
    for (const lf::mesh::Entity* node : nodes) {
      // Area of the cell
      // Obtain index of tent function associated with node
      // All indices of global shape functions sitting at node
      std::span<const lf::assemble::gdof_idx_t> dof_idx{
          dofh.InteriorGlobalDofIndices(*node)};
      LF_ASSERT_MSG(dof_idx.size() == 3, "Node must carry 3 dofs!");
      // The index of the tent function is the third one
      const lf::assemble::gdof_idx_t tent_idx = dof_idx[2];
      A.AddToEntry(n, tent_idx, area / 3.0);
      A.AddToEntry(tent_idx, n, area / 3.0);
    }
  }
  // Rely on return value optimization
  return A;
}
```

HINT 1 for (12-1.l): The main task to be accomplished in `solvePipeFlow()` is imposing the Dirichlet boundary condition. To that end follow [Lecture → § 2.7.6.13].

⌟

HINT 2 for (12-1.l): Of course, you should first invoke `buildTaylorHoodGalerkinMatrix()` from Sub-problem (12-1.k), which will give you an **lf::assemble::COOMatrix** object, on which LEHRFEM++ functions like `FixFlaggedSolutionComponents()` or `FixFlaggedSolutionCompAlt()` can operate. ⌟

SOLUTION of (12-1.l):

C++ code 12.1.32: Sub-problem (12-1.l): Code for `solvePipeFlow()` ⇝ GitLab

```
template <typename gFunctor>
Eigen::VectorXd solvePipeFlow(const lf::assemble::DofHandler &dofh,
                              gFunctor &&g, bool print = true) {
  // Number of d.o.f.  in FE spaces
  size_t n = dofh.NumDofs();
  if (print)
    std::cout << "Computing: solvePipeFlow() with N = " << n
              << ", assembling .. " << std::flush;
  // Obtain full Galerkin matrix in triplet format
  lf::assemble::COOMatrix<double> A{buildTaylorHoodGalerkinMatrix(dofh)};
  if (print) std::cout << "done. Imposing BDC ... " << std::flush;
  LF_VERIFY_MSG(A.cols() == A.rows(), "Matrix A must be square");
  // Auxiliary right-hnad side vector
  Eigen::VectorXd phi(A.cols());
  phi.setZero();
  // Impose Dirichlet boundary conditions
  const std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh.Mesh();
  // Flag any entity located on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p)};
  // Flag vector for d.o.f.  on the boundary
  std::vector<std::pair<bool, double>> ess_dof_select(n + 1, {false, 0.0});
  // Visit nodes on the boundary
  for (const lf::mesh::Entity *node : mesh_p->Entities(2)) {
    if (bd_flags(*node)) {
      // Indices of global shape functions sitting at node
      std::span<const lf::assemble::gdof_idx_t> dof_idx{
          dofh.InteriorGlobalDofIndices(*node)};
      LF_ASSERT_MSG(dof_idx.size() == 3, "Node must carry 3 dofs!");
      // Position of node
      const Eigen::Vector2d pos{Corners(*(node->Geometry())).col(0)};
      // Dirichlet data
      const Eigen::Vector2d g_val{g(pos)};
      // x-component of the velocity
      ess_dof_select[dof_idx[0]] = {true, g_val[0]};
      // y-component of the velocity
      ess_dof_select[dof_idx[1]] = {true, g_val[1]};
    }
  }
  // Visit edges on the boundasry
  for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
    if (bd_flags(*edge)) {
      // Indices of global shape functions associated with the edge
      std::span<const lf::assemble::gdof_idx_t> dof_idx{
          dofh.InteriorGlobalDofIndices(*edge)};
      LF_ASSERT_MSG(dof_idx.size() == 2, "Edge must carry 2 dofs!");
      // Midpoint of edge
      const Eigen::MatrixXd endpoints{Corners(*(edge->Geometry()))};
      const Eigen::Vector2d pos{0.5 * (endpoints.col(0) + endpoints.col(1))};
      // Dirichlet data
      const Eigen::Vector2d g_val{g(pos)};
      // x-component of the velocity
      ess_dof_select[dof_idx[0]] = {true, g_val[0]};
```

```
54          // y-component of the velocity
55          ess_dof_select[dof_idx[1]] = {true, g_val[1]};
56        }
57    }
58    // modify linear system of equations
59    lf::assemble::FixFlaggedSolutionComponents<double>(
60        [&ess_dof_select](lf::assemble::glb_idx_t dof_idx)
61            -> std::pair<bool, double> { return ess_dof_select[dof_idx]; },
62        A, phi);
63    // Assembly completed:  Convert COO matrix A into CRS format using
        Eigen's
64    // internal conversion routines.
65    if (print) std::cout << "done. Solving ..... " << std::flush;
66    const Eigen::SparseMatrix<double> A_crs = A.makeSparse();
67
68    // Solve linear system using Eigen's sparse direct elimination
69    // Examine return status of solver in case the matrix is singular
70    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
71    solver.compute(A_crs);
72    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
73    const Eigen::VectorXd dofvec = solver.solve(phi);
74    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
75    if (print) std::cout << "done. |dof vector| = " << dofvec.norm() << std::endl;
76    // This is the coefficient vector for the FE solution; Dirichlet
77    // boundary conditions are included
78    return dofvec;
79  }
```

SOLUTION of (12-1.m):

Thanks to linearity, we need only verify (12.1.34) for Cartesian unit vectors. Let us pick $\boldsymbol{d} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$. Then

$$\mathbf{v}_{\boldsymbol{d}}(\boldsymbol{x}) = \frac{1}{4\pi} \begin{bmatrix} -\frac{1}{2}\log(x_1^2 + x_2^2) + \frac{x_1^2}{x_1^2+x_2^2} \\ \frac{x_1 x_2}{x_1^2+x_2^2} \end{bmatrix} \quad , \quad p_{\boldsymbol{d}}(\boldsymbol{x}) = -\frac{1}{2\pi} \frac{x_1}{x_1^2 + x_2^2} \, .$$

We compute by "calculus I-style" differentiation using computational tools like SymPy, Maple, Mathematica or Wolfram Alpha[1]:

$$\begin{aligned} \Delta \left\{ \boldsymbol{x} \mapsto \frac{x_1^2}{x_1^2 + x_2^2} \right\} &= 2 \, \frac{x_2^2 - x_1^2}{(x_1^2 + x_2^2)^2} \, , \\ \Delta \left\{ \boldsymbol{x} \mapsto \frac{x_1 x_2}{x_1^2 + x_2^2} \right\} &= -4 \, \frac{x_1 x_2}{(x_1^2 + x_2^2)^2} \, , \qquad \text{for} \quad \boldsymbol{x} \neq \mathbf{0} \, . \\ \Delta \left\{ \boldsymbol{x} \mapsto \log(x_1^2 + x_2^2) \right\} &= 0 \, , \\ \mathbf{grad} \left\{ \boldsymbol{x} \mapsto \frac{x_1}{x_1^2 + x_2^2} \right\} &= \left[\frac{x_2^2 - x_1^2}{(x_1^2 + x_2^2)^2} \, , \, -2 \, \frac{x_1 x_2}{(x_1^2 + x_2^2)^2} \right] \, , \end{aligned}$$

[1] WolframAlpha Laplacian Calculator and WolframAlpha Gradient widget

HINT 1 for (12-1.n): You should rely on LEHRFEM++'s built-in tools for computing error norms, see [Lecture → Ex. 3.2.3.4]. To that end you have to set up appropriate **lf::fe::MeshFunctionFE** and **lf::fe::MeshFunctionGradFE** objects, which can only be done, if you *extract appropriate parts* of the vector of basis expansion coefficients of the Taylor-Hood FE solution. ⌟

SOLUTION of (12-1.n):

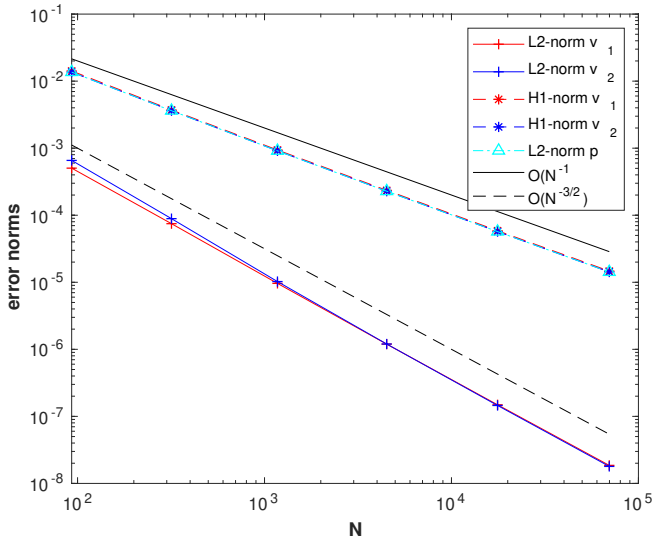
See the comments in the code for explanations.

C++ code 12.1.35: Sub-problem (12-1.n): Code for `testCvgTaylorHood()` ⇝ GitLab

```
void testCvgTaylorHood(unsigned int refsteps) {
  // Analytic solution for velocity and pressure
  Eigen::Vector2d d{2.0, 1.0};
  auto v_ex = [&d](Eigen::Vector2d x) -> Eigen::Vector2d {
    x += Eigen::Vector2d(0.5, 0.5);
    const double nx = x.norm();
    return 0.25 * (-std::log(nx) * d + x * (x.dot(d)) / (nx * nx));
  };
  auto p_ex = [&d](Eigen::Vector2d x) -> double {
    x += Eigen::Vector2d(0.5, 0.5);
    return -0.5 * x.dot(d) / x.squaredNorm() + (d[0] + d[1]) * 0.5 * 0.502128;
  };
  auto grad_v1 = [&d](Eigen::Vector2d x) -> Eigen::Vector2d {
    x += Eigen::Vector2d(0.5, 0.5);
    const double x2 = x[0] * x[0];
    const double x3 = x2 * x[0];
    const double y2 = x[1] * x[1];
    const double den = (x2 + y2) * (x2 + y2);
    return 0.25 * d[0] *
               Eigen::Vector2d{(-x3 + x[0] * y2) / den,
                               -x[1] * (3 * x2 + y2) / den} +
           0.25 * d[1] *
               Eigen::Vector2d{x[1] * (-x2 + y2) / den, x[0] * (x2 - y2) / den};
  };
  auto grad_v2 = [&d](Eigen::Vector2d x) -> Eigen::Vector2d {
    x += Eigen::Vector2d(0.5, 0.5);
    const double x2 = x[0] * x[0];
    const double x3 = x2 * x[0];
    const double y2 = x[1] * x[1];
    const double den = (x2 + y2) * (x2 + y2);
    return 0.25 * d[1] *
               Eigen::Vector2d{-x[0] * (x2 + 3 * y2) / den,
                               x[1] * (x2 - y2) / den} +
           0.25 * d[0] *
               Eigen::Vector2d{x[1] * (-x2 + y2) / den, x[0] * (x2 - y2) / den};
  };

  // Generate a small unstructured triangular mesh of the unit square
  const std::shared_ptr<lf::mesh::Mesh> mesh_ptr =
      lf::mesh::test_utils::GenerateHybrid2DTestMesh(3, 1.0 / 3.0);
  // Generate sequence of meshes by uniform regular refinement
  const std::shared_ptr<lf::refinement::MeshHierarchy> multi_mesh_p =
      lf::refinement::GenerateMeshHierarchyByUniformRefinemnt(mesh_ptr,
                                                              refsteps);
  lf::refinement::MeshHierarchy& multi_mesh{*multi_mesh_p};
  // Ouput summary information about hierarchy of nested meshes
  std::cout << "\t Sequence of nested meshes created\n";
  multi_mesh.PrintInfo(std::cout);

  // Number of levels
```

```cpp
  const int L = multi_mesh.NumLevels();

  // Table of various error norms
  std::vector<std::tuple<size_t, double, double, double, double, double>> errs;
  // Loop over the levels
  for (int level = 0; level < L; ++level) {
    const std::shared_ptr<const lf::mesh::Mesh> lev_mesh_p =
        multi_mesh.getMesh(level);
    // Initialize dof handler for Taylor-Hood FEM
    lf::assemble::UniformFEDofHandler dofh(lev_mesh_p,
                                           {{lf::base::RefEl::kPoint(), 3},
                                            {lf::base::RefEl::kSegment(), 2},
                                            {lf::base::RefEl::kTria(), 0},
                                            {lf::base::RefEl::kQuad(), 0}});
    // Define first- and second-order Lagrangian FE spaces for the
       piecewise
    // linear Taylor-Hood pressure approximation and the piecewise
       quadratic
    auto fes_o1_ptr =
        std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(lev_mesh_p);
    auto fes_o2_ptr =
        std::make_shared<lf::uscalfe::FeSpaceLagrangeO2<double>>(lev_mesh_p);
    // Fetch dof handler for the components of the velocity
    const lf::assemble::DofHandler& dofh_u = fes_o2_ptr->LocGlobMap();
    // Fetch dof handler for the pressure p
    const lf::assemble::DofHandler& dofh_p = fes_o1_ptr->LocGlobMap();
    LF_ASSERT_MSG(dofh.NumDofs() == 2 * dofh_u.NumDofs() + dofh_p.NumDofs(),
                  "No dof mismatch");
    // Solve the system with trace of the exact velocity solution as
       Dirichlet
    // data
    Eigen::VectorXd res = StokesPipeFlow::solvePipeFlow(dofh, v_ex);
    // Coefficient vectors for the first and second component of the
       velocity
    Eigen::VectorXd coeff_vec_u1 = Eigen::VectorXd::Zero(dofh_u.NumDofs());
    Eigen::VectorXd coeff_vec_u2 = Eigen::VectorXd::Zero(dofh_u.NumDofs());
    // Coefficient vector for the pressure
    Eigen::VectorXd coeff_vec_p = Eigen::VectorXd::Zero(dofh_p.NumDofs());

    // Remapping dofs:  We first loop over vertices, then over edges
    for (int codim = 2; codim >= 1; codim--) {
      for (auto e : lev_mesh_p->Entities(codim)) {
        // Global indices for u1, u2 for the respective vertex or edge
        auto glob_idxs = dofh.InteriorGlobalDofIndices(*e);
        auto glob_idx_o2 = dofh_u.InteriorGlobalDofIndices(*e)[0];
        // Extract the correct elements for the coefficient matrix of the
        // components of u
        coeff_vec_u1[glob_idx_o2] = res[glob_idxs[0]];
        coeff_vec_u2[glob_idx_o2] = res[glob_idxs[1]];
        // The pressure is only defined on vertices
        if (codim == 2) {
          // Global indices for p for the respective vertex
          lf::assemble::gdof_idx_t glob_idx_o1 = dofh_p.GlobalDofIndices(*e)[0];
          coeff_vec_p(glob_idx_o1) = res[glob_idxs[2]];
        }
      }
    }
    // Variables for storing the error norms
    double L2err_u1, L2err_u2, H1err_u1, H1err_u2, L2err_p;
```

```
    // Define finite-element mesh functions
    const lf::fe::MeshFunctionFE mf_o2_u1(fes_o2_ptr, coeff_vec_u1);
    const lf::fe::MeshFunctionFE mf_o2_u2(fes_o2_ptr, coeff_vec_u2);
    const lf::fe::MeshFunctionFE mf_o1_p(fes_o1_ptr, coeff_vec_p);
    const lf::fe::MeshFunctionGradFE mf_o2_grad_u1(fes_o2_ptr, coeff_vec_u1);
    const lf::fe::MeshFunctionGradFE mf_o2_grad_u2(fes_o2_ptr, coeff_vec_u2);

    // Exact solution for the first component of the velocity
    auto u1 = [&v_ex](Eigen::Vector2d x) -> double { return v_ex(x)[0]; };
    const lf::mesh::utils::MeshFunctionGlobal mf_u1{u1};
    // Exact solution for the gradient of v_1
    const lf::mesh::utils::MeshFunctionGlobal mf_grad_u1{grad_v1};
    // Exact solution second component of the velocity
    auto u2 = [&v_ex](Eigen::Vector2d x) -> double { return v_ex(x)[1]; };
    const lf::mesh::utils::MeshFunctionGlobal mf_u2{u2};
    // Exact solution for the gradient of v_2
    const lf::mesh::utils::MeshFunctionGlobal mf_grad_u2{grad_v2};
    // Mesh function for exact solution pressure
    const lf::mesh::utils::MeshFunctionGlobal mf_p{p_ex};
    // compute errors with 5th order quadrature rules
    L2err_u1 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_u1 - mf_u1), 4));
    L2err_u2 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_u2 - mf_u2), 4));
    H1err_u1 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_grad_u1 - mf_grad_u1),
        4));
    H1err_u2 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_grad_u2 - mf_grad_u2),
        4));
    L2err_p = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o1_p - mf_p), 4));
    errs.emplace_back(dofh.NumDofs(), L2err_u1, L2err_u2, H1err_u1, H1err_u2,
                      L2err_p);
  }
  // Output table of errors to file and terminal
  std::ofstream out_file("errors.txt");
  std::cout.precision(3);
  std::cout << std::endl
            << std::left << std::setw(10) << "N" << std::right << std::setw(16)
            << "L2 err(v1)" << std::setw(16) << "L2 err(v2)" << std::setw(16)
            << "H1 err(v1)" << std::setw(16) << "H1 err(v2)" << std::setw(16)
            << "L2 err(p)" << '\n';
  std::cout << "-----------------------------------------" << '\n';
  for (const auto& err : errs) {
    auto [N, L2err_u1, L2err_u2, H1err_u1, H1err_u2, L2err_p] = err;
    out_file << std::left << std::setw(10) << N << std::left << std::setw(16)
             << L2err_u1 << std::setw(16) << L2err_u2 << std::setw(16)
             << H1err_u1 << std::setw(16) << H1err_u2 << std::setw(16)
             << L2err_p << '\n';
    std::cout << std::left << std::setw(10) << N << std::left << std::setw(16)
              << L2err_u1 << std::setw(16) << L2err_u2 << std::setw(16)
              << H1err_u1 << std::setw(16) << H1err_u2 << std::setw(16)
              << L2err_p << '\n';
  }
}
```



The plotted error norms neatly lie on (almost) straight lines in a doubly logarithmic plot versus the number N of unknowns, which is strong evidence of **asymptotic algebraic convergence**.

Inspecting the slopes of the lines we observe

$$\begin{aligned} \|\mathbf{v}-\mathbf{v}_h\|_{L^2(\Omega)} &\sim O(N^{-3/2})\,, \\ |\mathbf{v}-\mathbf{v}_h|_{H^1(\Omega)} &\sim O(N^{-1})\,, \quad \text{for} \quad N\to\infty\,. \\ \|p-p_h\|_{L^2(\Omega)} &\sim O(N^{-1}) \end{aligned} \tag{12.1.36}$$

In terms of the mesh width $h \sim N^{-1/2}$ this means

$$\begin{aligned} \|\mathbf{v}-\mathbf{v}_h\|_{L^2(\Omega)} &\sim O(h^3)\,, \\ |\mathbf{v}-\mathbf{v}_h|_{H^1(\Omega)} &\sim O(h^2)\,, \quad \text{for} \quad h\to 0\,. \\ \|p-p_h\|_{L^2(\Omega)} &\sim O(h^2) \end{aligned} \tag{12.1.37}$$

This very well agrees with the observations made in [Lecture → Ex. 12.3.4.5].

HINT 1 for (12-1.o): The key LEHRFEM++ class is **lf::io::VtkWriter**, which can write a **lf::fe::MeshFunct** to a `.vtk`-file. Study its documentation for details. ⌟

SOLUTION of (12-1.o):

Since building an **lf::fe::MeshFunctionFE** object requires a scalar-valued finite element space, we have to set up auxiliary objects of type **lf::uscalfe::FeSpaceLagrangeO1** (for the pressure) and **lf::uscalfe::FeSpa** (for the components of the velocity).

Of course, we cannot take for granted anything about the numbering of the global shape functions managed by the **lf::assemble::DofHandler**s of those objects. This is why we have to remap the basis expansion coefficients.

C++ code 12.1.38: Sub-problem (12-1.o): Code for writing .vtk file. ⇝ GitLab

```
2   auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
3   lf::io::GmshReader reader(std::move(mesh_factory), meshfile_str);
4
5   const std::shared_ptr<const lf::mesh::Mesh> mesh_ptr = reader.mesh();
6   const lf::mesh::Mesh& mesh{*mesh_ptr};
7
8   // Initialize dof handler for Taylor-Hood FEM
9   lf::assemble::UniformFEDofHandler dofh(mesh_ptr,
10                                         {{lf::base::RefEl::kPoint(), 3},
11                                          {lf::base::RefEl::kSegment(), 2},
12                                          {lf::base::RefEl::kTria(), 0},
13                                          {lf::base::RefEl::kQuad(), 0}});
14  // We define function providing the boundary values for the velocity
15  auto g = [](Eigen::Vector2d x) -> Eigen::Vector2d {
16    if ((x[0] < 1E-8) and (x[1] >= 0.5) and (x[1] <= 1.0)) {
17      // Left boundary:  inlet, parabolic velocity profile
18      return {(1.0 - x[1]) * (x[1] - 0.5), 0.0};
19    }
20    if ((x[0] > 0.99999) and (x[1] >= 0.0) and (x[1] <= 0.5)) {
21      // Right boundary:  outlet
22      return {(0.5 - x[1]) * x[1], 0.0};
23    }
24    return {0.0, 0.0};
25  };
26  // Solve the system
27  Eigen::VectorXd res = StokesPipeFlow::solvePipeFlow(dofh, g);
28
29  double p_diss = 0.0;
30  switch (powerflag) {
31    case VOLUME: {
32      p_diss = compDissPowVolume(dofh, res);
33      break;
34    }
35    case BOUNDARY: {
36      p_diss = compDissPowBd(dofh, res);
37      break;
38    }
39    default: {
40      std::cout << "Dissipated power not computed\n";
41    }
42  }
43
44  std::cout << "Dissipated power: " << p_diss << std::endl;
45
```

```
46  if (producevtk) {
47    LF_VERIFY_MSG(outfile != nullptr,
48                  "Filename for .vtk files has to be provided");
49    // Define first- and second-order Lagrangian FE spaces for the
        piecewise
50    // linear Taylor-Hood pressure approximation and the piecewise
        quadratic
51    auto fes_o1_ptr =
52        std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_ptr);
53    auto fes_o2_ptr =
54        std::make_shared<lf::uscalfe::FeSpaceLagrangeO2<double>>(mesh_ptr);
55    // Fetch dof handler for the components of the velocity
56    const lf::assemble::DofHandler& dofh_u = fes_o2_ptr->LocGlobMap();
57    // Fetch dof handler for the pressure p
58    const lf::assemble::DofHandler& dofh_p = fes_o1_ptr->LocGlobMap();
59
60    // Coefficient vectors for the first and second component of the
        velocity
61    Eigen::VectorXd coeff_vec_u1(dofh_u.NumDofs());
62    Eigen::VectorXd coeff_vec_u2(dofh_u.NumDofs());
63    // Coefficient vector for the pressure
64    Eigen::VectorXd coeff_vec_p(dofh_p.NumDofs());
65
66    // Loop over vertices and edges to remap the basis expansion
        coefficients
67    for (int codim = 2; codim >= 1; codim--) {
68      for (auto e : mesh.Entities(codim)) {
69        // Global indices for u1, u2 for the respective vertex or edge
70        auto glob_idxs = dofh.GlobalDofIndices(*e);
71        auto glob_idx_o2 = dofh_u.GlobalDofIndices(*e)[0];
72
73        // Extract the correct elements for the coefficient matrix of the
74        // components of u
75        coeff_vec_u1(glob_idx_o2) = res(glob_idxs[0]);
76        coeff_vec_u2(glob_idx_o2) = res(glob_idxs[1]);
77
78        // Global indices for p for the respective vertex
79        lf::assemble::gdof_idx_t glob_idx_o1 = -1;
80        // The pressure is only defined on vertices
81        if (codim == 2) {
82          glob_idx_o1 = dofh_p.GlobalDofIndices(*e)[0];
83          coeff_vec_p(glob_idx_o1) = res(glob_idxs[2]);
84        }
85      }
86    }
87
88    // Define finite-element mesh functions
89    lf::fe::MeshFunctionFE<double, double> mf_o2_u1(fes_o2_ptr, coeff_vec_u1);
90    lf::fe::MeshFunctionFE<double, double> mf_o2_u2(fes_o2_ptr, coeff_vec_u2);
91    lf::fe::MeshFunctionFE<double, double> mf_o1_p(fes_o1_ptr, coeff_vec_p);
92
93    const std::string outfile_str = std::string(outfile);
94    std::cout << outfile_str << std::endl;
95    lf::io::VtkWriter vtk_writer(mesh_ptr, outfile_str);
96    vtk_writer.WritePointData("u1", mf_o2_u1);
97    vtk_writer.WritePointData("u2", mf_o2_u2);
98    vtk_writer.WritePointData("p", mf_o1_p);
99  }
```

SOLUTION of (12-1.p):

The volume-based formula for the dissipated power in the setting of this project is

$$P_{\text{diss}} = \int_{\Omega} |\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}|^2 \, \mathrm{d}\boldsymbol{x} \,, \tag{12.1.39}$$

into which we plug the finite element solution $\mathbf{v}_h \in \mathcal{S}_2^0(\mathcal{M})$.

The policy of the implementation is the same as for the member function `Eval()` in Sub-problem (12-1.h). We encapsulate the gradient of the local shape functions of $\mathcal{S}_2^0(\mathcal{M})$ in lambda functions and use the edge midpoint quadrature rule for the exact evaluation of the local integrals.

C++ code 12.1.40: Sub-problem (12-1.p): Volume formula for dissipated power ➙ GitLab

```
double compDissPowVolume(const lf::assemble::DofHandler& dofh,
                         const Eigen::VectorXd& mu_vec) {
  // First fetch underlying mesh
  const lf::mesh::Mesh& mesh{*dofh.Mesh()};
  // Summation variable
  double p_diss{0.0};
  // Loop over all cells and add up contributions of local integrals to
     the
  // dissipated power.
  for (const lf::mesh::Entity* cell : mesh.Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Only implemented for triangles");
    // Obtain coefficients for local shape functions
    // Local indices for LSFs fpr velocity components
    std::array<Eigen::Index, 6> vx_idx{0, 3, 6, 9, 11, 13};
    std::array<Eigen::Index, 6> vy_idx{1, 4, 7, 10, 12, 14};
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    // Workaround for a side-effect in LehrFEM++
    // Local shape functions associated with edges have to be swapped,
       if the
    // edge has "negative" orientation with respect to the triangle.
       This is
    // meant to offset such a swapping done by LehrFEM++'s
       UniformFEDofHandler.
    auto edge_orientations = cell->RelativeOrientations();
    LF_ASSERT_MSG(edge_orientations.size() == 3,
                  "Triangle should have 3 edges!?");
    for (int k = 0; k < 3; ++k) {
      if (edge_orientations[k] == lf::mesh::Orientation::negative) {
        // The rows and columns of the element matrix with numbers 9+2*k
           and
        // 9+2*k+1 have to be swapped.
        std::swap(vx_idx[3 + k], vy_idx[3 + k]);
      }
    }
    // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
    std::span<const lf::assemble::gdof_idx_t> dof_idx{
        dofh.GlobalDofIndices(*cell)};
    LF_ASSERT_MSG(dof_idx.size() == 15, "Taylor-Hood FEM involves 15 LSFs!");

    // Copied from Eval(); Found no good way of moving this to a
       function
    // Area of the triangle
    double area = lf::geometry::Volume(*cell->Geometry());
```

```
    // Compute gradients of barycentric coordinate functions, see
    // [Lecture → Code 2.4.5.11] Get vertices of the triangle
    auto endpoints = lf::geometry::Corners(*(cell->Geometry()));
    Eigen::Matrix<double, 3, 3> X;  // temporary matrix
    X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
    X.block<3, 2>(0, 1) = endpoints.transpose();
    // This matrix contains grad λ_i in its columns
    const auto G{X.inverse().block<2, 3>(1, 0)};
    // Gradients of local shape functions of quadratic Lagrangian finite
      element
    // space as lambda functions, see (12.1.24)
    std::array<std::function<Eigen::Vector2d(Eigen::Vector3d)>, 6> gradq{
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return (4 * c[0] - 1) * G.col(0);
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return (4 * c[1] - 1) * G.col(1);
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return (4 * c[2] - 1) * G.col(2);
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return 4 * (c[0] * G.col(1) + c[1] * G.col(0));
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return 4 * (c[1] * G.col(2) + c[2] * G.col(1));
        },
        [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
          return 4 * (c[2] * G.col(0) + c[0] * G.col(2));
        }};
    // Barycentric coordinates of the midpoints of the edges for
    // use with the 3-point edge midpoint quadrature rule (12.1.25)
    // Note that we only integrate a quadratic polynomial so that the
      edge
    // midpoint quadrature rule evaluates the integral exactly.
    const std::array<Eigen::Vector3d, 3> mp = {Eigen::Vector3d({0.5, 0.5, 0}),
                                               Eigen::Vector3d({0, 0.5, 0.5}),
                                               Eigen::Vector3d({0.5, 0, 0.5})};
    // We have to integrate |∂v_2/∂x_1 − ∂v_1/∂x_2|^2.
    // Loop over quadrature points
    double s = 0.0;
    for (int qp_idx = 0; qp_idx < 3; ++qp_idx) {
      // Compute grad v_1 and grad v_2 quadrature point
      Eigen::Vector2d grad_vx{0.0, 0.0};
      Eigen::Vector2d grad_vy{0.0, 0.0};
      for (int i = 0; i < 6; ++i) {
        grad_vx += mu_vec[dof_idx[vx_idx[i]]] * gradq[i](mp[qp_idx]);
        grad_vy += mu_vec[dof_idx[vy_idx[i]]] * gradq[i](mp[qp_idx]);
      }
      double curl_qp = grad_vy[0] - grad_vx[1];
      s += curl_qp * curl_qp;
    }
    p_diss += (area / 3.0 * s);
  }
  return p_diss;
}
```

SOLUTION of (12-1.q):

We already know the normal vector at those parts of the boundary, where the velocity does not vanish:

$$\text{On} \quad \Gamma_{\text{in}} \quad \boldsymbol{n}(\boldsymbol{x}) = \begin{bmatrix} -1 \\ 0 \end{bmatrix} \quad , \quad \text{on} \quad \Gamma_{\text{out}} \quad \boldsymbol{n}(\boldsymbol{x}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} .$$

So we have to add up integrals

$$\pm \int_e (\mathbf{v}_h)_1(\boldsymbol{x}) p_h(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) \, , \quad e \mathrel{\hat{=}} \text{mesh edge on } \Gamma_* \, .$$

If the two values of p_h in the endpoints of the edge e are denoted by p_0, p_1, and the values of $(\mathbf{v}_h)_1$ at the endpoints and in the midpoint are v_0, v_1, v_m, then

$$\int_e (\mathbf{v}_h)_1(\boldsymbol{x}) p_h(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) = \begin{bmatrix} p_0 & p_1 \end{bmatrix} \begin{bmatrix} \frac{1}{6} & \frac{1}{3} & 0 \\ 0 & \frac{1}{3} & \frac{1}{6} \end{bmatrix} \begin{bmatrix} v_0 \\ v_m \\ v_1 \end{bmatrix} . \tag{12.1.41}$$

The values p_0, p_1, v_0, v_1, v_m can directly be accessed as basis expansion coefficients through the global indices of the local shape functions *covering* the edge e.

C++ code 12.1.42: Sub-problem (12-1.q): Boundary integral formula for dissipated power
➥ GitLab

```
2  double compDissPowBd(const lf::assemble::DofHandler& dofh,
3                       const Eigen::VectorXd& muvec, bool print) {
4    // First fetch underlying mesh
5    const lf::mesh::Mesh& mesh{*dofh.Mesh()};
6    // Summation variable
7    double p_diss{0.0};
8  // Loop over the edges and check whether they are located on the inlet
       x_1 = 0
9  // or outlet x_1 = 1.
10   const Eigen::Matrix<double, 2, 3> M{
11       (Eigen::Matrix<double, 2, 3>(2, 3) << 1.0 / 6.0, 1.0 / 3.0, 0.0, 0.0,
12        1.0 / 3.0, 1.0 / 6.0)
13           .finished()};
14   for (const lf::mesh::Entity* edge : mesh.Entities(1)) {
15     // Length of edge
16     const double length = lf::geometry::Volume(*(edge->Geometry()));
17     const Eigen::MatrixXd endpoints{Corners(*(edge->Geometry()))};
18     const Eigen::Vector2d mp{0.5 * (endpoints.col(0) + endpoints.col(1))};
19     // Check by location whether the edge is on the inlet or outlet
20     int locflag = 0;
21     if (mp[0] < 1E-8) {
22       // On inlet boundary
23       locflag = -1;
24     } else if (mp[0] > 1 - 1E-8) {
25       locflag = 1;
26     }
27     if (locflag != 0) {
28       // Obtain indices of global shape functions associated with the
           edge
29       std::span<const lf::assemble::gdof_idx_t> dof_idx{
30           dofh.GlobalDofIndices(*edge)};
```

```
31        LF_ASSERT_MSG(dof_idx.size() == 8,
32                      "For TH FEM an edge should be covered by 8 GSFs");
33        // The pressure dofs sit in the nodes as third node dof.
34        // These are edge-local shape functions #2 and #5 (C++ indexing)
35        Eigen::Vector2d p_ldof{muvec[dof_idx[2]], muvec[dof_idx[5]]};
36        // The velocity x-component dofs sit in nodes and edges and are
              numbered
37        // first there.  Their edge-local numbers are #0, #3, and #6
38        Eigen::Vector3d vx_ldof{muvec[dof_idx[0]], muvec[dof_idx[6]],
39                                muvec[dof_idx[3]]};
40        // Evaluate the integral; can be done exactly, because the
              integrand is
41        // polynomial
42        const double loc_diss = locflag * length * p_ldof.dot(M * vx_ldof);
43        if (print) {
44          std::cout << "DPB: Edge with midpoint [" << mp.transpose()
45                    << "] : lf = " << locflag
46                    << ", * pvals = " << p_ldof.transpose()
47                    << ", vx_ldof = " << vx_ldof.transpose()
48                    << ", loc_diss = " << loc_diss << std::endl;
49        }
50        p_diss += loc_diss;
51      }
52    }
53    return p_diss;
54  }
```

HINT 1 for (12-1.b): Take into account the incompressibility constraint $\operatorname{div} \mathbf{v} = 0$ and remember Gauss' theorem [Lecture $\rightarrow$ Thm. 1.5.2.4]. ⌟

SOLUTION of (12-1.b):

By Gauss' theorem

$$\operatorname{div} \mathbf{v} = 0 \quad \blacktriangleright \quad \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) = 0 \,, \tag{12.1.9}$$

which means, since $\mathbf{v}|_{\Gamma_0} = 0$, $\partial\Omega = \overline{\Gamma}_0 \cup \overline{\Gamma}_{\text{in}} \cup \overline{\Gamma}_{\text{out}}$, and $\boldsymbol{n}|_{\Gamma_{\text{in}}} = \left[\begin{smallmatrix} -1 \\ 0 \end{smallmatrix}\right]$, $\boldsymbol{n}|_{\Gamma_{\text{out}}} = \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right]$,

$$\int_{\Gamma_{\text{in}}} \mathbf{v}(\boldsymbol{x}) \cdot \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] \mathrm{d}S\boldsymbol{x} = \int_{\Gamma_{\text{out}}} \mathbf{v}(\boldsymbol{x}) \cdot \left[\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right] \mathrm{d}S\boldsymbol{x} \,. \tag{12.1.10}$$

$$\Leftrightarrow \quad \int_{a_i}^{b_i} (t - a_i)(b_i - t) \, \mathrm{d}t = v_0 \int_{a_o}^{b_o} (t - a_o)(b_o - t) \, \mathrm{d}t$$

$$\Leftrightarrow \quad \tfrac{1}{6}(b_i - a_i)^3 = v_0 \, \tfrac{1}{6}(b_o - a_o)^3 \quad \Rightarrow \quad v_0 = \frac{(b_i - a_i)^3}{(b_o - a_o)^3} \,. \tag{12.1.11}$$

HINT 1 for (12-1.c): As in the proof of [Lecture → Lemma 12.2.1.2] use integration by parts based on Green's formula [Lecture → Thm. 1.5.2.7]. Boundary terms will be important this time, because $\mathbf{v}$ need not vanish on $\partial\Omega$. ⌟

SOLUTION of (12-1.c):

From Greens first formula

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \tag{12.1.13}$$

with $\mathbf{j}(\boldsymbol{x}) = u(\boldsymbol{x})\boldsymbol{e}_i$, $\boldsymbol{e}_i \hat{=}$ i-th Cartesian coordinate vector, $i = 1,2$, we obtain the integration by parts formula for simple partial derivatives ($\boldsymbol{n} = \left[\begin{smallmatrix} n_1 \\ n_2 \end{smallmatrix}\right]$),

$$\int_\Omega u(\boldsymbol{x}) \frac{\partial v}{\partial x_i}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \frac{\partial u}{\partial x_i}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u(\boldsymbol{x}) v(\boldsymbol{x})\, n_i(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x})\,. \tag{12.1.14}$$

We apply this integration by parts formula to the expression for the dissipated power using the definition of the scalar-valued 2D rotation operator:

$$\begin{aligned} P_{\mathrm{diss}} &= \int_\Omega \left(\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}\right)\left(\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}\right) \mathrm{d}\boldsymbol{x} \\ &= \int_\Omega \begin{bmatrix} -\frac{\partial}{\partial x_2}\left(\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}\right) \\ \frac{\partial}{\partial x_1}\left(\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}\right) \end{bmatrix} \cdot \mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \left(\frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1}\right) \underbrace{(v_1 n_2 - v_2 n_1)}_{=0 \text{ if } \mathbf{v} \| \boldsymbol{n} \text{ on } \partial\Omega} \mathrm{d}S(\boldsymbol{x}) \end{aligned}$$

The incompressibility condition $\operatorname{div} \mathbf{v} = 0$ permits us to introduce the vector Laplacian,

$$\frac{\partial v_1}{\partial x_1} + \frac{\partial v_2}{\partial x_2} = 0 \quad \Rightarrow \quad \begin{bmatrix} -\frac{\partial}{\partial x_2}\left(\frac{\partial v_1}{\partial x_2}(\boldsymbol{x}) - \frac{\partial v_2}{\partial x_1}(\boldsymbol{x})\right) \\ \frac{\partial}{\partial x_1}\left(\frac{\partial v_1}{\partial x_2}(\boldsymbol{x}) - \frac{\partial v_2}{\partial x_1}(\boldsymbol{x})\right) \end{bmatrix} = -\boldsymbol{\Delta}\mathbf{v}(\boldsymbol{x})\,, \tag{12.1.15}$$

which yields

$$\begin{aligned} P_{\mathrm{diss}} &= \int_\Omega -\boldsymbol{\Delta}\mathbf{v}(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \overset{(12.1.1a)}{=} \int_\Omega \mathbf{grad}\, p(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \\ &= -\int_\Omega \underbrace{\operatorname{div} \mathbf{v}(\boldsymbol{x})}_{=0}\, p(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot (p(\boldsymbol{x})\boldsymbol{n}(\boldsymbol{x})) \,\mathrm{d}\boldsymbol{x}\,. \end{aligned} \tag{12.1.16}$$

Remark. This formula reflects conservation of energy: the energy dissipated in the fluid has to be supplied by the mechanical power injected/extracted at inlet and outlet. Remember that mechanical power is "force×velocity" and that the normal force opposing the injection of fluid is given by the pressure.

SOLUTION of (12-1.d):

The following steps are necessary:

1. Read `pipe.geo` through the `File -> Open` menu item.
2. Through `Modules -> Mesh -> 2D` generate a triangular mesh
3. Use `Modules -> Mesh -> Save` to save the mesh to file.

SOLUTION of (12-1.e):

The local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ are just the barycentric coordinate functions. The local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ are given in [Lecture → (2.6.1.6)]; one is associated with every vertex, one with every edge. In total, there are $Q := 2 \cdot 6 + 3 = 15$ local shape functions

LEHRFEM++'s numbering convention numbers vertex associated local shape functions first, then those associated with edges. Regarding the three local shape functions belonging to a vertex, the first contributes for the x_1-component of the velocity $\mathbf{v}_h$, the second tor the x_2-component, and the third is for the pressure p_h.

The two local shape functions sitting on each edge correspond to the first and second component of the velocity.

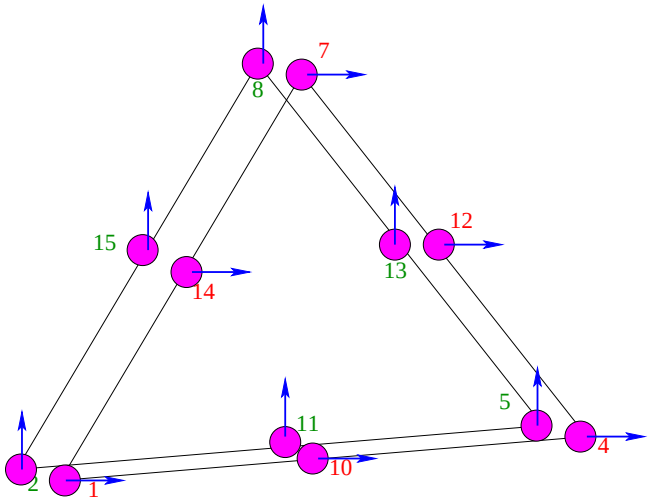
The following table gives the explicit formulas:

vertex 1	vertex 2	vertex 3
$b_K^1 = \begin{bmatrix} (2\lambda_1 - 1)\lambda_1 \\ 0 \end{bmatrix}$,	$b_K^4 = \begin{bmatrix} (2\lambda_2 - 1)\lambda_2 \\ 0 \end{bmatrix}$,	$b_K^7 = \begin{bmatrix} (2\lambda_3 - 1)\lambda_3 \\ 0 \end{bmatrix}$,
$b_K^2 = \begin{bmatrix} 0 \\ 2\lambda_1 - 1)\lambda_1 \end{bmatrix}$,	$b_K^5 = \begin{bmatrix} 0 \\ (2\lambda_2 - 1)\lambda_2 \end{bmatrix}$,	$b_K^8 = \begin{bmatrix} 0 \\ (2\lambda_3 - 1)\lambda_3 \end{bmatrix}$,
$b_K^3 = \lambda_1$,	$b_K^6 = \lambda_2$,	$b_K^9 = \lambda_3$,
edge 1	**edge 2**	**edge 3**
$b_K^{10} = \begin{bmatrix} 4\lambda_1\lambda_2 \\ 0 \end{bmatrix}$,	$b_K^{12} = \begin{bmatrix} 4\lambda_2\lambda_3 \\ 0 \end{bmatrix}$,	$b_K^{14} = \begin{bmatrix} 4\lambda_1\lambda_3 \\ 0 \end{bmatrix}$,
$b_K^{11} = \begin{bmatrix} 0 \\ 4\lambda_1\lambda_2 \end{bmatrix}$,	$b_K^{13} = \begin{bmatrix} 0 \\ 4\lambda_2\lambda_3 \end{bmatrix}$,	$b_K^{15} = \begin{bmatrix} 0 \\ 4\lambda_1\lambda_3 \end{bmatrix}$.

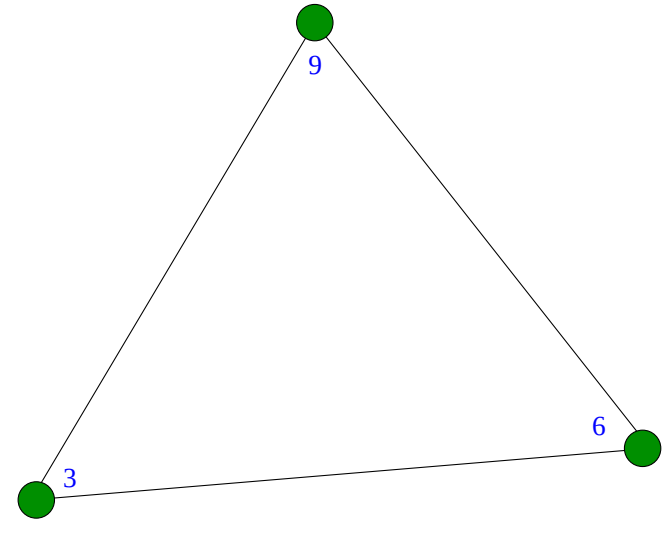
(12.1.21)

Obviously, the local shape functions b_K^3, b_K^6, b_K^9 belong to the pressure space $\mathcal{S}_1^0(\mathcal{M})$.

The six local shape functions $b_K^1, b_K^4, b_K^7, b_K^{10}, b_K^{12}, b_K^{14}$ are meant for the x_1-component of the velocity, the remaining local shape functions $b_K^2, b_K^5, b_K^8, b_K^{11}, b_K^{13}, b_K^{14}$ for the x_2-component.



Velocity d.o.f.: local numbers



Pressure d.o.f.: local numbers

SOLUTION of (12-1.f):

Straightforward from (12.1.20),

$$\dim V_{0,h} = \dim \mathcal{S}_2^0(\mathcal{M}) + \dim \mathcal{S}_2^0(\mathcal{M}) + \dim \mathcal{S}_1^0(\mathcal{M}) \ .$$

- As explained in [Lecture → Ex. 2.6.1.2] a single global shape funtion of $\mathcal{S}_2^0(\mathcal{M})$ is associated with every node and edge of the mesh, which means

$$\dim \mathcal{S}_2^0(\mathcal{M}) = \sharp\mathcal{V}(\mathcal{M}) + \sharp\mathcal{E}(\mathcal{M}) \ .$$

- A single tent basis function of $\mathcal{S}_1^0(\mathcal{M})$ belongs to every node of the mesh,

$$\dim \mathcal{S}_1^0(\mathcal{M}) = \sharp\mathcal{V}(\mathcal{M}) \ .$$

$$\dim V_{0,h} = 3 \cdot \sharp\mathcal{V}(\mathcal{M}) + 2 \cdot \sharp\mathcal{E}(\mathcal{M}) \ .$$

SOLUTION of (12-1.g):

Dropping the last equation and the Lagrange multiplier and splitting the velocity $\mathbf{v}_h$ into its components $v_{1,h}$ and $v_{2,h}$, the discrete variational problem (12.1.18) becomes: seek $v_{1,h} \in \mathcal{S}_2^0(\mathcal{M})$, $v_{2,h} \in \mathcal{S}_2^0(\mathcal{M})$, $v_{i,h}|_{\partial\Omega} = \mathsf{I}_2(\mathbf{g})_i|_{\partial\Omega}$, $p_h \in \mathcal{S}_1^0(\mathcal{M})$ such that

$$\begin{aligned} \int_\Omega \mathbf{grad}\, v_{1,h} \cdot \mathbf{grad}\, w_{1,h}\, \mathrm{d}\boldsymbol{x} \quad &+ \quad \int_\Omega \frac{\partial w_{1,h}}{\partial x_1} p_h \, \mathrm{d}\boldsymbol{x} \;=\; 0\,, \\ \int_\Omega \mathbf{grad}\, v_{2,h} \cdot \mathbf{grad}\, w_{2,h}\, \mathrm{d}\boldsymbol{x} \quad &+ \quad \int_\Omega \frac{\partial w_{2,h}}{\partial x_2} p_h \, \mathrm{d}\boldsymbol{x} \;=\; 0\,, \\ \int_\Omega \frac{\partial v_{1,h}}{\partial x_1} q_h \, \mathrm{d}\boldsymbol{x} \quad &+ \quad \int_\Omega \frac{\partial v_{2,h}}{\partial x_2} q_h \, \mathrm{d}\boldsymbol{x} \;=\; 0 \end{aligned} \tag{12.1.22}$$

for all $w_{1,h} \in \mathcal{S}_2^0(\mathcal{M})$, $w_{1,h} \in \mathcal{S}_2^0(\mathcal{M})$, $q_h \in \mathcal{S}_1^0(\mathcal{M})$.

The structure of this variational problem is reflected in the block structure of the element matrix $\mathbf{A}_K \in \mathbb{R}^{15,15}$

$$\mathbf{A}_K = \begin{bmatrix} \mathbf{A}_1 & \mathbf{O}_{6,6} & \mathbf{B}_1^\top \\ \mathbf{O}_{6,6} & \mathbf{A}_2 & \mathbf{B}_2^\top \\ \mathbf{B}_1 & \mathbf{B}_2 & \mathbf{O}_{3,3} \end{bmatrix}, \qquad \begin{aligned} \mathbf{A}_1 &:= \left[\int_K \mathbf{grad}\, b_K^j \cdot \mathbf{grad}\, b_K^i \, \mathrm{d}\boldsymbol{x} \right]_{i,j\in\{1,4,7,10,12,14\}}, \\ \mathbf{A}_2 &:= \left[\int_K \mathbf{grad}\, b_K^j \cdot \mathbf{grad}\, b_K^i \, \mathrm{d}\boldsymbol{x} \right]_{i,j\in\{2,5,8,11,13,15\}}, \\ \mathbf{B}_1 &:= \left[\int_K \frac{\partial b_K^j}{\partial x_1} b_K^i \, \mathrm{d}\boldsymbol{x} \right]_{\substack{i\in\{3,6,9\} \\ j\in\{1,4,7,10,12,14\}}}, \\ \mathbf{B}_2 &:= \left[\int_K \frac{\partial b_K^j}{\partial x_1} b_K^i \, \mathrm{d}\boldsymbol{x} \right]_{\substack{i\in\{3,6,9\} \\ j\in\{2,5,8,11,13,15\}}}. \end{aligned} \tag{12.1.23}$$

Note that the ordering of the local shape functions that gives rise to the block structure of the matrix $\mathbf{A}_K$ as displayed in (12.1.23) does *not match* the ordering used by LEHRFEM++ **!**

SOLUTION of (12-1.h):

We first examine the local shape functions $\{q_K^i\}_{i=1}^6$ for $\mathcal{S}_2^0(\mathcal{M})$, see [Lecture → (2.6.1.6)], and their gradients:

$$\begin{aligned}
q_K^1 &= (2\lambda_1 - 1)\lambda_1 &&\rightarrowtail \quad \mathbf{grad}\, q_K^1 = (4\lambda_1 - 1)\,\mathbf{grad}\,\lambda_1\,,\\
q_K^2 &= (2\lambda_2 - 1)\lambda_2 &&\rightarrowtail \quad \mathbf{grad}\, q_K^2 = (4\lambda_2 - 1)\,\mathbf{grad}\,\lambda_2\,,\\
q_K^3 &= (2\lambda_3 - 1)\lambda_3 &&\rightarrowtail \quad \mathbf{grad}\, q_3^3 = (4\lambda_3 - 1)\,\mathbf{grad}\,\lambda_3\,,\\
q_K^4 &= 4\lambda_1\lambda_2 &&\rightarrowtail \quad \mathbf{grad}\, q_K^4 = 4(\lambda_1\,\mathbf{grad}\,\lambda_2 + \lambda_2\,\mathbf{grad}\,\lambda_1)\,,\\
q_K^5 &= 4\lambda_2\lambda_3 &&\rightarrowtail \quad \mathbf{grad}\, q_K^5 = 4(\lambda_2\,\mathbf{grad}\,\lambda_3 + \lambda_3\,\mathbf{grad}\,\lambda_2)\,,\\
q_K^6 &= 4\lambda_1\lambda_3 &&\rightarrowtail \quad \mathbf{grad}\, q_K^6 = 4(\lambda_3\,\mathbf{grad}\,\lambda_1 + \lambda_1\,\mathbf{grad}\,\lambda_3)\,,
\end{aligned} \tag{12.1.24}$$

where $\lambda_1, \lambda_2, \lambda_3$ are the barycentric coordinate functions on K. The sequence of local shape functions as used by LEHRFEM++ is

$$\left(\begin{bmatrix} q_K^1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ q_K^1 \end{bmatrix}, \lambda_1, \begin{bmatrix} q_K^2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ q_K^2 \end{bmatrix}, \lambda_2, \begin{bmatrix} q_K^3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ q_K^3 \end{bmatrix}, \lambda_3, \begin{bmatrix} q_K^4 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ q_k^4 \end{bmatrix}, \begin{bmatrix} q_K^5 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ q_k^5 \end{bmatrix}, \begin{bmatrix} q_K^6 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ q_k^6 \end{bmatrix}\right) \quad .$$

The constant vectors $\mathbf{grad}\,\lambda_i$, $i = 1,2,3$, can be computed as in [Lecture → Code 2.4.5.11]. A corresponding auxiliary function

```
Eigen::Matrix<double, 2, 3> gradbarycoordinates(
const lf::mesh::Entity &entity) {
  LF_VERIFY_MSG(entity.RefEl() == lf::base::RefEl::kTria(),
  "Unsupported cell type " << entity.RefEl());
  // Get vertices of the triangle
  auto endpoints = lf::geometry::Corners(*(entity.Geometry()));
  Eigen::Matrix<double, 3, 3> X;  // temporary matrix
  X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
  X.block<3, 2>(0, 1) = endpoints.transpose();
  return X.inverse().block<2, 3>(1, 0);
}
```

can just copy parts of that code and then returns the vectors $\mathbf{grad}\,\lambda_i$ in the columns of a 2×3-matrix.

Represent the gradients $\mathbf{grad}\, q_K^i$ as lambda functions taking barycentric coordinates as arguments, with signature

`std::function<Eigen::Vector2d(Eigen::Vector3d)>`

We also observe that all integrands on K are polynomials of maximal degree 2. For such integrands the 3-point quadrature rule based on the midpoints of the edges of K provides the exact integral values. If the integrand $\varphi \in \mathcal{P}_2(\mathbb{R}^2)$ is a function of the three barycentric coordinates $\varphi = \varphi(\lambda_1, \lambda_2, \lambda_3)$, then we have

$$\int_K \varphi(\lambda_1(\boldsymbol{x}), \lambda_2(\boldsymbol{x}), \lambda_3(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} = \tfrac{1}{3}|K|\left(\varphi(\tfrac{1}{2}, \tfrac{1}{2}, 0) + \varphi(0, \tfrac{1}{2}, \tfrac{1}{2}) + \varphi(\tfrac{1}{2}, 0, \tfrac{1}{2})\right)\,. \tag{12.1.25}$$

We have chosen this way to write down the quadrature formula, because all local shape functions have simple representations in terms of barycentric coordinate functions.

Another important tool is index remapping through (C++ indexing!)

```
const std::array<Eigen::Index, 6> vx_idx{0, 3, 6, 9, 11, 13};
const std::array<Eigen::Index, 6> vy_idx{1, 4, 7, 10, 12, 14};
const std::array<Eigen::Index, 3> p_idx{2, 5, 8};
```

This means, using C++ indexing, the $\mathcal{S}_2^0$-LSF with number i, $i = 0,\ldots,5$, corresponds to the Taylor-Hood LSFs with number `vx_idx[i]` (x_1-component of the velocity) and `vy_idx[i]` (x_2-component of the velocity), respectively. The $\mathcal{S}_1^0$-LSF with number j, $j = 0,1,2$ is mapped to the Taylor-Hood LSF with number `p_idx[j]`;

As we see from (12.1.23), for the part of $\mathbf{A}_K$ related to the bilinar form $(\mathbf{v},q) \mapsto \int_\Omega \operatorname{div}\mathbf{v}\, q\,\mathrm{d}\boldsymbol{x}$ we have the entries

$$\int_K \operatorname{div}\begin{bmatrix} q_K^j \\ 0 \end{bmatrix} \lambda_i \,\mathrm{d}\boldsymbol{x} = \int_K \frac{\partial q_K^j}{\partial x_1}\,\lambda_i\,\mathrm{d}\boldsymbol{x} \quad , \quad \left[\int_K \operatorname{div}\begin{bmatrix} 0 \\ q_K^j \end{bmatrix} \lambda_i \,\mathrm{d}\boldsymbol{x}\right] = \int_K \frac{\partial q_K^j}{\partial x_2}\,\lambda_i\,\mathrm{d}\boldsymbol{x}\,, \tag{12.1.26}$$

$i = 1,2,3$, $j = 1,\ldots,6$. We can get them as the components of the vectors

$$\int_K \mathbf{grad}\, q_K^j(\boldsymbol{x})\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \in \mathbb{R}^2\,, \quad i = 1,2,3\,, \quad j = 1,\ldots,6\,. \tag{12.1.27}$$

Also in this case the integral can be evaluated exactly using the 3-point edge midpoint quadrature rule.

C++ code 12.1.28: Sub-problem (12-1.h): `Eval()` member function of TaylorHoodElement-MatrixProvider ↪ GitLab

```
2  TaylorHoodElementMatrixProvider::ElemMat TaylorHoodElementMatrixProvider::Eval(
3      const lf::mesh::Entity& cell) {
4    LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
5                  "Unsupported cell type " << cell.RefEl());
6    // Area of the triangle
7    double area = lf::geometry::Volume(*cell.Geometry());
8    // Compute gradients of barycentric coordinate functions, see
9    // [Lecture → Code 2.4.5.11] Get vertices of the triangle
10   auto endpoints = lf::geometry::Corners(*(cell.Geometry()));
11   Eigen::Matrix<double, 3, 3> X;  // temporary matrix
12   X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
13   X.block<3, 2>(0, 1) = endpoints.transpose();
14   // This matrix contains grad λ_i in its columns
15   const auto G{X.inverse().block<2, 3>(1, 0)};
16   // Dummy lambda functions for barycentric coordinates
17   std::array<std::function<double(Eigen::Vector3d)>, 3> lambda{
18       [](Eigen::Vector3d c) -> double { return c[0]; },
19       [](Eigen::Vector3d c) -> double { return c[1]; },
20       [](Eigen::Vector3d c) -> double { return c[2]; }};
21   // Gradients of local shape functions of quadratic Lagrangian finite
       element
22   // space as lambda functions, see (12.1.24)
23   std::array<std::function<Eigen::Vector2d(Eigen::Vector3d)>, 6> gradq{
24       [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
25         return (4 * c[0] - 1) * G.col(0);
26       },
27       [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
28         return (4 * c[1] - 1) * G.col(1);
29       },
30       [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
31         return (4 * c[2] - 1) * G.col(2);
32       },
```

```
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[0] * G.col(1) + c[1] * G.col(0));
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[1] * G.col(2) + c[2] * G.col(1));
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[2] * G.col(0) + c[0] * G.col(2));
      }};
  // Barycentric coordinates of the midpoints of the edges for
  // use with the 3-point edge midpoint quadrature rule (12.1.25)
  const std::array<Eigen::Vector3d, 3> mp = {Eigen::Vector3d({0.5, 0.5, 0}),
                                             Eigen::Vector3d({0, 0.5, 0.5}),
                                             Eigen::Vector3d({0.5, 0, 0.5})};
  // Compute the (scaled) element matrix for -Δ and S_2^0.
  Eigen::Matrix<double, 6, 6> L;
  for (int i = 0; i < 6; ++i) {
    for (int j = 0; j <= i; ++j) {
      // 3-point edge midpoint quadrature rule!
      L(i, j) = L(j, i) = gradq[i](mp[0]).dot(gradq[j](mp[0])) +
                          gradq[i](mp[1]).dot(gradq[j](mp[1])) +
                          gradq[i](mp[2]).dot(gradq[j](mp[2]));
    }
  }
  // Do not forget to set all non-initialized entries to zero
  AK_.setZero();
  // Distribute the entries of L to the final element matrix
  const std::array<Eigen::Index, 6> vx_idx{0, 3, 6, 9, 11, 13};
  const std::array<Eigen::Index, 6> vy_idx{1, 4, 7, 10, 12, 14};
  for (int i = 0; i < 6; ++i) {
    for (int j = 0; j < 6; ++j) {
      AK_(vx_idx[i], vx_idx[j]) = AK_(vy_idx[i], vy_idx[j]) = L(i, j);
    }
  }
  // Fill entries related to bilinear form b(.,.):  (12.1.26)
  const std::array<Eigen::Index, 3> p_idx{2, 5, 8};
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
      // (12.1.27) with 3-point edge midpoint quadrature rule!
      const Eigen::Vector2d gql_ij{gradq[j](mp[0]) * lambda[i](mp[0]) +
                                   gradq[j](mp[1]) * lambda[i](mp[1]) +
                                   gradq[j](mp[2]) * lambda[i](mp[2])};
      AK_(p_idx[i], vx_idx[j]) = AK_(vx_idx[j], p_idx[i]) = gql_ij[0];
      AK_(p_idx[i], vy_idx[j]) = AK_(vy_idx[j], p_idx[i]) = gql_ij[1];
    }
  }
  AK_ *= area / 3.0;
  // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
  // Workaround for a side-effect in LehrFEM++
  // Local shape functions associated with edges have to be swapped, if
     the edge
  // has "negative" orientation with respect to the triangle.  This is
     meant to
  // offset such a swapping done by LehrFEM++'s UniformFEDofHandler.
  auto edge_orientations = cell.RelativeOrientations();
  LF_ASSERT_MSG(edge_orientations.size() == 3,
                "Triangle should have 3 edges!?");
  Eigen::VectorXd tmp(15);
```

```
89    for (int k = 0; k < 3; ++k) {
90      const lf::assemble::ldof_idx_t ed_dofx = 9 + 2 * k;
91      const lf::assemble::ldof_idx_t ed_dofy = 10 + 2 * k;
92      if (edge_orientations[k] == lf::mesh::Orientation::negative) {
93        // The rows and columns of the element matrix with numbers 9+2*k
            and
94        // 9+2*k+1 have to be swapped.
95        tmp = AK_.col(ed_dofx);
96        AK_.col(ed_dofx) = AK_.col(ed_dofy);
97        AK_.col(ed_dofy) = tmp;
98        tmp = AK_.row(ed_dofx);
99        AK_.row(ed_dofx) = AK_.row(ed_dofy);
100       AK_.row(ed_dofy) = tmp;
101     }
102   }
103   // !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
104   // Finally multiply with the quadrature weight
105   return AK_;
106 }
```

SOLUTION of (12-1.i):

Let k be the index of the tent function s_h^i in the ordered set of all global shape functions of $V_{0,h}$. Then, from (12.1.18) we see

$$(\mathbf{A})_{N+M+1,k} = (\mathbf{A})_{k,N+M+1} = \int_\Omega s_h^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,. \tag{12.1.29}$$

SOLUTION of (12-2.a):

- The sum in (12.2.2) is a *direct sum*, which means that the dimensions add up,

$$\blacktriangleright \qquad \dim \mathcal{BS}_1^0(\mathcal{M}) = \dim \mathcal{S}_1^0(\mathcal{M}) + \sharp\mathcal{M} = \sharp\mathcal{V}(\mathcal{M}) + \sharp\mathcal{M} \ . \tag{12.2.7}$$

  In fact, this can immediately be inferred from the information about the global shape functions given above.

- Switching from $\mathcal{BS}_1^0(\mathcal{M})$ to $\mathcal{BS}_{1,0}^0(\mathcal{M})$ amounts to dropping the global shape functions sitting on nodes on $\partial\Omega$,

$$\blacktriangleright \qquad \dim \mathcal{BS}_{1,0}^0(\mathcal{M}) = \dim \mathcal{BS}_1^0(\mathcal{M}) - \sharp\mathcal{V}_\partial(\mathcal{M}) = \sharp\mathcal{V}_i(\mathcal{M}) + \sharp\mathcal{M} \ . \tag{12.2.8}$$

SOLUTION of (12-2.b):

Since we have no information about the geometry of the mesh, we start from the following premises.

> A pair of global shape functions from trial and test finite element spaces give rise to a non-zero entry of the Galerkin matrix, if their supports overlap.

Moreover, since the "elimination" of the boundary values has been done already in (12.2.9), we have to consider the Galerkin matrices with the trial and test space $\mathcal{BS}^0_{1,0}(\mathcal{M})$ for the velocity components.

(i) $\mathbf{A}_x$, $\mathbf{A}_y$ are $\mathcal{BS}^0_{1,0}(\mathcal{M}) \times \mathcal{BS}^0_{1,0}(\mathcal{M})$-Galerkin matrices. The global shape functions are associated with the interior nodes of the mesh and the cells of the mesh.

- Two node-associated global shape functions can contribute a non-zero matrix element, if "their" nodes coincide or are connected by an edge.
- A cell-associated global shape function and a node-associated global shape function have overlapping supports, if the node is a vertex of the triangle/cell.

$$\blacktriangleright \quad \operatorname{nnz}(\mathbf{A}_x) = \operatorname{nnz}(\mathbf{A}_y) \le \sharp\mathcal{V}_i(\mathcal{M}) + 2\sharp\mathcal{E}_{ii}(\mathcal{M}) + (1+2\cdot 3)\cdot\sharp\mathcal{M}\,,$$
$$\mathcal{E}_{ii}(\mathcal{M}) := \{e \in \mathcal{E}(\mathcal{M}) : \text{ both endpoints } \notin \partial\Omega\}\,.$$

Remark. A sharper bound can be stated taking into account that

$$\int_K \mathbf{grad}\,\beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall K \in \mathcal{M}\,, \quad \text{see Sub-problem (12-2.h) (ii).}$$

This means that no non-zero matrix entries will be spawned by the interaction of cubic bubble functions and tent functions. Thus we obtain the smaller bound

$$\operatorname{nnz}(\mathbf{A}_x) = \operatorname{nnz}(\mathbf{A}_y) \le \sharp\mathcal{V}_i(\mathcal{M}) + 2\sharp\mathcal{E}_{ii}(\mathcal{M}) + \sharp\mathcal{M}\,.$$

(ii) The rows of $\mathbf{B}_x$, $\mathbf{B}_y$ correspond to the node-associated global shape functions (tent functions))of $\mathcal{S}^0_1(\mathcal{M})$, their columns to the global shape functions of $\mathcal{BS}^0_{1,0}(\mathcal{M})$, the tent function belonging to *interior* nodes.

- The support of two tent functions overlap, if they belong to the same node or their nodes are connected by an edge.
- The supports of a node-associated tent function and a cell-associated cubic bubble function overlap, if the node is a vertex of the triangle.

$$\blacktriangleright \quad \operatorname{nnz}(\mathbf{B}_x) = \operatorname{nnz}(\mathbf{B}_y) \le \sharp\mathcal{V}_i(\mathcal{M}) + \sharp\mathcal{E}_i(\mathcal{M}) + 3\cdot\sharp\mathcal{M}\,.$$

(iii) $\mathbf{c}$ is a densely populated vector:

$$\blacktriangleright \quad \operatorname{nnz}(\mathbf{c}) \le M := \sharp\mathcal{V}(\mathcal{M})\,.$$

HINT 1 for (12-2.c): You may also examine the code of `testCvgSimpleFEM()` in `stokesminieler`
.

SOLUTION of (12-2.c):

- From Code 12.2.11 we see that there are 9 local shape functions.
- The matrix $\mathbf{L}$ in Code 12.2.12 is $\mathbf{L} := \left[\mathbf{grad}\,\lambda_i \cdot \mathbf{grad}\,\lambda_j\right]_{i,j=1}^{3}$. Moreover, the barycentric coordinate functions are the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$.
- The matrix

$$|K|\mathbf{L} = \left[\int_K \mathbf{grad}\,\lambda_i \cdot \mathbf{grad}\,\lambda_j \,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^{3} \in \mathbb{R}^{3,3}\,,$$

  the $\mathcal{S}_1^0(\mathcal{M})$-element matrix for the bilinear form $(u,v) \to \int_\Omega \mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}$, is copied into the element matrix $\mathbf{A}_k$ in Lines 25-29 of Code 12.2.11.
- We have, e.g.,

$$\int_K \mathrm{div}\begin{bmatrix}\lambda_i\\0\end{bmatrix}\mathrm{d}\boldsymbol{x}\,\lambda_j = \tfrac{1}{3}|K|(\mathbf{grad}\,\lambda_i)_1 = (\mathbf{G})_{1,i}\,,\quad j = 1,2,3\,.$$

- In `testCvgSimpleFEM()` we read

```
// Initialize dof handler for simple FEM
lf::assemble::UniformFEDofHandler dofh(lev_mesh_p,
  {{lf::base::RefEl::kPoint(), 3},
   {lf::base::RefEl::kSegment(), 0},
   {lf::base::RefEl::kTria(), 0},
   {lf::base::RefEl::kQuad(), 0}});
```

  So three global shape functions (two for the velocity components and one for the pressure) are associated to every (interior) node of the mesh.

All these pieces of information clearly hint that

$$\mathbf{v}_h \in \left(\mathcal{S}_1^0(\mathcal{M})\right)^2 \quad,\quad p_h \in \mathcal{S}_1^0(\mathcal{M})\,.$$

SOLUTION of (12-2.d):

From linear algebra we know that column transformations (adding a multiple of a column to another column) do not change the rank of the matrix. By successive column transformations

1. subtracting multiples of the 7-th columns from column 3, 4, 5, 6, and then
2. adding the 6-th column to the third,

we can transform $\mathbf{A}$ into

$$\mathbf{A} \rightarrow \begin{bmatrix} 4 & 0 & \frac{1}{6} & -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} & 0 & 0 \\ 0 & 4 & \frac{1}{6} & \frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ -\frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 4 & 0 & \frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 \\ 0 & 4 & \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 \\ \frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ \frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ -\frac{1}{6} & -\frac{1}{6} & 0 & 0 & 0 & 0 & 0 & \frac{1}{6} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{3} & 0 \end{bmatrix}$$

The upper left 4×4 block has full rank $= 4$, but there are two zero columns.

$$\blacktriangleright \qquad \text{rank}(\mathbf{A}) = 6 \, .$$

In particular, the matrix $\mathbf{A}$ is *singular*.

SOLUTION of (12-2.e):

- There is a blow-up of the basis expansion coefficients for the pressure p_h indicating **pressure instability** [Lecture → Section 12.3.1]. The system matrix will be singular or nearly singular with its kernel or near-kernel spanned by some vectors of pressure coefficients. Pressure components lying in that (near-kernel) can become very large (even ∞ in theory) in the solution; the pressure solution clearly displays a **blow-up** as $h \to 0$.
- The *velocity components converge* nicely as the meshes are refined, as long as those are not too fine.
- On very fine meshes, we observe a deterioration of the accuracy of the velocity solution. Due to finite-precision arithmetic the blown-up pressure solution starts "polluting" $\mathbf{v}_h$.

Remark. The message of the table matches the observation made in [Lecture → Exp. 12.3.1.7]: Though a method may be affected by pressure instability, it can still produce meaningful velocity solutions.

HINT 1 for (12-2.f): Some integrals can be evaluated more easily after apply Green's first formula of [Lecture → Thm. 1.5.2.7].

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad \text{[Lecture → (12.2.3.2)]}$$

HINT 2 for (12-2.f): Integrals of products of barycentric coordinate functions can be evaluated by means of [Lecture → Lemma 2.7.5.5].

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex[a] K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1!\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, .$$

[Lecture → (2.7.5.6)]

[a] A d-simplex is the convex hull of $d+1$ points in $\mathbb{R}^d$, which do not all lie in a single plane.

SOLUTION of (12-2.f):

(i) We apply [Lecture → Lemma 2.7.5.5] with $d = 2$, $\alpha_1 = \alpha_2 = \alpha_3 = 1$ and get

$$\int_K \beta_K(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = 2!|K| \, \frac{1}{5!} = \frac{|K|}{60} \, . \tag{12.2.13}$$

(ii) We apply Green's first formula [Lecture → Thm. 1.5.2.7] with $\Omega := K$, $\mathbf{j} := \mathbf{grad}\, \lambda_i$, and $v := \beta_K$ and note that

- $\operatorname{div} \mathbf{grad}\, \lambda_i = 0$, since $\lambda_i \in \mathcal{P}_1$,
- $\beta_K|_{\partial K} = 0$ by the very definition of β_K.

This yields

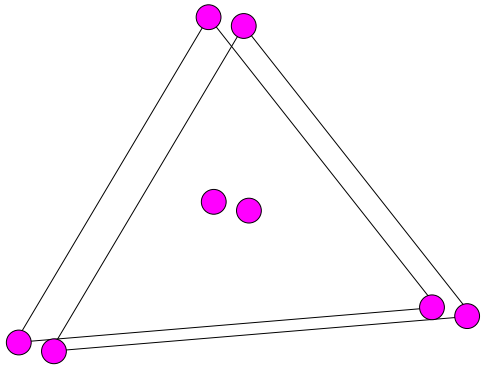
$$\begin{aligned} \int_K \mathbf{grad}\, \beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \\ = -\int_K \beta_K(\boldsymbol{x}) \, \underbrace{\Delta \lambda_i(\boldsymbol{x})}_{=0} \, \mathrm{d}\boldsymbol{x} + \int_{\partial K} \underbrace{\beta_K(\boldsymbol{x})}_{=0} \, \mathbf{grad}\, \lambda_i(\boldsymbol{x}) \cdot \boldsymbol{n}_K(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = 0 \, . \end{aligned} \tag{12.2.14}$$

(iii) Note that $\frac{\partial}{\partial x_k} = \mathbf{e}_k \cdot \mathbf{grad}$, where $\mathbf{e}_k$ is the k-th Cartesian coordinate vector. Thus we can apply Green's first formula [Lecture → Thm. 1.5.2.7] with $\Omega := K$, $\mathbf{j} := \boldsymbol{e}_k \lambda_i$, and $v := \beta_K$:

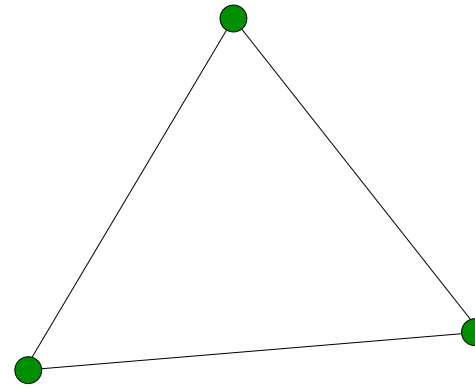
$$\begin{aligned} \int_K \frac{\partial \beta_K}{\partial x_k} \lambda_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} &= \int_K \mathbf{grad}\, \beta_K(\boldsymbol{x}) \cdot \boldsymbol{e}_k \lambda_i(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \\ &= -\int_K \beta_K(\boldsymbol{x}) \, \operatorname{div}(\boldsymbol{e}_k \lambda_i)(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} + \int_{\partial K} \underbrace{\beta_K(\boldsymbol{x})}_{=0} \lambda_i(\boldsymbol{x}) (\boldsymbol{n}_K)_k \, \mathrm{d}\boldsymbol{x} \\ &= -\frac{\partial \lambda_i}{\partial x_k} \int_K \beta_K \, \mathrm{d}\boldsymbol{x} \overset{(12.2.13)}{=} -\frac{|K|}{60} \frac{\partial \lambda_i}{\partial x_k} \, , \end{aligned} \tag{12.2.15}$$

where we used that $\mathbf{grad}\, \lambda_i$ is constant on K.

HINT 1 for (12-2.g): The local shape functions for a single component of the velocity in the MINI finite element method on a triangle K are its barycentric coordinate functions and the bubble function β_K. The "layout" of the local shape functions is



MINI-element: local d.o.f. for velocity space $\left(\mathcal{BS}_1^0(\mathcal{M})\right)^2$



MINI-element: local d.o.f. for pressure space Q_N

SOLUTION of (12-2.g):

As we have seen in Sub-problem (12-2.c), the simple FEM relies of $\mathcal{S}_1^0(\mathcal{M})$ for the approximation of the velocity components and on the same space for the pressure. Tent basis functions are used as global shape functions, which means that the ordered set of global shape functions will be

$$\left(\begin{bmatrix} \lambda_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_1 \end{bmatrix}, \lambda_1, \begin{bmatrix} \lambda_2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_2 \end{bmatrix}, \lambda_2, \begin{bmatrix} \lambda_3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_3 \end{bmatrix}, \lambda_3, \right) , \tag{12.2.16}$$

where we applied LEHRFEM++'s rules for ordering local shape functions from [Lecture → § 2.7.4.11].

In light of the specification of the basis functions for $\mathcal{BS}_1^0(\mathcal{M})$, we see that there are only *two extra new local shape functions* for the velocity space on a triangle K and those agree with β_K in either components.

These new basis functions are naturally associated with K. Therefore, according to LEHRFEM++'s numbering convention for local shape functions [Lecture → § 2.7.4.11], they will have the local numbers 10 (for the x_1-component of the velocity) and 11 (for the x_2-component), in mathematical indexing. Summing up, the element matrices for the MINI element method will be of size 11×11 with ordered local shape functions

$$\left(\begin{bmatrix} \lambda_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_1 \end{bmatrix}, \lambda_1, \begin{bmatrix} \lambda_2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_2 \end{bmatrix}, \lambda_2, \begin{bmatrix} \lambda_3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_3 \end{bmatrix}, \lambda_3, \begin{bmatrix} \beta_K \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \beta_K \end{bmatrix} \right) . \tag{12.2.17}$$

Comparing (12.2.16) and (12.2.17) we see that the element matrices for the simple FEM and the MINI element FEM *differ only in the last two rows and columns*.

Let us specify these new entries for the element matrix $\mathbf{A}_K$ of the MINI FEM and a triangle $K \in \mathcal{M}$:

$$(\mathbf{A}_K)_{10,k} = (\mathbf{A}_K)_{k,10} = \int_K \mathbf{grad}\, \beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} , \quad k = 3*(i-1)+1 , \quad i = 1,2,3 , \tag{12.2.18a}$$

$$(\mathbf{A}_K)_{11,k} = (\mathbf{A}_K)_{k,11} = \int_K \mathbf{grad}\, \beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} , \quad k = 3*(i-1)+2 , \quad i = 1,2,3 , \tag{12.2.18b}$$

$$(\mathbf{A}_K)_{10,10} = (\mathbf{A}_K)_{11,11} = \int_K \mathbf{grad}\, \beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\, \beta_K(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} , \tag{12.2.18c}$$

$$(\mathbf{A}_K)_{10,11} = (\mathbf{A}_K)_{11,10} = 0 , \tag{12.2.18d}$$

$$(\mathbf{A}_K)_{9+\ell,j} = (\mathbf{A}_K)_{j,9+\ell} = \int_K \frac{\partial \beta_K}{\partial x_\ell}(\boldsymbol{x})\, \lambda_i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} , \qquad j = 3*i , \quad i = 1,2,3 , \quad \ell = 1,2 . \tag{12.2.18e}$$

Unspecified entries are supposed to be zero. In addition, in light of (12.2.14) we find that

$$\begin{aligned} (\mathbf{A}_K)_{10,k} = (\mathbf{A}_K)_{k,10} = 0 , \quad k = 3*(i-1)+1 , \\ (\mathbf{A}_K)_{11,k} = (\mathbf{A}_K)_{k,11} = 0 , \quad k = 3*(i-1)+2 , \end{aligned} \qquad i = 1,2,3 .$$

Using (12.2.15) we also have

$$(\mathbf{A}_K)_{9+\ell,j} = (\mathbf{A}_K)_{j,9+\ell} = -\frac{|K|}{60} \frac{\partial \lambda_i}{\partial x_\ell} . \tag{12.2.19}$$

It remains to compute $\int_K \mathbf{grad}\,\beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\,\beta_K(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, starting with the product rule

$$\begin{aligned}\mathbf{grad}\,\beta_k &= \underbrace{\lambda_1\lambda_2}_{=:\xi_3}\mathbf{grad}\,\lambda_3 + \underbrace{\lambda_1\lambda_3}_{=:\xi_2}\mathbf{grad}\,\lambda_2 + \underbrace{\lambda_2\lambda_3}_{\xi_1}\mathbf{grad}\,\lambda_1 \\ &= \underbrace{\begin{bmatrix}\mathbf{grad}\,\lambda_1 & \mathbf{grad}\,\lambda_2 & \mathbf{grad}\,\lambda_3\end{bmatrix}}_{=:\mathbf{G}\in\mathbb{R}^{2,3}}\begin{bmatrix}\xi_1\\ \xi_2\\ \xi_3\end{bmatrix}\,.\end{aligned}$$

From [Lecture → Lemma 2.7.5.5] we deduced

$$\int_K \xi_i^2(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = |K|\,\frac{8}{6!} = \frac{|K|}{90} \quad , \quad \int_K \xi_i(\boldsymbol{x})\xi_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = |K|\,\frac{2\cdot 2}{6!} = \frac{|K|}{180}\,.$$

This means

$$\begin{aligned}\int_K \mathbf{grad}\,\beta_K(\boldsymbol{x}) \cdot \mathbf{grad}\,\beta_K(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_K \begin{bmatrix}\xi_1 & \xi_2 & \xi_3\end{bmatrix}\underbrace{\mathbf{G}^\top\mathbf{G}}_{=:\mathbf{L}\in\mathbb{R}^3}\begin{bmatrix}\xi_1\\ \xi_2\\ \xi_3\end{bmatrix}\mathrm{d}\boldsymbol{x} \\ = \sum_{i=1}^3\sum_{j=1}^3 (\mathbf{L})_{i,j}\int_K \xi_i\xi_j\,\mathrm{d}\boldsymbol{x} &= \frac{|K|}{90}\Big((\mathbf{L})_{1,1}+(\mathbf{L})_{2,2}+(\mathbf{L})_{3,3}+(\mathbf{L})_{1,2}+(\mathbf{L})_{1,3}+(\mathbf{L})_{2,3}\Big)\,. \quad (12.2.20)\end{aligned}$$

The structure of the element matrix for the Stokes MINI element is

$$\mathbf{A}_K = \begin{bmatrix}
* & * & * & * & * & * & * & * & * & 0 & 0\\
* & * & * & * & * & * & * & * & * & 0 & 0\\
* & * & * & * & * & * & * & * & * & * & *\\
* & * & * & * & * & * & * & * & * & 0 & 0\\
* & * & * & * & * & * & * & * & * & 0 & 0\\
* & * & * & * & * & * & * & * & * & * & *\\
* & * & * & * & * & * & * & * & * & 0 & 0\\
* & * & * & * & * & * & * & * & * & 0 & 0\\
* & * & * & * & * & * & * & * & * & * & *\\
0 & 0 & * & 0 & 0 & * & 0 & 0 & * & * & 0\\
0 & 0 & * & 0 & 0 & * & 0 & 0 & * & 0 & *
\end{bmatrix} \in \mathbb{R}^{11,11}\,, \quad (12.2.21)$$

$*$ $\hat{=}$ same as for simple FEM, $*$ $\hat{=}$ entries from (12.2.19), $*$ $\hat{=}$ entries according to (12.2.20).

C++ code 12.2.22: Implementation of `Eval()` member function for MINIElementMatrixProvider

```
2  MINIElementMatrixProvider::ElemMat MINIElementMatrixProvider::Eval(
3      const lf::mesh::Entity& cell) {
4    LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
5                  "Unsupported cell type " << cell.RefEl());
6    // Area of the triangle
7    double area = lf::geometry::Volume(*cell.Geometry());
8    // Compute gradients of barycentric coordinate functions, see
9    // [Lecture → Code 2.4.5.11]. First fetch vertices of the triangle
10   auto endpoints = lf::geometry::Corners(*(cell.Geometry()));
11   Eigen::Matrix<double, 3, 3> X;  // temporary matrix
12   X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
13   X.block<3, 2>(0, 1) = endpoints.transpose();
```

```
  // This matrix contains grad λ_i in its columns
  const auto G{X.inverse().block<2, 3>(1, 0)};
  // Inner products of gradients of barycentric coordinate functions
  // (Scaled) element matrix for −Δ
  const Eigen::Matrix3d L{G.transpose() * G};
  // Fixed-size matrix containing the element matrix
  ElemMat AK;
  AK.setZero();
  // indices of local shape function for various solution components
  constexpr std::array<Eigen::Index, 4> vx_idx_{0, 3, 6, 9};
  constexpr std::array<Eigen::Index, 4> vy_idx_{1, 4, 7, 10};
  constexpr std::array<Eigen::Index, 3> p_idx_{2, 5, 8};
  // Place the element matrix for −Δ in the element matrix
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      AK(vx_idx_[i], vx_idx_[j]) = L(i, j);
      AK(vy_idx_[i], vy_idx_[j]) = L(i, j);
    }
  }
  // Insert contributions to "pressure parts" of element matrix
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      AK(vx_idx_[j], p_idx_[i]) = AK(p_idx_[i], vx_idx_[j]) = G(0, j) / 3.0;
      AK(vy_idx_[j], p_idx_[i]) = AK(p_idx_[i], vy_idx_[j]) = G(1, j) / 3.0;
    }
  }
  // New contribution due to bubble functions
  // First:  B-parts
  for (int i = 0; i < 3; ++i) {
    AK.block<2, 1>(9, 3 * i + 2) = -G.col(i) / 60.0;
    AK.block<1, 2>(3 * i + 2, 9) = AK.block<2, 1>(9, 3 * i + 2).transpose();
  }
  // Second:  Contribution of cubic bubbles to A-parts
  const double int_gbgb =
      (L(0, 0) + L(1, 1) + L(2, 2) + L(0, 1) + L(0, 2) + L(1, 2)) / 90;
  AK(9, 9) = AK(10, 10) = int_gbgb;
  // Finally scale with the area of the triangle
  AK *= area;
  return AK;
}
```

SOLUTION of (12-2.h):

The main changes are to

1. to tell the **lf::assemble::DofHandler** that two local shape functions will be associated with the triangles of the mesh
2. to rely on **MINIElementMatrixProvider** as **ENTITY_MATRIX_PROVIDER**.

Note that nothing changes concerning the handling of the non-zero boundary values, because the additional cubic bubble functions vanish on the boundary.

C++ code 12.2.24: Part of `testCvgMINIFEM()`.

```
2    // ********** Part III: Solving on a single mesh **********
3    // Initialize dof handler for simple FEM
4    lf::assemble::UniformFEDofHandler dofh(lev_mesh_p,
5                                           {{lf::base::RefEl::kPoint(), 3},
6                                            {lf::base::RefEl::kSegment(), 0},
7                                            {lf::base::RefEl::kTria(), 2},
8                                            {lf::base::RefEl::kQuad(), 0}});
9    LF_ASSERT_MSG(dofh.NumDofs() == 2 * dofh_u.NumDofs() + dofh_p.NumDofs() +
10                                        2 * lev_mesh_p->NumEntities(0),
11                "No dof mismatch");
12   // Build and solve the linear system with trace of the exact
         velocity
13   // solution as Dirichlet data.
14   // Total number of d.o.f.  in monolithic FE spaces
15   size_t n = dofh.NumDofs();
16   std::cout << "Computing with " << n << " d.o.f.s, assembling LSE .. "
17             << std::flush;
18   // Obtain full Galerkin matrix in triplet format
19   // Full Galerkin matrix in triplet format taking into account the
         zero mean
20   // constraint on the pressure.
21   lf::assemble::COOMatrix<double> A(n + 1, n + 1);
22   // Set up computation of element matrix
23   MINIElementMatrixProvider themp{};
24   // Assemble full Galerkin matrix for simple Stokes FEM
25   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, themp, A);
26
27   // Add bottom row and right column corresponding to Lagrange
         multiplier
28   for (const lf::mesh::Entity* cell : dofh.Mesh()->Entities(0)) {
29     LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
30                   "Only implemented for triangles");
31     // Obtain area of triangle
32     const double area = lf::geometry::Volume(*cell->Geometry());
33     // The pressure GSFs are associated with the nodes
34     const std::span<const lf::mesh::Entity* const> nodes{
35         cell->SubEntities(2)};
36     // Loop over nodes
37     for (const lf::mesh::Entity* node : nodes) {
38       // Area of the cell
39       // Obtain index of tent function associated with node
40       // All indices of global shape functions sitting at node
41       std::span<const lf::assemble::gdof_idx_t> dof_idx{
42           dofh.InteriorGlobalDofIndices(*node)};
```

```
43          LF_ASSERT_MSG(dof_idx.size() == 3, "Node must carry 3 dofs!");
44          // The index of the tent function is the third one
45          const lf::assemble::gdof_idx_t tent_idx = dof_idx[2];
46          A.AddToEntry(n, tent_idx, area / 3.0);
47          A.AddToEntry(tent_idx, n, area / 3.0);
48        }
49      }
50      // Auxiliary right-hnad side vector
51      Eigen::VectorXd phi(A.cols());
52      phi.setZero();
53      // Impose Dirichlet boundary conditions
54      const std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh.Mesh();
55      // Flag nodes located on the boundary
56      auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 2)};
57      // Flag vector for d.o.f.  on the boundary
58      std::vector<std::pair<bool, double>> ess_dof_select(n + 1, {false, 0.0});
59      // Visit nodes on the boundary
60      for (const lf::mesh::Entity* node : mesh_p->Entities(2)) {
61        if (bd_flags(*node)) {
62          // Indices of global shape functions sitting at node
63          std::span<const lf::assemble::gdof_idx_t> dof_idx{
64              dofh.InteriorGlobalDofIndices(*node)};
65          LF_ASSERT_MSG(dof_idx.size() == 3, "Node must carry 3 dofs!");
66          // Position of node
67          const Eigen::Vector2d pos{Corners(*(node->Geometry())).col(0)};
68          // Dirichlet data
69          const Eigen::Vector2d g_val{v_ex(pos)};
70          // x-component of the velocity
71          ess_dof_select[dof_idx[0]] = {true, g_val[0]};
72          // y-component of the velocity
73          ess_dof_select[dof_idx[1]] = {true, g_val[1]};
74        }
75      }
76      // modify linear system of equations
77      lf::assemble::FixFlaggedSolutionComponents<double>(
78          [&ess_dof_select](lf::assemble::glb_idx_t dof_idx)
79              -> std::pair<bool, double> { return ess_dof_select[dof_idx]; },
80          A, phi);
81      // Assembly completed:  Convert COO matrix A into CRS format using
        Eigen's
82      // internal conversion routines.
83      Eigen::SparseMatrix<double> A_crs{A.makeSparse()};
84
85      std::cout << "done. Solving ..... " << std::flush;
86      // Solve linear system using Eigen's sparse direct elimination
87      // Examine return status of solver in case the matrix is singular
88      Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
89      solver.compute(A_crs);
90      LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
91      // This is the coefficient vector for the FE solution; Dirichlet
92      // boundary conditions are included
93      const Eigen::VectorXd res = solver.solve(phi);
94      LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
95      std::cout << "done. |dof vector| = " << res.norm() << std::endl;
96      // ********** Part IV: Compute error norms **********
97      // Coefficient vectors for the first and second component of the
        velocity
98      Eigen::VectorXd coeff_vec_u1 = Eigen::VectorXd::Zero(dofh_u.NumDofs());
```

```
    Eigen::VectorXd coeff_vec_u2 = Eigen::VectorXd::Zero(dofh_u.NumDofs());
    // Coefficient vector for the pressure
    Eigen::VectorXd coeff_vec_p = Eigen::VectorXd::Zero(dofh_p.NumDofs());
    // Remapping dofs in orde to be able to use MeshFunctionFE
    for (auto e : lev_mesh_p->Entities(2)) {
      // Global indices for u1, u2 for the respective vertex or edge
      auto glob_idxs = dofh.InteriorGlobalDofIndices(*e);
      auto glob_idx_o1 = dofh_u.InteriorGlobalDofIndices(*e)[0];
      // Extract the correct elements for the coefficient vector of the
      // components of u and the pressure p
      coeff_vec_u1[glob_idx_o1] = res[glob_idxs[0]];
      coeff_vec_u2[glob_idx_o1] = res[glob_idxs[1]];
      coeff_vec_p(glob_idx_o1) = res[glob_idxs[2]];
    }
    // Variables for storing the error norms
    double L2err_u1, L2err_u2, H1err_u1, H1err_u2, L2err_p;

    // Define finite-element mesh functions
    const lf::fe::MeshFunctionFE mf_o2_u1(fes_o1_ptr, coeff_vec_u1);
    const lf::fe::MeshFunctionFE mf_o2_u2(fes_o1_ptr, coeff_vec_u2);
    const lf::fe::MeshFunctionFE mf_o1_p(fes_o1_ptr, coeff_vec_p);
    const lf::fe::MeshFunctionGradFE mf_o2_grad_u1(fes_o1_ptr, coeff_vec_u1);
    const lf::fe::MeshFunctionGradFE mf_o2_grad_u2(fes_o1_ptr, coeff_vec_u2);

    // Exact solution for the first component of the velocity
    auto u1 = [&v_ex](Eigen::Vector2d x) -> double { return v_ex(x)[0]; };
    const lf::mesh::utils::MeshFunctionGlobal mf_u1{u1};
    // Exact solution for the gradient of v_1
    const lf::mesh::utils::MeshFunctionGlobal mf_grad_u1{grad_v1};
    // Exact solution second component of the velocity
    auto u2 = [&v_ex](Eigen::Vector2d x) -> double { return v_ex(x)[1]; };
    const lf::mesh::utils::MeshFunctionGlobal mf_u2{u2};
    // Exact solution for the gradient of v_2
    const lf::mesh::utils::MeshFunctionGlobal mf_grad_u2{grad_v2};
    // Mesh function for exact solution pressure
    const lf::mesh::utils::MeshFunctionGlobal mf_p{p_ex};
    // compute errors with 5th order quadrature rules
    L2err_u1 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_u1 - mf_u1), 4));
    L2err_u2 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_u2 - mf_u2), 4));
    H1err_u1 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_grad_u1 - mf_grad_u1),
        4));
    H1err_u2 = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o2_grad_u2 - mf_grad_u2),
        4));
    L2err_p = std::sqrt(lf::fe::IntegrateMeshFunction(
        *lev_mesh_p, lf::mesh::utils::squaredNorm(mf_o1_p - mf_p), 4));
    errs.emplace_back(dofh.NumDofs(), L2err_u1, L2err_u2, H1err_u1, H1err_u2,
                      L2err_p);
  }
  // Output table of errors to file and terminal
  std::ofstream out_file("errors.txt");
  std::cout.precision(3);
  std::cout << std::endl
            << std::left << std::setw(10) << "N" << std::right << std::setw(16)
```

```
156             << "L2 err(v1)" << std::setw(16) << "L2 err(v2)" << std::setw(16)
157             << "H1 err(v1)" << std::setw(16) << "H1 err(v2)" << std::setw(16)
158             << "L2 err(p)" << '\n';
159   std::cout << "-----------------------------------------" << '\n';
160   for (const auto& err : errs) {
161     auto [N, L2err_u1, L2err_u2, H1err_u1, H1err_u2, L2err_p] = err;
162     out_file << std::left << std::setw(10) << N << std::left << std::setw(16)
163              << L2err_u1 << std::setw(16) << L2err_u2 << std::setw(16)
164              << H1err_u1 << std::setw(16) << H1err_u2 << std::setw(16)
165              << L2err_p << '\n';
166     std::cout << std::left << std::setw(10) << N << std::left << std::setw(16)
167               << L2err_u1 << std::setw(16) << L2err_u2 << std::setw(16)
168               << H1err_u1 << std::setw(16) << H1err_u2 << std::setw(16)
169               << L2err_p << '\n';
170   }
171 }
172
173 }  // namespace StokesMINIElement
```

SOLUTION of (12-2.i):

We appeal to the following result from the course.

Theorem [Lecture → Thm. 12.3.3.13]. Convergence of stable FE for Stokes problem

If U_h, Q_h is a stable finite element pair (→ [Lecture → Def. 12.3.2.7]) for the Stokes variational saddle point problem [Lecture → (12.2.2.19)], then the corresponding finite element Galerkin solution $(\mathbf{v}_h, p_h)$ satisfies

$$\|\mathbf{v}-\mathbf{v}_h\|_{H^1(\Omega)} + \|p-p_h\|_{L^2(\Omega)} \leq C\left(\inf_{\mathbf{w}_h\in U_h}\|\mathbf{v}-\mathbf{w}_h\|_{H^1(\Omega)} + \inf_{q_h\in Q_h}\|p-q_h\|_{L^2(\Omega)}\right),$$

with a constant $C > 0$ that depends only on Ω, μ, and the shape regularity of the finite element mesh.

The asymptotic convergence of the best approximation errors we derive from the following estimate.

Corollary [Lecture → Cor. 3.3.3.4]. Error estimate for piecewise linear interpolation in 2D

Under the assumptions/with notations of [Lecture → Thm. 3.3.2.21]

$$\begin{aligned}\|u-\mathrm{I}_1 u\|_{L^2(\Omega)} &\leq \sqrt{\tfrac{3}{8}}\, h_{\mathcal{M}}^2 |u|_{H^2(\Omega)}\,,\\ |u-\mathrm{I}_1 u|_{H^1(\Omega)} &\leq \sqrt{\tfrac{3}{24}}\,\rho_{\mathcal{M}}\, h_{\mathcal{M}} |u|_{H^2(\Omega)}\,,\end{aligned} \qquad \forall u\in H^2(\Omega)\,.$$

A key insight is that the lowest-order Lagrangian finite element space is a subspace of the bubble-enriched space,

$$\mathcal{S}^0_{1,0}(\mathcal{M}) \subset \mathcal{BS}^0_{1,0}(\mathcal{M})\,.$$

Since $\mathbf{v}\in H^2(\Omega)\cap H^1_0(\Omega)$ and $p\in H^2(\Omega)$ this implies

$$\inf_{\mathbf{w}_h\in\left(\mathcal{BS}^0_{1,0}(\mathcal{M}_\ell)\right)^2}|\mathbf{v}_h-\mathbf{v}|_{H^1(\Omega)} \leq \inf_{\mathbf{w}_h\in\left(\mathcal{S}^0_{1,0}(\mathcal{M}_\ell)\right)^2}|\mathbf{v}_h-\mathbf{v}|_{H^1(\Omega)} = O(h_\ell) \quad \text{for} \quad h_\ell\to 0\,, \tag{12.2.27}$$

$$\inf_{q_h\in\mathcal{S}^0_1(\mathcal{M}_\ell)}\|q_h-p\|_{L^2(\Omega)} = O(h_\ell^2) \quad \text{for} \quad h_\ell\to 0\,. \tag{12.2.28}$$

Note that the cubic bubble functions do not contribute anything to the approximation power of $\mathcal{BS}^0_1(\mathcal{M})$ in the Sobolev scale.

Also note that the estimate of [Lecture → Thm. 12.3.3.13] does not permit us to benefit from the fast convergence of the norm best approximation in p. The more slowly converging norm of the best approximation error for the velocity will determine the overall asymptotic convergence and we can only infer

$$\begin{aligned}|\mathbf{v}_\ell-\mathbf{v}|_{H^1(\Omega)} &= O(h_\ell) \quad \text{for} \quad h_\ell\to 0\,,\\ \|p_\ell-p\|_{L^2(\Omega)} &= O(h_\ell) \quad \text{for} \quad h_\ell\to 0\,,\end{aligned}$$

Remark. The empiric convergence test conducted with `testCvgMINIFEM()` gives us the following output:

N	L2 err(**v1**)	L2 err(**v2**)	H1 err(**v1**)	H1 err(**v2**)	L2 err(**p**)
69	0.00916	0.00568	0.106	0.0857	0.156
240	0.00228	0.00152	0.0514	0.0443	0.0612
897	0.000561	0.000379	0.0253	0.022	0.0181
3471	0.000139	9.39e−05	0.0126	0.0109	0.00562
13659	3.48e−05	2.33e−05	0.00626	0.00544	0.00183
54195	8.69e−06	5.82e−06	0.00312	0.00272	0.000611
215907	2.17e−06	1.45e−06	0.00156	0.00136	0.000208

We clearly observe algebraic convergence with rate 1 of $\mathbf{v}_\ell - \mathbf{v}$. However, the $L^2(\Omega)$-norm of the pressure error seems to converge faster than with rate 1. A similar empiric finding is reported in

SOLUTION of (12-3.a):

1. The basis functions of $\mathcal{S}_1^0(\mathcal{M})$ are associated with the nodes of $\mathcal{M}$, which implies

$$M = \dim \mathcal{S}_1^0(\mathcal{M}) = \sharp\mathcal{V}(\mathcal{M}) = \sharp\mathcal{V}_i(\mathcal{M}) + \sharp\mathcal{V}_\partial(\mathcal{M}) \ . \tag{12.3.5}$$

   No boundary conditions are imposed.

2. The basis functions of $\mathcal{S}_2^0(\mathcal{M})$ sit on edges and nodes. Since functions in $\mathcal{S}_{2,0}^0(\mathcal{M})$ vanish on $\partial\Omega$ we have to remove those belonging to edges and nodes on the boundary, which leaves us with a single global shape function for every *interior* node and edge,

$$N = \dim \mathcal{S}_{2,0}^0(\mathcal{M}) = \sharp\mathcal{E}_i(\mathcal{M}) + \sharp\mathcal{V}_i(\mathcal{M}) \ . \tag{12.3.6}$$

SOLUTION of (12-3.b):

Recall that the : operator means componentwise multiplication of matrices. Thus we can rewrite (12.3.3) as, $\mathbf{f} = \begin{bmatrix} f_x \\ f_y \end{bmatrix}$,

$$\begin{aligned}
\int_\Omega \mathbf{grad}\, v_{x,h} \cdot \mathbf{grad}\, w_{x,h}\, \mathrm{d}\boldsymbol{x} + \int_\Omega \frac{\partial w_{x,h}}{\partial x_1}\, p_h\, \mathrm{d}\boldsymbol{x} &= \int_\Omega f_x \cdot w_{x,h} \mathrm{d}\boldsymbol{x} \quad \forall w_{x,h} \in \mathcal{S}^0_{2,0}(\mathcal{M})\,, \\
\int_\Omega \mathbf{grad}\, v_{y,h} \cdot \mathbf{grad}\, w_{y,h}\, \mathrm{d}\boldsymbol{x} + \int_\Omega \frac{\partial w_{y,h}}{\partial x_2}\, p_h\, \mathrm{d}\boldsymbol{x} &= \int_\Omega f_y \cdot w_{y,h} \mathrm{d}\boldsymbol{x} \quad \forall w_{y,h} \in \mathcal{S}^0_{2,0}(\mathcal{M})\,, \\
\int_\Omega \frac{\partial v_{x,h}}{\partial x_1}\, q_h\, \mathrm{d}\boldsymbol{x} + \int_\Omega \frac{\partial v_{y,h}}{\partial x_2}\, q_h\, \mathrm{d}\boldsymbol{x} &= 0 \quad \forall q_h \in \mathcal{S}^0_1(\mathcal{M})\,.
\end{aligned}$$

(i) $\mathbf{A}$ is the $\mathcal{S}^0_{2,0}(\mathcal{M}) \times \mathcal{S}^0_{2,0}(\mathcal{M})$-Galerking matrix for the bilinear form

$$(u,v) \mapsto \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1(\Omega)\,. \tag{12.3.7}$$

(ii) $\mathbf{B}_x$ is the $\mathcal{S}^0_{2,0}(\mathcal{M}) \times \mathcal{S}^0_1(\mathcal{M})$-Galerking matrix for the bilinear form

$$(u,q) \mapsto \int_\Omega \frac{\partial u}{\partial x_1}(\boldsymbol{x})\, q(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad u \in H^1(\Omega),\ q \in L^2(\Omega)\,. \tag{12.3.8}$$

(iii) $\mathbf{B}_y$ is the $\mathcal{S}^0_{2,0}(\mathcal{M}) \times \mathcal{S}^0_1(\mathcal{M})$-Galerking matrix for the bilinear form

$$(u,q) \mapsto \int_\Omega \frac{\partial u}{\partial x_2}(\boldsymbol{x})\, q(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad u \in H^1(\Omega),\ q \in L^2(\Omega)\,. \tag{12.3.9}$$

For both $\mathbf{B}_x$ and $\mathbf{B}_y$ trial and test space must not be swapped, because in (12.3.4) both matrices act on vectors $\vec{\boldsymbol{v}}_x$ and $\vec{\boldsymbol{v}}_y$, respectively, which are basis expansion coefficient vectors for $\mathcal{S}^0_{2,0}(\mathcal{M})$. This is the trial space and comes first.

SOLUTION of (12-3.c):

As we have seen in Sub-problem (12-3.b) the columns of $\mathbf{B}_x$ are associated with the global shape functions b_h^j, $j = 1, \ldots, N$, of $\mathcal{S}^0_{2,0}(\mathcal{M})$, which are sitting on nodes and edges, and its rows are induced by the nodal basis functions q_h^j, $j = 1, \ldots, M$, of $\mathcal{S}^0_1(\mathcal{M})$ (associated with mesh nodes).

We have to take into account that zero boundary conditions are imposed for the space $\mathcal{S}^0_{2,0}(\mathcal{M})$. Thus one global shape function of $\mathcal{S}^0_{2,0}(\mathcal{M})$ is associated with each *interior* node, and one with each *interior* edge. The dimension of $\mathcal{S}^0_1(\mathcal{M})$ agrees with the total number of nodes:

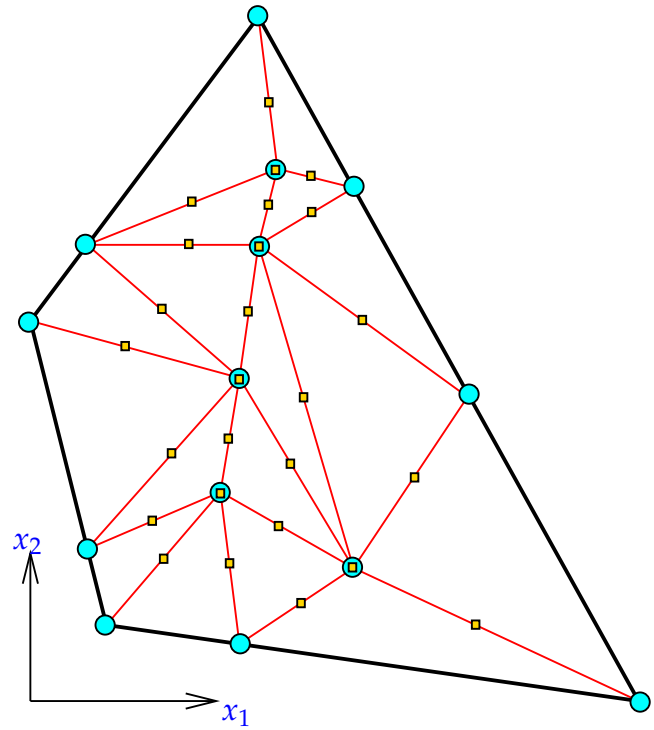
$$\blacktriangleright \quad \mathbf{B}_x \in \mathbb{R}^{\sharp\mathcal{V}_\partial(\mathcal{M})+\sharp\mathcal{V}_i(\mathcal{M}),\sharp\mathcal{V}_i(\mathcal{M})+\sharp\mathcal{E}_i(\mathcal{M})} = \mathbb{R}^{14,26} \,.$$

Since the nodes of $\mathcal{M}$ seems to be in rather "general positions", it is reasonable to assume

$$(\mathbf{B}_x)_{i,j} \neq 0 \quad \Leftrightarrow \quad \int_\Omega \frac{\partial b_h^j}{\partial x_1}(\boldsymbol{x})\, q_h^j(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \neq 0 \quad \Leftrightarrow \quad \mathrm{vol}(\mathrm{supp}(b_h^j) \cap \mathrm{supp}(q_h^j)) > 0 \,. \tag{12.3.10}$$

In other words,

> entries of $\mathbf{B}_x$ are non-zero if and only if the supports of the corresponding two global shape functions overlap.



◁ Cyan disks marks the nodes carrying a global shape function of $\mathcal{S}^0_1(\mathcal{M})$.

Orange squares indicated the locations of the global shape functions of $\mathcal{S}^0_{2,0}(\mathcal{M})$

➢ Column of $\mathbf{B}_x$ associated with the node •: Non zero entries of that column arise from tent functions whose supports overlap with the support of the global shape function of $\mathcal{S}^0_2(\mathcal{M})$ attached to •. Their number agrees with the number of nodes connected to • with an edge plus one (for the tent function sitting in •). We count,

$$\mathrm{nnz}((\mathbf{B}_x)_{:,\bullet}) = 7 \,.$$

➢ Row of $\mathbf{B}_x$ associated with the node •: Non-zero entries of that row are induced by global shape functions of $\mathcal{S}^0_{2,0}(\mathcal{M})$, whose supports overlap the support of the tent function associated with •. There is one for each edge emanating from •, one for each *interior* node connected/coinciding with • and one for each *interior* edge opposite to •. We count

$$\operatorname{nnz}((\mathbf{B}_x)_{\bullet,:}) = 4+6+4 = 14 \ .$$

Now let us bound the total number of non-zero entries of the matrix $\mathbf{B}_x$.

- The support of a global shape function b^j_h sitting on an interior node $\boldsymbol{p}$ overlaps with the supports of all q^j_h associated with endpoints of edges emanating from $\boldsymbol{p}$ and with that of that q_h sitting on $\boldsymbol{p}$ itself. We count that there is a total of $27 = 4+6+6+5+6$ edges starting in interior nodes (edges with two interior endpoints counted twice) and $5 = \sharp\mathbf{a}_i(\mathcal{M})$ interior nodes themselves. Hence the columns of $\mathbf{B}_x$ belonging to interior nodes contain a total of $32 = 5+27$ non-zero entries.
- The support of global shape functions b^j_h belonging to an interior edge (with two adjacent triangles) always overlaps with the support of four node-associated basis functions q^i_h, which leads to $4 \cdot \sharp\mathcal{E}_i(\mathcal{M}) = 4 \cdot 21 = 84$ non-zero entries.

$$\operatorname{nnz}(\mathbf{B}_x) = 32+84 = 116 \ . \tag{12.3.11}$$

SOLUTION of (12-3.d):

C++ code 12.3.12: Part of `buildStokesLSE()`

```
template <typename FFUNCTOR>
std::tuple<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>,
           Eigen::SparseMatrix<double>, Eigen::VectorXd, Eigen::VectorXd>
buildStokesLSE(std::shared_ptr<const lf::mesh::Mesh> mesh_p, FFUNCTOR &&force) {
  LF_ASSERT_MSG(mesh_p != nullptr, "Mesh must be supplied!");
  // I. Assemble full Galerkin matrix A in triplet format
  // Initialize quadratic Lagrange FE space
  auto fes_velo_comp =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO2<double>>(mesh_p);
  // Unit "coefficient function", μ = 1!
  auto one = [](const Eigen::Vector2d & /*x*/) -> double { return 1.0; };
  // Create unit mesh function
  const lf::mesh::utils::MeshFunctionGlobal mf_one{one};
  // Element matrix provider to −Δ bilinear form
  lf::fe::DiffusionElementMatrixProvider Laplace_emp(fes_velo_comp, mf_one);
  // local → global index mapping
  const lf::assemble::DofHandler &dofh_velo_comp{fes_velo_comp->LocGlobMap()};
  const lf::assemble::size_type N_velo_comp = dofh_velo_comp.NumDofs();
  // Full matrix A in triplet format
  lf::assemble::COOMatrix<double> A_velo_comp(N_velo_comp, N_velo_comp);
  lf::assemble::AssembleMatrixLocally(0, dofh_velo_comp, dofh_velo_comp,
                                      Laplace_emp, A_velo_comp);
```

HINT 1 for (12-3.e): This is the provided incomplete code with comments, which should be read carefully.

```
THBElementMatrixProvider::ElemMat THBElementMatrixProvider::Eval(
    const lf::mesh::Entity &cell) {
  ElemMat BK;  // Element matrix to be filled
  // Area of the triangle
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions λ_i, see
  // [Lecture → Code 2.4.5.11].
  // Get the coordinates of the vertices of the triangle
  auto endpoints = lf::geometry::Corners(*(cell.Geometry()));
  Eigen::Matrix<double, 3, 3> X;  // temporary matrix
  X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
  X.block<3, 2>(0, 1) = endpoints.transpose();
  // This matrix contains grad λ_i in its columns
  const auto G{X.inverse().block<2, 3>(1, 0)};
  // Dummy lambda functions for barycentric coordinates
  std::array<std::function<double(Eigen::Vector3d)>, 3> lambda{
      [](Eigen::Vector3d c) -> double { return c[0]; },
      [](Eigen::Vector3d c) -> double { return c[1]; },
      [](Eigen::Vector3d c) -> double { return c[2]; }};
  // Gradients of local shape functions of quadratic Lagrangian finite
  // element space as lambda functions, see [Lecture → (2.6.1.6)].
  std::array<std::function<Eigen::Vector2d(Eigen::Vector3d)>, 6> gradbK{
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return (4 * c[0] - 1) * G.col(0);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return (4 * c[1] - 1) * G.col(1);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return (4 * c[2] - 1) * G.col(2);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[0] * G.col(1) + c[1] * G.col(0));
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[1] * G.col(2) + c[2] * G.col(1));
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[2] * G.col(0) + c[0] * G.col(2));
      }};
  // Barycentric coordinates of the midpoints of the edges for
  // use with the 3-point edge midpoint quadrature rule (12.3.13)
  const std::array<Eigen::Vector3d, 3> mp = {Eigen::Vector3d({0.5, 0.5, 0}),
                                             Eigen::Vector3d({0, 0.5, 0.5}),
                                             Eigen::Vector3d({0.5, 0, 0.5})};
  /* **********************************************************************
     Your code here
     ********************************************************************** */
  return BK;
}
```

Using the lambda functions in `lambda` and `gradbK` is key to solving this sub-problem in a few lines.

Note that the lambda functions `lambda[i]` and `gradbK[j]` take vectors of barycentric coordinates as arguments, *cf.* (12.3.13). The array `mp` contains the barycentric coordinate vectors of the midpoints of the

edges.

SOLUTION of (12-3.e):

Local shape functions:

- Let λ_i, $i = 1,2,3$, denote the barycentric coordinate functions for the triangle K.
- Write b_K^j, $j = 1,\ldots,6$, for the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ as defined in [Lecture → (2.6.1.6)].

$$\begin{aligned} &b_K^1 = (2\lambda_1 - 1)\lambda_1\,, \quad b_K^2 = (2\lambda_2 - 1)\lambda_2\,, \quad b_K^3 = (2\lambda_3 - 1)\lambda_3\,,\\ &b_K^4 = 4\lambda_1\lambda_2\,, \qquad b_K^5 = 4\lambda_2\lambda_3\,, \qquad b_K^6 = 4\lambda_1\lambda_3\,. \end{aligned} \qquad \text{[Lecture → (2.6.1.6)]}$$

  The numbering already agrees with the numbering conventions of LEHRFEM++, see [Lecture → § 2.7.4.11].

The entries of the element matrix for the triangle K are obtained by plugging the local shape functions into the underlying localized bilinear form:

$$(\mathbf{B}_K)_{i,j} = \int_K \frac{\partial b_K^j}{\partial x_*}(\boldsymbol{x})\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad \begin{array}{l} *=1,2\,, \quad \text{(controlled by } \texttt{dirflag_}\text{)}\\ i \in \{1,2,3\}\,,\\ j \in \{1,\ldots,6\}\,. \end{array} \tag{12.3.14}$$

The derivative of any b_K^j is an affine linear function and so are the λ_i. Therefore the integrand in (12.3.14) is a quadratic polynomial and the midpoint quadrature rule provides the exact value

$$(\mathbf{B}_K)_{i,j} = \tfrac{1}{3}|K| \sum_{\ell=1}^{3} \frac{\partial b_K^j}{\partial x_*}(\boldsymbol{m}_\ell)\,\lambda_i(\boldsymbol{m}_\ell)\,, \tag{12.3.15}$$

where $\boldsymbol{m}_1, \boldsymbol{m}_2, \boldsymbol{m}_3$ are the midpoints of the edges of K. This is the formula that is implemented in the code below.

C++ code 12.3.16: Implementation of `Eval()` member function of THBElementMatrixProvider.

```
THBElementMatrixProvider::ElemMat THBElementMatrixProvider::Eval(
    const lf::mesh::Entity &cell) {
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type " << cell.RefEl());
  // Element matrix to be filled
  ElemMat BK;
  // Area of the triangle
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions, see
  // [Lecture → Code 2.4.5.11].
  // Get vertices of the triangle
  auto endpoints = lf::geometry::Corners(*(cell.Geometry()));
  Eigen::Matrix<double, 3, 3> X;  // temporary matrix
  X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
  X.block<3, 2>(0, 1) = endpoints.transpose();
  // This matrix contains grad λ_i in its columns
  const auto G{X.inverse().block<2, 3>(1, 0)};
  // Dummy lambda functions for barycentric coordinates
  std::array<std::function<double(Eigen::Vector3d)>, 3> lambda{
      [](Eigen::Vector3d c) -> double { return c[0]; },
      [](Eigen::Vector3d c) -> double { return c[1]; },
```

```
      [](Eigen::Vector3d c) -> double { return c[2]; }};
  // Gradients of local shape functions of quadratic Lagrangian finite
     element
  // space as lambda functions, see (??)
  std::array<std::function<Eigen::Vector2d(Eigen::Vector3d)>, 6> gradbK{
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return (4 * c[0] - 1) * G.col(0);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return (4 * c[1] - 1) * G.col(1);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return (4 * c[2] - 1) * G.col(2);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[0] * G.col(1) + c[1] * G.col(0));
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[1] * G.col(2) + c[2] * G.col(1));
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return 4 * (c[2] * G.col(0) + c[0] * G.col(2));
      }};
  // Barycentric coordinates of the midpoints of the edges for
  // use with the 3-point edge midpoint quadrature rule (12.3.13)
  const std::array<Eigen::Vector3d, 3> mp = {Eigen::Vector3d({0.5, 0.5, 0}),
                                             Eigen::Vector3d({0, 0.5, 0.5}),
                                             Eigen::Vector3d({0.5, 0, 0.5})};
  // Initialize the element matrix for the bilinear form b(.,.)
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
      // (12.3.15) without the area scaling
      const Eigen::Vector2d gql_ij{gradbK[j](mp[0]) * lambda[i](mp[0]) +
                                   gradbK[j](mp[1]) * lambda[i](mp[1]) +
                                   gradbK[j](mp[2]) * lambda[i](mp[2])};
      BK(i, j) = gql_ij[dirflag_];
    }
  }
  // Do not forget to scale by \frac{1}{3}|K|.
  BK *= area / 3.0;
  return BK;
}
```

SOLUTION of (12-3.f):

The task boils down to coding Lines 14-23 of Code 12.3.18 in C++ using EIGEN's high-level numerical linear algebra operations.

- Multiplication with $\mathbf{A}^{-1}$ means solving a linear system of equations with system matrix $\mathbf{A}$. We use EIGEN's sparse direct solver. For the sake of efficiency it is important to precompute the LU-decomposition of $\mathbf{A}$.
- Multiplication with $\mathbf{B}$ involves multiplications with $\mathbf{B}_x$ and $\mathbf{B}_y$ and summation:

$$\begin{bmatrix}\mathbf{B}_x & \mathbf{B}_y\end{bmatrix}\begin{bmatrix}\vec{\eta}_x \\ \vec{\eta}_y\end{bmatrix} = \mathbf{B}_x\vec{\eta}_x + \mathbf{B}_y\vec{\eta}_y \,.$$

- Multiplication with $\mathbf{B}^\top$ results in two vectors

$$\begin{bmatrix}\mathbf{B}_x & \mathbf{B}_y\end{bmatrix}^\top\vec{\nu} = \begin{bmatrix}\mathbf{B}_x\vec{\nu} \\ \mathbf{B}_y\vec{\nu}\end{bmatrix} \,.$$

C++ code 12.3.19: Implementation of `CGUzawa()`

```
2  template <typename RECORDER = std::function<void(const Eigen::VectorXd &,
3                                                   const Eigen::VectorXd &,
4                                                   const Eigen::VectorXd &)>>
5  std::tuple<Eigen::VectorXd, Eigen::VectorXd, Eigen::VectorXd> CGUzawa(
6      const Eigen::SparseMatrix<double> &A,
7      const Eigen::SparseMatrix<double> &B_x,
8      const Eigen::SparseMatrix<double> &B_y, const Eigen::VectorXd &phi_x,
9      const Eigen::VectorXd &phi_y, double rtol = 1E-6, double atol = 1E-8,
10     unsigned int itmax = 100,
11     RECORDER &&rec = [](const Eigen::VectorXd &mu_x,
12                         const Eigen::VectorXd &mu_y,
13                         const Eigen::VectorXd &pi) -> void {}) {
14   const Eigen::Index N = A.cols();
15   const Eigen::Index M = B_x.rows();
16   // Check consistent sizes of matrices
17   LF_VERIFY_MSG(A.cols() == A.rows(), "A must be square");
18   LF_VERIFY_MSG(B_x.cols() == N, "B_x.cols() == A.cols() required");
19   LF_VERIFY_MSG(B_y.cols() == N, "B_y.cols() == A.cols() required");
20   LF_VERIFY_MSG(B_y.rows() == M, "B_x and B_y must have the same size");
21   // (Sparse) LU-decomposition of A: done once for the sake of
       efficiency!
22   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_A;
23   solver_A.compute(A);
24   LF_VERIFY_MSG(solver_A.info() == Eigen::Success, "LU decomposition failed");
25   // Initialize the remaining vectors
26   Eigen::VectorXd pi(M);
27   pi.setZero();
28   Eigen::VectorXd mu_x = solver_A.solve(phi_x);
29   LF_VERIFY_MSG(solver_A.info() == Eigen::Success, "Solving x LSE failed");
30   Eigen::VectorXd mu_y = solver_A.solve(phi_y);
31   LF_VERIFY_MSG(solver_A.info() == Eigen::Success, "Solving y LSE failed");
32   Eigen::VectorXd delta = B_x * mu_x + B_y * mu_y;
33   Eigen::VectorXd rho = -delta;
34   const double rn0 = rho.norm();
```

```
35    double rn = rn0;
36    Eigen::VectorXd omega_x(N);
37    Eigen::VectorXd omega_y(N);
38    Eigen::VectorXd eta_x(N);
39    Eigen::VectorXd eta_y(N);
40    // Main iteration loop
41    unsigned int step = 0;
42    do {
43      if (rn >= atol) {
44        // ω⃗ := B^⊤ δ⃗;
45        omega_x = B_x.transpose() * delta;
46        omega_y = B_y.transpose() * delta;
47        // η⃗ := A^{-1} ω⃗;
48        eta_x = solver_A.solve(omega_x);
49        eta_y = solver_A.solve(omega_y);
50        // α = (rn * rn) / ω⃗^⊤ η⃗;
51        const double alpha =
52            (rn * rn) / (omega_x.dot(eta_x) + omega_y.dot(eta_y));
53        // π⃗ := π⃗ + α δ⃗;
54        pi += alpha * delta;
55        // μ⃗ := μ⃗ − α η⃗;
56        mu_x -= alpha * eta_x;
57        mu_y -= alpha * eta_y;
58        // ρ⃗ := γ⃗ − B μ⃗;
59        rho = -(B_x * mu_x + B_y * mu_y);
60        const double rn1 = rho.norm();
61        const double beta = (rn1 * rn1) / (rn * rn);
62        rn = rn1;
63        // δ⃗ := −ρ⃗ + β δ⃗;
64        delta = -rho + beta * delta;
65      }
66      rec(mu_x, mu_y, pi);
67      step++;
68    } while ((rn >= rn0 * rtol) and (rn >= atol) and (step < itmax));
69    return {mu_x, mu_y, pi};
70  }
```

SOLUTION of (12-3.g):

Writing n for the number of iteration steps on $\mathcal{M}_0$, we have on $\mathcal{M}_0$

$$\left\| \mathbf{v}_h^{(n)} - \mathbf{v}_h \right\|_{H^1(\Omega)} \approx \| \mathbf{v}_h - \mathbf{v} \|_{H^1(\Omega)} \,. \tag{12.3.21}$$

By linear convergence with *sharp rate* q and initial guess $= \mathbf{0}$ we know

$$\left\| \mathbf{v}_h^{(n)} - \mathbf{v}_h \right\|_{H^1(\Omega)} \approx q^n \| \mathbf{v}_h \|_{H^1(\Omega)} \,. \tag{12.3.22}$$

If the Stokes BVP has smooth solution we known from [Lecture → Thm. 12.3.4.4] that

$$\| \mathbf{v}_h - \mathbf{v} \|_{H^1(\Omega)} = O(h_{\mathcal{M}}^2) \quad \text{for} \quad h_{\mathcal{M}} \to 0 \,.$$

We are allowed to use the sharper version

$$\| \mathbf{v}_h - \mathbf{v} \|_{H^1(\Omega)} \approx C h_{\mathcal{M}}^2 \quad \text{for} \quad h_{\mathcal{M}} \ll 1 \,. \tag{12.3.23}$$

Uniform regular refinement halves the meshwidth. Thus, writing $\widetilde{\mathbf{v}}_h$ for the Taylor-Hood velocity solution on the refined mesh we infer from (12.3.23)

$$\| \widetilde{\mathbf{v}}_h - \mathbf{v} \|_{H^1(\Omega)} \approx \tfrac{1}{4} \| \mathbf{v}_h - \mathbf{v} \|_{H^1(\Omega)} \,. \tag{12.3.24}$$

Let $m > n$ stand for the number of iteration steps required on the refined mesh. It should satisfy

$$\left\| \widetilde{\mathbf{v}}_h^{(m)} - \widetilde{\mathbf{v}}_h \right\|_{H^1(\Omega)} \approx \| \widetilde{\mathbf{v}}_h - \mathbf{v} \|_{H^1(\Omega)} \,.$$

Combine this with (12.3.21) and (12.3.22):

$$q^m \, \| \widetilde{\mathbf{v}}_h \|_{H^1(\Omega)} \approx \tfrac{1}{4} \, \| \mathbf{v}_h - \mathbf{v} \|_{H^1(\Omega)} \approx \tfrac{1}{4} \, q^n \| \mathbf{v}_h \|_{H^1(\Omega)} \,.$$

Since the meshes are rather fine we expect $\| \widetilde{\mathbf{v}}_h \|_{H^1(\Omega)} \approx \| \mathbf{v}_h \|_{H^1(\Omega)}$, which implies

$$q^m \approx \tfrac{1}{4} \, q^n \quad \Rightarrow \quad \boxed{m - n = \left\lceil \left| \frac{\log 4}{\log q} \right| \right\rceil} \,.$$

Short-cut considerations:

- According to [Lecture → Thm. 12.3.4.4] the $H^1(\Omega)$-norm of the velocity discretization error for Taylor-Hood FEM in the case of smooth exact solution converges algebraically with rate 2.
- This means that the $H^1(\Omega)$-norm of the finite element discretization error of $\mathbf{v}_h$ drops by a factor of 4 when halving the meshwidth as it happens during uniform regular refinement.
- So the additional iterations have to achieve a reduction of the iteration error by a factor of 4, too. Thus, their number k must satisfy

$$q^k \leq \frac{1}{4} \quad \Rightarrow \quad \boxed{k = \left\lceil \left| \frac{\log 4}{\log q} \right| \right\rceil} \,.$$

SOLUTION of (12-4.a):

Just recall [Lecture → (12.2.2.44)]: seek velocity $\mathbf{v} \in (H_0^1(\Omega))^d$, pressure $p \in L^2(\Omega)$, *Lagrange multiplier* $\lambda \in \mathbb{R}$ such that

$$\begin{aligned}
\int_\Omega \mu \mathrm{D}\mathbf{v} : \mathrm{D}\mathbf{w}\,\mathrm{d}\boldsymbol{x} \quad + \quad \int_\Omega \operatorname{div}\mathbf{w}\, p\,\mathrm{d}\boldsymbol{x} \qquad\qquad &= \int_\Omega \mathbf{f}\cdot\mathbf{w}\,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{w} \in (H_0^1(\Omega))^d\,, \\
\int_\Omega \operatorname{div}\mathbf{v}\, q\,\mathrm{d}\boldsymbol{x} \qquad\qquad + \quad \lambda \int_\Omega q\,\mathrm{d}\boldsymbol{x} \;&= \; 0 \qquad\quad \forall q \in L^2(\Omega)\,, \\
\int_\Omega p\,\mathrm{d}\boldsymbol{x} \qquad\qquad\qquad\quad &= \; 0 \qquad\quad .
\end{aligned}$$

Here : denotes the componentwise dot-product of two matrices. Zero "0 (or nothing) should be written in the third (from left) box in the first line and the second box in the second line.

SOLUTION of (12-4.b):

As we saw in [Lecture → Section 12.3.1] – refer to this section for explanations –,

$\dim U_h \geq \dim Q_h$ is a *necessary* condition for the uniqueness of the pressure solution p_h of the discrete saddle point variational problem [Lecture → (12.3.0.4)].

Strict inequalities cannot be inferred from the stability of the pair of finite element spaces.

SOLUTION of (12-4.c):

In the course notes you find the theorem stated as follows:

Theorem [Lecture → Thm. 12.3.2.15]. inf-sup condition

The finite element spaces $U_h \subset \boldsymbol{H}_0^1(\Omega)$, $Q_h \subset L_*^2(\Omega)$ provide a stable finite element pair (→ [Lecture → Def. 12.3.2.7]) for the Stokes problem [Lecture → (12.2.2.19)]/[Lecture → (12.3.0.2)] if there is a constant $\beta > 0$ depending only on Ω and the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$ such that

$$\sup_{\mathbf{w}_h \in U_h} \frac{|\mathsf{b}(\mathbf{w}_h, q_h)|}{\|\mathbf{w}_h\|_{\mathsf{a}}} \geq \beta \|q_h\|_{L^2(\Omega)} \quad \forall q_h \in Q_h \,. \qquad \text{([Lecture → (12.3.3.25)])}$$

Here, it was defined

$$\mathsf{b}(\mathbf{w}, q) := \int_\Omega \operatorname{div} \mathbf{w}(\boldsymbol{x})\, q(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,.$$
$$\|\mathbf{w}\|_{\mathsf{a}}^2 := \int_\Omega \mu \mathsf{D}\mathbf{w}(\boldsymbol{x}) : \mathsf{D}\mathbf{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,.$$

Therefore you have to write

Theorem [Lecture → Thm. 12.3.2.15]. inf-sup condition

The finite element spaces $U_h \subset \boldsymbol{H}_0^1(\Omega)$, $Q_h \subset L_*^2(\Omega)$ provide a stable finite element pair (→ [Lecture → Def. 12.3.2.7]) for the Stokes problem [Lecture → (12.2.2.19)]/[Lecture → (12.3.0.2)] if there is a constant $\beta > 0$ depending only on Ω and the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$ such that

$$\sup_{\mathbf{w}_h \in U_h} \frac{\int_\Omega \operatorname{div} \mathbf{w}_h(\boldsymbol{x})\, q_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}{\sqrt{\int_\Omega \mu \mathsf{D}\mathbf{w}_h(\boldsymbol{x}) : \mathsf{D}\mathbf{w}_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}} \geq \beta \, \|q_h\|_{L^2(\Omega)} \quad \forall q_h \in Q_h \,. \quad \text{([Lecture → (12.3.3.25)])}$$

Note that in the weighting by the coefficient μ in the denominator can be dropped. In fact, one could also write

$$\sup_{\mathbf{w}_h \in U_h} \frac{\int_\Omega \operatorname{div} \mathbf{w}_h(\boldsymbol{x})\, q_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}{|\mathbf{w}_h|_{H^1(\Omega)}} \geq \beta \|q_h\|_{L^2(\Omega)} \quad \forall q_h \in Q_h \,.$$

Owing to the Poincaré-Friedrichs inequality [Lecture → Thm. 1.3.4.17], in this formula the semi-norm in the denominator could even be replaced with the full $(H^1(\Omega))^2$-norm:

$$\sup_{\mathbf{w}_h \in U_h} \frac{\int_\Omega \operatorname{div} \mathbf{w}_h(\boldsymbol{x})\, q_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}{\|\mathbf{w}_h\|_{H^1(\Omega)}} \geq \beta \|q_h\|_{L^2(\Omega)} \quad \forall q_h \in Q_h \,.$$

SOLUTION of (12-4.d):

We appeal to [Lecture → Thm. 12.3.2.15] and wonder how replacing one or both of the finite element spaces in [Lecture → (12.3.3.25)] will affect the inequality.

- Thanks to the sup replacing U_h with a larger space $\widetilde{U}_h$ the left-hand side of [Lecture → (12.3.3.25)] will remain at least as big. So the $\mathcal{M}$-uniform inf-sup condition is preserved.
- Replacing Q_h with a larger space $\widetilde{Q}_h$ can destroy stability. Just choose $\widetilde{Q}_h$ so large that $\dim \widetilde{Q}_h >$ di and recall the solution of Sub-problem (12-4.b).
- Replacing both spaces with larger spaces has an effect on [Lecture → (12.3.3.25)] that is a priori undetermined. We can not predict that the resulting finite element scheme will be stable.

Thus $(\widetilde{U}_h, Q_h)$ is the only finite element pair that is predictably stable.

Pair of spaces	Guaranteed to be a stable pair?			
$(U_h, \widetilde{Q}_h)$	YES	○	NO	✔
$(\widetilde{U}_h, Q_h)$	YES	✔	NO	○
$(\widetilde{U}_h, \widetilde{Q}_h)$	YES	○	NO	✔

SOLUTION of (12-4.e):

Of course we invoke the following result

Theorem [Lecture → Thm. 12.3.3.13]. Convergence of stable FE for Stokes problem

If U_h, Q_h is a stable finite element pair (→ [Lecture → Def. 12.3.2.7]) for the Stokes variational saddle point problem [Lecture → (12.2.2.19)], then the corresponding finite element Galerkin solution $(\mathbf{v}_h, p_h)$ satisfies

$$\|\mathbf{v}-\mathbf{v}_h\|_{H^1(\Omega)} + \|p-p_h\|_{L^2(\Omega)} \leq C\left(\inf_{\mathbf{w}_h\in U_h}\|\mathbf{v}-\mathbf{w}_h\|_{H^1(\Omega)} + \inf_{q_h\in Q_h}\|p-q_h\|_{L^2(\Omega)}\right),$$

with a constant $C>0$ that depends only on Ω, μ, and the shape regularity of the finite element mesh.

for $U_h := \mathcal{S}^0_{2,0}(\mathcal{M})$ and $Q_h := \mathcal{S}^{-1}_0(\mathcal{M})$. This means that we have to recall best approximation estimates

- For $\mathcal{S}^0_{2,0}(\mathcal{M})$ appeal to [Lecture → Cor. 3.3.3.4] applied componentwise, which tells us

  $$\inf_{\mathbf{w}_h\in\mathcal{S}^0_{2,0}(\mathcal{M})}\|\mathbf{v}-\mathbf{w}_h\|_{H^1(\Omega)} \leq Ch^2_{\mathcal{M}}\|\mathbf{v}\|_{H^2(\Omega)}$$

  with a constant depending only on the shape regularity measure of $\mathcal{M}$.

- For $\mathcal{S}^{-1}_0(\mathcal{M})$ we can only guess guided by the convergence properties of the $(\mathcal{RM}(\mathcal{M}), \mathcal{S}^{-1}_0(\mathcal{M}))$ stable pair of finite element spaces discussed in [Lecture → Section 12.3.5]. The we observed first-order asymptotic convergence in the $H^1(\Omega)\times L^2(\Omega)$-norm for velocity and pressure, which hints at

  $$\inf_{q_h\in\mathcal{S}^{-1}_0(\mathcal{M})}\|p-q_h\|_{L^2(\Omega)} = O(h_{\mathcal{M}}) \quad \text{for} \quad h_{\mathcal{M}}\to 0\,. \tag{12.4.1}$$

  In fact, we already have the tools to prove this. Recall

  Theorem [Lecture → Thm. 1.8.0.20]. Second Poincaré-Friedrichs inequality

  If $\Omega\subset\mathbb{R}^d$, $d\in\mathbb{N}$, is bounded and connected, then

  $$\exists C = C(\Omega)>0:\quad \|u\|_0 \leq C\,\operatorname{diam}(\Omega)\|\mathbf{grad}\,u\|_0 \quad \forall u\in H^1_*(\Omega)\,.$$

  Now, on every cell K of the mesh let $q_h\in\mathcal{S}^{-1}_0(\mathcal{M})$ be defined by a local $L^2(K)$-orthogonal projection,

  $$\int_\Omega p(\boldsymbol{x}) - q_h|_K(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0\,. \tag{12.4.2}$$

  Then apply [Lecture → Thm. 1.8.0.20] with $\Omega = k$ and $u := p - q_h|_K \in H^1_*(K)$. We get

  $$\|p-q_h|_K\|_{L^2(K)} \leq Ch_K|p|_K|_{H^1(\Omega)}\,, \tag{12.4.3}$$

  with $C>0$ merely depending on the shape (regularity) of K. Squaring and adding these estimates gives (12.4.1). Choosing p as a simple affine linear function on Ω then shows that the power of $h_{\mathcal{M}}$ in (12.4.1) is the highest possible.

Hence, the approximation error of (12.4.1) limits the overall rate of asymptotic convergence and we obtain first-order convergence,

$$\|\mathbf{v} - \mathbf{v}_h\|_{H^1(\Omega)} + \|p - p_h\|_{L^2(\Omega)} = O(h_{\mathcal{M}}) \quad \text{for meshwidth} \quad h_{\mathcal{M}} \to 0 \,.$$

SOLUTION of (12-5.a):

The operator I_1 relies on point evaluations of its arguments in the nodes of the mesh.

Point evaluation, regarded as a linear functional, is not well-defined on $H^1(\Omega)$, because that space can contain unbounded functions as we have learned in [Lecture → Ex. 1.2.3.45]. Of course, then the product pq of $p, q \in H^1(\Omega)$ may also be unbounded in a point.

So the extra bilinear form introduced for the sake of stabilization

$$(p,q) \mapsto \int_\Omega \mathsf{I}_1(pq)(\boldsymbol{x}) - p(\boldsymbol{x})q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$

is *not defined* on $H^1(\Omega)$; it cannot be evaluated for all $p, q \in H^1(\Omega)$.

Remark. Since $\mathcal{S}_1^0(\mathcal{M}) \subset C^0(\overline{\Omega})$, the stabilization bilinear form can easily be evaluated for any $p_h, q_h \in \mathcal{S}$

SOLUTION of (12-5.b):

Preliminary considerations. By the very way the question is posed, it is clear that $\vec{\zeta}$ cannot depend on $\mathcal{M}$. The only remaining realistic option is a partly constant vector.

By Gauss theorem [Lecture $\rightarrow$ Thm. 1.5.2.4] we have

$$\int_\Omega \operatorname{div} \mathbf{w}_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall \mathbf{w}_h \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2 .$$

Thus,

$$\mathbf{B}^\top \mathbf{1} = \left[\sum_{j=1}^N \int_\Omega \operatorname{div} \mathbf{b}_h^i(\boldsymbol{x})\, b_h^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i=1}^{2n} = \left[\int_\Omega \operatorname{div} \mathbf{b}_h^i \underbrace{\sum_{j=1}^N b_h^j(\boldsymbol{x})}_{\equiv 1}\,\mathrm{d}\boldsymbol{x}\right]_{i=1}^{2n} = \mathbf{0}\,,$$

where $\mathbf{1} \in \mathbb{R}^N$ is a vector with all components $= 1$.

The basis expansion coefficient vector $\mathbf{1}$ for a $p_h \in \mathcal{S}^0_1(\mathcal{M})$ means $p_h \equiv 1$. In particular, we have

$$\mathsf{I}_1(p_h q_h) = \mathsf{I}_1 q_h = q_h \quad \forall q_h \in \mathcal{S}^0_1(\mathcal{M})\,.$$

Thus for $p_h \equiv 1$

$$\int_\Omega \mathsf{I}_1(p_h q_h)(\boldsymbol{x}) - p_h(\boldsymbol{x}) q_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega q_h(\boldsymbol{x}) - q_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall q_h \in \mathcal{S}^0_1(\mathcal{M})\,.$$

We infer that also $\mathbf{S}\mathbf{1} = \mathbf{0}$, which means that

$$\begin{bmatrix} \mathbf{A} & \mathbf{B}^\top \\ \mathbf{B} & -\mathbf{S} \end{bmatrix} \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix} = \mathbf{0} \quad \blacktriangleright \quad \vec{\zeta} = \begin{bmatrix} \mathbf{0} \\ \mathbf{1} \end{bmatrix}\,.$$

Note that the vector $[\mathbf{1}\,\mathbf{0}]^\top$ is not a solution, because $\mathbf{A}$ is *positive definite* due to the zero Dirichlet boundary conditions imposed on the components of the velocity.

SOLUTION of (12-5.c):

Recall that the **local shape functions** for $\mathcal{S}_1^0(\mathcal{M})$ and a triangular cell are the barycentric coordinate functions λ_1, λ_2, and λ_3 of the triangle, their numbering induced by the numbering of the vertices [Lecture → § 2.4.5.2].

Thus, for the monolithic stabilized Stokes P1-FEM the ordered set of local shape functions is

$$\left\{ \begin{bmatrix} \lambda_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_1 \end{bmatrix}, \lambda_1, \begin{bmatrix} \lambda_2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_2 \end{bmatrix}, \lambda_2, \begin{bmatrix} \lambda_3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_3 \end{bmatrix}, \lambda_3 \right\} . \tag{12.5.5}$$

Here we followed LEHRFEM++'s ordering convention from [Lecture → § 2.7.4.11] and our own rule stated above.

However, note that for the sake of simplicity of presentation, we are going to rely on a different ordering

$$\left\{ \begin{bmatrix} \lambda_1 \\ 0 \end{bmatrix}, \begin{bmatrix} \lambda_2 \\ 0 \end{bmatrix}, \begin{bmatrix} \lambda_3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_1 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_2 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_3 \end{bmatrix}, \lambda_1, \lambda_2, \lambda_3 \right\} . \tag{12.5.6}$$

The mapping between the two orderings (12.5.5) and (12.5.6) is done through the index mapping arrays `vx_idx`, `vy_idx`, and `p_idx`. With respect to (12.5.6) the 9×9 element matrix has the block structure

$$\mathbf{M}_K = \begin{bmatrix} \mathbf{A}_{K,1} & \mathbf{O} & \mathbf{B}_{K,1}^\top \\ \mathbf{O} & \mathbf{A}_{K,2} & \mathbf{B}_{K,2}^\top \\ \mathbf{B}_{K,1} & \mathbf{B}_{K,2} & -\mathbf{S}_K \end{bmatrix} , \quad \mathbf{A}_{K,1}, \mathbf{A}_{K,2}, \mathbf{B}_{K,1}, \mathbf{B}_{K,2}, \mathbf{S} \in \mathbb{R}^{3,3} . \tag{12.5.7}$$

The formulas can be expressed more compactly when using the matrix

$$\mathbf{G} := \begin{bmatrix} \mathbf{grad}\,\lambda_1 & \mathbf{grad}\,\lambda_2 & \mathbf{grad}\,\lambda_3 \end{bmatrix} \in \mathbb{R}^{2,3}$$

collecting the constant gradients of the barycentric coordinate functions in its columns.

❶ The (symmetric) matrix blocks $\mathbf{A}_{K,1}$ and $\mathbf{A}_{k,2}$ are the same and equal to the $\mathcal{S}_1^0(\mathcal{M})$ element matrix for $-\Delta$:

$$\mathbf{A}_{K,1} = \mathbf{A}_{K,2} = \left[\int_K \mathbf{grad}\,\lambda_j(\boldsymbol{x}) \cdot \mathbf{grad}\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \right]_{i,j=1}^3 = |K| \mathbf{G}^\top \mathbf{G} .$$

❷ We have

$$\mathbf{B}_{K,\ell} = \left[\int_K \frac{\partial \lambda_j}{\partial x_\ell}(\boldsymbol{x})\, \lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \right]_{i,j=1}^3 = \tfrac{1}{3}|K| \begin{bmatrix} \mathbf{G}_{\ell,:} \\ \mathbf{G}_{\ell,:} \\ \mathbf{G}_{\ell,:} \end{bmatrix} , \quad \ell = 1,2 , \tag{12.5.8}$$

because $\int_K \lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{1}{3}|K|$.

❸ The contribution due to stabilization is the symmetric matrix

$$\mathbf{S}_K = \left[\int_K \mathsf{I}_1(\lambda_i\lambda_j)(\boldsymbol{x}) - \lambda_i(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \right]_{i,j=1}^3 . \tag{12.5.9}$$

Locally we have

$$\mathsf{I}_1 f = f(\boldsymbol{a}_K^1)\lambda_1 + f(\boldsymbol{a}_K^2)\lambda_2 + f(\boldsymbol{a}_K^3)\lambda_3 , \quad \boldsymbol{a}_K^1, \boldsymbol{a}_K^2, \boldsymbol{a}_K^3 = \text{vertices of } K ,$$

which means, since $\lambda_k(\boldsymbol{a}_K^\ell) = \delta_{\ell,k}$,

$$\mathsf{I}_1(\lambda_i\lambda_j) = \begin{cases} \lambda_i & \text{, if } \quad i = j\,, \\ 0 & \text{else,} \end{cases} \qquad i,j = 1,2,3\,.$$

$$\blacktriangleright \quad \mathbf{S}_K = \tfrac{1}{3}|K| \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \begin{bmatrix} \int_K \lambda_1^2 \,\mathrm{d}\boldsymbol{x} & \int_K \lambda_1\lambda_2 \,\mathrm{d}\boldsymbol{x} & \int_K \lambda_1\lambda_3 \,\mathrm{d}\boldsymbol{x} \\ \int_K \lambda_2\lambda_1 \,\mathrm{d}\boldsymbol{x} & \int_K \lambda_2^2 \,\mathrm{d}\boldsymbol{x} & \int_K \lambda_2\lambda_3 \,\mathrm{d}\boldsymbol{x} \\ \int_K \lambda_3\lambda_1 \,\mathrm{d}\boldsymbol{x} & \int_K \lambda_3\lambda_3 \,\mathrm{d}\boldsymbol{x} & \int_K \lambda_3^2 \,\mathrm{d}\boldsymbol{x} \end{bmatrix}\,.$$

We evaluate the right matrix using

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex[a] K with barycentric coordinate functions $\lambda_1,\ldots,\lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1,\ldots,d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \,\mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1!\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1+\alpha_2+\cdots+\alpha_{d+1}+d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1}\,.$$

[Lecture → (2.7.5.6)]

[a] A d-simplex is the convex hull of $d+1$ points in $\mathbb{R}^d$, which do not all lie in a single plane.

$$\mathbf{S}_K = \frac{|K|}{3} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \frac{|K|}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} = \frac{|K|}{12} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{bmatrix}\,. \tag{12.5.10}$$

C++ code 12.5.11: `Eval()` method of ENTITY_MATRIX_PROVIDER class for the computation of the element matrix for the monolithic stabilized P1 FEM for Stokes BVP

```
P1StabFEMElementMatrixProvider::ElemMat P1StabFEMElementMatrixProvider::Eval(
    const lf::mesh::Entity& cell) {
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type " << cell.RefEl());
  LF_VERIFY_MSG(cell.Geometry()->isAffine(),
                "Triangle must have straight edges");
  // Area of the triangle
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions and store
     them in the
  // columns of the 2×3-matrix G
  // clang-format off
  const Eigen::MatrixXd dpt = (Eigen::MatrixXd(2, 1) << 0.0, 0.0).finished();
  const Eigen::Matrix<double, 2, 3> G =
      cell.Geometry()->JacobianInverseGramian(dpt).block(0, 0, 2, 2) *
    (Eigen::Matrix<double, 2, 3>(2,3) << -1, 1, 0,
                                         -1, 0, 1).finished();
  // clang-format on
  // Compute the element matrix for −Δ and S_1^0.
  // See also [Lecture → Code 2.4.5.13].
  Eigen::Matrix<double, 3, 3> L = area * G.transpose() * G;

  // Do not forget to set all non-initialized entries to zero
```

```
24    MK_.setZero();
25    // Arrays for local index remapping
26    const std::array<Eigen::Index, 3> vx_idx{0, 3, 6};
27    const std::array<Eigen::Index, 3> vy_idx{1, 4, 7};
28    const std::array<Eigen::Index, 3> p_idx{2, 5, 8};
29    // Distribute the entries of L to the final element matrix
30    for (int i = 0; i < 3; ++i) {
31      for (int j = 0; j < 3; ++j) {
32        MK_(vx_idx[i], vx_idx[j]) = MK_(vy_idx[i], vy_idx[j]) = L(i, j);
33      }
34    }
35    // Fill entries related to the B-blocks of the element matrix
36    for (int i = 0; i < 3; ++i) {
37      for (int j = 0; j < 3; ++j) {
38        // (12.5.8) with 3-point edge midpoint quadrature rule!
39        MK_(p_idx[i], vx_idx[j]) = MK_(vx_idx[j], p_idx[i]) =
40            G(0, j) * area / 3.0;
41        MK_(p_idx[i], vy_idx[j]) = MK_(vy_idx[j], p_idx[i]) =
42            G(1, j) * area / 3.0;
43      }
44    }
45    // Fill 3×3 block due to stabilization terms
46    // Element mass matrix for S_1^0(M)
47    /* A more compact option for implementation
48    const Eigen::Matrix3d SK =
49     (area / 12.0) *
50        (Eigen::Matrix3d() << 2, -1, -1, -1, 2, -1, -1, -1, 2).finished();
51    */
52    Eigen::Matrix3d SK;
53    SK(0, 0) = (area / 12.0) * 2;
54    SK(1, 1) = (area / 12.0) * 2;
55    SK(2, 2) = (area / 12.0) * 2;
56    SK(0, 1) = SK(1, 0) = -(area / 12.0);
57    SK(0, 2) = SK(2, 0) = -(area / 12.0);
58    SK(2, 1) = SK(1, 2) = -(area / 12.0);
59    for (int i = 0; i < 3; ++i) {
60      for (int j = 0; j < 3; ++j) {
61        MK_(p_idx[i], p_idx[j]) = -SK(i, j);
62      }
63    }
64    return MK_;
65  }
```

SOLUTION of (12-5.d):

We have to estimate the inf expressions on the right-hand side of (12.5.12). To do so we can use the results reported in [Lecture → Section 3.3.2]. By [Lecture → Thm. 3.3.2.21]

$$\inf_{\mathbf{w}_h \in \left(\mathcal{S}^0_{1,0}(\mathcal{M})\right)^2} |\mathbf{v} - \mathbf{w}_h|_{H^1(\Omega)} \leq |\mathbf{v} - \mathsf{I}_1\mathbf{v}|_{H^1(\Omega)} \leq C\, h_{\mathcal{M}} |\mathbf{v}|_{H^2(\Omega)} \,.$$

The other estimate of [Lecture → Thm. 3.3.2.21] gives

$$\inf_{q_h \in \mathcal{S}^0_1(\mathcal{M})} \|p - q_h\|_{L^2(\Omega)} \leq \|p - \mathsf{I}_1 p\|_{L^2(\Omega)} \leq C\, h^2_{\mathcal{M}} |p|_{H^2(\Omega)} \,.$$

Unfortunately, we cannot benefit from the higher rate of algebraic convergence in $h_{\mathcal{M}}$ of the second bound, because the first bound will dominate the sum of the best approximation errors. Thus theory predicts the same first-order asymptotic algebraic convergence for both error norms:

$$|\mathbf{v} - \mathbf{v}_h|_{H^1(\Omega)} = O(h_{\mathcal{M}}) \quad \text{and} \quad \|p - p_h\|_{L^2(\Omega)} = O(h_{\mathcal{M}}) \quad \text{for} \quad h_{\mathcal{M}} \to 0 \,.$$

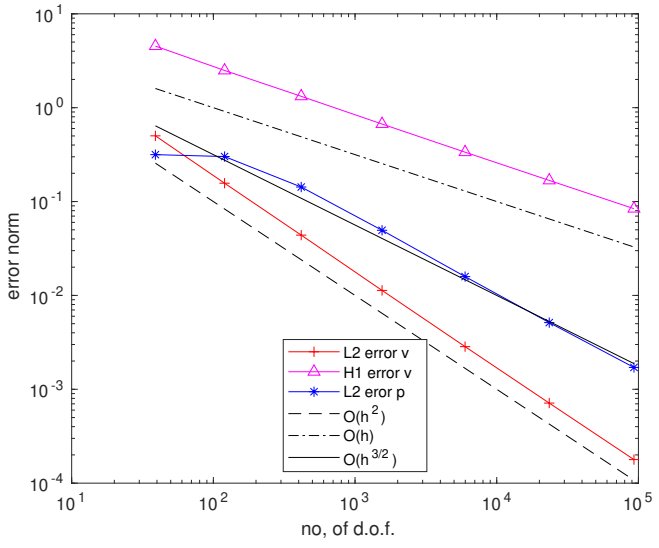
SOLUTION of (12-5.e):

Since we look at data for a sequence of regularly refined meshes, the meshwidth $h_{\mathcal{M}}$ is halved when proceeding to the next row. The following table gives the *ratios* of the error norms on two consecutive meshes.

L2err(v1)	L2err(v2)	H1err(v1)	H1err(v2)	L2err(p)
3.5294	2.8506	1.9144	1.7153	1.0476
3.5893	3.5573	1.8907	1.8788	2.1123
3.8755	3.8873	1.9764	1.9713	2.8894
3.9692	3.9741	1.9950	1.9945	3.1161
3.9939	3.9944	1.9991	1.9994	3.0895
3.9992	3.9989	2.0000	2.0002	3.0053

We clearly see that $\|\mathbf{v}-\mathbf{v}_h\|_{L^2(\Omega)}$ shrinks by a factor of 4 when $h_{\mathcal{M}} \to h_{\mathcal{M}}/2$, while $|\mathbf{v}-\mathbf{v}_h|_{H^1(\Omega)}$ decreases by a factor of 2. The picture is blurred for $\|p-p_h\|_{L^2(\Omega)}$, but we may conjecture an asymptotic decrease by a factor ≈ 3 in each step of refinement. From these observations we conclude that all errors **convergence algebraically** for $h_{\mathcal{M}} \to 0$,

- $\|\mathbf{v}-\mathbf{v}_h\|_{L^2(\Omega)}$ with rate 2 : $\|p-p_h\|_{L^2(\Omega)} = O(h_{\mathcal{M}}^2)$ for $h_{\mathcal{M}} \to 0$,
- $|\mathbf{v}-\mathbf{v}_h|_{H^1(\Omega)}$ with rate 1 : $|\mathbf{v}-\mathbf{v}_h|_{H^1(\Omega)} = O(h_{\mathcal{M}})$ for $h_{\mathcal{M}} \to 0$,
- $\|p-p_h\|_{L^2(\Omega)}$ with rate $\approx \log 3/\log 2 \approx 1.58$: $\|p-p_h\|_{L^2(\Omega)} = O(h^{1.58})$ for $h_{\mathcal{M}} \to 0$.



◁ Plots of different error norms versus the total number of degrees of freedom.

The error points in the *doubly logarithmic* plot approximately lie on straight lines, which is evidence of algebraic convergence as we know from [Lecture → § 3.2.2.5]. The slopes of these lines tell the rates of algebraic convergence.

SOLUTION of (13-1.a):

We have to specialize the formula

$$\mathbf{D}_\ell = \left[o_r(f_j, F_i)\right]_{\substack{i=1,\dots,N_{\ell+1}\\ j=1,\dots,N_\ell}} \in \{-1,0,1\}^{N_{\ell+1},N_\ell} \qquad \begin{array}{l}\mathcal{F}_\ell(\mathcal{M}) = \{f_1,\dots,f_{N_\ell}\}\,,\\ \mathcal{F}_{\ell+1}(\mathcal{M}) = \{F_1,\dots,F_{N_{\ell+1}}\}\,.\end{array}$$

([Lecture → (13.2.1.18)])

This gives

$$\begin{aligned}(\mathbf{D}_0)_{i,j} &:= o_r(\boldsymbol{p}_j, E_i)\,, \quad i \in \{1,\dots,N_1\}\,, \quad j \in \{1,\dots,N_0\}\,,\\ (\mathbf{D}_1)_{i,j} &:= o_r(E_j, K_i)\,, \quad i \in \{1,\dots,N_2\}\,, \quad j \in \{1,\dots,N_1\}\,.\end{aligned}$$

HINT 1 for (13-1.j): Use the fundamental result about the kernels and image/column spaces of a matrix and its transpose:

$$\mathcal{N}(\mathbf{M}^\top) = \mathcal{R}(\mathbf{M})^\perp \quad \text{and} \quad \mathcal{R}(\mathbf{M}^\top) = \mathcal{N}(\mathbf{M})^\perp \quad \forall \mathbf{M} \in \mathbb{R}^{m,n}\,, \quad \forall m,n \in \mathbb{N}\,. \tag{13.1.12}$$

⌟

HINT 2 for (13-1.j): We know that the column space of $\mathbf{D}_0$ is contained in the kernel of $\mathbf{D}_1$. If they are not the same, then $\mathcal{R}(\mathbf{D}_0)$ is a genuine subspace of $\mathcal{N}(\mathbf{d}_1)$ and as such, it has a non-trivial orthogonal complement inside $\mathcal{N}(\mathbf{D}_1)$:

$$\mathcal{R}(\mathbf{D}_0) \neq \mathcal{N}(\mathbf{D}_1) \quad \Leftrightarrow \quad \exists \vec{\zeta} \in \mathbb{R}^{N_1} \setminus \{\mathbf{0}\}: \quad \mathbf{D}_1 \vec{\zeta} = \mathbf{0}\,, \quad \vec{\zeta} \perp \mathcal{R}(\mathbf{D}_0)\,.$$

⌟

SOLUTION of (13-1.j):

From [Lecture → Thm. 13.2.1.21] we already know $\mathcal{R}(\mathbf{D}_0) \subset \mathcal{N}(\mathbf{D}_1)$. We verify

$$\mathcal{R}(\mathbf{D}_0) \neq \mathcal{N}(\mathbf{D}_1) \quad \Leftrightarrow \quad \mathcal{N}(\mathbf{L}) \neq \{\mathbf{0}\} \,.$$

Starting with the argument explained in the second hint,

$$\begin{aligned}
\mathcal{R}(\mathbf{D}_0) \neq \mathcal{N}(\mathbf{D}_1) \quad &\Leftrightarrow \quad \exists \vec{\zeta} \in \mathbb{R}^{N_1} \setminus \{\mathbf{0}\}: \quad \begin{array}{l} \mathbf{D}_1 \vec{\zeta} = \mathbf{0} \,, \\ \vec{\zeta} \in \mathcal{R}(\mathbf{D}_0)^{\perp} \overset{(13.1.12)}{=} \mathcal{N}(\mathbf{D}_0^{\top}) \end{array} \\
&\Leftrightarrow \quad \exists \vec{\zeta} \in \mathbb{R}^{N_1} \setminus \{\mathbf{0}\}: \quad \begin{array}{l} \mathbf{D}_1 \vec{\zeta} = \mathbf{0} \,, \\ \mathbf{D}_0^{\top} \vec{\zeta} = \mathbf{0} \end{array} \\
&\Leftrightarrow \quad \exists \vec{\zeta} \in \mathbb{R}^{N_1} \setminus \{\mathbf{0}\}: \quad \left(\mathbf{D}_1^{\top} \mathbf{D}_1 + \mathbf{D}_0 \mathbf{D}_0^{\top}\right) \vec{\zeta} = \mathbf{0} \,.
\end{aligned}$$

The last equivalence is due to the reasoning

$$\left(\mathbf{D}_1^{\top} \mathbf{D}_1 + \mathbf{D}_0 \mathbf{D}_0^{\top}\right) \vec{\zeta} = \mathbf{0} \;\Rightarrow\; \vec{\zeta}^{\top} \left(\mathbf{D}_1^{\top} \mathbf{D}_1 + \mathbf{D}_0 \mathbf{D}_0^{\top}\right) \vec{\zeta} = 0 \;\Leftrightarrow\; \left\| \mathbf{D}_1 \vec{\zeta} \right\|^2 + \left\| \mathbf{D}_0^{\top} \vec{\zeta} \right\|^2 = 0 \,.$$

HINT 1 for (13-1.k): It is advisable to rely on the initialization of the sparse matrix through the COO/triplet format, see .

SOLUTION of (13-1.k):

We consider the matrices $\mathbf{L}_1 := \mathbf{D}_1^\top \mathbf{D}_1$ and $\mathbf{L}_0 := \mathbf{D}_0 \mathbf{D}_0^\top$ separately.

❶ By definition we have

$$(\mathbf{L}_1)_{e,e'} = \sum_{K \in \mathcal{F}_2(\mathcal{M})} o_r(e,K) o_r(e',K) \,, \quad e,e' \in \mathcal{F}_1(\mathcal{M}) \,.$$

The sum can be restricted to cell adjacent to both edges e and e', which is a single cell. This suggests a *cell-oriented* algorithm.

- Run over all cells of the mesh, and
- for each pair of edges multiply their relative orientations and initialize the corresponding triplet (e,e',v) with the result $v \in \{-1,+1\}$.

❷ Again by definition

$$(\mathbf{L}_0)_{e,e'} = \sum_{p \in \mathcal{F}_0(\mathcal{M})} o_r(p,e) o_r(p,e') \,, \quad e,e' \in \mathcal{F}_1(\mathcal{M}) \,.$$

So we can compute the triplets contributing to $\mathbf{L}_0$ in an *node-oriented* way:

- Run over all nodes of the mesh, and
- for each pair (e,e') of adjacent edges multiply the node's relative orientations, store the result in $v \in \{-1,+1\}$, and add the triplet (e,e',v).

For step ❶ an auxiliary data structure has to be set up, which stores a list of adjacent edges for every node of the mesh.

Both algorithms run with optimal asymptotic complexity $O(N_1)$ for sequences of uniformly shape regular meshes with $N_1 \to \infty$.

code 13.1.13: Computation of Hodge-Laplacian matrix $\mathbf{L}$ ↪ GitLab

```
2  Eigen::SparseMatrix<int> computeHodgeLaplaceMatrix(const lf::mesh::Mesh &mesh) {
3    // Size of Hodge Laplacian matrix for discrete 1-forms is equal to the
       number
4    // of edges of the mesh
5    const size_t N = mesh.NumEntities(1);
6    // Store cell-edge incidence matrix here
7    Eigen::SparseMatrix<int, Eigen::RowMajor> L(N, N);
8    // Triplet vector to be used for the initialization of the sparse
       matrix
9    std::vector<Eigen::Triplet<int>> triplets;
10   // Pass I: Cell-oriented assembly of D_1^T D_1.
11   // Visit all cells and compute contributions of pairs of faces
12   for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
13     // Array of pointers to the edges of the cell, see [Lecture →
         Ex. 2.7.2.12]
14     std::span<const lf::mesh::Entity *const> edges{cell->SubEntities(1)};
15     const unsigned int n_ed = edges.size();
16     LF_ASSERT_MSG(n_ed <= 4, "Illegal number of edges");
17     // Inquire about relative orientations, see [Lecture →
         Rem. 2.7.2.15]
```

```
    std::span<const lf::mesh::Orientation> ror{cell->RelativeOrientations()};
    LF_ASSERT_MSG(ror.size() == n_ed, "Size mismatch for orientation array");
    // Obtain global indices of edges
    std::array<double, 4> edge_idx;
    for (unsigned int j = 0; j < n_ed; ++j) {
      edge_idx[j] = mesh.Index(*edges[j]);
    }
    // Initialize triplets corresponding to pairs of edges
    for (unsigned int j = 0; j < n_ed; ++j) {
      for (unsigned int i = 0; i < n_ed; ++i) {
        triplets.emplace_back(edge_idx[j], edge_idx[i],
                              lf::mesh::to_sign(ror[j]) * to_sign(ror[i]));
      }
    }
  }
  // Pass II: Node-oriented assembly of D_0 D_0^T
  // Preparatory step:  build lists of edges adjacent to nodes
  std::vector<std::vector<std::pair<size_t, int>>> eds_node(
      mesh.NumEntities(2));
  // Run through the edges and collect their endpoints
  for (const lf::mesh::Entity *edge : mesh.Entities(1)) {
    // Get index of this edge
    lf::mesh::Mesh::size_type edgeIdx = mesh.Index(*edge);
    // Get the nodes and their indices.
    // Note, that seen from the edges the nodes have codim 1, not 2,
    // hence we call SubEntities(1).  This is a relative codimension!
    auto nodes = edge->SubEntities(1);
    lf::mesh::Mesh::size_type firstNodeIdx = mesh.Index(*nodes[0]);
    lf::mesh::Mesh::size_type lastNodeIdx = mesh.Index(*nodes[1]);
    LF_ASSERT_MSG(firstNodeIdx < mesh.NumEntities(2),
                  "First node index out of range");
    LF_ASSERT_MSG(lastNodeIdx < mesh.NumEntities(2),
                  "Last node index out of range");
    eds_node[firstNodeIdx].emplace_back(edgeIdx, -1);
    eds_node[lastNodeIdx].emplace_back(edgeIdx, 1);
  }
  // Run through nodes and pairs ofv adjacent edges
  for (const std::vector<std::pair<size_t, int>> &edgesinfo : eds_node) {
    for (const std::pair<size_t, int> &idxori1 : edgesinfo) {
      for (const std::pair<size_t, int> &idxori2 : edgesinfo) {
        triplets.emplace_back(idxori1.first, idxori2.first,
                              idxori1.second * idxori2.second);
      }
    }
  }
  L.setFromTriplets(triplets.begin(), triplets.end());
  return L;
}
```

Remark. Of course, the straightforward way to compute $\mathbf{L}$ is to obtain $\mathbf{D}_0$ and $\mathbf{D}_1$ in sparse-matrix format and the simply use eigen's sparse matrix×matrix multiplication ➙ documentation. Presumably, the asymptotic complexity of this is N_1^2, but authoritative information about this is not readily available.

SOLUTION of (13-1.b):

Remember from [Lecture → Section 13.2.1.2] that the incidence matrix $\mathbf{D}_\ell$ represents the exterior derivative $\vec{\mathrm{d}} : \mathcal{C}^\ell(\mathcal{M}) \to \mathcal{C}^{\ell+1}(\mathcal{M})$ of cochains after the latter have been identified with $\mathbb{R}^{N_\ell}$ and $\mathbb{R}^{N_{\ell+1}}$, respectively.

There is a one-to-one correspondence of integral ℓ-forms in the Whitney finite element space and $\mathbb{R}^{N_\ell}$ mediated through the basis expansion with respect to the basis $\left\{\beta_f^\ell\right\}_{f\in\mathcal{F}_\ell(\mathcal{M})}$ of Whitney ℓ-forms: The mappings

$$\mathsf{I}_\ell : \mathbb{R}^{N_\ell} \to \mathcal{W}^\ell(\mathcal{M}) \,, \quad \mathsf{I}_\ell(\vec{\boldsymbol{\mu}}) = \sum_{i=1}^{N_\ell} (\vec{\boldsymbol{\mu}})_i \, \beta_{f_i}^\ell \,, \quad \vec{\boldsymbol{\mu}} \in \mathbb{R}^{N_\ell}$$

are isomorphisms, taking for granted a numbering of the ℓ-facets, $\mathcal{F}_\ell(\mathcal{M}) = \{f_1, \ldots, f_{N_\ell}\}$.

By the close connections between the various exterior derivatives expressed in [Lecture → Thm. 13.2.2.48] and [Lecture → Lemma 13.2.2.22], we have the commuting diagram

$$\underline{\mathrm{d}}_\ell \circ \mathsf{I}_\ell = \mathsf{I}_{\ell+1} \circ \mathbf{D}_\ell \quad \text{on} \quad \mathbb{R}^{N_\ell} \,. \tag{13.1.2}$$

Now the exterior derivative $\underline{\mathrm{d}}_\ell$ acting on elements of $\mathcal{W}^\ell(\mathcal{M})$ boils down to the $\mathcal{M}$-piecewise application of a standard exterior derivative d_ℓ of differential forms.

At this point we remember from [Lecture → § 13.1.4.49] that in the vector proxy model of differential forms on 2D Euclidean space the exterior derivative is realized through a classical partial differential operator from vector analysis. Concretely,

- d_0 corresponds to a **gradient** (Option I) or a planar rotation (Option II), see [Lecture → (13.1.4.51)],
- d_1 gives rise to a scalar rotation (Option I, [Lecture → (13.1.4.53)]) or a **divergence** (Option II, [Lecture → (13.1.4.54)]).

In fact the naming of $\mathbf{D}_0$ and $\mathbf{D}_1$ as mentioned in the problem statement is inconsistent, because it is based on different conventions for choosing vector proxies for 1-forms in 2D Euclidean space.

HINT 1 for (13-1.c): Take into account that we are dealing with a *hybrid* 2D mesh. ⌟

SOLUTION of (13-1.c):

The rows of $\mathbf{D}_0$ are in one-on-one relationship with the edges of the mesh, whereas the columns are related to the nodes of the mesh. A row for a particular edge has a non-zero entry exactly for those nodes that are endpoints of the edge. Hence $\mathbf{D}_0$ has exactly 2 non-zero entries per row.

The rows of $\mathbf{D}_1$ are associated with cells and have non-zero entries in exactly those columns, which belong to the edges of the cell. Therefore the rows of $\mathbf{D}_1$ have either 3 (for a TRIA) or 4 (for a QUAD) non-zero entries. -

SOLUTION of (13-1.d):

Recall the explanations for Sub-problem (13-1.c). They tell us:

- The number of non-zero entries in a column of $\mathbf{D}_0$ tells the number of edges adjacent to the node to which that column is associated.
- The number of non-zero entries in a column of $\mathbf{D}_1$ is the number of cells adjacent to the associated edge.

SOLUTION of (13-1.e):

For the mesh from Fig. 234 we have $N_0 = 2$, $N_1 = 6$, and $N_2 = 5$. The black arrows indicates which is the first and second endpoint of an edge. This gives the node-edge incidence matrix:

$$\mathbf{D}_0 = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

To determine the cell-edge incidence matrix, look up the induced orientations in Fig. 232 and Fig. 233 and mark them in Fig. 234. Thus you can read off

$$\mathbf{D}_1 = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & 0 & 1 \end{bmatrix}.$$

SOLUTION of (13-1.f):

Code 13.1.9 shows the straightforward implementation based on an outer loop traversing all edges. Refer to the copious comments in the code for further explanations.

C++ code 13.1.4: Computation of node-edge incidence matrix $\mathbf{D}_0$ ↪ GitLab

```
2  Eigen::SparseMatrix<int> computeEdgeVertexIncidenceMatrix(
3      const lf::mesh::Mesh &mesh) {
4    // Store edge-vertex incidence matrix here
5    Eigen::SparseMatrix<int, Eigen::RowMajor> G;
6
7    // Mesh::NumEntities(unsigned codim) returns the number of elements
8    // with given codimension.  Codim(Edge) = 1, Codim(Node) = 2.
9    const lf::mesh::Mesh::size_type numEdges = mesh.NumEntities(1),
10                                    numNodes = mesh.NumEntities(2);
11   // Following the demo for the reserve()-initialising the sparse matrix
         given
12   // in the exercise sheet.  From (2.1a) we know that G has exactly 2
         entries
13   // per row.
14   G = Eigen::SparseMatrix<int, Eigen::RowMajor>(numEdges, numNodes);
15   G.reserve(Eigen::VectorXi::Constant(numEdges, 2));
16
17   // To compute G efficiently we iterate over all edges and check the
         index
18   // of the nodes at its end.  This is the efficient way to do the
         assembly,
19   // introduced as "distribute scheme" in class.  We cannot iterative
         over
20   // vertices, because LehrFEM++ does not allow to visit the edges
21   // adjacent to a vertex
22   for (const lf::mesh::Entity *edge : mesh.Entities(1)) {
23     // Get index of this edge
24     lf::mesh::Mesh::size_type edgeIdx = mesh.Index(*edge);
25     // Get the nodes and their indices.
26     // Note, that seen from the edges the nodes have codim 1, not 2,
27     // hence we call SubEntities(1).  This is a relative codimension!
28     auto nodes = edge->SubEntities(1);
29     lf::mesh::Mesh::size_type firstNodeIdx = mesh.Index(*nodes[0]);
30     lf::mesh::Mesh::size_type lastNodeIdx = mesh.Index(*nodes[1]);
31     // Add the matrix entries according to the definition
32     G.coeffRef(edgeIdx, firstNodeIdx) += 1;
33     G.coeffRef(edgeIdx, lastNodeIdx) -= 1;
34   }
35
36   return G;
37 }
```

HINT 1 for (13-1.g): If you find it too cumbersome to test for the induced orientation of edges by just relying on calls to `SubEntities()`, LEHRFEM++ mak make your life easier with the member function

```
lf::nonstd::span<const Orientation>
   lf::mesh::Entity::RelativeOrientations() const;
```

which, for a cell, returns an array of **lf::mesh::Orientation** "mismatch flags" between the intrinsic and induced orientations of its edges. The function `lf::mesh::to_sign()` converts an orientation flag to an integer $\in \{-1, 1\}$, ⌟

SOLUTION of (13-1.g):

Code 13.1.5 runs over all cells and then sets the entries of $\mathbf{D}_1$ according to what is returned by `relativeO` for all the edges of a cell.

C++ code 13.1.5: Computation of cell-edge incidence matrix $\mathbf{D}_1$ ↪ GitLab

```
Eigen::SparseMatrix<int> computeCellEdgeIncidenceMatrix(
    const lf::mesh::Mesh &mesh) {
  // Store cell-edge incidence matrix here
  Eigen::SparseMatrix<int, Eigen::RowMajor> D;

  // Mesh::NumEntities(unsigned codim) returns the number of elements
  // with given codimension.  Codim(Edge) = 0, Codim(Node) = 1.
  const lf::mesh::Mesh::size_type numCells = mesh.NumEntities(0),
                                  numEdges = mesh.NumEntities(1);
  // Following the demo for the reserve()-initialising the sparse matrix
     given
  // in the exercise sheet.  From (2.1a) we know that D has at most 4
     entries
  // per row.
  D = Eigen::SparseMatrix<int, Eigen::RowMajor>(numCells, numEdges);
  D.reserve(Eigen::VectorXi::Constant(numCells, 4));

  // To compute D efficiently we iterate over all cells and check the
  // orientations (+1 or -1, same as in the definition of the matrix D)
  // of its edges.  For this we may use RelativeOrientations().
  for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
    // Get cell index
    lf::mesh::Mesh::size_type cellIdx = mesh.Index(*cell);
    // Get edges and their orientations (these already the entries for
       D!)
    auto edges = cell->SubEntities(1);
    auto edgeOrientations = cell->RelativeOrientations();

    // Iterate over both and add to D
    auto edgeIt = edges.begin();
    auto orntIt = edgeOrientations.begin();
    for (; edgeIt != edges.end() && orntIt != edgeOrientations.end();
         ++edgeIt, ++orntIt) {
      // Get the edge index and add its orientation to D
      lf::mesh::Mesh::size_type edgeIdx = mesh.Index(**edgeIt);
      D.coeffRef(cellIdx, edgeIdx) += lf::mesh::to_sign(*orntIt);
    }
  }

  return D;
}
```

HINT 1 for (13-1.h): For the proof you need to consider only a single triangular/quadrilateral cell. Then introduce the numbers d_ℓ, where $d_\ell = 1$, if the intrinsic orientation of the ℓ-th edge matches its induced orientation, $d_\ell = -1$ otherwise. ⌟

SOLUTION of (13-1.h):

We repeat the arguments of the proof of [Lecture → Thm. 13.2.1.21] in two dimensions. We examine a single entry of $\mathbf{D}_1\mathbf{D}_0$:

$$(\mathbf{D}_1\mathbf{D}_0)_{j,i} = (\mathbf{D}_1)_{j,:} \cdot (\mathbf{D}_0)_{:,i} \,, \quad j \in \{1,\ldots,N_0\} \,, \quad i \in \{1,\ldots,N_2\} \,. \tag{13.1.8}$$

If the node $\boldsymbol{x}^i \in \mathcal{F}_0(\mathcal{M})$ is not a vertex of the cell K_j, then non-zero entries will not meet in the dot product in (13.1.8). In this case $(\mathbf{D}_1\mathbf{D}_0)_{j,i} = 0$.

Otherwise, let $e_k, e_\ell \in \mathcal{F}_1(\mathcal{M})$ be the two edges which are

- both adjacent to $\boldsymbol{x}^i$,
- and both belong to ∂K_j.

Let $d_k, d_\ell \in \{-1,1\}$ denote the induced orientations of e_k and e_ℓ with respect to K. Without loss of generality assume that $\boldsymbol{x}^i$ is the first vertex of K_j and that $d_k = d_\ell = 1$, that is the edges are oriented as in Fig. 232 and Fig. 233. Then one points awway from $\boldsymbol{x}^i$ and the other towwards $\boldsymbol{x}^i$, which means that $(\mathbf{D}_0)_{\ell,i}$ and $(\mathbf{D}_0)_{k,i}$ have opposite signs. Hence,

$$\begin{aligned}(\mathbf{D}_1\mathbf{D}_0)_{j,i} &= (\mathbf{D}_1)_{j,k}(\mathbf{D}_0)_{k,i} + (\mathbf{D}_1)_{j,\ell}(\mathbf{D}_0)_{\ell,i} = \underbrace{d_k}_{=1}(\mathbf{D}_0)_{k,i} + \underbrace{d_\ell}_{=1}(\mathbf{D}_0)_{\ell,i} \\ &= (\mathbf{D}_0)_{k,i} + (\mathbf{D}_0)_{\ell,i} = 0 \,.\end{aligned}$$

SOLUTION of (13-1.i):

code 13.1.9: Computation of node-edge incidence matrix $\mathbf{D}_0$ ↪ GitLab

```
bool testZeroIncidenceMatrixProduct(const lf::mesh::Mesh &mesh) {
  bool isZero = false;

  Eigen::SparseMatrix<int> G = computeEdgeVertexIncidenceMatrix(mesh),
                           D = computeCellEdgeIncidenceMatrix(mesh);

  Eigen::SparseMatrix<int> O = D * G;
  // Possibility 1:
  // Not prone to roundoff errors, since O is an integer matrix!
  isZero = O.norm() == 0;
  // Possibility 2:  But this doesn't use the fact that O is sparse.
  // isZero = Eigen::MatrixXi(O).isZero(0);
  return isZero;
}
```

Note that roundoff errors do not have to be taken into account, because we rely on matrices with integer entries.

SOLUTION of (13-2.a):

Just cite

Definition [Lecture → Def. 13.1.3.16]. Differential form (DF)

A (continuous) **differential ℓ-form** on an n-dimensional C^1-submanifold Σ of $\mathbb{R}^d$, $\ell \in \{0,\ldots,n\}$ is a (continuous) mapping $\omega : \Sigma \to \Lambda^\ell(\mathcal{T}(\Sigma))$, that is, ω assigns to every $\boldsymbol{x} \in \Sigma$ a unique alternating ℓ-multilinear form on the tangent space $\mathcal{T}_{\boldsymbol{x}}(\Sigma)$: $\omega(\boldsymbol{x}) \in \Lambda^\ell(\mathcal{T}_{\boldsymbol{x}}(\Sigma))$.

and specialize this definition for $\Sigma := \Omega \subset \mathbb{R}^2$, for which $\mathcal{T}_{\boldsymbol{x}}(\Omega) = \mathbb{R}^2$ for all $\boldsymbol{x} \in \Omega$, *c.f.* [Lecture → § 13.1.3.17].

Definition 13.2.4. Continuous differential forms on $\Omega \subset \mathbb{R}^2$

A continuous **differential ℓ-form** $\omega \in C^0\Lambda^\ell(\Omega)$ on a domain $\Omega \subset \mathbb{R}^2$ is a continuous mapping

$$\omega : \Omega \to \Lambda^\ell(\mathbb{R}^2) ,$$

where $\Lambda^\ell(\mathbb{R}^2)$ is the vector space of alternating ℓ-linear forms on $\mathbb{R}^2$.

Remember from [Lecture → Thm. 13.1.3.6] that $\dim \Lambda^\ell(\mathbb{R}^2) = \binom{2}{\ell}$. This will tell you the "number of vector components" for vector proxies.

SOLUTION of (13-2.j):

Matching the building blocks of (13.2.3) and (13.2.19) we find

- $U = H\Lambda^1(\mathsf{d},\Omega)$, $Q = H\Lambda^0(\mathsf{d},\Omega)$
- a $\leftrightarrow$ $(\omega,\omega') \mapsto \left(\mathsf{d}_1\omega, \mathsf{d}_1\omega'\right)_{L^2\Lambda^2(\Omega)}$
- b $\leftrightarrow$ $(\omega,\pi') \mapsto \left(\omega, \mathsf{d}_0\pi'\right)_{L^2\Lambda^1(\Omega)}$
- c $\leftrightarrow$ $(\pi,\pi') \mapsto \left(\pi,\pi'\right)_{L^2\Lambda^0(\Omega)}$
- ℓ $\leftrightarrow$ $\omega' \mapsto \left(\varphi,\omega'\right)_{L^2\Lambda^1(\Omega)}$
- g $\leftrightarrow$ $\pi' \mapsto 0$

All these (bi-)linear forms are obviously continuous/bounded [Lecture $\rightarrow$ Def. 1.2.3.42].

In terms of 2D Euclidean vector proxies the identifications are:

- $U = \boldsymbol{H}(\mathrm{curl}_{2D},\Omega)$, $Q = H^1(\Omega)$
- a $\leftrightarrow$ $(\mathbf{u},\mathbf{u}') \mapsto \int_\Omega \mathbf{curl}_{2D}\,\mathbf{u}(\boldsymbol{x})\ \mathbf{curl}_{2D}\,\mathbf{u}'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
- b $\leftrightarrow$ $(\mathbf{u},p') \mapsto \int_\Omega \mathbf{u}(\boldsymbol{x})\cdot\mathbf{grad}\,p'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
- c $\leftrightarrow$ $(p,p') \mapsto \int_\Omega p(\boldsymbol{x})\,p'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
- ℓ $\leftrightarrow$ $\mathbf{u}' \mapsto \int_\Omega \mathbf{f}(\boldsymbol{x})\cdot\mathbf{u}'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$
- g $\leftrightarrow$ $p' \mapsto 0$

HINT 1 for (13-2.k): Exploit that

$$\mathsf{d}_0 H\Lambda^0(\mathsf{d},\Omega) \subset H\Lambda^1(\mathsf{d},\Omega)\ , \tag{13.2.25}$$

cf. [Lecture → (13.1.5.3)].

SOLUTION of (13-2.k):

In the concrete setting of (13.2.3) we have

$$\mathsf{b}(\omega,\pi') := \big(\omega, \mathsf{d}_0\pi'\big)_{L^2\Lambda^1(\Omega)}\,, \quad \omega \in H\Lambda^1(\mathsf{d},\Omega),\ \pi' \in H\Lambda^0(\mathsf{d},\Omega)\,,$$

which means

$$\mathcal{N}^*(\mathsf{b}) = \Big\{\pi \in H\Lambda^0(\mathsf{d},\Omega):\ \big(\omega', \mathsf{d}_0\pi\big) = 0\ \forall \omega' \in H\Lambda^1(\mathsf{d},\Omega)\Big\} = \Big\{\pi \in H\Lambda^0(\mathsf{d},\Omega):\ \mathsf{d}_0\pi = 0\Big\}\,.$$

Since Ω is connected 0-forms with vanishing exterior derivative must be constant. Hence,

$$\mathcal{N}^*(\mathsf{b}) = \mathcal{P}_0\Lambda^0(\Omega) = \{\boldsymbol{x} \mapsto c,\ c \in \mathbb{R}\}\,. \tag{13.2.26}$$

The same in terms of 2D Euclidean vector proxies: Starting from

$$\mathsf{b}(\mathbf{u},p) := \int_\Omega \mathbf{u}(\boldsymbol{x})\cdot\mathbf{grad}\,p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad \mathbf{u} \in \boldsymbol{H}(\mathrm{curl}_{2D},\Omega)\,, \quad p \in H^1(\Omega)\,.$$

and rewriting (13.2.25) as

$$\mathbf{grad}\,H^1(\Omega) \subset \boldsymbol{H}(\mathrm{curl}_{2D},\Omega)\,,$$

we conclude

$$\begin{aligned}\mathcal{N}^*(\mathsf{b}) &= \Big\{p \in H^1(\Omega):\ \int_\Omega \mathbf{u}'(\boldsymbol{x})\cdot\mathbf{grad}\,p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0\ \forall \mathbf{u}' \in \boldsymbol{H}(\mathrm{curl}_{2D},\Omega)\Big\}\\ &= \Big\{p \in H^1(\Omega):\ \mathbf{grad}\,p = 0\Big\}\,,\end{aligned}$$

because we can set $\mathbf{u}' := \mathbf{grad}\,p$. Obviously, this is the space of globally constant functions on Ω.

HINT 1 for (13-2.l): Take into account the hint for Sub-problem (13-2.k) and remember

Theorem [Lecture → Thm. 1.8.0.20]. Second Poincaré-Friedrichs inequality

If $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$, is bounded and connected, then

$$\exists C = C(\Omega) > 0: \quad \|u\|_0 \leq C \operatorname{diam}(\Omega) \|\mathbf{grad}\, u\|_0 \quad \forall u \in H^1_*(\Omega) .$$

⌟

HINT 2 for (13-2.l): The expression $\inf_{\nu\in\mathcal{N}^*(\mathrm{b})} \|\pi-\nu\|_{H\Lambda^0(\mathrm{d},\Omega)}$ is the $H\Lambda^0(\mathrm{d},\Omega)$-*distance* of π from $\mathcal{N}^*(\mathrm{b})$, which is the distance of π and its $H\Lambda^0(\mathrm{d},\Omega)$-orthogonal projection onto $\mathcal{N}^*(\mathrm{b})$.

Since 0-forms are just functions and $\mathcal{N}^*(\mathrm{b}) = \mathcal{P}_0\Lambda^0(\Omega)$ (space of constant 0-forms on Ω), we thus find

$$\inf_{\nu\in\mathcal{N}^*(\mathrm{b})} \|\pi-\nu\|_{H\Lambda^0(\mathrm{d},\Omega)} = \left\| \boldsymbol{x} \mapsto \pi(\boldsymbol{x}) - \frac{1}{|\Omega|}\int_\Omega \pi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \right\|_{H\Lambda^0(\mathrm{d},\Omega)}.$$

Then note that the function in the norm has vanishing mean. ⌟

SOLUTION of (13-2.l):

We have to show

$$\exists \beta > 0: \quad \sup_{\omega' \in H\Lambda^1(\mathsf{d},\Omega)} \frac{(\omega', \mathsf{d}_0\pi)_{L^2\Lambda^1(\Omega)}}{\|\omega'\|_{H\Lambda^1(\mathsf{d},\Omega)}} \geq \beta \inf_{\nu \in \mathcal{P}_0\Lambda^0(\Omega)} \|\pi - \nu\|_{H\Lambda^0(\mathsf{d},\Omega)} \quad \forall \pi \in H\Lambda^0(\mathsf{d},\Omega) \,.$$

❶ By the reasoning put forth in the hint we have

$$\inf_{\nu \in \mathcal{P}_0\Lambda^0(\Omega)} \|\pi - \nu\|_{H\Lambda^0(\mathsf{d},\Omega)} = \|\pi^*\|_{H\Lambda\,(\mathsf{d},\Omega)}, \quad \pi^*(\boldsymbol{x}) := \pi(\boldsymbol{x}) - \frac{1}{|\Omega|}\int_\Omega \pi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \,.$$

By construction π^* has vanishing average, $\pi^* \in H^1_*(\Omega)$, which permits us to invoke [Lecture → Thm. 1.8.0 and conclude

$$\|\pi^*\|_{H\Lambda^0(\mathsf{d},\Omega)} \leq C_1 \|\mathsf{d}_0\pi^*\|_{L^2\Lambda^1(\Omega)} \tag{13.2.27}$$

with some $C_1 > 0$ depending only on Ω.

❷ We exploit (13.2.25) and replace ω' in the supremum with $\mathsf{d}_0\pi'$:

$$\begin{aligned}\sup_{\omega' \in H\Lambda^1(\mathsf{d},\Omega)} \frac{(\omega', \mathsf{d}_0\pi)_{L^2\Lambda^1(\Omega)}}{\|\omega'\|_{H\Lambda^1(\mathsf{d},\Omega)}} &\geq \sup_{\pi' \in H\Lambda^0(\mathsf{d},\Omega)} \frac{(\mathsf{d}_0\pi', \mathsf{d}_0\pi)_{L^2\Lambda^1(\Omega)}}{\|\mathsf{d}_0\pi'\|_{H\Lambda^1(\mathsf{d},\Omega)}} \\ &\geq \frac{(\mathsf{d}_0\pi, \mathsf{d}_0\pi)_{L^2\Lambda^1(\Omega)}}{\|\mathsf{d}_0\pi\|_{L^2\Lambda^1(\Omega)}} = \|\mathsf{d}_0\pi\|_{L^2\Lambda^1(\Omega)} \geq C_1^{-1}\|\pi^*\|_{H\Lambda^0(\mathsf{d},\Omega)} \,.\end{aligned}$$

This is (LBB2) with $\beta := C_1^{-1}$.

The same arguments can be made in terms of vector proxies: In this case we have to show

$$\exists \beta > 0: \quad \sup_{\mathbf{u}' \in \boldsymbol{H}(\mathrm{curl}_{2D},\Omega)} \frac{(\mathbf{u}', \mathbf{grad}\, p)_{L^2(\Omega)}}{\|\mathbf{u}'\|_{\boldsymbol{H}(\mathrm{curl}_{2D},\Omega)}} \geq \beta \inf_{c \in \mathbb{R}} \|p - c\|_{H^1(\Omega)} \quad \forall p \in H^1(\Omega) \,.$$

❶ From the second hint we lean

$$\inf_{c \in \mathbb{R}} \|p - c\|_{H^1(\Omega)} = \|p^*\|_{H^1(\Omega)} \,, \quad p^*(\boldsymbol{x}) := p(\boldsymbol{x}) - \frac{1}{|\Omega|}\int_\Omega p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \,.$$

By construction p^* has vanishing average, $p^* \in H^1_*(\Omega)$, which permits us to invoke [Lecture → Thm. 1.8.0. and conclude

$$\|p^*\|_{H^1(\Omega)} \leq C_1 \|\mathbf{grad}\, p^*\|_{L^2(\Omega)} \tag{13.2.28}$$

with some $C_1 > 0$ depending only on Ω.

❷ We exploit (13.2.25) and replace $\mathbf{u}'$ in the supremum with $\mathbf{grad}\, p'$:

$$\begin{aligned}\sup_{\mathbf{u}' \in \boldsymbol{H}(\mathrm{curl}_{2D},\Omega)} \frac{(\mathbf{u}', \mathbf{grad}\, p)_{L^2\Lambda^1(\Omega)}}{\|\mathbf{u}'\|_{\boldsymbol{H}(\mathrm{curl}_{2D},\Omega)}} &\geq \sup_{p' \in H^1(\Omega)} \frac{(\mathbf{grad}\, p', \mathbf{grad}\, p)_{L^2(\Omega)}}{\|\mathbf{grad}\, p'\|_{\boldsymbol{H}(\mathrm{curl}_{2D},\Omega)}} \\ &\geq \frac{(\mathbf{grad}\, p, \mathbf{grad}\, p)_{L^2(\Omega)}}{\|\mathbf{grad}\, p\|_{\boldsymbol{H}(\mathrm{curl}_{2D},\Omega)}} = \|\mathbf{grad}\, p\|_{L^2(\Omega)} \geq C_1^{-1}\|p^*\|_{H^1(\Omega)} \,.\end{aligned}$$

SOLUTION of (13-2.m):

In this concrete case we have

$$\mathsf{c}(\pi,\pi') := \left(\pi,\pi'\right)_{L^2\Lambda^0(\Omega)}, \quad \pi,\pi' \in H\Lambda^0(\mathsf{d},\Omega) .$$

The fact that this bilinear form is positive definite is an immediate consequence of the inner-product property of $(\cdot,\cdot)_{L^2\Lambda^0(\Omega)}$.

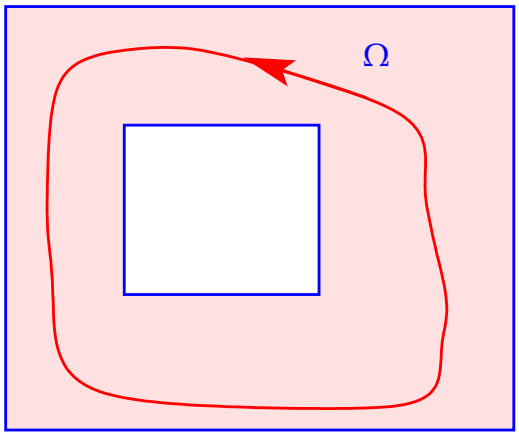
As we have seen in Sub-problem (13-2.k) the space $\mathcal{N}^*(\mathsf{b})$ is the space of constant functions, such that

$$\mathsf{c}(\pi,\pi) = (\pi,\pi)_{L^2\Lambda^0(\Omega)} = \|\pi\|^2_{L^2\Lambda^0(\Omega)} = \|\pi\|^2_{H\Lambda^0(\mathsf{d},\Omega)} \quad \forall \pi \in \mathcal{N}^*(\mathsf{b}) .$$

This is (13.2.23) with $\gamma = 1$.

HINT 1 for (13-2.n): The boundary of Ω need not be connected! ⌟

SOLUTION of (13-2.n):



◁ “Domain with a hole”

The closed red directed curve does not bound any subdomain $\subset \Omega$, because if you take the intersection of the interior of the curve with Ω, its boundary is not only the curve, but also the boundary of the square hole.

SOLUTION of (13-2.o):

❶ To begin with, it is clear that a is positive semi-definite: $\mathsf{a}(\omega,\omega) \geq 0$ for all $\omega \in H\Lambda^1(\mathsf{d},\Omega)$.

❷ If Ass. 13.2.29 holds true, we can appeal to (13.2.30)

$$\left\{\omega \in H\Lambda^1(\mathsf{d},\Omega) : \ \mathsf{d}_1\omega = 0\right\} = \mathsf{d}_0 H\Lambda^0(\mathsf{d},\Omega)\,, \tag{13.2.30}$$

and obtain the representation

$$\begin{aligned}\mathcal{N}(\mathsf{b}) &= \left\{\omega \in H\Lambda^1(\mathsf{d},\Omega) : \ (\omega,\mathsf{d}_0\pi)_{L^2\Lambda^1(\Omega)} = 0 \ \forall \pi \in H\Lambda^0(\mathsf{d},\Omega)\right\}\\ &= \left\{\omega \in H\Lambda^1(\mathsf{d},\Omega) : \ \mathsf{d}_1\omega = 0\right\}^{\perp}\,,\end{aligned}$$

where $\perp$ denotes the $L^2\Lambda^1(\Omega)$-orthogonal complement. Therefore,

$$\omega \in \mathcal{N}(\mathsf{b}) \quad \text{and} \quad \left\{\ \mathsf{a}(\omega,\omega) = \|\mathsf{d}_1\omega\|^2_{L^2\Lambda^1(\Omega)} = 0 \quad \Leftrightarrow \quad \mathsf{d}_1\omega = 0\ \right\} \quad \Rightarrow \quad \omega = 0\,,$$

because $\mathbf{0}$ is the only element that belongs to a space and its orthogonal complement simultaneously.

Remark. The space $\mathcal{N}(\mathsf{b})$ is $L^2\Lambda^1(\Omega)$-orthogonal to the kernel of d_1, which suggests that $\mathsf{a}(\omega,\omega') := (\mathsf{d}_1$ is *uniformly* positive definite on $\mathcal{N}(\mathsf{b})$, which would amount to (LBB1). However, this conclusion is premature and the fact that

$$\exists \alpha > 0\colon \quad (\mathsf{d}_1\omega,\mathsf{d}_1\omega)_{L^2\Lambda^2(\Omega)} \geq \alpha\|\omega\|^2_{H\Lambda^1(\mathsf{d},\Omega)} \quad \forall \omega \in \mathcal{N}(\mathsf{b}) \tag{13.2.32}$$

is a rather deep mathematical result and a consequence of the compact embedding of $\mathcal{N}(\mathsf{b})$ into $L^2\Lambda^1(\Omega)$. The proof follows the reasoning in [Lecture → Suppl. 1.8.0.24].

SOLUTION of (13-2.p):

The Galerkin matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ arise from plugging pairs of global shape functions into the underlying bilinear forms.

$$\mathbf{A} = \left[\left(\mathrm{d}_1 \beta_j^1, \mathrm{d}_1 \beta_i^1 \right)_{L^2\Lambda^1(\Omega)} \right]_{i,j=1}^{N} = \left[\int_\Omega \mathrm{curl}_{2D}\, \check{\beta}_j^1(\boldsymbol{x})\ \mathrm{curl}_{2D}\, \check{\beta}_i^1(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \right]_{i,j=1}^{N},$$

$$\mathbf{B} = \left[\left(\beta_j^1, \mathrm{d}_0 \beta_i^0 \right)_{L^2\Lambda^1(\Omega)} \right]_{\substack{i=1,\dots,M \\ j=1,\dots,N}} = \left[\int_\Omega \check{\beta}_j^1(\boldsymbol{x}) \cdot \mathbf{grad}\, \check{\beta}_i^0\, \mathrm{d}\boldsymbol{x} \right]_{\substack{i=1,\dots,M \\ j=1,\dots,N}},$$

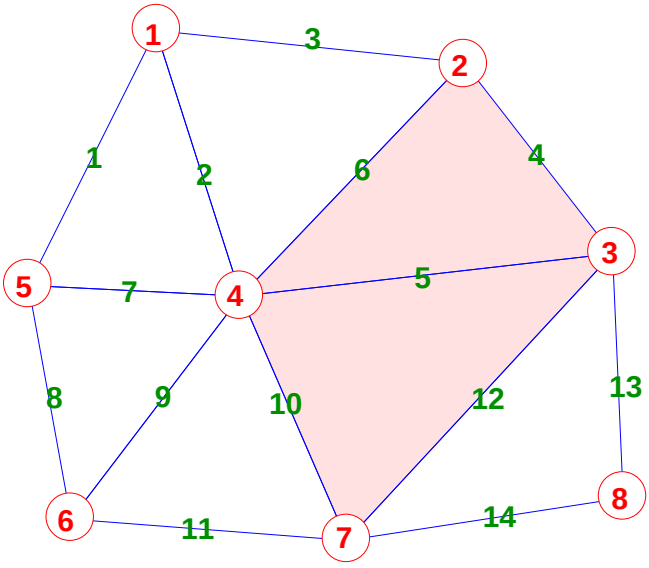
$$\mathbf{C} = \left[\left(\beta_j^0, \beta_i^0 \right)_{L^2\Lambda^0(\Omega)} \right]_{i,j=1}^{M} = \left[\int_\Omega \check{\beta}_j^0(\boldsymbol{x})\, \check{\beta}_j^0(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \right]_{i,j=1}^{M}.$$

For convenience, also the expressions in the language of 2D Euclidean vector proxies are given, $\check{\beta}_i^* := \mathrm{v.p.}$

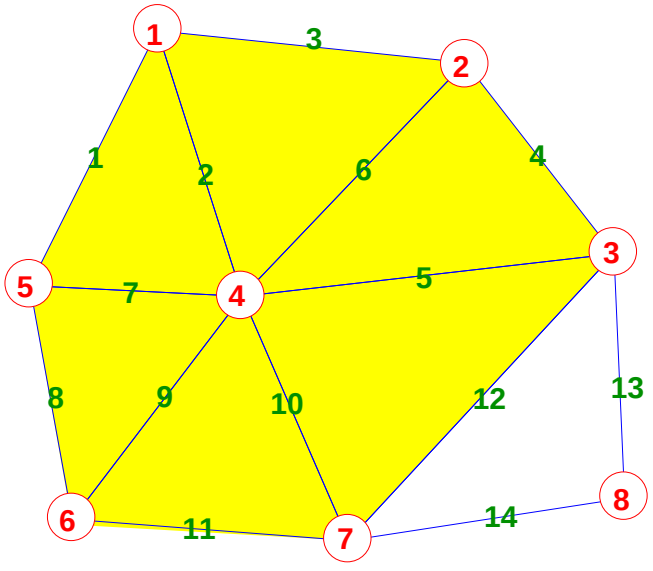
SOLUTION of (13-2.q):

Remember from [Lecture → Section 2.4.4] and [Lecture → (13.2.2.52)] that

- the support of β_e^1 is the union of (one or two) triangle adjacent to the edge e, and
- that the support of the tent function β_p^0 is the union of all triangles sharing the node p.



Support of β_e^1 associated with edge $\sharp 5$



Support of β_p^0 associated with node $\sharp 4$

Moreover, an entry of a Galerkin matrix can be non-zero, if and only if the supports of the involved global shape functions *overlap*. So for $\mathbf{A}$ non-zero entries correspond to pairs of edges belonging to the same triangle. For $\mathbf{B}$ a non-zero entry can arise, if a node and and edge belong to a common triangle. The non-zero entries of $\mathbf{C}$ are associated with pairs of endpoints of edges of the mesh.

$$\mathbf{A} = \begin{bmatrix} \star & \star & 0 & 0 & 0 & 0 & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \star & \star & \star & 0 & 0 & \star & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \star & \star & 0 & 0 & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 & \star & 0 & \star & 0 & 0 \\ 0 & \star & \star & \star & \star & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \star & \star & 0 & 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star & \star & \star & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \star & 0 & 0 & 0 & \star & \star & 0 & \star & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \star & 0 & 0 & 0 & 0 & \star & 0 & \star & \star & \star \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star \end{bmatrix},$$

$$\mathbf{B} = \begin{bmatrix} \star & \star & \star & 0 & 0 & \star & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \star & \star & \star & \star & \star & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 & \star & 0 & \star & \star & \star \\ \star & \star & \star & \star & \star & \star & \star & \star & \star & \star & \star & \star & 0 & 0 \\ \star & \star & 0 & 0 & 0 & 0 & \star & \star & \star & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star & \star & \star & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \star & 0 & 0 & 0 & 0 & \star & \star & \star & \star & \star \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \star & \star & \star \end{bmatrix}, \quad \mathbf{C} = \begin{bmatrix} \star & \star & 0 & \star & \star & 0 & 0 & 0 \\ \star & \star & \star & \star & 0 & 0 & 0 & 0 \\ 0 & \star & \star & \star & 0 & 0 & \star & \star \\ \star & \star & \star & \star & \star & \star & \star & \star \\ \star & 0 & 0 & \star & \star & \star & 0 & 0 \\ 0 & 0 & 0 & \star & \star & \star & \star & 0 \\ 0 & 0 & \star & \star & 0 & \star & \star & \star \\ 0 & 0 & \star & 0 & 0 & 0 & \star & \star \end{bmatrix}.$$

Sanity check: the matrices $\mathbf{A}$ and $\mathbf{C}$ are symmetric.

SOLUTION of (13-2.r):

We have

- a single global shape function (of $\mathcal{W}^1(\mathcal{M})$) associated with each edge of $\mathcal{M}$,
- a single global shape function (of $\mathcal{W}^0(\mathcal{M})$) associated with each node of $\mathcal{M}$, and
- no global shape functions associated with cells.

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(), 1 },
  {lf::base::RefEl::kSegment(), 1 },
  {lf::base::RefEl::kTria(), 0 },
  {lf::base::RefEl::kQuad(), 0 }});
```

HINT 1 for (13-2.s): By [Lecture → (13.2.2.21)] and the generalized Stokes theorem [Lecture → Thm. 13.1.4.57]

$$\int_K \mathrm{d}_1 \beta^1_{i,K} = \int_{\partial K} \beta^1_{i,K} = 1\,, \quad i = 1,2,3\,. \tag{13.2.41}$$

⌟

SOLUTION of (13-2.s):

❶ Let $q := \mathrm{v.p.}(\mathrm{d}_1\beta^1_{i,K})$ be the function proxy of the exterior derivative of the local Whitney 1-form $\beta^1_{i,K}$. In [Lecture → § 13.1.3.30] we learned that

$$\int_K \mathrm{d}_1\beta^1_{i,K} = \int_K q(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

Moreover, from [Lecture → (13.2.2.42)] we know that q will be *constant*! Thus from

$$\int_K \mathrm{d}_1\beta^1_{i,K} = \int_{\partial K} \beta^1_{i,K} = 1\,, \quad i = 1,2,3\,, \tag{13.2.41}$$

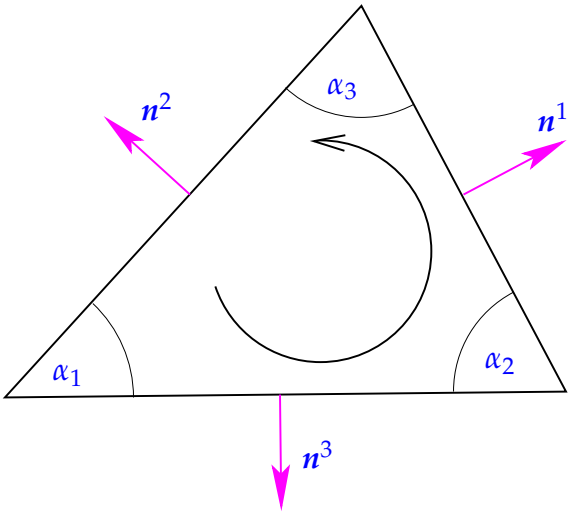
we conclude that $q = \dfrac{1}{|K|}$.

❷ We can also embark on direct computations using the vector proxies from (13.2.40), the product rule

$$\mathrm{curl}_{2D}(f\mathbf{u}) = f\ \mathrm{curl}_{2D}\,\mathbf{u} - \mathbf{curl}_{2D}\,f\cdot\mathbf{u}\,, \quad \mathbf{u}(\boldsymbol{x}) = \begin{bmatrix} u_1(\boldsymbol{x}) \\ u_2(\boldsymbol{x}) \end{bmatrix}\,, \tag{13.2.42}$$

and the fact $\mathrm{curl}_{2D} \circ \mathbf{grad} = 0$:

$$\begin{aligned} \mathrm{curl}_{2D}\,\check{\beta}^1_{i,K} &= -\,\mathbf{curl}_{2D}\,\lambda_i\cdot\mathbf{grad}\,\lambda_k + \mathbf{curl}_{2D}\,\lambda_k\cdot\mathbf{grad}\,\lambda_i \\ &= \mathbf{grad}\,\lambda_i^{\perp}\cdot\mathbf{grad}\,\lambda_k - \mathbf{grad}\,\lambda_k^{\perp}\cdot\mathbf{grad}\,\lambda_i\,, \quad k := i \mod 3+1\,. \end{aligned}$$



From [Lecture → (2.4.5.4)] we get

$$\begin{aligned} \mathbf{grad}\,\lambda_1 &= -\frac{|e_1|}{2|K|}\,\boldsymbol{n}^1\,, \\ \mathbf{grad}\,\lambda_2 &= -\frac{|e_2|}{2|K|}\,\boldsymbol{n}^2\,, \\ \mathbf{grad}\,\lambda_3 &= -\frac{|e_3|}{2|K|}\,\boldsymbol{n}^3\,. \end{aligned}$$

$$\begin{aligned} \mathrm{curl}_{2D}\,\check{\beta}^1_{i,K} &= \frac{|e_i||e_k|}{4|K|^2}\left(\left(\boldsymbol{n}^i\right)^{\perp}\cdot\boldsymbol{n}^k - \left(\left(\boldsymbol{n}^k\right)^{\perp}\cdot\boldsymbol{n}^i\right)\right. \\ &= \frac{2}{2|K|}\,. \end{aligned}$$

where we used the area formula $|K| = 2|e_i||e_k|\sin(\alpha_k)$ from elementary trigonometry and $\angle((\boldsymbol{n}^i)^{\perp}, \boldsymbol{n}^k) =$

❸ By the very construction of local Whitney forms in [Lecture → § 13.2.2.15] we know that (edges carry the induced orientation!) $\mathrm{d}_1\beta^1_{i,K}$ is $\beta^2_{1,K}$. Since there is only one local Whitney 2-forms in 2D, which is constant, and $\int_K \beta^2_{1,K} = 1$, the result $\mathrm{v.p.}(\mathrm{d}_1\beta^1_{i,K})(\boldsymbol{x}) = \frac{1}{K}$ follows.

HINT 1 for (13-2.b): Note that according to [Lecture → (13.1.4.105)] we have

$$\omega \wedge \eta = -\eta \wedge \omega \quad \text{for} \quad \omega, \eta \in C^0\Lambda^1(\Omega) \,. \tag{13.2.5}$$

This can be used as a sanity check for the formula you have found. ⌟

HINT 2 for (13-2.b): Note that

$$\det(\mathbf{p},\mathbf{q}) = \mathbf{p}^\top \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \mathbf{q}\,, \quad \mathbf{p},\mathbf{q} \in \mathbb{R}^2\,. \tag{13.2.6}$$

⌟

SOLUTION of (13-2.b):

According to [Lecture → Def. 13.1.4.102]

$$(\omega \wedge \eta)(x)(\mathbf{p},\mathbf{q}) = \omega(x)(\mathbf{p})\,\eta(x)(\mathbf{q}) - \omega(x)(\mathbf{q})\,\eta(x)(\mathbf{p}) \quad \begin{array}{l} \forall \mathbf{p},\mathbf{q} \in \mathbb{R}^2,\ x \in \Omega\,, \\ \omega,\eta \in C^0\Lambda^1(\Omega)\,. \end{array} \tag{13.2.7}$$

We write $\mathbf{u} := \mathrm{v.p.}(\omega)$, $\mathbf{v} := \mathrm{v.p.}(\eta)$. Using the above conventions for proxy vectorfields of 1-forms and then (13.2.6) we find

$$\begin{aligned} (\omega \wedge \eta)(x)(\mathbf{p},\mathbf{q}) &= (\mathbf{u}(x)\cdot\mathbf{p})(\mathbf{v}(x)\cdot\mathbf{q}) - (\mathbf{u}(x)\cdot\mathbf{q})(\mathbf{v}(x)\cdot\mathbf{p}) \\ &= \mathbf{p}^\top(\mathbf{u}(x)\mathbf{v}(x)^\top)\mathbf{q} - \mathbf{p}^\top(\mathbf{v}(x)\mathbf{u}(x)^\top)\mathbf{q} \\ &= \mathbf{p}^\top \begin{bmatrix} 0 & u_1v_2 - v_1u_2 \\ u_2v_1 - v_2u_1 & \end{bmatrix} \mathbf{q} \quad [\,\mathbf{u}(x) =: \left[\begin{smallmatrix} u_1 \\ u_2 \end{smallmatrix}\right],\ \mathbf{v}(x) =: \left[\begin{smallmatrix} v_1 \\ v_2 \end{smallmatrix}\right]] \\ &= (u_1v_2 - v_1u_2)(p_1q_2 - p_2q_1) \overset{(13.2.6)}{=} \mathbf{u}(x)^\perp \cdot \mathbf{v}(x)\det(\mathbf{p},\mathbf{q})\,. \end{aligned}$$

Recall that the counter-clockwise $\frac{\pi}{2}$-rotated vector is

$$\mathbf{u}(x)^\perp = \begin{bmatrix} -u_2 \\ u_1 \end{bmatrix}.$$

Comparing the above formula with he definition of the function proxy for a 2-form in 2D [Lecture → (13.1.3.22)], we conclude

$$\mathrm{v.p.}(\omega \wedge \eta) = \mathrm{v.p.}(\omega)^\perp \cdot \mathrm{v.p.}(\eta)\,, \quad \omega,\eta \in C^0\Lambda^1(\Omega)\,. \tag{13.2.8}$$

Since $\left(\mathbf{u}^\perp\right)^\perp = -\mathbf{u}$ for all $\mathbf{u} \in \mathbb{R}^2$, we have

$$\mathbf{u}(x)^\perp \cdot \mathbf{v}(x) = \left(\mathbf{u}(x)^\perp\right)^\perp \cdot \mathbf{v}(x)^\perp = -\mathbf{v}(x)^\perp \cdot \mathbf{u}(x)\,,$$

which matches (13.2.5).

SOLUTION of (13-2.t):

In Sub-problem (13-2.s) we have found $\mathrm{curl}_{2D}\,\check{\beta}^1_{i,K}(\boldsymbol{x}) \equiv \frac{1}{|K|}$ so that $\mathbf{A}_K$ is straightforward:

$$\mathbf{A}_K = \frac{1}{|K|}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} . \tag{13.2.44}$$

SOLUTION of (13-2.u):

We use the formulas for $\check{\beta}^1_{i,K}$ from (13.2.40) and find

$$(\mathbf{B}_K)_{i,j} = \int_K \left(\lambda_j(\boldsymbol{x})\,\mathbf{grad}\,\lambda_k(\boldsymbol{x}) - \lambda_k(\boldsymbol{x})\,\mathbf{grad}\,\lambda_j(\boldsymbol{x})\right)\cdot\mathbf{grad}\,\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad \begin{array}{l} i,j=1,2,3\,,\\ k:=j \mod 3+1\,. \end{array}$$

We know that $\mathbf{grad}\,\lambda_i(\boldsymbol{x}) \equiv \mathrm{const}$ and $\int_K \lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{1}{3}|K|$. This means

$$(\mathbf{B}_K)_{i,j} = \tfrac{1}{3}|K|\left(\mathbf{grad}\,\lambda_k\cdot\mathbf{grad}\,\lambda_i - \mathbf{grad}\,\lambda_j\cdot\mathbf{grad}\,\lambda_i\right)\,,\quad i,j=1,2,3\,,\quad k:=j \mod 3+1\,.$$

$$\blacktriangleright\quad \mathbf{B}_K = \frac{1}{3}\begin{bmatrix} (\mathbf{L}_K)_{1,2}-(\mathbf{L}_K)_{1,1} & (\mathbf{L}_K)_{1,3}-(\mathbf{L}_K)_{1,2} & (\mathbf{L}_K)_{1,1}-(\mathbf{L}_K)_{1,3}\\ (\mathbf{L}_K)_{2,2}-(\mathbf{L}_K)_{2,1} & (\mathbf{L}_K)_{2,3}-(\mathbf{L}_K)_{2,2} & (\mathbf{L}_K)_{2,1}-(\mathbf{L}_K)_{2,3}\\ (\mathbf{L}_K)_{3,2}-(\mathbf{L}_K)_{3,1} & (\mathbf{L}_K)_{3,3}-(\mathbf{L}_K)_{3,2} & (\mathbf{L}_K)_{3,1}-(\mathbf{L}_K)_{3,3}\end{bmatrix}\,. \tag{13.2.47}$$

Since, $\sum_i \lambda_i \equiv 1$, we obtain $\sum_i (\mathbf{B}_K)_{i,j} = 0$, which is a useful test, e.g.,

$$\begin{aligned}(\mathbf{L}_K)_{1,2}-(\mathbf{L}_K)_{1,1}+(\mathbf{L}_K)_{2,2}-(\mathbf{L}_K)_{2,1}+(\mathbf{L}_K)_{3,2}-(\mathbf{L}_K)_{3,1}&\\ = \Big((\mathbf{L}_K)_{1,2}+(\mathbf{L}_K)_{2,2}+(\mathbf{L}_K)_{3,2}\Big) - \Big((\mathbf{L}_K)_{1,1}+(\mathbf{L}_K)_{2,1}+(\mathbf{L}_K)_{3,1}\Big) &= 0\,,\end{aligned}$$

because the row and column sums of entries of $\mathbf{L}_K$ vanish.

HINT 1 for (13-2.v): You can compute the entries of the element matrix $\mathbf{C}_K$ by analytic evaluation of the two-dimensional integrals. In order to avoid cumbersome computations, you may rely on the general formula from

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1! \alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, . \tag{13.2.49}$$

that gives the integral of products of barycentric coordinate functions over triangles. ⌟

SOLUTION of (13-2.v):

Appealing to [Lecture → Lemma 2.7.5.5] for $d = 2$ and $\alpha_1 + \alpha_2 + \alpha_3 = 2$ we find for the element matrix the general formula:

$$\mathbf{C}_K = \frac{|K|}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} , \tag{13.2.50}$$

where $|K|$ stands for the area of the triangle K.

The same question is also asked in Sub-problem (2-5.b) and Sub-problem (2-8.g).

SOLUTION of (13-2.w):

When relating (13.2.3) to (13.2.19) in Sub-problem (13-2.j) we have identified several bilinear forms as building blocks. How can we all merge them into one?

We have encountered this issue on several occasions, whenever we wanted to apply results about general linear variational problems to saddle point variational problems, as, for instance, in [Lecture → § 12.3.3.11].

Regard (13.2.3) as a variational problem on the "monolithic space" $V := H\Lambda^1(\mathsf{d},\Omega) \times H\Lambda^0(\mathsf{d},\Omega)$.

From this point of view (13.2.3) becomes: Seek $(\omega,\pi) \in V$ such that

$$\begin{aligned}\left(\mathsf{d}_1\omega,\mathsf{d}_1\omega'\right)_{L^2\Lambda^2(\Omega)} + \left(\omega',\mathsf{d}_0\pi\right)_{L^2\Lambda^1(\Omega)} + \left(\omega,\mathsf{d}_0\pi'\right)_{L^2\Lambda^1(\Omega)} - \left(\pi,\pi'\right)_{L^2\Lambda^0(\Omega)} \\ = \left(\varphi,\omega'\right)_{L^2\Lambda^1(\Omega)} \quad \forall (\omega',\pi') \in V\,.\end{aligned}$$

This is a standard linear variational problem

$$\widetilde{u} \in V\colon \quad \widetilde{\mathsf{a}}(\widetilde{u},\widetilde{v}) = \widetilde{\ell}(\widetilde{v}) \quad \forall \widetilde{v} \in V\,,$$

with

- $V := H\Lambda^1(\mathsf{d},\Omega) \times H\Lambda^0(\mathsf{d},\Omega)$,
- $\widetilde{u} := \left[\begin{smallmatrix}\omega\\ \pi\end{smallmatrix}\right]$, $\widetilde{v} := \left[\begin{smallmatrix}\omega'\\ \pi'\end{smallmatrix}\right]$,
- $\widetilde{\mathsf{a}} \quad \leftrightarrow$
$$\left(\left[\begin{smallmatrix}\omega\\ \pi\end{smallmatrix}\right], \left[\begin{smallmatrix}\omega'\\ \pi'\end{smallmatrix}\right]\right) \mapsto \left(\mathsf{d}_1\omega,\mathsf{d}_1\omega'\right)_{L^2\Lambda^2(\Omega)} + \left(\omega',\mathsf{d}_0\pi\right)_{L^2\Lambda^1(\Omega)} + \left(\omega,\mathsf{d}_0\pi'\right)_{L^2\Lambda^1(\Omega)} - \left(\pi,\pi'\right)_{L^2\Lambda^0(\Omega)}\,,$$
- $\widetilde{\ell} \quad \leftrightarrow \quad \left[\begin{smallmatrix}\omega'\\ \pi'\end{smallmatrix}\right] \mapsto \left(\varphi,\omega'\right)_{L^2\Lambda^1(\Omega)}$.

Hence the localized bilinear form on K is

$$\begin{aligned}&\widetilde{\mathsf{a}}_K\left(\left[\begin{smallmatrix}\omega\\ \pi\end{smallmatrix}\right], \left[\begin{smallmatrix}\omega'\\ \pi'\end{smallmatrix}\right]\right) \\ &\quad := \left(\mathsf{d}_1\omega,\mathsf{d}_1\omega'\right)_{L^2\Lambda^2(K)} + \left(\omega',\mathsf{d}_0\pi\right)_{L^2\Lambda^1(K)} + \left(\omega,\mathsf{d}_0\pi'\right)_{L^2\Lambda^1(K)} - \left(\pi,\pi'\right)_{L^2\Lambda^0(K)}\,. \qquad (13.2.53)\end{aligned}$$

The ordered set of local shape functions is

$$\left\{b_K^1, b_K^2, b_K^3, b_K^4, b_K^5, b_K^6\right\} = \left\{\left[\begin{smallmatrix}0\\ \lambda_1\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\ \lambda_2\end{smallmatrix}\right], \left[\begin{smallmatrix}0\\ \lambda_3\end{smallmatrix}\right], \left[\begin{smallmatrix}\beta^1_{1,K}\\ 0\end{smallmatrix}\right], \left[\begin{smallmatrix}\beta^1_{2,K}\\ 0\end{smallmatrix}\right], \left[\begin{smallmatrix}\beta^1_{3,K}\\ 0\end{smallmatrix}\right]\right\}. \qquad (13.2.54)$$

The ordering is determined by LEHRFEM++'s numbering rule for local shape functions [Lecture → § 2.7.4.1

- First number those associated with the nodes of the mesh taking into account the order of the vertices of the triangle, and
- then number the local shape functions attached to edges following the numbering of the edges.

SOLUTION of (13-2.x):

Using the notations from Sub-problem (13-2.w), in particular from (13.2.53) and (13.2.54), we have

$$(\mathbf{M}_K)_{i,j} = \widetilde{\mathsf{a}}(b_K^j, b_K^i) \;, \quad i,j \in \{1,\dots,6\} \;.$$

Inserting the concrete definitions we find

$$\begin{aligned}
\mathbf{M}_K &= \begin{bmatrix} \left[\widetilde{\mathsf{a}}\left(\begin{bmatrix}0\\ \lambda_j\end{bmatrix}, \begin{bmatrix}0\\ \lambda_i\end{bmatrix}\right)\right]_{i,j=1}^3 & \left[\widetilde{\mathsf{a}}\left(\begin{bmatrix}0\\ \lambda_i\end{bmatrix}, \begin{bmatrix}\beta^1_{j,K}\\ 0\end{bmatrix}\right)\right]_{i,j=1}^3 \\ \left[\widetilde{\mathsf{a}}\left(\begin{bmatrix}\beta^1_{i,K}\\ 0\end{bmatrix}, \begin{bmatrix}0\\ \lambda_j\end{bmatrix}\right)\right]_{i,j=1}^3 & \left[\widetilde{\mathsf{a}}\left(\begin{bmatrix}\beta^1_{j,K}\\ 0\end{bmatrix}, \begin{bmatrix}\beta^1_{i,K}\\ 0\end{bmatrix}\right)\right]_{i,j=1}^3 \end{bmatrix} \\
&= \begin{bmatrix} \left[(\lambda_j, \lambda_i)_{L^2\Lambda^0(K)}\right]_{i,j=1}^3 & \left[\left(\beta^1_{j,K}, \mathsf{d}_0\lambda_j\right)_{L^2\Lambda^1(K)}\right]_{i,j=1}^3 \\ \left[\left(\beta^1_{i,K}, \mathsf{d}_0\lambda_j\right)_{L^2\Lambda^1(K)}\right]_{i,j=1}^3 & \left[\left(\mathsf{d}_1\beta^1_{j,K}, \mathsf{d}_1\beta^1_{i,K}\right)_{L^2\Lambda^2(K)}\right]_{i,j=1}^3 \end{bmatrix} = \begin{bmatrix} \mathbf{C}_K & \mathbf{B}_K \\ \mathbf{B}_K^\top & \mathbf{A}_K \end{bmatrix} .
\end{aligned}$$

SOLUTION of (13-2.y):

On $\widehat{K}$ we have with $\boldsymbol{x} = \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right]$

$$\begin{aligned} &\lambda_1(\boldsymbol{x}) = 1 - x_1 - x_2\,, \quad \lambda_2(\boldsymbol{x}) = x_1\,, \qquad \lambda_3(\boldsymbol{x}) = x_2\,, \\ &\mathbf{grad}\,\lambda_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, \qquad \mathbf{grad}\,\lambda_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \mathbf{grad}\,\lambda_3 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \end{aligned}$$

and, consequently,

$$\begin{aligned} \check{\beta}_{1,\widehat{K}}(\boldsymbol{x}) &= \lambda_1(\boldsymbol{x})\,\mathbf{grad}\,\lambda_2(\boldsymbol{x}) - \lambda_2(\boldsymbol{x})\,\mathbf{grad}\,\lambda_1(\boldsymbol{x}) = (1-x_1-x_2)\begin{bmatrix} 1 \\ 0 \end{bmatrix} - x_1\begin{bmatrix} -1 \\ -1 \end{bmatrix} = \begin{bmatrix} 1-x_2 \\ x_1 \end{bmatrix}, \\ \check{\beta}_{2,\widehat{K}}(\boldsymbol{x}) &= \lambda_2(\boldsymbol{x})\,\mathbf{grad}\,\lambda_3(\boldsymbol{x}) - \lambda_3(\boldsymbol{x})\,\mathbf{grad}\,\lambda_2(\boldsymbol{x}) = x_1\begin{bmatrix} 0 \\ 1 \end{bmatrix} - x_2\begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}, \\ \check{\beta}_{3,\widehat{K}}(\boldsymbol{x}) &= \lambda_3(\boldsymbol{x})\,\mathbf{grad}\,\lambda_1(\boldsymbol{x}) - \lambda_1(\boldsymbol{x})\,\mathbf{grad}\,\lambda_3(\boldsymbol{x}) = x_2\begin{bmatrix} -1 \\ -1 \end{bmatrix} - (1-x_1-x_2)\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -x_2 \\ x_1-1 \end{bmatrix}. \end{aligned}$$

As a sanity check we can verify [Lecture → (13.2.2.21)]

$$\int_{\widehat{e}_i} \beta_{j,\widehat{K}} = \int_{\widehat{e}_i} \check{\beta}_{j,\widehat{K}}(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}}(\boldsymbol{x}) = \int_{\widehat{e}_i} \check{\beta}_{j,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_i \, dS(\boldsymbol{x}) = \begin{cases} 1 & \text{, if } \quad i = j\,, \\ 0 & \text{else.} \end{cases} \tag{13.2.55}$$

The *oriented* edge unit tangent vectors (green arrows in Fig. 241) are

$$\widehat{\boldsymbol{t}}_1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad , \quad \widehat{\boldsymbol{t}}_2 = \tfrac{1}{\sqrt{2}}\begin{bmatrix} -1 \\ 1 \end{bmatrix} \quad , \quad \widehat{\boldsymbol{t}}_3 = \begin{bmatrix} 0 \\ -1 \end{bmatrix}.$$

We find

$$\begin{aligned} &\check{\beta}_{1,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_1 = 1 - x_2 = 1 \quad \text{on} \quad \widehat{e}_1\,, \\ &\check{\beta}_{1,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_2 = \tfrac{1}{\sqrt{2}}(1 - x_1 + x_2) = 0 \quad \text{on} \quad \widehat{e}_2\,, \\ &\check{\beta}_{1,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_3 = -x_1 = 0 \quad \text{on} \quad \widehat{e}_3\,, \\ &\check{\beta}_{2,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_1 = -x_2 = 0 \quad \text{on} \quad \widehat{e}_1\,, \\ &\check{\beta}_{2,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_2 = \tfrac{1}{\sqrt{2}}(x_1 + x_2) = \tfrac{1}{\sqrt{2}} \quad \text{on} \quad \widehat{e}_2\,, \\ &\check{\beta}_{2,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_3 = x_1 = 0 \quad \text{on} \quad \widehat{e}_3\,, \\ &\check{\beta}_{3,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_1 = -x_2 = 0 \quad \text{on} \quad \widehat{e}_1\,, \\ &\check{\beta}_{3,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_2 = \tfrac{1}{\sqrt{2}}(1 - x_1 + x_2) = 0 \quad \text{on} \quad \widehat{e}_2\,, \\ &\check{\beta}_{3,\widehat{K}}(\boldsymbol{x}) \cdot \widehat{\boldsymbol{t}}_3 = 1 - x_1 = 1 \quad \text{on} \quad \widehat{e}_3\,. \end{aligned}$$

From these formulas (13.2.55) is immediate.

In addition, using $\mathrm{curl}_{2D}\,\mathbf{u} = \frac{\partial u_2}{\partial x_1} - \frac{\partial u_1}{\partial x_2}$, we can directly compute

$$\mathrm{curl}_{2D}\,\check{\beta}_{i,\widehat{K}}(\boldsymbol{x}) = 2 = \frac{1}{|\widehat{K}|}\,, \quad i = 1,2,3\,.$$

This agrees with the result of Sub-problem (13-2.s) and we recover (13.2.44)

$$\mathbf{A}_{\widehat{K}} = \begin{bmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \\ 2 & 2 & 2 \end{bmatrix} . \tag{13.2.56}$$

From [Lecture → § 4.1.3.3] or [Lecture → (2.4.5.8)] we learn that the $\mathcal{S}_1^0(\mathcal{M})$ element matrix for $-\Delta$ on $\widehat{K}$ is

$$\mathbf{L}_{\widehat{K}} = \frac{1}{2}\begin{bmatrix} 2 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} . \tag{13.2.57}$$

Returning to the result of Sub-problem (13-2.u), we obtain

$$\mathbf{B}_{\widehat{K}} = \frac{1}{6}\begin{bmatrix} -1-2 & -1+1 & 2+1 \\ 1+1 & 0-1 & -1-0 \\ 0+1 & 1-0 & -1-1 \end{bmatrix} = \frac{1}{6}\begin{bmatrix} -3 & 0 & 3 \\ 2 & -1 & -1 \\ 1 & 1 & -2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{3} & -\frac{1}{6} & -\frac{1}{6} \\ \frac{1}{6} & \frac{1}{6} & -\frac{1}{3} \end{bmatrix} . \tag{13.2.58}$$

We can also do a direct computation exploiting that $\boldsymbol{x} \mapsto \check{\beta}_{i,K} \cdot \mathbf{grad}\, \lambda_j \in (\mathcal{P}_1(K))^2$ which enables the application of a 1-point "quadrature rule":

$$\int_{\widehat{K}} \check{\beta}^1_{i,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_j \,\mathrm{d}\boldsymbol{x} = |K| \check{\beta}^1_{i,K}(\begin{bmatrix} 1/3 \\ 1/3 \end{bmatrix}) \cdot \mathbf{grad}\, \lambda_j :$$

$$\begin{aligned}
\int_{\widehat{K}} \check{\beta}^1_{1,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_1 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} -1 \\ -1 \end{bmatrix} = -\frac{1}{2} = (\mathbf{B}_{\widehat{K}})_{1,1} , \\
\int_{\widehat{K}} \check{\beta}^1_{1,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_2 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{3} = (\mathbf{B}_{\widehat{K}})_{2,1} , \\
\int_{\widehat{K}} \check{\beta}^1_{1,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_3 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} \frac{2}{3} \\ \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{6} = (\mathbf{B}_{\widehat{K}})_{3,1} , \\
\int_{\widehat{K}} \check{\beta}^1_{2,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_1 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} -\frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} -1 \\ -1 \end{bmatrix} = 0 = (\mathbf{B}_{\widehat{K}})_{1,2} , \\
\int_{\widehat{K}} \check{\beta}^1_{2,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_2 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} -\frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = -\frac{1}{6} = (\mathbf{B}_{\widehat{K}})_{2,2} , \\
\int_{\widehat{K}} \check{\beta}^1_{2,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_3 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} -\frac{1}{3} \\ \frac{1}{3} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \frac{1}{6} = (\mathbf{B}_{\widehat{K}})_{3,2} , \\
\int_{\widehat{K}} \check{\beta}^1_{3,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_1 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} -\frac{1}{3} \\ -\frac{2}{3} \end{bmatrix} \cdot \begin{bmatrix} -1 \\ -1 \end{bmatrix} = \frac{1}{2} = (\mathbf{B}_{\widehat{K}})_{1,3} , \\
\int_{\widehat{K}} \check{\beta}^1_{3,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_2 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} -\frac{1}{3} \\ -\frac{2}{3} \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} = -\frac{1}{6} = (\mathbf{B}_{\widehat{K}})_{2,3} , \\
\int_{\widehat{K}} \check{\beta}^1_{3,K}(\boldsymbol{x}) \cdot \mathbf{grad}\, \lambda_3 \,\mathrm{d}\boldsymbol{x} &= \tfrac{1}{2}\begin{bmatrix} -\frac{1}{3} \\ -\frac{2}{3} \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} = -\frac{1}{3} = (\mathbf{B}_{\widehat{K}})_{3,3} .
\end{aligned}$$

Finally, for $K = \widehat{K}$ we get from (13.2.50)

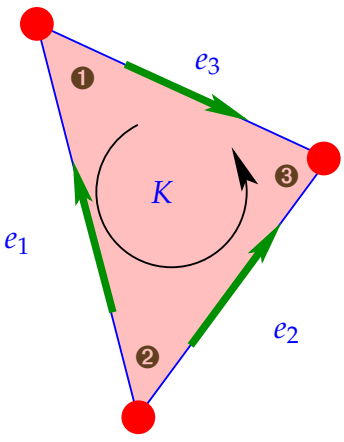
$$\mathbf{C}_{\widehat{K}} = \frac{1}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}.$$

Assembling these matrices into $\mathbf{M}_K$ according to the formula from Sub-problem (13-2.x), we arrive at

$$\mathbf{M}_{\widehat{K}} = \begin{bmatrix} -\mathbf{C}_{\widehat{K}} & \mathbf{B}_{\widehat{K}} \\ \mathbf{B}_{\widehat{K}}^{\top} & \mathbf{A}_{\widehat{K}} \end{bmatrix} = \left[\begin{array}{ccc|ccc} -\frac{1}{12} & -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{2} & 0 & \frac{1}{2} \\ -\frac{1}{24} & -\frac{1}{12} & -\frac{1}{24} & \frac{1}{3} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{24} & -\frac{1}{24} & -\frac{1}{12} & \frac{1}{6} & \frac{1}{6} & -\frac{1}{3} \\ \hline -\frac{1}{2} & \frac{1}{3} & \frac{1}{6} & 2 & 2 & 2 \\ 0 & -\frac{1}{6} & \frac{1}{6} & 2 & 2 & 2 \\ \frac{1}{2} & -\frac{1}{6} & -\frac{1}{3} & 2 & 2 & 2 \end{array}\right]. \tag{13.2.59}$$

By construction this matrix must be symmetric.

SOLUTION of (13-2.z):



We have to take into account the **relative orientations** of the edges of the marked triangle, whose own orientation is determined by the numbering of its vertices ❶ → ❷ → ❸. Refer to [Lecture → Ex. 13.2.1.14] for more explanations.
The relative orientations are given by

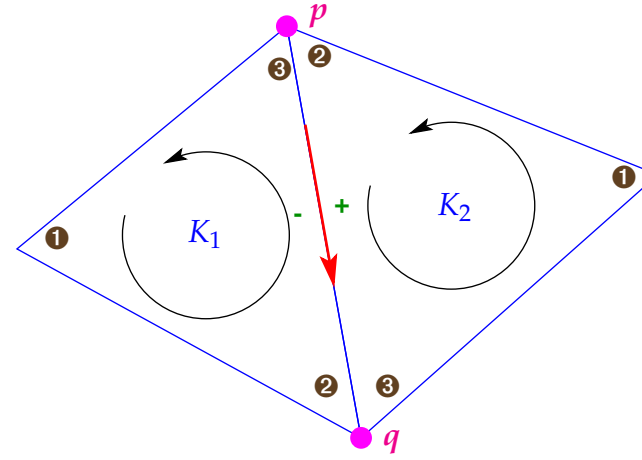
$$o_r(e_1,K) = -1\,, \quad o_r(e_2,K) = 1\,, \quad o_r(e_3,K) = -1\,.$$
$$\blacktriangleright \quad \mathbf{R} = \operatorname{diag}(1,1,1,-1,1,-1)\,.$$

LEHRFEM++'s local numbering scheme from Fig. 239 applies.
Note that the local shape functions associated with nodes are numbered first and those are not affected by issues of orientation. So the first three rows an columns of $\mathbf{M}_K$ will never undergo a sign change.

Remark. The global shape function of the Whitney finite element space $\mathcal{W}^1(\mathcal{M})$ associated with the edge $e = [\boldsymbol{p},\boldsymbol{q}] \in \mathcal{F}_1(\mathcal{M})$ is

$$\beta^1_e := \beta^0_{\boldsymbol{p}}\mathrm{d}_0\beta^0_{\boldsymbol{q}} - \beta^0_{\boldsymbol{q}}\mathrm{d}_0\beta^0_{\boldsymbol{p}}\,,$$

where $\beta^0_{\boldsymbol{p}}$, $\beta^0_{\boldsymbol{q}}$ are the Whitney 0-forms, the "tent functions" [Lecture → (2.4.3.5)], belonging to the mesh nodes $\boldsymbol{p},\boldsymbol{q} \in \mathcal{F}_0(\mathcal{M})$. See [Lecture → Fig. 695] for a plot of the vectorfield proxy of β^1_e.



Hence, referring to Fig. 244 and using λ_i to denote the barycentric coordinate functions for both e-adjacent triangles K_1 and K_2, we have

$$\begin{aligned} \beta^1_e\big|_{K_1} &= \lambda_3\mathrm{d}_0\lambda_2 - \lambda_2\mathrm{d}_0\lambda_3 \quad \text{on} \quad K_1\,, \\ \beta^1_e\big|_{K_2} &= \lambda_2\mathrm{d}_0\lambda_3 - \lambda_3\mathrm{d}_0\lambda_2 \quad \text{on} \quad K_2\,. \end{aligned} \tag{13.2.60}$$

Now remember, what local shape functions we used in Sub-problem (13-2.t) and Sub-problem (13-2.u) to compute the element matrices, see (13.2.40),

$$\text{for} \quad K_1\text{:} \quad \begin{aligned} \beta^1_{1,K_1} &:= \lambda_1\mathrm{d}_0\lambda_2 - \lambda_2\mathrm{d}_0\lambda_1\,, \\ \beta^1_{2,K_1} &:= \lambda_2\mathrm{d}_0\lambda_3 - \lambda_3\mathrm{d}_0\lambda_2\,, \\ \beta^1_{3,K_1} &:= \lambda_3\mathrm{d}_0\lambda_1 - \lambda_1\mathrm{d}_0\lambda_3 \end{aligned} \quad \text{and for} \quad K_2\text{:} \quad \begin{aligned} \beta^1_{1,K_2} &:= \lambda_1\mathrm{d}_0\lambda_2 - \lambda_2\mathrm{d}_0\lambda_1\,, \\ \beta^1_{2,K_2} &:= \lambda_2\mathrm{d}_0\lambda_3 - \lambda_3\mathrm{d}_0\lambda_2\,, \\ \beta^1_{3,K_2} &:= \lambda_3\mathrm{d}_0\lambda_1 - \lambda_1\mathrm{d}_0\lambda_3\,. \end{aligned} \tag{13.2.61}$$

Also remember that by the strict definition [Lecture → Def. 2.5.3.4] of local shape functions those are the restrictions of global shape functions to mesh cells. Yet, there is an obvious mismatch between (13.2.60) and (13.2.61), in case the intrinsic orientation of e is opposite to its induced orientation with respect to an adjacent triangle. This can be corrected by *flipping the sign* of the local shape function, if $o_r(e,K_i) = -1$.

SOLUTION of (13-2.aa):

The member function `Eval()` takes a cell object for $K \in \mathcal{M}$ as argument and returns $\mathbf{M}_K$ as an `Eigen::6,6>` object.

As elaborated in Sub-problem (13-2.x) we have

$$\mathbf{M}_K = \begin{bmatrix} \mathbf{C}_K & \mathbf{B}_K \\ \mathbf{B}_K^\top & \mathbf{A}_K \end{bmatrix} \quad , \quad \begin{array}{l} \mathbf{C}_K \in \mathbb{R}^{3,3} \text{ given in (13.2.50)} , \\ \mathbf{B}_K \in \mathbb{R}^{3,3} \text{ according to (13.2.47)} , \\ \mathbf{A}_K \in \mathbb{R}^{3,3} \text{ as in (13.2.43)} . \end{array}$$

The following implementation does not exploit the symmetry of $\mathbf{A}_K$ and $\mathbf{C}_K$.

The entries of the matrices $\mathbf{B}_K$ and $\mathbf{C}_K$ are given by integrals over K, see (13.2.46) and (13.2.48). These integrals are computed by means of an 3-point quadrature rule, which is exact for integrands $\in \mathcal{P}_2(\mathbb{R}^2)$. More direct evaluations would be possible, but would lead to more complicated code.

C++ code 13.2.62: Implementation of `Eval()` member function computing $\mathbf{M}_K$ for a mesh cell entity. ↪ GitLab

```
HodgeLaplacian2DElementMatrixProvider::ElemMat
HodgeLaplacian2DElementMatrixProvider::Eval_ref(const lf::mesh::Entity& cell) {
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type " << cell.RefEl());
  LF_VERIFY_MSG(cell.Geometry()->isAffine(),
                "Triangle must have straight edges");
  // Area of the triangle
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions, see
  // [Lecture → Code 2.4.5.11]. Get vertices of the triangle
  auto endpoints = lf::geometry::Corners(*(cell.Geometry()));
  Eigen::Matrix<double, 3, 3> X;  // temporary matrix
  X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
  X.block<3, 2>(0, 1) = endpoints.transpose();
  // This matrix contains grad λ_i in its columns
  const auto G{X.inverse().block<2, 3>(1, 0)};
  // Dummy lambda functions for barycentric coordinates
  const std::array<std::function<double(Eigen::Vector3d)>, 3> lambda{
      [](Eigen::Vector3d c) -> double { return c[0]; },
      [](Eigen::Vector3d c) -> double { return c[1]; },
      [](Eigen::Vector3d c) -> double { return c[2]; }};
  // Lambda functions for local Whitney 1-forms, see (13.2.40).
  // Note that grad λ_i is accessed as G.col(i-1).
  const std::array<std::function<Eigen::Vector2d(Eigen::Vector3d)>, 3> beta{
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return c[0] * G.col(1) - c[1] * G.col(0);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return c[1] * G.col(2) - c[2] * G.col(1);
      },
      [&G](Eigen::Vector3d c) -> Eigen::Vector2d {
        return c[2] * G.col(0) - c[0] * G.col(2);
      }};
  // Barycentric coordinates of the midpoints of the edges for
  // use with the third order 3-point edge midpoint quadrature rule
  // [Lecture → (2.7.5.35)]
```

```
  const std::array<Eigen::Vector3d, 3> mp = {Eigen::Vector3d({0.5, 0.5, 0}),
                                             Eigen::Vector3d({0, 0.5, 0.5}),
                                             Eigen::Vector3d({0.5, 0, 0.5})};
  // Compute the element matrix for -Δ and S_1^0.
  // See also [Lecture → Code 2.4.5.13].
  // Eigen::Matrix<double, 3, 3> L = area * G.transpose() * G;
  // Fill the four 3x3 blocks of M_K
  const double A_ent = 3.0 / (area * area);
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      // Left upper block -C_K:
      MK_(i, j) = -(lambda[i](mp[0]) * lambda[j](mp[0]) +
                    lambda[i](mp[1]) * lambda[j](mp[1]) +
                    lambda[i](mp[2]) * lambda[j](mp[2]));
      // Upper right block B_K and lower left block B_K^⊤:
      const double val =
          ((beta[j](mp[0])) + (beta[j](mp[1])) + (beta[j](mp[2])))
              .dot(G.col(i));
      MK_(i, j + 3) = val;
      MK_(j + 3, i) = val;
      // Lower right block
      MK_(i + 3, j + 3) = A_ent;
    }
  }
  // Correct for orientation mismatch, cf. Sub-problem (13-2.z).
  auto relor = cell.RelativeOrientations();
  LF_ASSERT_MSG(relor.size() == 3, "Triangle should have 3 edges!?");
  for (int k = 0; k < 3; ++k) {
    if (relor[k] == lf::mesh::Orientation::negative) {
      // Flip sign of 3+k-th rown and column
      MK_.col(3 + k) *= -1.0;
      MK_.row(3 + k) *= -1.0;
    }
  }
  // Finally multiply with with the quadrature weight.
  MK_ *= area / 3.0;
  return MK_;
}
```

An alternative implementation directly relies on the formulas derived in the various sub problems.

C++ code 13.2.63: Implementation of `Eval()` member function computing $\mathbf{M}_K$ for a mesh cell entity. ↬ GitLab

```
HodgeLaplacian2DElementMatrixProvider::ElemMat
HodgeLaplacian2DElementMatrixProvider::Eval(const lf::mesh::Entity& cell) {
  LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type " << cell.RefEl());
  LF_VERIFY_MSG(cell.Geometry()->isAffine(),
                "Triangle must have straight edges");
  // Area of the triangle
  double area = lf::geometry::Volume(*cell.Geometry());
  // Compute gradients of barycentric coordinate functions and store
     them in the
  // columns of the 2×3-matrix G
  // clang-format off
```

```
  const Eigen::MatrixXd dpt = (Eigen::MatrixXd(2, 1) << 0.0, 0.0).finished();
  const Eigen::Matrix<double, 2, 3> G =
      cell.Geometry()->JacobianInverseGramian(dpt).block(0, 0, 2, 2) *
    (Eigen::Matrix<double, 2, 3>(2,3) << -1, 1, 0,
                                         -1, 0, 1).finished();
  // clang-format on
  // Compute the element matrix for -Δ and S_1^0.
  // See also [Lecture → Code 2.4.5.13].
  Eigen::Matrix<double, 3, 3> L = area * G.transpose() * G;

  // Fill left upper block C_K, (13.2.50)
  const Eigen::Matrix3d C =
      (Eigen::Matrix3d() << 2, 1, 1, 1, 2, 1, 1, 1, 2).finished();
  MK_.block(0, 0, 3, 3) = -(area / 12.0) * C;
  // Initialize right upper block B_K and left lower block B^⊤,
  // see (13.2.47).
  Eigen::Matrix3d B;
  B.col(0) = (L.col(1) - L.col(0)) / 3.0;
  B.col(1) = (L.col(2) - L.col(1)) / 3.0;
  B.col(2) = (L.col(0) - L.col(2)) / 3.0;
  MK_.block(0, 3, 3, 3) = B;
  MK_.block(3, 0, 3, 3) = B.transpose();
  // Set lower right block A_K, see (13.2.56)
  MK_.block(3, 3, 3, 3) = Eigen::Matrix3d::Constant(1.0 / area);
  // Correct for orientation mismatch, cf. Sub-problem (13-2.z).
  auto relor = cell.RelativeOrientations();
  LF_ASSERT_MSG(relor.size() == 3, "Triangle should have 3 edges!?");
  for (int k = 0; k < 3; ++k) {
    if (relor[k] == lf::mesh::Orientation::negative) {
      // Flip sign of 3+k-th rown and column
      MK_.col(3 + k) *= -1.0;
      MK_.row(3 + k) *= -1.0;
    }
  }
  return MK_;
}
```

The highlighted parts of the functions implement the recipe for adjusting the sign of local shape functions in order to take into account the relative orientation of edges.

SOLUTION of (13-2.ab):

Recall that in a single step of uniform regular refinement the meshwidth is halved.

The tabulated data support the prediction that

- halving the meshwidth $h_{\mathcal{M}}$ leads to a reduction of $\|\omega - \omega_h\|_{L^2\Lambda^1(\Omega)}$ by a factor of 2, and
- halving the meshwidth makes the error $\|\pi - \pi_h\|_{L^2\Lambda^0(\Omega)}$ decrease by a factor of 4

We conclude asymptotic **algebraic convergence**,

$$\|\omega - \omega_h\|_{L^2\Lambda^1(\Omega)} = O(h_{\mathcal{M}}) \quad \text{and} \quad \|\pi - \pi_h\|_{L^2\Lambda^0(\Omega)} = O(h_{\mathcal{M}}^2) \quad \text{for} \quad h_{\mathcal{M}} \to 0\,.$$

Since $N \sim h_{\mathcal{M}}^{-2}$ this implies

$$\|\omega - \omega_h\|_{L^2\Lambda^1(\Omega)} = O(N^{-1/2}) \quad \text{and} \quad \|\pi - \pi_h\|_{L^2\Lambda^0(\Omega)} = O(N^{-1}) \quad \text{for} \quad N \to \infty\,.$$

Assuming that the Whitney finite element discretization of (13.2.3) is stable, that is, the assumptions of Thm. 13.2.35 are satisfied with constants independent of $h_{\mathcal{M}}$ (which holds true), we can predict from (13.2.38)

$$\begin{aligned}\|\omega - \omega_h\|_{H\Lambda^1(\mathsf{d},\Omega)} + \|\pi - \pi_h\|_{H\Lambda^0(\mathsf{d},\Omega)} \\ \leq C\left(\inf_{\eta_h \in \mathcal{W}^1(\mathcal{M})} \|\omega - \eta_h\|_{H\Lambda^1(\mathsf{d},\Omega)} + \inf_{\kappa_h \in \mathcal{W}^0(\mathcal{M})} \|\pi - \kappa_h\|_{H\Lambda^0(\mathsf{d},\Omega)}\right)\,, \quad (13.2.64)\end{aligned}$$

with $C > 0$ independent of $h_{\mathcal{M}}$. The results reported in [Lecture → Thm. 13.2.2.61] then tell us that

$$\|\omega - \omega_h\|_{H\Lambda^1(\mathsf{d},\Omega)} + \|\pi - \pi_h\|_{H\Lambda^0(\mathsf{d},\Omega)} = O(h_{\mathcal{M}}) \quad \text{for} \quad h_{\mathcal{M}} \to 0\,.$$

This fits the observed behavior of $\|\omega - \omega_h\|_{L^2\Lambda^1(\Omega)}$, but $\|\pi - \pi_h\|_{L^2\Lambda^0(\Omega)}$ converges with twice the rate compared to the predictions of the theory.

HINT 1 for (13-2.c): Consult [Lecture → § 13.1.4.49], in particular [Lecture → (13.1.4.55)]. ⌟

SOLUTION of (13-2.c):

Definition [Lecture → Def. 13.1.5.2]. Sobolev space of differentiable differential forms

For a domain $\Omega \subset \mathbb{R}^d$ and $\ell \in \{0,\ldots,d-1\}$ we define

$$H\Lambda^\ell(\mathrm{d},\Omega) := \left\{\omega \in L^2\Lambda^\ell(\Omega) : \mathrm{d}_\ell\omega \in L^2\Lambda^{\ell+1}(\Omega)\right\},$$

which becomes a Hilbert space with the *basis-dependent* norm

$$\|\omega\|^2_{H\Lambda^\ell(\mathrm{d},\Omega)} := \|\omega\|^2_{L^2\Lambda^\ell(\Omega)} + \|\mathrm{d}_\ell\omega\|^2_{L^2\Lambda^{\ell+1}(\Omega)}.$$

For 2D Euclidean vector proxies (Option I) we have the correspondences:

$$\mathrm{d}_0 \;\leftrightarrow\; \mathbf{grad} \quad , \quad \mathrm{d}_1 \;\leftrightarrow\; \mathrm{curl}_{2D}\,. \tag{13.2.9}$$

$$\blacktriangleright\quad H\Lambda^\ell(\mathrm{d},\Omega) \hat{=} \begin{cases} H^1(\Omega) := \left\{v \in L^2(\Omega) : \mathbf{grad}\, v \in \left(L^2(\Omega)\right)^2\right\} & \text{for} \quad \ell = 0\,, \\ \boldsymbol{H}(\mathrm{curl}_{2D},\Omega) := \left\{\mathbf{v} \in \left(L^2(\Omega)\right)^2 : \mathrm{curl}_{2D}\,\mathbf{v} \in L^2(\Omega)\right\} & \text{for} \quad \ell = 1\,, \\ L^2(\Omega) & \text{for} \quad \ell = 2\,. \end{cases}$$

SOLUTION of (13-2.d):

Recall the convention that we have adopted:

$$\omega \in C^0\Lambda^1(\Omega): \quad \mathbf{u} := \mathrm{v.p.}(\omega) : \Omega \to \mathbb{R}^2\,, \quad \mathrm{v.p.}(\omega)(\boldsymbol{x}) \cdot \mathbf{v} := \omega(\boldsymbol{x})(\mathbf{v}) \quad \forall \mathbf{v} \in \mathbb{R}^2\,. \tag{13.2.10}$$

This is the same as for 1-form vector proxies in 3D. So the computations done in [Lecture → (13.1.3.31)] remain valid:

$$\int_\gamma \omega = \int_0^1 \omega(\gamma(s))\left(\frac{d\gamma}{ds}(s)\right) \mathrm{d}s = \int_0^1 \mathbf{u}(\gamma(s)) \cdot \frac{d\gamma}{ds}(s)\, \mathrm{d}s = \int_\gamma \mathbf{u}(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}}(\boldsymbol{x})\,. \tag{13.2.11}$$

Integrating a 1-form over a curve can be realized by evaluating the **path integral** of its vector proxy.

SOLUTION of (13-2.e):

1-forms in $H_0\Lambda^1(\mathrm{d},\Omega)$ when evaluated for $\boldsymbol{x} \in \partial\Omega$ are supposed to have all tangential vectors to $\partial\Omega$ in $\boldsymbol{x}$ in their kernel. In light of (13.2.10) this means for the proxy vectorfield

$$\mathrm{v.p.}(\omega)(\boldsymbol{x}) \cdot \mathbf{t} = 0 \quad \text{for all tangential vectors } \mathbf{t} \text{ in } \boldsymbol{x} \in \partial\Omega \ .$$

So we have to enforce *zero tangential components* on $\partial\Omega$ for $\mathrm{v.p.}(\omega)$ of $\omega \in H_0\Lambda^1(\mathrm{d},\Omega)$,

$$\mathrm{v.p.}(H_0\Lambda^1(\mathrm{d},\Omega)) = \left\{ \mathbf{v} \in \left(L^2(\Omega)\right)^2 : \mathrm{curl}_{2D}\,\mathbf{v} \in L^2(\Omega),\ \mathbf{v}\cdot\mathbf{t} = 0 \text{ on } \partial\Omega \right\} ,$$

where $\mathbf{t} : \partial\Omega \to \mathbb{R}^2$ is a unit tangent vectorfield on $\partial\Omega$.

SOLUTION of (13-2.f):

We use

- the vector proxy incarnations of the exterior derivatives from (13.2.9),
- the vector proxy way of computing the exterior product of two 1-forms from (13.2.8), and
- the path integral counterpart (13.2.11) of the integral of a 1-form over the curve $\partial\Omega$,

and get

$$\int_{\partial\Omega} \omega \wedge \eta = \int_{\Omega} \mathrm{d}_1\omega \wedge \eta - \omega \wedge \mathrm{d}_0\eta \;, \quad \omega \in C^1\Lambda^1(\Omega),\ \eta \in C^0\Lambda^0(\Omega)$$

$$\Updownarrow \quad [\mathbf{u} := \mathrm{v.p.}(\omega),\ p := \mathrm{v.p.}(\eta)\,]$$

$$\int_{\partial\Omega} p(\boldsymbol{x})\mathbf{u}(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}}(\boldsymbol{x}) = \int_{\Omega} \mathrm{curl}_{2D}\,\mathbf{u}(\boldsymbol{x})\,p(\boldsymbol{x}) - \mathbf{u}(\boldsymbol{x})^{\perp} \cdot \mathbf{grad}\,p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$

$$\Updownarrow$$

$$\int_{\partial\Omega} p(\boldsymbol{x})\mathbf{u}(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}}(\boldsymbol{x}) = \int_{\Omega} \mathrm{curl}_{2D}\,\mathbf{u}(\boldsymbol{x})\,p(\boldsymbol{x}) - \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{curl}_{2D}\,p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;. \tag{13.2.13}$$

Note that the two rotation operators are different:

$$\mathbf{curl}_{2D}\,p(\boldsymbol{x}) := \begin{bmatrix} \frac{\partial p}{\partial x_2}(\boldsymbol{x}) \\ -\frac{\partial p}{\partial x_1}(\boldsymbol{x}) \end{bmatrix} \quad , \quad \mathrm{curl}_{2D}\,\mathbf{u}(\boldsymbol{x}) := \frac{\partial u_2}{\partial x_1}(\boldsymbol{x}) - \frac{\partial u_1}{\partial x_2}(\boldsymbol{x}) \;.$$

Remark. Replacing $\mathbf{u}$ with $\mathbf{u}^{\perp}$ in the above formula, we end up with

$$-\int_{\partial\Omega} p(\boldsymbol{x})\,\mathbf{u}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S = \int_{\Omega} -\mathrm{div}\,\mathbf{u}(\boldsymbol{x})\,p(\boldsymbol{x}) - \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{grad}\,p(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;,$$

which is Green's first formula from [Lecture → Thm. 1.5.2.7].

SOLUTION of (13-2.g):

Invoking (13.2.9) and (13.2.1) we obtain:

Seek $\mathbf{u} \in \boldsymbol{H}(\mathrm{curl}_{2D}, \Omega)$ and $p \in H^1(\Omega)$ such that

$$\begin{array}{rcl} \int_\Omega \mathrm{curl}_{2D}\, \mathbf{u}(\boldsymbol{x})\ \mathrm{curl}_{2D}\, \mathbf{u}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \ + \ \int_\Omega \mathbf{u}'(\boldsymbol{x}) \cdot \mathbf{grad}\, p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} & = & \int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{u}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{grad}\, p'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \ - \ \int_\Omega p(\boldsymbol{x})\, p'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} & = & 0\,. \end{array} \tag{13.2.3}$$

for all $\mathbf{u}' \in \boldsymbol{H}(\mathrm{curl}_{2D}, \Omega)$, $p' \in H^1(\Omega)$.

HINT 1 for (13-2.h): A key tool will be the integration by parts formulae

$$\int_{\partial\Omega} p(\boldsymbol{x})\mathbf{u}(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}}(\boldsymbol{x}) = \int_{\Omega} \mathrm{curl}_{2D}\, \mathbf{u}(\boldsymbol{x})\, p(\boldsymbol{x}) - \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{curl}_{2D}\, p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \tag{13.2.13}$$

$$\int_{\partial\Omega} p(\boldsymbol{x})\, \mathbf{u}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) = \int_{\Omega} \mathrm{div}\, \mathbf{u}(\boldsymbol{x})\, p(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{grad}\, p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \tag{13.2.14}$$

for $\mathbf{u} \in (C^1(\overline{\Omega}))^2$, $p \in C^1(\overline{\Omega})$, established in Sub-problem (13-2.f). ⌟

SOLUTION of (13-2.h):

Throughout, we manipulate vector proxies in this sub-problem.

We follow the approach of [Lecture → Section 1.5.3] for extracting boundary value problems from variational problems. We start from the variational problem found in Sub-problem (13-2.g):

$$\begin{aligned} \int_\Omega \operatorname{curl}_{2D} \mathbf{u}(\boldsymbol{x})\, \operatorname{curl}_{2D} \mathbf{u}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \mathbf{u}'(\boldsymbol{x}) \cdot \mathbf{grad}\, p(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{u}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{grad}\, p'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \;-\; \int_\Omega p(\boldsymbol{x})\, p'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned} \tag{13.2.15}$$

for all $\mathbf{u}' \in \boldsymbol{H}(\operatorname{curl}_{2D}, \Omega)$, $p' \in H^1(\Omega)$. As in [Lecture → Section 1.5.3] extra smoothness of the solution $(\mathbf{u}, p)$ of (13.2.15) will be assumed:

$$\mathbf{u} \in (C^2(\overline{\Omega}))^2 \quad \text{and} \quad p \in C^1(\overline{\Omega})\,.$$

❶ Test with compactly supported smooth functions $\mathbf{u}' \in (C_0^\infty(\Omega))^2$, $p' \in C_0(\Omega)$ and apply integration by parts. Boundary terms do not arise, because the test functions vanish on $\partial\Omega$.

$$\begin{aligned} \int_\Omega (\mathbf{curl}_{2D} \operatorname{curl}_{2D} \mathbf{u}(\boldsymbol{x}) + \mathbf{grad}\, p(\boldsymbol{x}) - \mathbf{f}(\boldsymbol{x})) \cdot \mathbf{u}'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall \mathbf{u}' \in (C_0^\infty(\Omega))^2\,, \\ \int_\Omega (-\operatorname{div} \mathbf{u}(\boldsymbol{x}) - p(\boldsymbol{x}))\, p'(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall p' \in C_0(\Omega)\,. \end{aligned}$$

Appealing to the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4], we conclude that $(\mathbf{u}, p)$ solve the *PDEs*

$$\mathbf{curl}_{2D} \operatorname{curl}_{2D} \mathbf{u} + \mathbf{grad}\, p = \mathbf{f} \quad , \quad -\operatorname{div} \mathbf{u} - p = 0 \quad \text{in} \quad \Omega\,. \tag{13.2.16}$$

❷ Next, we test with $\mathbf{u}' \in (C^\infty(\overline{\Omega}))^2$, $p' \in C^\infty(\overline{\Omega})$ and again apply integration by parts. Invoking (13.2.16), the volume integrals can be dropped and we are left with only boundary terms.

$$\begin{aligned} \int_{\partial\Omega} \operatorname{curl}_{2D} \mathbf{u}(\boldsymbol{x})\, \mathbf{u}'(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}}(\boldsymbol{x}) = 0 \quad \forall \mathbf{u}' \in (C^\infty(\overline{\Omega}))^2\,, \\ \int_{\partial\Omega} \mathbf{u}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, p'(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall p' \in C^\infty(\overline{\Omega})\,. \end{aligned}$$

This implies the *boundary conditions*

$$\operatorname{curl}_{2D} \mathbf{u} = 0 \quad , \quad \mathbf{u} \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega\,. \tag{13.2.17}$$

These are **natural boundary conditions**, because they were hidden in the variational formulation and not imposed directly on the trial/test space.

SOLUTION of (13-2.i):

We start from the PDE obtained in Sub-problem (13-2.h):

$$\mathbf{curl}_{2D}\,\mathrm{curl}_{2D}\,\mathbf{u} + \mathbf{grad}\,p = \mathbf{f} \quad , \quad -\mathrm{div}\,\mathbf{u} - p = 0 \quad \text{in} \quad \Omega\,. \tag{13.2.16}$$

Elimination of p yields

$$\begin{gathered}
-\mathbf{grad}\,\mathrm{div}\,\mathbf{u} + \mathbf{curl}_{2D}\,\mathrm{curl}_{2D}\,\mathbf{u} = \mathbf{f} \quad \text{in} \quad \Omega\,, \\
\Updownarrow \\
\begin{bmatrix} -\frac{\partial}{\partial x_1}\left(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2}\right) \\ \frac{\partial}{\partial x_2}\left(\frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2}\right) \end{bmatrix} + \begin{bmatrix} \frac{\partial}{\partial x_2}\left(\frac{\partial u_2}{\partial x_1} - \frac{\partial u_1}{\partial x_2}\right) \\ -\frac{\partial}{\partial x_1}\left(\frac{\partial u_2}{\partial x_1} - \frac{\partial u_1}{\partial x_2}\right) \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}, \\
\Updownarrow \\
\begin{cases} -\dfrac{\partial^2 u_1}{\partial {x_1}^2} - \dfrac{\partial^2 u_1}{\partial {x_2}^2} = f_1 \\ -\dfrac{\partial^2 u_2}{\partial {x_1}^2} - \dfrac{\partial^2 u_2}{\partial {x_2}^2} = f_2 \end{cases} \quad \text{in} \quad \Omega\,,
\end{gathered}$$

because the partial derivatives $\frac{\partial}{\partial x_1}$ and $\frac{\partial}{\partial x_2}$ commute.

HINT 1 for (13-3.a): From [Lecture → § 13.1.4.49] recall the scalar and vectorial 2D rotation differential operators,

$$\operatorname{curl}_{2D} \mathbf{v}(\boldsymbol{x}) := \frac{\partial v_2}{\partial x_1}(\boldsymbol{x}) - \frac{\partial v_1}{\partial x_2}(\boldsymbol{x}) \quad , \quad \mathbf{curl}_{2D}\, w(\boldsymbol{x}) := \begin{bmatrix} \frac{\partial w}{\partial x_2}(\boldsymbol{x}) \\ -\frac{\partial w}{\partial x_1}(\boldsymbol{x}) \end{bmatrix} ,$$

and the two integration-by-parts formulas

$$\int_\Omega \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, w(\boldsymbol{x}) + \operatorname{div} \mathbf{v}(\boldsymbol{x})\, w(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} w(\boldsymbol{x})\, \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \tag{13.3.3}$$

and

$$\int_\Omega \operatorname{curl}_{2D} \mathbf{v}(\boldsymbol{x})\, w(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} - \int_\Omega \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{curl}_{2D}\, w(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{t}(\boldsymbol{x})\, w(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) = \int_{\partial\Omega} w(\boldsymbol{x}) \mathbf{v}(\boldsymbol{x}) \cdot \mathrm{d}\vec{\boldsymbol{s}} \tag{13.3.4}$$

for differentiable $\mathbf{v} : \overline{\Omega} \to \mathbb{R}^2$, $w : \overline{\Omega} \to \mathbb{R}$. Here $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ is the exterior unit normal vector field, and $\boldsymbol{t} := \boldsymbol{n}^\perp$ is a unit tangential vector field, directed counterclockwise. ⌟

SOLUTION of (13-3.a):

We face zero **esssential boundary conditions** [Lecture → Section 1.9]. Both spaces $\boldsymbol{H}_0(\mathrm{curl}_{2D},\Omega)$ and $H_0^1(\Omega)$ include zero (tangential) boundary conditions:

$$\mathbf{a}\cdot\boldsymbol{t}=0 \quad \text{on} \quad \partial\Omega \quad , \quad p=0 \quad \text{on} \quad \partial\Omega \ .$$

To find the PDEs implied by (13.3.1) we move all derivatives off the test functions by means of integration by parts.

We apply (13.3.4) to the first equation and get

$$\int_\Omega \mathbf{curl}_{2D}\left(\frac{1}{\mu(\boldsymbol{x})}\,\mathrm{curl}_{2D}\,\mathbf{a}(\boldsymbol{x})+\mathbf{grad}\,p(\boldsymbol{x})\right)\cdot\mathbf{a}'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}=\int_\Omega \mathbf{j}(\boldsymbol{x})\cdot\mathbf{a}'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall\mathbf{a}'\in(C_0^\infty(\Omega))^2 \ .$$

By the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4] we conclude that

$$\mathbf{curl}_{2D}\left(\frac{1}{\mu(\boldsymbol{x})}\,\mathrm{curl}_{2D}\,\mathbf{a}(\boldsymbol{x})\right)+\mathbf{grad}\,p(\boldsymbol{x})=\mathbf{j}(\boldsymbol{x}) \quad \text{in} \quad \Omega \ .$$

We tackle the second equation with (13.3.3) and end up with

$$-\int_\Omega \mathrm{div}\,\mathbf{a}(\boldsymbol{x})\,p'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}=0 \quad \forall p'\in C_0^\infty(\Omega) \ ,$$

which leads to the PDE

$$\mathrm{div}\,\mathbf{a}(\boldsymbol{x})=0 \quad \text{in} \quad \Omega \ .$$

HINT 1 for (13-3.b): The inclusion [Lecture → (13.1.5.17)] plus $\mathrm{d}_{\ell+1} \circ \mathrm{d}_\ell = 0$ means that in terms of 2D Euclidean vector proxies (Option I)

$$\mathbf{grad}\, H_0^1(\Omega) \subset \{\mathbf{v} \in \boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega) : \mathrm{curl}_{2D}\, \mathbf{v} = 0\} \,. \tag{13.3.5}$$

⌟

SOLUTION of (13-3.b):

The key facts, already given as a hint, are

1. $\mathbf{grad}\, H_0^1(\Omega) \subset \boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega)$, and
2. $\mathrm{curl}_{2D} \circ \mathbf{grad} = 0$.

Hence, from the first equation of (13.3.1) when tested with $\mathbf{grad}\, v$, $v \in H_0^1(\Omega)$, we infer

$$\int_\Omega \mathbf{grad}\, p(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{j}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,, \tag{13.3.6}$$

if $p \in H_0^1(\Omega)$ is a component of a solution of (13.3.1).

The variational problem (13.3.6) is a standard 2nd-order elliptic variational problem. Thanks to the first Poincaré-Friedrichs inequality [Lecture → Thm. 1.3.4.17] it has a unique solution and we conclude

$$p = 0 \quad \Longleftrightarrow \quad \int_\Omega \mathbf{j}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H_0^1(\Omega)\,,$$

which means

$$Z = \left\{ \mathbf{j} \in \left(L^2(\Omega)\right)^2 : \int_\Omega \mathbf{j}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H_0^1(\Omega) \right\}\,.$$

Remark. The theory of Sobolev spaces confirms that there is an equivalent characterization of Z,

$$Z = \{ \mathbf{j} \in \boldsymbol{H}(\mathrm{div}, \Omega) : \ \mathrm{div}\, \mathbf{j} = 0 \}\,.$$

SOLUTION of (13-3.c):

Key to the solution is (13.3.8), because it permits us to choose

$$\mathbf{v}_h := \mathbf{grad}\, q_h \in \boldsymbol{H}_0(\mathrm{curl}_{2D}, \Omega) .$$

Then, by the very definition of the norms

$$\|\mathbf{v}_h\|_{L^2(\Omega)} = \|\mathbf{grad}\, q_h\|_{L^2(\Omega)} = |q_h|_{H^1(\Omega)} \leq \|q_h\|_{H^1(\Omega)} ,$$

and

$$\frac{\int_\Omega \mathbf{v}_h(\boldsymbol{x}) \cdot \mathbf{grad}\, q_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}{\|\mathbf{v}_h\|_{L^2(\Omega)} + \|\mathrm{curl}_{2D}\, \mathbf{v}_h\|_{L^2(\Omega)}} = \frac{\int_\Omega \mathbf{grad}\, q_h(\boldsymbol{x}) \cdot \mathbf{grad}\, q_h(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}{\|\mathbf{grad}\, q_h\|_{L^2(\Omega)}}$$
$$= \|\mathbf{grad}\, q_h\|_{L^2(\Omega)} \geq \frac{1}{c}\|q_h\|_{H^1(\Omega)} \quad \forall q_h \in \mathcal{S}^0_{1,0}(\mathcal{M}) ,$$

due to

Theorem [Lecture → Thm. 1.3.4.17]. Poincaré-Friedrichs inequality

If $\Omega \subset \mathbb{R}^d$, $d \in \mathbb{N}$, *is* bounded, *then*

$$\|u\|_0 \leq \mathrm{diam}(\Omega)\, \|\mathbf{grad}\, u\|_0 \quad \forall u \in H^1_0(\Omega) .$$

SOLUTION of (13-3.d):

Note that in (13.3.9) equations and unknowns have been reordered compared to (13.3.1):

- $\mathbf{A}$ is the $\mathcal{W}_0^1(\mathcal{M})$ Galerkin matrix related to $(\mathbf{a}, \mathbf{a}') \mapsto \int_\Omega \mu(\boldsymbol{x})^{-1} \operatorname{curl}_{2D} \mathbf{a}(\boldsymbol{x}) \operatorname{curl}_{2D} \mathbf{a}'(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$, and
- $\mathbf{B}$ arises from the Galerkin discretization of $(\mathbf{a}, p') \mapsto \int_\Omega \mathbf{a}(\boldsymbol{x}) \cdot \mathbf{grad}\, p'(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$ based on $\mathcal{W}_0^1(\mathcal{M}) \times$

The corresponding matrix entries are obtained by inserting pairs of suitable global shape functions into the bilinear forms:

$$(\mathbf{A})_{i,j} = \int_\Omega \frac{1}{\mu(\boldsymbol{x})} \operatorname{curl}_{2D} \boldsymbol{\beta}_j^1(\boldsymbol{x}) \operatorname{curl}_{2D} \boldsymbol{\beta}_i^1(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \,, \quad i = 1, \ldots, N_1, \; j = 1, \ldots, N_1 \,,$$

$$(\mathbf{B})_{i,j} = \int_\Omega \boldsymbol{\beta}_j^1(\boldsymbol{x}) \cdot \mathbf{grad}\, \beta_i^0(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \,, \qquad i = 1, \ldots, N_0, \; j = 1, \ldots, N_1 \,.$$

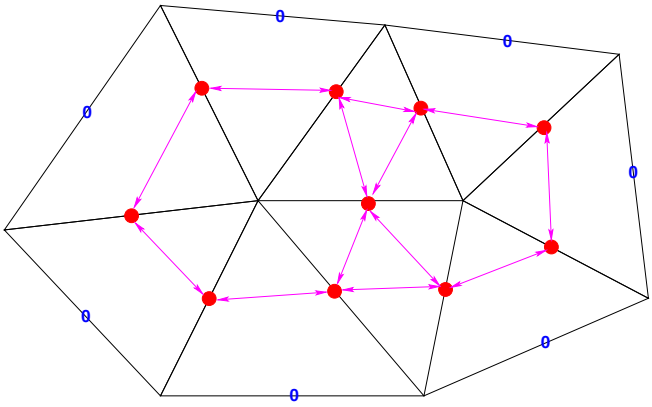
Note that $\dim \mathcal{S}_{1,0}^0(\mathcal{M}) = N_0$ and $\dim \mathcal{W}_0^1(\mathcal{M}) = N_1$.

SOLUTION of (13-3.e):

Since $\mathbf{A}$ is the $\mathcal{W}_0^1(\mathcal{M})$ Galerkin matrix for $(\mathbf{a},\mathbf{a}') \mapsto \int_\Omega \mu(\boldsymbol{x})^{-1}\,\mathrm{curl}_{2D}\,\mathbf{a}(\boldsymbol{x})\;\mathrm{curl}_{2D}\,\mathbf{a}'(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ we know

$$\begin{aligned}(\mathbf{A})_{i,j} \neq 0 \quad &\Rightarrow \quad \text{the supports of } \boldsymbol{\beta}_i^1 \text{ and } \boldsymbol{\beta}_j^1 \text{ overlap}\\ &\Rightarrow \quad i\text{-th and } j\text{-th edge share a triangle.}\end{aligned}$$

This is consequence of the fact that the global shape functions of $\mathcal{W}_0^1(\mathcal{M})$ are associated with interior edges of $\mathcal{M}$ and that the support of $\boldsymbol{\beta}_i^1$ is the union of the triangles adjacent to the i-th edge.



Thus, every triangle can contribute three pairs of potentially non-zero off-diagonal entries of $\mathbf{A}$. Triangles with one edge on the boundary $\partial\Omega$ contribute only a single pair. However, for want of information we cannot take this into account.

◁ Arrows $\leftrightarrow$ represent pairs of non-zero off-diagonal entries of $\mathbf{A}$ for a simple mesh.

Each interior edge corresponds to a necessarily non-zero diagonal entry of $\mathbf{A}$

$$\mathrm{nnz}(\mathbf{A}) \leq N_1 + 6N_2 \ .$$

SOLUTION of (13-3.f):

We have

- a single global shape function (of $\mathcal{W}^1(\mathcal{M})$) associated with each edge of $\mathcal{M}$,
- a single global shape function (of $\mathcal{S}_1^0(\mathcal{M}) = \mathcal{W}^0(\mathcal{M})$) associated with each node of $\mathcal{M}$, and
- no global shape functions associated with cells.

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(), 1 },
  {lf::base::RefEl::kSegment(), 1 },
  {lf::base::RefEl::kTria(), 0 },
  {lf::base::RefEl::kQuad(), 0 /*arbitrary*/ }});
```

Since quadrilateral cells are not allowed, it does not matter what number is put in the bottom box.

SOLUTION of (13-3.h):

For the treatment of essential boundary conditions in LEHRFEM++ consult [Lecture → Section 2.7.6], [Lecture → Code 2.7.6.20].

The array `ess_dof_select` has to have length = `dofh.NumDofs()` and for d.o.f. with global index i has to contain `{true, 0.0}`, whenever the corresponding global shape function is associated with a mesh entity on $\partial\Omega$. Otherwise the entry should be n `{false, 0.0}` ➙ (documentation).

C++ code 13.3.13: FEM solver for (13.3.1)

```
template <lf::mesh::utils::MeshFunction MESH_FUNCTION_J,
          lf::mesh::utils::MeshFunction MESH_FUNCTION_MU>
Eigen::VectorXd solveMagStat2DBVP(const lf::assemble::DofHandler &dofh,
                                  MESH_FUNCTION_MU mu,
                                  MESH_FUNCTION_J j_source) {
  // Size of linear system
  const size_t N_dofs = dofh.NumDofs();
  // Galerkin matrix in COO format
  lf::assemble::COOMatrix<double> M_COO{
      buildMagStat2DGalerkinMatrix<MESH_FUNCTION_MU>(dofh, mu)};
  // Right-hand side vector
  Eigen::VectorXd phi{
      computeMagStat2DRhsVector<MESH_FUNCTION_J>(dofh, j_source)};
  // Enforce zero essential boundary conditions
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh.Mesh();
  // Set boundary flags for both mesh nodes and edges
  lf::mesh::utils::AllCodimMeshDataSet<bool> bd_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(mesh_p)};
  // Flag all d.o.f.s associated with mesh entities on the boundary
  std::vector<std::pair<bool, double>> ess_dof_select{};
  for (lf::assemble::gdof_idx_t dofnum = 0; dofnum < N_dofs; ++dofnum) {
    const lf::mesh::Entity &entity{dofh.Entity(dofnum)};
    if (bd_flags(entity)) {
      // Entity carrying a d.o.f. is located on the boundary.
      // The value of the d.o.f. will be set to zero
      ess_dof_select.emplace_back(true, 0.0);
    } else {
      // Interior node: d.o.f. remains active
      ess_dof_select.emplace_back(false, 0.0);
    }
  }
  // Modify linear system of equations enforcing zero value for d.o.f.s
  // belonging to mesh entities on the boundary
  lf::assemble::FixFlaggedSolutionCompAlt(
      [&ess_dof_select](lf::assemble::glb_idx_t dof_idx)
          -> std::pair<bool, double> { return ess_dof_select[dof_idx]; },
      M_COO, phi);

  // Solve linear system using Eigen's sparse direct elimination
  // Examine return status of solver in case the matrix is singular
  const Eigen::SparseMatrix<double> M_crs = M_COO.makeSparse();
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(M_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  const Eigen::VectorXd dofvec = solver.solve(phi);
```

```
47    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
48    return dofvec;
49  }
```

SOLUTION of (1-1.a):

The trick is to apply the binomial formula $(a+b)^2 = a^2 + 2ab + b^2$:

$$J(v) = \int_{-\pi}^{\pi} x^2 \left(v^2(x) + 2v(x) + 1\right) \mathrm{d}x = \int_{-\pi}^{\pi} x^2 v(x)^2 \,\mathrm{d}x + \int_{-\pi}^{\pi} 2x^2 v(x) \,\mathrm{d}x + \int_{-\pi}^{\pi} x^2 \,\mathrm{d}x \,. \tag{1.1.3}$$

(i) Yes, J is a quadratic functionals, with

(ii)

$$\mathsf{a}(u,v) = 2 \int_{-\pi}^{\pi} x^2 u(x)\, v(x) \,\mathrm{d}x \,, \quad u, v \in \mathrm{C}^0_{\mathrm{pw}}([-\pi,\pi]) \,,$$

$$\ell(v) = -2 \int_{-\pi}^{\pi} x^2 v(x) \,\mathrm{d}x \,,$$

which emerge from the different terms in (1.1.3).

(iii) The bilinear form a is clearly positive semi-definite, because squares cannot be negative:

$$\mathsf{a}(v,v) = 2 \int_{-\pi}^{\pi} x^2 |v(x)|^2 \,\mathrm{d}x \geq 0 \,.$$

If $\mathsf{a}(v,v) = 0$, then the piecewise continuous function v must vanish everywhere, because $x^2 > 0$ *almost everywhere* in $]-\pi,\pi[$.

HINT 1 for (1-1.b): Note that in this case $J(\lambda v) = \lambda^2 J(v)$ for all $\lambda \in \mathbb{R}$, which means that $J(v) = \frac{1}{2}\mathsf{a}(v, v$ with a suitable *symmetric* bilinear form a. Then try to guess a. ⌟

HINT 2 for (1-1.b): If you cannot "guess" the building blocks of J, you may peek at Sub-problem (1-1.g) and use Lemma 1.1.10 to find the underlying bilinear form. ⌟

HINT 3 for (1-1.b): In this case the bilinear form a defining J is **not** positive semi-definite. Prove this by finding $v \in C^2([-1,1])$ with $J(v) < 0$. ⌟

SOLUTION of (1-1.b):

(i) This is a quadratic functional on $V_0 = C^2([-1,1])$ with

(ii)

$$\mathsf{a}(u,v) = \int_{-1}^{1} \frac{d^2u}{dx^2}(x)\,\frac{dv}{dx}(x) + \frac{d^2v}{dx^2}(x)\,\frac{du}{dx}(x)\,\mathrm{d}x\,,$$
$$\ell(v) = 0\,.$$

Note that the bilinear form is supposed to be *symmetric*.

(iii) Set $v(x) := \frac{1}{2}(x \pm 1)^2$ with $v'(x) = x - 1$, $v''(x) = 1$, and observe that in this case $\mathsf{a}(v,v) = \pm 2$. Hence a has "no sign", it is definitely not positive semi-definite.

HINT 1 for (1-1.c): As in Sub-problem (1-1.b) we face a functional for which $J(\lambda v) = \lambda^2 J(v)$ for all $\lambda \in \mathbb{R}$, $v \in \mathbb{R}^n$. Hence, J will be of the form $J(v) = \frac{1}{2}\mathsf{a}(v, v)$ with a symmetric bilinear form a on $\mathbb{R}^n$. ⌟

SOLUTION of (1-1.c):

(i) Yes, J is a quadratic functional and we find, with a notation analogous to the one introduced above

(ii)

$$\mathsf{a}(\boldsymbol{u},\boldsymbol{v}) = 2\sum_{j=1}^{n}(u_j - u_{j-1})(v_j - v_{j-1}) \;,$$
$$\ell(\boldsymbol{v}) = 0 \;.$$

(iii) The bilinear form a is positive definite, because $\mathsf{a}(\boldsymbol{v},\boldsymbol{v})$ is a sum of squares, and

$$\mathsf{a}(\boldsymbol{v},\boldsymbol{v}) = 2\sum_{j=1}^{n}|v_j - v_{j-1}|^2 = 0 \quad \Rightarrow \quad v_j = v_{j-1} \quad \forall j = 1,\ldots,n \;.$$

As we imposed $v_0 = 0$, we immediately conclude $\boldsymbol{v} = \mathbf{0}$.

To find the representation (1.1.6) we rewrite

$$J(\boldsymbol{v}) = v_n^2 + 2\sum_{j=1}^{n-1} v_j^2 - \sum_{j=1}^{n-1} v_{j+1}v_j - \sum_{j=2}^{n} v_j v_{j-1} \;.$$

Hence we have by the elementary rules of matrix$\times$vector multiplication

$$J(\boldsymbol{v}) = \boldsymbol{v}^\top \begin{bmatrix} 2 & -1 & & & & & \\ -1 & 2 & -1 & & & & \\ & \ddots & \ddots & \ddots & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & -1 & 2 & -1 \\ & & & & & -1 & 1 \end{bmatrix} \boldsymbol{v} = \tfrac{1}{2}\boldsymbol{v}^\top \mathbf{A}\boldsymbol{v} \;.$$

Of course $\ell \equiv 0$ means $\boldsymbol{b} = \mathbf{0}$.

HINT 1 for (1-1.d): Note the structural similarity of the functionals from (1.1.2) and (1.1.7). The binomial formula is useful again. ⌟

SOLUTION of (1-1.d):

(i) Yes, J is a quadratic functional with

(ii)

$$\mathsf{a}(u,v) = 2\int_\Omega (\mathbf{f}(\boldsymbol{x})\cdot\mathbf{grad}\,u)(\mathbf{f}(\boldsymbol{x})\cdot\mathbf{grad}\,v)\,\mathrm{d}\boldsymbol{x}\,,$$
$$\ell(v) = -2\int_\Omega \mathbf{f}(\boldsymbol{x})\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}\,,$$

as one sees by applying the binomial formula.

(iii) From

$$\mathsf{a}(v,v) = 2\int_\Omega |\mathbf{f}(\boldsymbol{x})\cdot\mathbf{grad}\,v|^2\,\mathrm{d}\boldsymbol{x}$$

we see that $\mathsf{a}(v,v)\geq 0$, that is, a is positiv semi-definite, but $\mathsf{a}(v,v)=0$ for $v\equiv 1$, and, therefore, a is not positive definite.

HINT 1 for (1-1.e): If $v \mapsto J(v)$ is a quadratic functional then, for every *fixed* v the function $\lambda \in \mathbb{R} \mapsto J(\lambda$ is a polynomial of degree 2 in λ. ⌟

SOLUTION of (1-1.e):

J is not a quadratic functional, because $J(\lambda v) = \lambda^4 J(v)$, for any v, which can never occur with a quadratic functional.

HINT 1 for (1-1.f): Of course, the first step is to apply the binomial formula. ⌟

SOLUTION of (1-1.f):

(i) This J is a quadratic functional with

(ii)

$$\mathsf{a}(u,v) = 2\left(\int_\Omega u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right)\left(\int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right),$$
$$\ell(v) = -2\int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\int_\Omega 1\,\mathrm{d}\boldsymbol{x}\,.$$

(iii) $\mathsf{a}(v,v) \geq 0$ is immediate, but $\mathsf{a}(v,v) = 0$ for any function v with vanishing mean over Ω. Hence, a is positive semi-definite, but not positive definite.

SOLUTION of (1-1.g):

To prove Lemma 1.1.10 use the bi-linearity and symmetry of a and the linearity of ℓ in a straightforward manner:

$$\begin{aligned} J(u+v) - J(u) - J(v) + J(\mathbf{0}) \\ &= \tfrac{1}{2}\mathsf{a}(u+v,u+v) - \ell(u+v) + c - \tfrac{1}{2}\mathsf{a}(u,u) + \ell(u) - c - \tfrac{1}{2}\mathsf{a}(v,v) + \ell(v) - c + c \\ &\overset{①}{=} \tfrac{1}{2}\mathsf{a}(u,v) + \tfrac{1}{2}\mathsf{a}(v,u) = \mathsf{a}(u,v) \,, \end{aligned}$$

where, in step ①, we use the bilinearity of a:

$$\mathsf{a}(u+v,u+v) = \mathsf{a}(u,u) + \mathsf{a}(u,v) + \mathsf{a}(v,u) + \mathsf{a}(v,v) \,,$$

and the linearity of ℓ:

$$\ell(u+v) = \ell(u) + \ell(v) \,.$$

The last step exploits the symmetry of a.

HINT 1 for (1-10.): For all the sub-problems multi-dimensional integration by parts will play a key role. More precisely, you will have to invoke Green's first formula for $d = 2$.

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad \text{[Lecture → Eq. (13.2.3.2)]}$$

SOLUTION of (1-10.a):

It is important to notice that the PDE in (1.10.1) can be expressed using the differential operators [Lecture → § 0.3.2.19]

$$\operatorname{div} \mathbf{j}(\boldsymbol{x}) := \frac{\partial j_1}{\partial x_1}(\boldsymbol{x}) + \frac{\partial j_2}{\partial x_2}(\boldsymbol{x}) \quad \text{and} \quad \mathbf{grad}\, v(\boldsymbol{x}) = \begin{bmatrix} \frac{\partial v}{\partial x_1}(\boldsymbol{x}) \\ \frac{\partial v}{\partial x_2}(\boldsymbol{x}) \end{bmatrix} .$$

We get

$$\frac{\partial}{\partial x_1}\left(e^{x_2} \frac{\partial u}{\partial x_1}(\boldsymbol{x}) \right) + \frac{\partial}{\partial x_2}\left(e^{x_1} \frac{\partial u}{\partial x_2}(\boldsymbol{x}) \right) = \operatorname{div}\left(\begin{bmatrix} e^{x_2} & 0 \\ 0 & e^{x_1} \end{bmatrix} \mathbf{grad}\, u \right)(\boldsymbol{x}) .$$

Then we can apply the "algorithm" of [Lecture → Section 1.8] as in [Lecture → Ex. 1.8.0.2]. We test with $v \in C_0^\infty(\Omega)$ and integrate, which yields

$$\int_\Omega \operatorname{div}\left(\begin{bmatrix} e^{x_2} & 0 \\ 0 & e^{x_1} \end{bmatrix} \mathbf{grad}\, u \right)(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega (x_1^2 + x_2^2) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C_0^\infty(\Omega) .$$

We test with functions vanishing on $\partial\Omega$, because the Dirichlet boundary conditions already fix u there.

Then we integrate by parts by means of [Lecture → Thm. 1.5.2.7] with $\mathbf{j} := \left[\begin{smallmatrix} e^{x_2} & 0 \\ 0 & e^{x_1} \end{smallmatrix} \right] \mathbf{grad}\, u$ and arrive at

$$-\int_\Omega \begin{bmatrix} e^{x_2} & 0 \\ 0 & e^{x_1} \end{bmatrix} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega (x_1^2 + x_2^2) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C_0^\infty(\Omega) . \tag{1.10.2}$$

Since the matrix function $\boldsymbol{x} \mapsto \left[\begin{smallmatrix} e^{x_2} & 0 \\ 0 & e^{x_1} \end{smallmatrix} \right]$ is uniformly positive definite, the energy norm induced by the bilinear form of (1.10.2) is equivalent to $|\cdot|_{H^1(\Omega)}$. Moreover, the homogeneous Dirichlet boundary conditions are **essential** [Lecture → Section 1.9] and will be imposed on both trial and test space. Thus $H_0^1(\Omega)$ is the right Sobolev space and the final variational problem reads

$$u \in H_0^1(\Omega): \quad -\int_\Omega \begin{bmatrix} e^{x_2} & 0 \\ 0 & e^{x_1} \end{bmatrix} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega (x_1^2 + x_2^2) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) .$$

Equivalently, this can be written as

$$u \in H_0^1(\Omega): \quad -\int_\Omega e^{x_2} \frac{\partial u}{\partial x_1}(\boldsymbol{x}) \frac{\partial v}{\partial x_1}(\boldsymbol{x}) + e^{x_1} \frac{\partial u}{\partial x_2}(\boldsymbol{x}) \frac{\partial v}{\partial x_2}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega \|\boldsymbol{x}\|^2 v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) .$$

SOLUTION of (1-10.b):

The boundary value problem (1.10.3) agrees with that considered in [Lecture → Ex. 1.8.0.6] for $f \equiv 1$ and $\Psi(u) = u$.

Since u is not known on $\partial\Omega$ we have to test the PDE of (1.10.3) with $v \in C^\infty(\Omega)$ and get

$$\int_\Omega (-\Delta u)(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^\infty(\Omega) \,.$$

Next we carry out integration by parts according to [Lecture → Thm. 1.5.2.7], which yields

$$\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^\infty(\Omega) \,.$$

Finally, we recast the boundary term using the boundary conditions and end up with

$$\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^\infty(\Omega) \,.$$

Obviously, this variational problem is naturally posed in the "energy space" $H^1(\Omega)$.

$$u \in H^1(\Omega): \quad \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \,.$$

SOLUTION of (1-10.c):

The PDE in (1.10.4) expresses an equation of vector fields $\Omega \mapsto \mathbb{R}^2$. Therefore, it has to be tested with a vector field $\mathbf{v} \in C^\infty(\Omega) := (C^\infty(\Omega))^2$. Note that $\mathbf{u}$ is not immediately known on $\partial\Omega$ so that we test with functions that do not vanish on (any part of) the boundary $\partial\Omega$. After integrating over Ω we get

$$\int_\Omega (-\mathbf{grad}(\alpha(\boldsymbol{x}) \operatorname{div} \mathbf{u}(\boldsymbol{x})) + \mathbf{u}(\boldsymbol{x}))\mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall \mathbf{v} \in C^\infty(\Omega) \,.$$

Next, we resort to integration by parts using [Lecture → Thm. 1.5.2.7] in "reverse mode" with $\mathbf{j} := \mathbf{v}$ and $v := \alpha(\boldsymbol{x}) \operatorname{div} \mathbf{u}$:

$$\int_\Omega \alpha(\boldsymbol{x}) \operatorname{div}(\mathbf{u}(\boldsymbol{x})) \operatorname{div} \mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x})\mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \alpha(\boldsymbol{x}) \operatorname{div} \mathbf{u}(\boldsymbol{x})(\mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})) \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall \mathbf{v} \in C^\infty(\Omega) \,.$$

We use the boundary conditions from (1.10.4) to rephrase the boundary term and move it to the right-hand side.

$$\underbrace{\int_\Omega \alpha(\boldsymbol{x}) \operatorname{div}(\mathbf{u}(\boldsymbol{x})) \operatorname{div} \mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x})\mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(\mathbf{u},\mathbf{v})} = \int_{\partial\Omega} 1 \,(\mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})) \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{v} \in C^\infty(\Omega) \,. \tag{1.10.5}$$

The energy norm spawned by $\mathsf{a}(\mathbf{u}, \mathbf{v})$ is

$$\|\mathbf{v}\|_{\mathsf{a}}^2 := \int_\Omega |\operatorname{div} \mathbf{v}(\boldsymbol{x})|^2 + \|\mathbf{v}(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \,.$$

The right Sobolev space is the vector space of vector fields with bounded energy norm

$$V = \left\{ \mathbf{v} : \Omega \to \mathbb{R}^2 : \int_\Omega |\operatorname{div} \mathbf{v}(\boldsymbol{x})|^2 + \|\mathbf{v}(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} < \infty \right\} \,.$$

The final variational formulations is posed on V:

$$\mathbf{u} \in V: \quad \int_\Omega \alpha(\boldsymbol{x}) \operatorname{div}(\mathbf{u}(\boldsymbol{x})) \operatorname{div} \mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x})\mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{v} \in V \,.$$

SOLUTION of (1-11.a):

In order to be able to apply the standard approach of [Lecture → Section 1.8], it is useful to rewrite the left-hand-side of the PDE:

$$\frac{\partial^2 u}{\partial {x_1}^2} + \frac{\partial^2 u}{\partial x_1 \partial x_2} + \frac{\partial^2 u}{\partial {x_2}^2} = \operatorname{div}\left(\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix} \mathbf{grad}\, u \right),$$

which is possible,because of the well-known fact from analysis $\frac{\partial^2 u}{\partial x_1 \partial x_2} = \frac{\partial^2 u}{\partial x_2 \partial x_1}$. So we can equivalently state the BVP as

$$\begin{aligned} -\operatorname{div}(\boldsymbol{\alpha}\, \mathbf{grad}\, u(\boldsymbol{x})) + \|\boldsymbol{x}\|^2 u(\boldsymbol{x}) = 0 = 0 \quad &\text{in} \quad \Omega\,, \\ u = 1 \quad &\text{on} \quad \partial\Omega\,, \end{aligned} \qquad \boldsymbol{\alpha} := \begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{bmatrix}.$$

From now we can simply apply the steps detailed in [Lecture → Ex. 1.8.0.2] and we arrive at a formula analogous to [Lecture → Eq. (1.8.0.5)]:

$$u \in H^1(\Omega),\ u = 1 \text{ on } \partial\Omega \quad : \quad \int_\Omega \boldsymbol{\alpha}\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \|\mathbf{x}\|^2 u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall\, v \in H^1_0(\Omega)\,.$$

Note. The coefficient matrix $\boldsymbol{\alpha}$ need not be symmetric. Correct, though peculiar, solutions would be, for instance

$$\boldsymbol{\alpha} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \quad \text{or} \quad \boldsymbol{\alpha} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix},$$

because partial derivatives commute. Many other solutions are possible.

SOLUTION of (1-11.b):

We follow the considerations of [Lecture → Section 1.5.3] and assume $u \in H^2(\Omega)$. First, we rewrite the variational problem as

$$u \in H^1(\Omega): \quad \int_\Omega \begin{bmatrix} 1 & 0 \\ 0 & \alpha(\boldsymbol{x}) \end{bmatrix} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \alpha(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \,.$$

The we remove the **grad** from the test function using Green's formula [Lecture → Thm. 1.5.2.7], which yields

$$\int_\Omega -\operatorname{div}\left(\begin{bmatrix} 1 & 0 \\ 0 & \alpha(\boldsymbol{x}) \end{bmatrix} \mathbf{grad}\, u(\boldsymbol{x}) \right) v(\boldsymbol{x}) + \alpha(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \begin{bmatrix} 1 & 0 \\ 0 & \alpha(\boldsymbol{x}) \end{bmatrix} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x})$$
$$= \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)$$

Testing with $v \in C_0^\infty(\Omega)$ (compactly supoorted inside Ω!), we can tease out the PDE:

$$-\operatorname{div}\left(\begin{bmatrix} 1 & 0 \\ 0 & \alpha(\boldsymbol{x}) \end{bmatrix} \mathbf{grad}\, u(\boldsymbol{x}) \right) + \alpha(\boldsymbol{x}) u(\boldsymbol{x}) = 1 \quad \text{in} \quad \Omega \,.$$

Then, testing with generic $v \in C^\infty(\overline{\Omega})$ we find the homogeneous Neumann boundary condition:

$$\begin{bmatrix} 1 & 0 \\ 0 & \alpha \end{bmatrix} \mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega \,.$$

In both cases we have tacitly used the appropriate version of the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4] and [Lecture → Lemma 1.5.3.16], respectively.

SOLUTION of (1-11.c):

By the very definition of $\|\cdot\|_{H^1(\Omega)}$,

$$\|v\|^2_{H^1(\Omega)} := \int_\Omega \|\mathbf{grad}\, v\|^2(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega |v(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\,,$$

we have

$$\mathsf{a}(\boldsymbol{\alpha};v,v) = \int_\Omega |\frac{\partial v}{\partial x_1}|^2 + \alpha(\boldsymbol{x})|\frac{\partial v}{\partial x_2}|^2 + \alpha(\boldsymbol{x})|v|^2\,\mathrm{d}\boldsymbol{x} \geq \min\{1, \min_{x\in\Omega}\alpha(\boldsymbol{x})\}\|v\|^2_{H^1(\Omega)}\,.$$

This suggests

$$\mathcal{A} = \{\alpha \in C^0(\overline{\Omega}):\ \exists\alpha_0 > 0:\ \alpha(\boldsymbol{x}) \geq \alpha_0\ \forall \boldsymbol{x} \in \overline{\Omega}\}\ .$$

Using the terminology introduced in [Lecture → § 1.2.2.7] we may also write

$$\mathcal{A} = \{\alpha \in C^0(\overline{\Omega}):\ \alpha \textbf{ uniformly positive}\}\ .$$

If α is not uniformly positive, then we can pick a function $v \in H^1(\Omega)$ supported in a neighborhood of $\{\boldsymbol{x} \in \Omega:\ \alpha(\boldsymbol{x}) \leq 0\}$ to render the quotient $\mathsf{a}(\boldsymbol{\alpha};v,v)/\|v\|_{H^1(\Omega)}$ arbitrarily small or even negative.

Remark. Note that $\alpha \in C^0(\overline{\Omega})$. Hence $\alpha(\boldsymbol{x}) > 0$ for all $\boldsymbol{x} \in \overline{\Omega}$ already implies

$$\exists\alpha_0 > 0:\ \alpha(\boldsymbol{x}) \geq \alpha_0\ \forall \boldsymbol{x} \in \overline{\Omega}\,,$$

because a continuous function attains its minimum on a closed set. Thus, another correct answer is

$$\mathcal{A} = \{\alpha \in C^0(\overline{\Omega}):\ \alpha(\boldsymbol{x}) > 0 \quad \forall \boldsymbol{x} \in \overline{\Omega}\}\ .$$

HINT 1 for (1-2.): Since the domain Ω is a disk, the use of polar coordinates (r, φ) sometimes facilitates the evaluation of integrals, see [Lecture $\rightarrow$ Eq. (1.2.3.47)]. ⌟

HINT 1 for (1-2.a): The occurrence of derivatives in the definition of ℓ_1 suggests that ℓ_1 is **not** continuous on $L^2(\Omega)$, because there are no derivatives in the $L^2(\Omega)$-norm. ⌟

SOLUTION of (1-2.a):

We claim that the functional ℓ_1 is *not* continuous on $L^2(\Omega)$.

We follow the recommended policy and try to find $v \in L^2(\Omega)$ for which the integral defining ℓ_1 diverges. Note that $\mathbf{c} \cdot \mathbf{grad}\, v$ is a directional derivative. Due to rotational symmetry it is enough to consider the special choice $\mathbf{c} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, that is, $\ell_1(v) = \int_\Omega \frac{\partial v}{\partial x_1}\, \mathrm{d}\boldsymbol{x}$ (Choose a coordinate system, in which $\mathbf{c}$ is aligned with the x_1-axis.).

"Counterexample functions" can be constructed in many different ways:

❶ Obviously, only the variation of v in x_1-direction matters, so that we try $v(\boldsymbol{x}) = w(x_1)$ with a function $w : [-1,1] \to \mathbb{R}$, which is to satisfy

$$\|v\|^2_{L^2(\Omega)} = \int_\Omega |v(\boldsymbol{x})|^2\, \mathrm{d}\boldsymbol{x} = 2\int_{-1}^1 \sqrt{1-\xi^2}\, |w(\xi)|^2\, \mathrm{d}\xi < \infty\,, \qquad \ell_1(v) = \int_{-1}^1 \sqrt{1-\xi^2}\, w'(\xi)\, \mathrm{d}\xi = \infty\,. \tag{1.2.6}$$

Let us choose an odd w that, moreover, is smooth away from $\xi = 0$. Thus the conditions of (1.2.6) become

$$\int_0^1 \sqrt{1-\xi^2}\, |w(\xi)|^2\, \mathrm{d}\xi < \infty \quad , \quad \int_0^1 \sqrt{1-\xi^2}\, w'(\xi)\, \mathrm{d}\xi = \infty\,. \tag{1.2.7}$$

We look for a function $]0,1[\to \mathbb{R}$, whose square is integrable, but not so its derivative. We narrow our search to functions with a singularity in $\xi = 0$. A natural candidate is $w(\xi) = \operatorname{sgn}\xi\, \log\xi \triangleq v(\boldsymbol{x}) := \operatorname{sgn} x_1$ because then (1.2.7) becomes

$$\int_0^1 \sqrt{1-\xi^2}\, |\log\xi|^2\, \mathrm{d}\xi < \infty \quad , \quad \int_0^1 \frac{\sqrt{1-\xi^2}}{\xi}\, \mathrm{d}\xi = \infty\,.$$

This is certainly satisfied, since

$$\int_0^1 \log^2\xi\, \mathrm{d}\xi = \int_0^\infty \eta^2 \exp(-\eta)\, \mathrm{d}\eta < \infty\,, \quad \text{"} \int_0^1 \xi^{-1}\, \mathrm{d}\xi = \infty \text{"}\,, \tag{1.2.8}$$

and the uniformly continuous function $\xi \in [0,1] \mapsto \sqrt{1-\xi^2}$ does not affect the (non-)existence of the improper integrals.

❷ One could also attempt to come up with a function that is "very steep close to $\partial\Omega$, leading to a blow-up of $\frac{\partial v}{\partial x_1}$ there. Using polar coordinates, let us try $v(r,\varphi) := -\log(1-r)$, $0 \le r < 1$, $0 \le \varphi < 2\pi$. To begin with, we find

$$\|v\|^2_{L^2(\Omega)} = 2\pi \int_0^1 |log(1-r)|^2\, \mathrm{d}r < \infty\,, \quad \text{see (1.2.8)}\,,$$

which implies $v \in L^2(\Omega)$. Next, we compute

$$\ell_1(v) = \int_\Omega \frac{\partial v}{\partial x_1}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_0^{2\pi}\int_0^1 \frac{\partial v}{\partial r}(r,\varphi) \cos\varphi\, r\, \mathrm{d}r\mathrm{d}\varphi = \int_0^{2\pi} \cos\varphi \left(\int_0^1 \frac{r}{1-r}\, \mathrm{d}r \right) \mathrm{d}\varphi = \infty\,.$$

Thus, the chosen v is a suitable counterexample function.

HINT 1 for (1-2.b): Recall the **Cauchy-Schwarz inequality** [Lecture → Eq. (1.3.4.15)] in $L^2(\Omega)$. ⌟

SOLUTION of (1-2.b):

The functional ℓ_2 is continuous on $L^2(\Omega)$. Indeed, using the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)], we obtain:

$$\left| \int_\Omega v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \right| \leq |\Omega|^{\frac{1}{2}} \, \|v\|_{L^2(\Omega)} \quad \text{for all } v \in L^2(\Omega) \, .$$

HINT 1 for (1-2.c): Compute $\|v\|_{L^2(\Omega)}$ and $\ell_3(v)$ for $v(\boldsymbol{x}) := \log(1 - \|\boldsymbol{x}\|)$ and heed Rem. 1.2.5. ⌟

SOLUTION of (1-2.c):

The functional ℓ_3 is *not* continuous on $L^2(\Omega)$.

We follow the policy outlined above, seeking a function in $L^2(\Omega)$ whose gradient blows up for $\|\boldsymbol{x}\| \to 1$. Since the domain is the unit disk, it is natural to look for a function of $\|\boldsymbol{x}\|$ alone. We try the function $v(\boldsymbol{x}) = \log(1 - \|\boldsymbol{x}\|)$, which blows up for $\|\boldsymbol{x}\| \to 1$. Since $\int_0^1 \log^2 \xi \, \mathrm{d}\xi < \infty$, we have (integration in polar coordinates, see [Lecture → Eq. (1.2.3.47)])

$$\|v\|^2_{L^2(\Omega)} = \int_\Omega v^2(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} = 2\pi \int_0^1 \log^2(1-r) r \mathrm{d}r \leq 2\pi \int_0^1 \log^2(r) \, \mathrm{d}r < +\infty,$$

and thus $v \in L^2(\Omega)$. However,

$$\mathbf{grad}\, v(\boldsymbol{x}) = -\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|(1-\|\boldsymbol{x}\|)} \,, \quad \boldsymbol{x} \in \Omega \,,$$

and, formally,

$$\int_{\partial\Omega} \mathbf{grad}\, v(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \mathrm{d}S(\boldsymbol{x}) = -\int_{\partial\Omega} \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|(1-\|\boldsymbol{x}\|)} \cdot \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|} \mathrm{d}S(\boldsymbol{x}) = -\int_{\partial\Omega} \frac{1}{1-\|\boldsymbol{x}\|} = -\infty \,.$$

HINT 1 for (1-2.d): The functional (1.2.4) can be rewritten in terms of an integral over $\partial\Omega$. ⌟

HINT 2 for (1-2.d): As in Sub-problem (1-2.c) use the "probing function" $v(\boldsymbol{x}) := \log(1 - \|\boldsymbol{x}\|)$. ⌟

SOLUTION of (1-2.d):

Using polar coordinates [Lecture → Eq. (1.2.3.47)], the functional ℓ_4 can be rewritten as

$$\begin{aligned} \ell_4(v) &= \int_\Omega v\left(\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}\right)\mathrm{d}\boldsymbol{x} = \int_0^1\int_0^{2\pi} v\left(\begin{bmatrix}\cos\phi\\ \sin\phi\end{bmatrix}\right) r\mathrm{d}\phi\mathrm{d}r = \\ &= \frac{1}{2}\int_0^{2\pi} v\left(\begin{bmatrix}\cos\phi\\ \sin\phi\end{bmatrix}\right)\mathrm{d}\phi = \frac{1}{2}\int_{\partial\Omega} v(\boldsymbol{x})\mathrm{d}S(\boldsymbol{x}). \end{aligned} \tag{1.2.9}$$

If we consider again $v(\boldsymbol{x}) = \log(1-\|\boldsymbol{x}\|) \in L^2(\Omega)$, then $\int_{\partial\Omega}\log(1-\|\boldsymbol{x}\|)\mathrm{d}S(\boldsymbol{x}) = -\infty$, which means that ℓ_4 is *not* continuous on $L^2(\Omega)$.

Alternatively, following the considerations of [Lecture → Ex. 1.2.3.45], we can also identify a sequence $(v_n)_{n\in\mathbb{N}} \subset L^2(\Omega)$ for which $\|v_n\|_{L^2(\Omega)} \le C$ for all $n \in \mathbb{N}$, whereas $\ell_4(v_n) \to \infty$ for $n \to \infty$. Appealing to the computations above, we may choose

$$v_n(\boldsymbol{x}) = \log(1+n^{-1}-\|\boldsymbol{x}\|)\,, \quad n \in \mathbb{N}\,.$$

HINT 1 for (1-2.e): Of course, the Cauchy-Schwarz inequality is the key tool for proving continuity of ℓ_1. ⌟

SOLUTION of (1-2.e):

ℓ_1 is continuous on $H^1(\Omega)$, which is confirmed by applying the Cauchy-Schwarz inequality, first in $\mathbb{R}^2$ in the form

$$|\boldsymbol{x}\cdot\boldsymbol{y}| \le \|\boldsymbol{x}\|\,\|\boldsymbol{y}\| \quad \forall \boldsymbol{x},\boldsymbol{y}\in\mathbb{R}^2\,, \tag{1.2.10}$$

where $\|\cdot\|$ denotes the Euclidean norm, and then in the form

$$\left|\int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right| \le \left(\int_\Omega |u(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2}\left(\int_\Omega |v(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2} = \|u\|_{L^2(\Omega)}\|v\|_{L^2(\Omega)}\,:$$

[Lecture → Eq. (1.3.4.15)]

$$\begin{aligned}|\ell_1(v)| = \left|\int_\Omega \boldsymbol{c}\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right| &\overset{(1.2.10)}{\le} \int_\Omega \|\boldsymbol{c}\|\,\|\mathbf{grad}\,v(\boldsymbol{x})\|\,\mathrm{d}\boldsymbol{x} \le \|\boldsymbol{c}\|\int_\Omega 1\cdot\|\mathbf{grad}\,v\|\mathrm{d}\boldsymbol{x}\\ &\underset{\text{[Lecture → Eq. (1.3.4.15)]}}{\le} \|\boldsymbol{c}\||\Omega||v|_{H^1(\Omega)} \le \|\boldsymbol{c}\||\Omega|\|v\|_{H^1(\Omega)} \quad \forall v\in H^1(\Omega)\,,\end{aligned}$$

where in the first and second step we have used Cauchy-Schwarz inequality.

HINT 1 for (1-2.f): Again, the Cauchy-Schwarz inequality can be used. ⌟

SOLUTION of (1-2.f):

Also ℓ_2 is continuous on $H^1(\Omega)$. Indeed, using Cauchy-Schwarz inequality we obtain:

$$|\ell_2(v)| = \left| \int_\Omega v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \right| \leq |\Omega| \|v\|_{L^2(\Omega)} \leq |\Omega| \|v\|_{H^1(\Omega)}, \quad \text{for all } v \in H^1(\Omega).$$

Here we have also invoked the trivial estimate $\|v\|_{L^2(\Omega)} \leq \|v\|_{H^1(\Omega)}$, which is a direct consequence of [Lecture → Def. 1.3.4.8], because

$$\|v\|^2_{H^1(\Omega)} = \|v\|^2_{L^2(\Omega)} + \underbrace{\|\mathbf{grad}\, u\|^2_{L^2(\Omega)}}_{\geq 0} .$$

HINT 1 for (1-2.g): The functional ℓ_3 relies on bounday values for $\mathbf{grad}\, v$. However, for $v \in H^1(\Omega)$ we have $\mathbf{grad}\, v \in L^2(\Omega)$. Then remember [Lecture $\rightarrow$ Ex. 1.3.2.5]. ⌟

HINT 2 for (1-2.g): Follow Rem. 1.2.5 and use the rotationally symmetric probing function $v(\boldsymbol{x}) = (1 - \|\boldsymbol{x}$

⌟

SOLUTION of (1-2.g):

ℓ_3 is *not* continuous on $H^1(\Omega)$. Again we look for a function $v \in H^1(\Omega)$ for which ℓ_3 blows up. As in Sub-problem (1-2.c) we try a radially symmetric function. For example, if we take $v(\boldsymbol{x}) = (1 - \|\boldsymbol{x}\|)\log(1 - \|\boldsymbol{x}\|)$, then $\mathbf{grad}\, v = -\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}(\log(1 - \|\boldsymbol{x}\|) + 1))$ and therefore

$$\int_{\partial\Omega} \mathbf{grad}\, v(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \mathrm{d}S(\boldsymbol{x}) = -\int_{\partial\Omega} \log(1 - \|x\|) + 1 \mathrm{d}S(\boldsymbol{x}) = +\infty.$$

The fact that $v \in H^1(\Omega)$ is a consequence of $\xi \mapsto \xi \log \xi$ belonging to $H^1(]0,1[)$, which can easily verified by computing the integral.

HINT 1 for (1-2.h): The functional ℓ_4 can be rewritten in terms of an integral over $\partial\Omega$. ⌟

SOLUTION of (1-2.h):

ℓ_4 is continuous on $H^1(\Omega)$. This can be proved by starting with similar manipulations as in (1.2.9)

$$\begin{aligned}\left|\int_\Omega v\left(\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}\right)\mathrm{d}\boldsymbol{x}\right| &= \tfrac{1}{2}\left|\int_{\partial\Omega} v(\boldsymbol{x})\mathrm{d}S(\boldsymbol{x})\right| \leq \\ &\leq |\partial\Omega|^{\frac{1}{2}}\,\|v\|_{L^2(\partial\Omega)} \leq \\ &\leq |\partial\Omega|^{\frac{1}{2}}\,C(\Omega)\sqrt{\|v\|_{L^2(\Omega)}\|v\|_{H^1(\Omega)}} \leq \\ &\leq |\partial\Omega|C(\Omega)\|v\|_{H^1(\Omega)},\end{aligned}$$

for all $v \in H^1(\Omega)$, where for the first inequality we have used Cauchy-Schwarz inequality, and for the second one we have used the multiplicative trace inequality from [Lecture → Thm. 1.9.0.10]:

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

$$\exists C = C(\Omega) > 0: \quad \|u\|^2_{L^2(\partial\Omega)} \leq C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega)\ .$$

HINT 1 for (1-3.a): Study the proof of [Lecture → Thm. 1.3.4.17] in 1D. Apply the fundamental theorem of calculus [Lecture → Eq. (1.5.1.6)]. ⌟

SOLUTION of (1-3.a):

By virtue of a density argument it is sufficient for prove the inequality (1.3.1) for

$$u \in C_0^1([0,1]) := \{v \in C^1([0,1]) : v(0) = v(1) = 0\} .$$

Further explanations are given in the proof of [Lecture → Thm. 1.3.4.17].

With the help of the fundamental theorem of calculus and the Cauchy-Schwarz inequality (CSI) [Lecture → Eq. (1.3.4.15)], $\forall x \in]0,1[$ it holds that:

$$|u(x)| = \left| \int_0^x u'(\tau)\,\mathrm{d}\tau + \underbrace{u(0)}_{=0} \right| \overset{\text{(CSI)}}{\leq} \left(\int_0^x 1\,\mathrm{d}\tau \right)^{\frac{1}{2}} \left(\int_0^x |u'(\tau)|^2\,\mathrm{d}\tau \right)^{\frac{1}{2}}$$

$$\leq \left(\int_0^1 |u'(\tau)|^2\,\mathrm{d}\tau \right)^{\frac{1}{2}} = |u|_{H^1(]0,1[)} \quad \forall u \in C_0^1(]0,1[) .$$

To finish, take the supremum over all $x \in]0,1[$:

$$\|u\|_{L^\infty(]0,1[)} = \sup_{x \in]0,1[} |u(x)| \leq |u|_{H^1(]0,1[)} .$$

There is an alternative "**physical reasoning**": One first observes that multiplying the function u in (1.3.1) with a scalar does not change anything. This, without loss of generality we can assume $\|u\|_{L^\infty(]0,1[)} = 1$. Let us assume $u(x) = 1$ for some $x \in]0,1[$.

How small can $|u|_{H^1(]0,1[)}$ become in this case. Remember that $|u|^2_{H^1(]0,1[)}$ is the *elastic energy* of a homogeneous elastic string whose shape is described by the graph of u. What is the minimal energy of such a string under the *pinning conditions* $u(0) = u(1) = 0$ and $u(x) = 1$. Since we are interested in the pure elastic energy, no forces come into play. Thus the string will just be straight between the pinning points.

The elastic energy of such a piecewise straight string, that is, $|u|^2_{H^1(]0,1[)}$ for a function piecewise linear in $[0,x] \cup [x,1]$, evaluates to

$$|u|^2_{H^1(]0,1[)} = x \cdot \left| \frac{1}{x} \right|^2 + (1-x) \left| \frac{1}{1-x} \right|^2 = \frac{1}{x} + \frac{1}{1-x} = \frac{1}{x(1-x)} \geq 4 > 1 .$$

HINT 1 for (1-3.b): Fix a point $x \in]0,1[$ and then use [Lecture $\rightarrow$ Thm. 1.9.0.10] with the domain $\Omega =]0,x[$. ⌟

HINT 2 for (1-3.b): The boundary of an interval are its two endpoints. Computing the “integral” of a function “over a point” boils down to evaluating the function in that point. ⌟

SOLUTION of (1-3.b):

Given $x \in]0,1[$, let us restate the assertion of [Lecture → Thm. 1.9.0.10] for $\Omega =]0,x[$, taking into account that computing the "integral" of a function "over a point" boils down to evaluating the function in that point. We obtain

$$u(0)^2 + u(x)^2 \leq C(x)\|u\|_{L^2(]0,x[)}\,\|u\|_{H^1(]0,x[)} \quad \forall u \in H^1(]0,x[)\ .$$

with some $C(x) > 0$ depending on x. Now assume $u \in C^1_0([0,1])$, which implies $u(0) = 0$. Moreover, from the definitions [Lecture → Def. 1.3.2.3] and [Lecture → Def. 1.3.4.8] it is clear that

$$\|u\|_{L^2(]0,x[)} \leq \|u\|_{L^2(]0,1[)} \quad , \quad \|u\|_{H^1(]0,x[)} \leq \|u\|_{H^1(]0,1[)}\ .$$

Moreover, by the Poincaré-Friedrichs inequality [Lecture → Thm. 1.3.4.17] for $\Omega =]0,1[$ we know

$$\|u\|_{L^2(]0,1[)} \leq |u|_{H^1(]0,1[)} \quad \text{and} \quad \|u\|_{H^1(]0,1[)} \leq 2|u|_{H^1(]0,1[)}\ .$$

Thus setting $C := 2\sup_{x\in]0,1[} C(x)$, we conclude (1.3.4).

Remark. The constant in the multiplicative trace inequality of [Lecture → Thm. 1.9.0.10] depends "continuously" on Ω. For the interval $\Omega =]0,x[$ this means that the supremum of $C(x)$ is finite.

HINT 3 for (1-4.): Use

1. the fundamental theorem of calculus [Lecture → Eq. (1.5.1.6)] in the form
$$u(x) = u(y) + \int_y^x u'(t)\mathrm{d}t \,,$$
2. the simple estimate $(a+b)^2 \leq (a+b)^2 + (a-b)^2 = 2(a^2+b^2)$, $a, b \in \mathbb{R}$, also a consequence of the arithmetic-geometric mean inequality.
3. and the Cauchy-Schwartz inequality for integrals in 1D, *cf.* [Lecture → Eq. (1.3.4.15)].

⌟

SOLUTION of (1-4.):

Let us write

$$u(x) = \frac{1}{b-a}\int_a^b u(x)\mathrm{d}y = \frac{1}{b-a}\int_a^b \left(u(y) + \int_y^x u'(t)\mathrm{d}t\right)\mathrm{d}y.$$

This gives us, using the hint about $(f+g)^2 \leq 2(f^2+g^2)$:

$$\begin{aligned}\int_a^b |u(x)|^2\mathrm{d}x &= \int_a^b \left|\frac{1}{b-a}\int_a^b \left(u(y) + \int_y^x u'(t)\mathrm{d}t\right)\mathrm{d}y\right|^2\mathrm{d}x \\ &= \frac{1}{(b-a)^2}\int_a^b \left|\int_a^b \left(u(y) + \int_y^x u'(t)\mathrm{d}t\right)\mathrm{d}y\right|^2\mathrm{d}x \\ &\leq \frac{2}{(b-a)^2}\int_a^b \left|\int_a^b u(y)\mathrm{d}y\right|^2\mathrm{d}x + \frac{2}{(b-a)^2}\int_a^b \left|\int_a^b \int_x^y u'(t)\mathrm{d}t\mathrm{d}y\right|^2\mathrm{d}x.\end{aligned}$$

We will now treat each term separately. For the first, we see that the integrand is independent of x, so

$$\frac{2}{(b-a)^2}\int_a^b \left|\int_a^b u(y)\mathrm{d}y\right|^2\mathrm{d}x \leq \frac{2}{b-a}\left|\int_a^b u(y)\mathrm{d}y\right|^2.$$

For the second, we use Cauchy-Schwartz twice, to get

$$\begin{aligned}\frac{2}{(b-a)^2}\int_a^b \left|\int_a^b \int_x^y u'(t)\mathrm{d}t\mathrm{d}y\right|^2\mathrm{d}x &= \frac{2}{(b-a)^2}\int_a^b \left|\int_a^b 1\cdot\left(\int_x^y u'(t)\mathrm{d}t\right)\mathrm{d}y\right|^2\mathrm{d}x \\ &\leq \frac{2}{(b-a)^2}\int_a^b \int_a^b \mathrm{d}y \int_a^b \left|\int_y^x u'(t)\mathrm{d}t\right|^2\mathrm{d}y\mathrm{d}x \\ &= \frac{2}{b-a}\int_a^b \int_a^b \left|\int_y^x 1\cdot u'(t)\mathrm{d}t\right|^2\mathrm{d}y\mathrm{d}x \\ &\leq \frac{2}{b-a}\int_a^b \int_a^b |x-y|\int_y^x |u'(t)|^2\mathrm{d}t\mathrm{d}y\mathrm{d}x\end{aligned}$$

Now, use the facts that $|x-y| \leq b-a$ and $\int_y^x |u'(t)|^2\mathrm{d}t \leq \int_a^b |u'(t)|^2\mathrm{d}t$ to finish:

$$\begin{aligned}\frac{2}{b-a}\int_a^b \int_a^b |x-y|\int_y^x |u'(t)|^2\mathrm{d}t\mathrm{d}y\mathrm{d}x &\leq 2\int_a^b \int_a^b \int_a^b |u'(t)|^2\mathrm{d}t\mathrm{d}y\mathrm{d}x \\ &= 2(b-a)^2\int_a^b |u'(t)|^2\mathrm{d}t.\end{aligned}$$

This concludes the proof.

HINT 1 for (1-5.a): The continuity of a bilinear form on a normed vector space is defined in [Lecture → Def. 1.2.3.42].

HINT 2 for (1-5.a): You may use (without proof) the result (1.5.5). ⌟

SOLUTION of (1-5.a):

According to [Lecture → Def. 1.2.3.42] we have to show

$$\exists C > 0: \quad |\mathsf{a}(u,v)| \leq C\|u\|_{H^1(]0,1[)} \cdot \|v\|_{H^1(]0,1[)} \quad \forall u, v \in V_0 \,. \tag{1.5.6}$$

The norm of $H^1(]0,1[)$ is

$$\|v\|^2_{H^1(]0,1[)} := \int_0^1 \left(\frac{dv}{dx}(x)\right)^2 + v(x)^2 \,\mathrm{d}x \,.$$

Morover, by

$$|v(x)| \leq |v|_{H^1(]0,1[)} \quad \forall x \in [0,1] \,, \quad \forall v \in V_0 \,, \tag{1.5.5}$$

we know that

$$v(1) \leq \|v\|_{H^1(]0,1[)} \quad \forall v \in V_0 \,.$$

Using the Cauchy-Schwarz inequality (CSI) [Lecture → Eq. (1.3.4.15)] together with (1.5.5) we get

$$\begin{aligned} |\mathsf{a}(u,v)| =& \left| \int_0^1 \frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x + u(1)v(1) \right| + \\ &\overset{\text{CSI}}{\leq} \left(\int_0^1 \left|\frac{du}{dx}(x)\right|^2 \mathrm{d}x \right)^{\frac{1}{2}} \cdot \left(\int_0^1 \left|\frac{dv}{dx}(x)\right|^2 \mathrm{d}x \right)^{\frac{1}{2}} + |u(1)||v(1)| \\ &\leq |u|_{H^1(]0,1[)} \cdot |v|_{H^1(]0,1[)} + \|u\|_{H^1(]0,1[)}\|v\|_{H^1(]0,1[)} \leq 2\|u\|_{H^1(]0,1[)} \cdot \|v\|_{H^1(]0,1[)} \,. \end{aligned}$$

HINT 1 for (1-5.b): Follow the reasoning of [Lecture → § 1.2.3.36]. ⌟

SOLUTION of (1-5.b):

According to [Lecture → Def. 0.3.1.15], [Lecture → Def. 1.2.3.24], and [Lecture → Def. 1.2.3.27], we have to verify three properties of the bilinear form

$$\mathsf{a}(u,v) = \int_0^1 \frac{du}{dx}\frac{dv}{dx}\,\mathrm{d}x + u(1)v(1)\,, \quad u,v \in V_0 := \left\{ v \in H^1(]0,1[) : v(0) = 0 \right\}. \tag{1.5.7}$$

- The symmetry of a, $\mathsf{a}(\mathsf{u},\mathsf{v}) = \mathsf{a}(\mathsf{v},\mathsf{u})$ for all $u, v \in V_0$, is an obvious[1] structural property.
- The non-negativity of a, $\mathsf{a}(v,v) \geq 0$ for all $v \in V_0$ is obvious as well:

$$\mathsf{a}(v,v) = \int_0^1 \left| \frac{dv}{dx}(x) \right|^2 \mathrm{d}x + |v(1)|^2 \geq 0\,, \tag{1.5.8}$$

  because no square can be negative.

- Definiteness, $\mathsf{a}(v,v) > 0$, if $v \neq 0$:

  If $\mathsf{a}(v,v) = 0$ for some $v \in V_0$, we conclude from (1.5.8) that the derivative $v' \equiv 0$, which means $v \equiv \mathrm{const}$, and that $v(1) = 0$. Thus, in addition taking into account the constraint $v(0) = 0$ built into V_0, we conclude $v \equiv 0$:

  ◤ $\mathsf{a}(v,v) > 0 \quad \forall v \in V_0,\ v \neq 0\,.$

[1] Calling properties or results "obvious" is a widespread bad habit in mathematics literature, but in this case I dare to indulge in it.

HINT 1 for (1-5.c): Appeal to (1.5.5) to verify [Lecture → Eq. (1.2.3.40)]. ⌟

SOLUTION of (1-5.c):

We simply infer from (1.5.5) that

$$\ell(v) := v(1/2) \leq |v|_{H^1(]0,1[)} \leq \|v\|_a := \mathsf{a}(v,v)^{1/2} \quad \forall v \in V_0 \ ,$$

because, obviously,

$$|v|^2_{H^1(]0,1[)} = \int_0^1 \left|\frac{dv}{dx}(x)\right|^2 \mathrm{d}x \leq \int_0^1 \left|\frac{dv}{dx}(x)\right|^2 \mathrm{d}x + v(1)^2 = \mathsf{a}(v,v) \ .$$

HINT 1 for (1-5.d): To extract the differential equation on an interval use smooth test functions compactly supported in that interval, see [Lecture → § 1.5.1.9]. ⌟

SOLUTION of (1-5.d):

We focus on $]0,1/2[$; the same considerations apply to the right half of $]0,1[$. The arguments parallel those of [Lecture → § 1.5.1.9].

We consider smooth test functions v *compactly supported in* $]0,1/2[$. We perform integration by parts [Lecture → Eq. (1.5.1.8)] on (1.5.1) and get

$$-\int_0^{1/2} \frac{d^2u}{dx^2}(x)\, v(x)\, \mathrm{d}x = \underbrace{v(1/2)}_{=0} \quad \forall v \in C_0^\infty(]0,1/2[) \ .$$

Appealing to the fundamental lemma of the calculus of variations, we end up with the ordinary differential equation (ODE)

$$\frac{d^2u}{dx^2} = 0 \quad \text{in} \quad]0,1/2[\quad \text{and} \quad]1/2,1[\ . \tag{1.5.11}$$

HINT 1 for (1-5.e): Adapt the arguments of [Lecture → Ex. 1.5.3.11] to (1.5.1). ⌟

SOLUTION of (1-5.e):

Continuing what was done in Sub-problem (1-5.d) we now resort to smooth test functions v that *vanish on* $[0, 1/2]$. Then integration by parts results in

$$-\int_{1/2}^{1} \frac{d^2u}{dx^2}(x)\, v(x)\, \mathrm{d}x + \frac{du}{dx}(1)v(1) + u(1)v(1) = 0 \quad \forall v \in C^\infty([1/2, 1]),\ v(1/2) = 0\ .$$

Appeal to (1.5.11) to see that the integral on the left vanishes, if u is a solution of (1.5.1) and $u|_{[1/2,1]} \in C^2([1$
This yields the **impedance boundary condition** [Lecture → Ex. 1.7.0.5]

$$\frac{du}{dx}(1) + u(1) = 0\ . \tag{1.5.12}$$

HINT 1 for (1-5.f): Apply [Lecture → Thm. 1.3.4.23] to infer the first so-called **transmission condition** ($\hat{=}$ a continuity condition) for u at $x = 1/2$. ⌟

HINT 2 for (1-5.f): Test with functions compactly supported in $]0,1[$. To obtain the second tranmission condition, you have to rely on integration by parts and the differential equation satisfied by u in $]0,1/2[$ and $]1/2,1[$. That was found in Sub-problem (1-5.d). ⌟

SOLUTION of (1-5.f):

(I) The first transmission condition implied by [Lecture → Thm. 1.3.4.23] reads

$$u(1/2-) = u(1/2+) \quad \text{with} \quad u(1/2\pm) := \lim_{\epsilon \to 0} u(1/2 \pm \epsilon) \,. \tag{1.5.13}$$

(II) Integration by parts *in the sub-intervals* $[0,1/2]$ *and* $[1/2,1]$ yields

$$-\int_{[0,1]\setminus\{1/2\}} \underbrace{\frac{d^2u}{dx^2}(x)}_{=0 \text{ by (1.5.11)}} v(x)\,\mathrm{d}x + \frac{du}{dx}(1/2-)v(1/2) - \frac{du}{dx}(1/2+)v(1/2) = v(1/2) \quad \forall v \in C_0^\infty(]0,1[) \,,$$

which implies the **jump condition** for the first derivative of the piecewise smooth solution u of (1.5.1):

$$\frac{du}{dx}(1/2-) - \frac{du}{dx}(1/2+) = 1 \,. \tag{1.5.14}$$

State an explicit formula for the solution of (1.5.1).

HINT 1 for (1-5.g): What functions satisfy (1.5.11)? ⌟

SOLUTION of (1-5.g):

Thanks to (1.5.11), the solution of (1.5.1) must be *piecewise linear*. The transmission condition (1.5.13) implies continuity, (1.5.14) a kink at $x = 1/2$. Since $u(0) = 0$ we can set

$$u(x) = \begin{cases} \alpha x & \text{in} \quad [0, 1/2] \,, \\ \beta x + \gamma & \text{in} \quad [1/2, 1] \,, \end{cases}$$

and obtain a linear system of equations for the coefficients:

$$\text{(1.5.13)} \quad \Rightarrow \qquad \tfrac{1}{2}\alpha = \tfrac{1}{2}\beta + \gamma \,, \tag{1.5.15}$$

$$\text{(1.5.14)} \quad \Rightarrow \qquad \alpha = 1 + \beta \,, \tag{1.5.16}$$

$$\text{(1.5.12)} \quad \Rightarrow \qquad \beta + \beta + \gamma = 0 \,, \tag{1.5.17}$$

which has the solution $\alpha = \frac{3}{4}$, $\beta = -\frac{1}{4}$, $\gamma = \frac{1}{2}$, which means

$$u(x) = \begin{cases} \frac{3}{4}x & \text{in} \quad [0, 1/2] \,, \\ -\frac{1}{4}x + \frac{1}{2} & \text{in} \quad [1/2, 1] \,. \end{cases} \tag{1.5.18}$$

SOLUTION of (1-6.a):

Consult [Lecture → Def. 1.4.1.6]. The components of the linear variational problem can be read off (1.6.1) straightforwardly. For the bilinear form we have

$$\mathsf{a}(u,v) = \int_{\Omega} \kappa(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x},$$

and the linear form is easily seen to be

$$\ell(v) = \int_{\Gamma_1} v(\boldsymbol{x}) \mathrm{d}S.$$

HINT 1 for (1-6.b): Adapt the argument of [Lecture → § 1.2.3.36]. ⌟

SOLUTION of (1-6.b):

If $\mathsf{a}(u,u) = 0$ for an $u \in V_0$, then, since $\kappa(\boldsymbol{x}) \geq \underline{\kappa} > 0$ for all $\boldsymbol{x} \in \Omega$, we conclude that $\mathbf{grad}\, u \equiv 0$, which implies that $u \equiv \mathrm{const}$. Thus $u \equiv 0$, because it vanishes on the non-empty part Γ_0 of the boundary.

HINT 1 for (1-6.c): The energy norm is defined as in [Lecture → Def. 1.2.3.35], and ℓ must satisfy [Lecture → Eq. (1.2.3.40)] to be continuous with respect to this norm. Then use the *trace theorem* [Lecture → Thm. 1.9.0.10]. ⌟

SOLUTION of (1-6.c):

Take the cue from the proof of [Lecture → Cor. 1.3.4.19]. Use the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)], the multiplicative trace theorem [Lecture → Thm. 1.9.0.10], the triangle inequality and (1.6.3):

$$\begin{aligned}|\ell(\mathbf{v})|^2 &= \left|\int_{\Gamma_1} 1 \cdot v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\right|^2 \le \left(\int_{\Gamma_1} 1\mathrm{d}S(\boldsymbol{x})\right)\left(\int_{\Gamma_1} |v(\boldsymbol{x})|^2\,\mathrm{d}S(\boldsymbol{x})\right) \le C_0\|v\|^2_{L^2(\Gamma_1)}\\ &= C_0\|v\|^2_{L^2(\partial\Omega)} \le C_1\|v\|_{L^2(\Omega)}\|v\|_{H^1(\Omega)} \le C_1\|v\|_{L^2(\Omega)}\Big(\|v\|_{L^2(\Omega)} + |v|_{H^1(\Omega)}\Big)\\ &\le C_2|v|_{H^1(\Omega)}\Big(C_3|v|_{H^1(\Omega)} + |v|_{H^1(\Omega)}\Big) \le C_4|v|^2_{H^1(\Omega)} \le \frac{C_4}{\underline{\kappa}}\|v\|^2_{\mathsf{a}},\end{aligned}$$

which concludes the proof. Here, $\underline{\kappa}$ is a lower bound for $\kappa(\boldsymbol{x})$: $\kappa(\boldsymbol{x}) \ge \underline{\kappa} > 0$ for all $\boldsymbol{x} \in \Omega$.

HINT 1 for (1-6.d): Follow the approach of [Lecture → Section 1.5]: as test functions v use functions in $C_0^1(\Omega)$, that is, they should be zero on both boundaries Γ_0, Γ_1. Use [Lecture → Thm. 1.5.2.7] (with $\mathbf{grad}\, u$ in place of j). Argue what happens to the boundary terms. Then appeal to [Lecture → Lemma 1.5.3.4]. ⌟

SOLUTION of (1-6.d):

As discussed in [Lecture → Rem. 1.5.3.9], we have to assume $u \in C^2(\overline{\Omega})$ and $\kappa \in C^1(\overline{\Omega})$, which makes all derivatives exists in a classical pointwise sense. To begin with, we apply Green's formula [Lecture → Thm. 1.5.2.7]: For $v \in C^1(\Omega)$ we have

$$\begin{aligned}\int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = &\int_{\partial\Omega}\kappa(\boldsymbol{x})v(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\\ &-\int_\Omega v(\boldsymbol{x})\,\mathrm{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\mathrm{d}\boldsymbol{x}\,.\end{aligned} \tag{1.6.4}$$

Now, if we further restrict testing to functions $v \in C_0^1(\Omega)$, then we can exploit $v = 0$ on $\partial\Omega$, which makes all boundary terms in (1.6.4) vanish. Using (1.6.4), we reduce (1.6.1) to

$$-\int_\Omega v(\boldsymbol{x})\,\mathrm{div}\big(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\big)\,\mathrm{d}\boldsymbol{x} = 0\,.$$

This holds for every $v \in C_0^1(\Omega)$ and

Lemma [Lecture → Lemma 1.5.3.4]. Fundamental lemma of calculus of variations in higher dimensions

If $f \in L^2(\Omega)$ satisfies

$$\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega)\,,$$

then $f \equiv 0$ can be concluded.

implies $\mathrm{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = 0$ in Ω. Note that if $\kappa = \mathrm{const}$, then this becomes $-\Delta u = 0$, the familiar Laplace equation.

HINT 1 for (1-6.e): Follow the strategy from [Lecture → Ex. 1.5.3.11] and use the PDE derived in the previous sub-problem. ⌟

SOLUTION of (1-6.e):

First we notice that $u \in V_0$ means that we have a homogeneous Dirichlet boundary condition on Γ_0 (essential boundary condition). We still need to figure out the boundary condition on Γ_1. To do this we have to take into account the constraint on test functions implied by the definition (1.6.2) of V_0.

The reasoning starts as in [Lecture → Ex. 1.5.3.11], which you should study again. For $v \in V_0$ in (1.6.1) we first use (1.6.4) and then use the PDE $\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = 0$ derived in the previous sub-problem to cancel all terms inside the domain. The only remaining terms are then

$$\int_{\partial\Omega} \kappa(\boldsymbol{x}) v(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \mathbf{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_{\Gamma_1} v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\ .$$

We note that the integral over Γ_0 will disappear because $v = 0$ there. On Γ_1 v is not a general function, but we must take into account $v = \text{const}$, which only shows that (test with a function that has value 1 on Γ_1)

$$\int_{\Gamma_1} \kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \mathbf{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_{\Gamma_1} 1\,\mathrm{d}S(\boldsymbol{x}) = |\Gamma_1|\ .$$

This integral condition has to be met by the boundary values of $\kappa\,\mathbf{grad}\,u$ on Γ_1. So, this boundary condition is not given as a pointwise condition.

SOLUTION of (1-6.f):

According to [Lecture → § 1.6.0.4], the term $\mathbf{j} := -\kappa\,\mathbf{grad}\,u$ where u is the temperature and κ the heat conductivity represents the heat flux, see also [Lecture → § 1.6.0.1]. Thus the non-local boundary condition derived in sub-problem (1-6.e) represents the the total heat flux through the part Γ_1 of the boundary of the insulator. In the situation described in the introduction to this problem, this total heat flux will agree with the total heat production inside the conductor.

SOLUTION of (1-7.a):

The energy norm is

$$\|\mathbf{v}\|_{\mathsf{a}} = (\mathsf{a}(\mathbf{v},\mathbf{v}))^{1/2} = \left(\int_\Omega \alpha(\boldsymbol{x}) |\operatorname{div} \mathbf{v}(\boldsymbol{x})|^2 + \|\mathbf{v}(\boldsymbol{x})\|^2 \, \mathrm{d}\boldsymbol{x} \right)^{1/2} . \tag{1.7.5}$$

The norm properties, in particular definiteness, are easily verified since this norm is stronger than the $L^2(\Omega)$-norm. However, a verification of norm properties was not asked for.

SOLUTION of (1-7.b):

The energy space is just the Sobolev space of vector fields whose energy norm has a finite value

$$V_0 := \{v : \Omega \to \mathbb{R}^2 \text{ integrable : } \|\mathbf{v}\|_{\mathsf{a}} < \infty\} \ .$$

We remark, that for this space mathematicians use the abbreviation $\boldsymbol{H}(\mathrm{div}, \Omega)$.

SOLUTION of (1-7.c):

The continuity of ℓ is an immediate consequence of the Cauchy-Schwarz inequality in $L^2(\Omega)$ [Lecture → Eq. (1.3.4.15)] applied component-wise:

$$\int_\Omega \mathbf{f}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} = \int_\Omega f_1v_1 + f_2v_2\,\mathrm{d}\boldsymbol{x} \overset{\text{[Lecture → Eq. (1.3.4.15)]}}{\leq} \|f_1\|_{L^2(\Omega)}\|v_1\|_{L^2(\Omega)} + \|f_2\|_{L^2(\Omega)}\|v_2\|_{L^2(\Omega)}$$
$$\overset{(*)}{\leq} \left(\|f_1\|^2_{L^2(\Omega)} + \|f_2\|^2_{L^2(\Omega)}\right)^{1/2}\left(\|v_1\|^2_{L^2(\Omega)} + \|v_2\|^2_{L^2(\Omega)}\right)^{1/2} = \|\mathbf{f}\|_{L^2(\Omega)}\,\|\mathbf{v}\|_{L^2(\Omega)}\,.$$

In fact in the step $(*)$ we used the Cauchy-Schwarz inequality in $\mathbb{R}^2$:

$$\boldsymbol{x}\cdot\boldsymbol{y} = x_1y_1 + x_2y_2 \leq \|\boldsymbol{x}\|_2\|\boldsymbol{y}\|_2 = \left(x_1^2+x_2^2\right)^{1/2}\left(y_1^2+y_2^2\right)^{1/2}. \tag{1.7.7}$$

Thus we get

$$|\ell(\mathbf{v})| \leq \|\mathbf{f}\|_{L^2(\Omega)}\|\mathbf{v}\|_{L^2(\Omega)} \leq \|\mathbf{f}\|_{L^2(\Omega)}\|\mathbf{v}\|_{\mathsf{a}}\,,$$

because, trivially, $\|\mathbf{v}\|_{L^2(\Omega)} \leq \|\mathbf{v}\|_{\mathsf{a}}$.

Of course, as a shortcut, we can directly apply (1.7.7) along with the Cauchy-Schwarz inequality in $L^2(\Omega)$:

$$\int_\Omega \mathbf{f}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} \leq \int_\Omega \|\mathbf{f}\|_2\|\mathbf{v}\|_2\,\mathrm{d}\boldsymbol{x} \leq \|\|\mathbf{f}\|_2\|_{L^2(\Omega)}\|\|\mathbf{v}\|_2\|_{L^2(\Omega)} = \|\mathbf{f}\|_{L^2(\Omega)}\|\mathbf{v}\|_{L^2(\Omega)} \leq \|\mathbf{f}\|_{L^2(\Omega)}\|\mathbf{v}\|_{\mathsf{a}}\,.$$

HINT 1 for (1-7.d): Appeal to Gauss' theorem [Lecture → Thm. 1.5.2.4] and the Cauchy-Schwarz inequaliy in $L^2(\Omega)$ [Lecture → Eq. (1.3.4.15)]. ⌟

SOLUTION of (1-7.d):

By Gauss' theorem followed by an application of the Cauchy-Schwarz inequality:

$$
\begin{aligned}
|\ell(\mathbf{v})| &= \left| \int_{\partial\Omega} \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) \right| = \left| \int_{\Omega} 1 \ \mathrm{div}\, \mathbf{v}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \right| \\
&\overset{(*)}{\leq} \left(\int_{\Omega} 1^2 \, \mathrm{d}\boldsymbol{x} \right)^{1/2} \left(\int_{\Omega} |\, \mathrm{div}\, \mathbf{v}(\boldsymbol{x})|^2 \, \mathrm{d}\boldsymbol{x} \right)^{1/2} \leq |\Omega|^{1/2} \|\mathrm{div}\, \mathbf{v}\|_{L^2(\Omega)} \\
&\leq |\Omega|^{1/2} \alpha_0^{-1/2} \|v\|_{\mathsf{a}} \,,
\end{aligned}
$$

where in the last step we used (1.7.2) and in step $(*)$ we applied the Cauchy-Schwarz inequality for integrals [Lecture $\rightarrow$ Eq. (1.3.4.15)], where one of the functions is the constant function $\{\boldsymbol{x} \mapsto 1\}$.

HINT 1 for (1-7.e): Use **Green's first formula** [Lecture → Thm. 1.5.2.7] to shift the divergence operator off the test function $\mathbf{v}$:

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \mathrm{div}\, \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad \text{[Lecture → Eq. (13.2.3.2)]}$$

You must not neglect the boundary terms! ⌟

SOLUTION of (1-7.e):

By Green's first formula we can convert

$$\int_\Omega \alpha(\boldsymbol{x})\ \operatorname{div}\mathbf{u}(\boldsymbol{x})\ \operatorname{div}\mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x})\cdot\mathbf{v}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{f}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{v} \in (C^1(\overline{\Omega}))^2 \tag{1.7.1}$$

into

$$\int_\Omega -\mathbf{grad}\left(\alpha(\boldsymbol{x})\operatorname{div}\mathbf{u}\right)\cdot\mathbf{v} + \mathbf{u}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \alpha(\boldsymbol{x})\operatorname{div}\mathbf{u}(\boldsymbol{x})(\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x}))\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega \mathbf{f}(\boldsymbol{x})\cdot\mathbf{v}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

Testing with smooth vectorfields compactly supported inside Ω yields the partial differential equation

$$-\mathbf{grad}(\alpha(\boldsymbol{x})\operatorname{div}\mathbf{u}) + \mathbf{u} = \mathbf{f} \quad \text{in } \Omega\ . \tag{1.7.9}$$

Then we use this identity and test with general vectorfields $\mathbf{v}$ that do not vanish on $\partial\Omega$, which gives

$$\int_{\partial\Omega} \alpha(\boldsymbol{x})\operatorname{div}\mathbf{u}(\boldsymbol{x})\,\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0\,.$$

This can only hold for arbitrary $\mathbf{v}$, if the boundary condition

$$\alpha(\boldsymbol{x})\operatorname{div}\mathbf{u}(\boldsymbol{x}) = 0\,, \quad \boldsymbol{x}\in\partial\Omega \tag{1.7.10}$$

is satisfied. Since $\alpha(\boldsymbol{x}) > 0$, we arrive at:

$$\operatorname{div}\mathbf{u} = 0 \quad \text{on} \quad \partial\Omega\ . \tag{1.7.11}$$

HINT 1 for (1-7.f): The hint for Sub-problem (1-7.e) still applies. ⌟

SOLUTION of (1-7.f):

The solution involves the same arguments as that of Sub-problem (1-7.e). Now we first obtain that for any test vectorfield $\mathbf{v}$

$$\int_\Omega -\mathbf{grad}\big(\alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}\big)\cdot\mathbf{v}+\mathbf{u}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x}+\int_{\partial\Omega}\alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}(\boldsymbol{x})\,\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})=\int_{\partial\Omega}\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}\,\mathrm{d}S(\boldsymbol{x})\;.$$

Restricting ourselves to compactly supported vectorfields we arrive at the partial differential equation

$$-\mathbf{grad}\big(\alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}\big)+\mathbf{u}=\mathbf{0}\quad\text{in }\Omega\;. \tag{1.7.12}$$

Using the PDE when testing with general $\mathbf{v}$ yields

$$\int_{\partial\Omega}\alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}(\boldsymbol{x})\,\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})=\int_{\partial\Omega}1\,\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}\,\mathrm{d}S(\boldsymbol{x})\;,$$

for all $\mathbf{v}\in(C^1(\overline{\Omega}))^2$. This means that we have to impose

$$\alpha(\boldsymbol{x})\,\mathrm{div}\,\mathbf{u}(\boldsymbol{x})=1\;,\quad\boldsymbol{x}\in\partial\Omega\;. \tag{1.7.13}$$

SOLUTION of (1-7.g):

The linear variational problem (1.7.1) involves the symmetric positive definite bilinear form

$$\mathsf{a}(\mathbf{u},\mathbf{v}) = \int_\Omega \alpha(\boldsymbol{x}) \operatorname{div}\mathbf{u}(\boldsymbol{x}) \operatorname{div}\mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x})\cdot\mathbf{v}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,. \tag{1.7.14}$$

From [Lecture $\rightarrow$ Rem. 1.3.3.9] we know that the quadratic minimization problem corresponding to a linear variational problem

$$u \in V_0\colon \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0$$

reads

$$u = \operatorname*{argmin}_{v\in V_0} \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v)\,. \tag{1.7.15}$$

Hence, the quadratic minimization problem for (1.7.1) with (1.7.8) is

$$\mathbf{u} = \operatorname*{argmin}_{\mathbf{v}\in(C^1(\overline{\Omega}))^2} \tfrac{1}{2}\int_\Omega |\operatorname{div}\mathbf{v}(\boldsymbol{x})|^2 + \|\mathbf{v}\|^2\,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega}\mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,. \tag{1.7.16}$$

This formula is all the problem asks for.

HINT 1 for (1-7.h):

(i) Start from the left-hand side of (1.7.1) and perform integration by parts based on Green's formula.

(ii) Note that the sought linear form ℓ will feature a boundary term.

⌟

SOLUTION of (1-7.h):

We manipulate the bilinear form from (1.7.1)

$$\int_\Omega \operatorname{div}\mathbf{u}\ \operatorname{div}\mathbf{v} + \mathbf{u}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} = \int_\Omega \left(-\,\mathbf{grad}\operatorname{div}\mathbf{u} + \mathbf{u}\right)\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \operatorname{div}\mathbf{u}(\mathbf{v}\cdot\boldsymbol{n})\,\mathrm{d}S$$

$$= \underbrace{\int_\Omega 2\begin{bmatrix}\sin(x_1)\\ \cos(x_2)\end{bmatrix}\cdot\mathbf{v}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}(\cos(x_1)-\sin(x_2))(\mathbf{v}\cdot\boldsymbol{n})\,\mathrm{d}S(\boldsymbol{x})}_{=\ell(\mathbf{v})}\ ,$$

because for $\mathbf{u}$ given as in (1.7.17) we have by elementary computations

$$-\,\mathbf{grad}\operatorname{div}\mathbf{u}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x}) = \begin{bmatrix}\sin(x_1)\\ \cos(x_2)\end{bmatrix} \quad , \quad \operatorname{div}\mathbf{u}(\boldsymbol{x}) = \cos(x_1) - \sin(x_2)\ .$$

SOLUTION of (1-7.i):

The variational problem satisfies the assumptions of Cea's lemma [Lecture → Thm. 3.1.3.7], which implies

$$\|\mathbf{u}-\mathbf{u}_h\|_{\mathsf{a}} = \inf_{\mathbf{v}_h\in V_{0,h}} \|\mathbf{u}-\mathbf{v}_h\|_{\mathsf{a}} \quad \leq 2 \inf_{\mathbf{v}_h\in V_{0,h}} \|\mathbf{u}-\mathbf{v}_h\|_{H^1(\Omega)} \,.$$

Here, we have used the simple estimate

$$|\operatorname{div}\mathbf{v}|^2 = \left|\frac{\partial v_1}{\partial x_1}+\frac{\partial v_2}{\partial x_2}\right| \leq 2\left\{\left|\frac{\partial v_1}{\partial x_1}\right|^2+\left|\frac{\partial v_2}{\partial x_2}\right|^2\right\},$$

which implies

$$\int_\Omega |\operatorname{div}\mathbf{v}|^2\,\mathrm{d}\boldsymbol{x} \leq 2\int_\Omega \left|\frac{\partial v_1}{\partial x_1}\right|^2+\left|\frac{\partial v_2}{\partial x_2}\right|^2 \mathrm{d}\boldsymbol{x}$$

$$2\int_\Omega \left|\frac{\partial v_1}{\partial x_1}\right|^2+\left|\frac{\partial v_1}{\partial x_2}\right|^2+\left|\frac{\partial v_2}{\partial x_1}\right|^2+\left|\frac{\partial v_2}{\partial x_2}\right|^2 \mathrm{d}\boldsymbol{x}$$

$$= \int_\Omega \|\mathbf{grad}\, v_1\|^2+\|\mathbf{grad}\, v_2\|^2\,\mathrm{d}\boldsymbol{x} = 2|\mathbf{v}|^2_{H^1(\Omega)} \,.$$

For $V_{0,N}=\mathcal{S}^0_2(\mathcal{M})$ we learn from [Lecture → Thm. 3.3.5.6] applied component-wise

$$\inf_{\mathbf{v}_N\in(\mathcal{S}^0_2(\mathcal{M}))^2} \|\mathbf{u}-\mathbf{v}_N\|_{H^1(\Omega)} \leq Ch^2_{\mathcal{M}}\|\mathbf{u}\|_{H^3(\Omega)} \,, \tag{1.7.18}$$

because $\mathbf{u}$ is smooth. In the case of regular refinement we have $h_{\mathcal{M}}\approx 2^{-L}$, from which we conclude

$$\|\mathbf{u}-\mathbf{u}_h\|_{\mathsf{a}} = O(4^{-L}) \quad \text{for} \quad L\to\infty \,. \tag{1.7.19}$$

The last estimate is all that is expected as an answer.

HINT 1 for (1-8.a): First, consider the equations corresponding to the first and second line of (1.8.1) separately. For the first, regard u and unknown and w as a mere parameter, for the second reverse their roles. ⌟

SOLUTION of (1-8.a):

The first line of (1.8.1) is a homogeneous Neumann problem for u and we can proceed as [Lecture → Ex. 1.8.0.10]. We get: seek $u \in H^1(\Omega)$

$$\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, p \,\mathrm{d}\boldsymbol{x} = \int_\Omega (c(\boldsymbol{x})w + f)\, p \,\mathrm{d}\boldsymbol{x} \quad \forall p \in H^1(\Omega)\ . \tag{1.8.2}$$

The second line of (1.8.1) represents an inhomogeneous Neumann problem for w, with corresponding variational formulation [Lecture → Eq. (1.8.0.12)]: seek $w \in H^1(\Omega)$

$$\int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, q \,\mathrm{d}\boldsymbol{x} = -\int_\Omega u\, q \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} u\, q \,\mathrm{d}S \quad \forall q \in H^1(\Omega)\ . \tag{1.8.3}$$

Simply combining both equations gives the variational formulation of the coupled problem.

SOLUTION of (1-8.b):

We have to “merge” (1.8.3) and (1.8.2) into a single variational problem.

We start from: seek $(u,w) \in V_0 := H^1(\Omega) \times H^1(\Omega)$

$$\begin{array}{rcrcll} \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, p \,\mathrm{d}\boldsymbol{x} & - & \int_\Omega c(\boldsymbol{x}) w\, p \,\mathrm{d}\boldsymbol{x} & = & \int_\Omega f\, p \,\mathrm{d}\boldsymbol{x} & \forall p \in H^1(\Omega)\,, \\ \int_\Omega u\, q \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u\, q \,\mathrm{d}S & + & \int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, q \,\mathrm{d}\boldsymbol{x} & = & 0 & \forall q \in H^1(\Omega)\,. \end{array} \tag{1.8.4}$$

Then we add both equations: seek $(u,w) \in V_0$

$$\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, p \,\mathrm{d}\boldsymbol{x} - \int_\Omega c(\boldsymbol{x}) w\, p \,\mathrm{d}\boldsymbol{x} + \int_\Omega u\, q \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u\, q \,\mathrm{d}S + \int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, q \,\mathrm{d}\boldsymbol{x} = \int_\Omega f\, p \,\mathrm{d}\boldsymbol{x} \quad \forall (p,q) \in V_0\,. \tag{1.8.5}$$

This permits us to read off the bilinear form and linear form

$$\mathsf{a}((u,w),(p,q)) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, p \,\mathrm{d}\boldsymbol{x} - \int_\Omega c(\boldsymbol{x}) w\, p \,\mathrm{d}\boldsymbol{x} + \int_\Omega u\, q \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u\, q \,\mathrm{d}S + \int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, q \,\mathrm{d}\boldsymbol{x}\,, \tag{1.8.6}$$

$$\ell((p,q)) = \int_\Omega f\, p \,\mathrm{d}\boldsymbol{x}\,. \tag{1.8.7}$$

HINT 1 for (1-8.c): For a *linear* equation $\mathsf{L}x = b$, uniqueness of solutions is equivalent to the fact that the problem with $b = 0$ has only the trivial solution $x = 0$. In the case of linear variational problems this can be expressed as follows:

Lemma 1.8.8. Uniqueness of solutions of linear variational problems

Assuming they exist, solutions of a linear variational problem [*Lecture* → *Def. 1.4.1.6*]

$$u \in V_0\text{:} \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0\,,$$

are unique, if and only if

$$\mathsf{a}(u,v) = 0 \quad \forall v \in V_0 \quad \Longleftrightarrow \quad u = 0\,.$$

This is closely related to the well known fact from linear algebra that solutions of a linear system of equations are unique, if and only if the kernel of the system matrix is trivial. ⌟

HINT 2 for (1-8.c): In the case of $f = 0$ test the first equation with the negative solution of the second, and the second with the solution of the first. Then subtract the resulting equations. ⌟

SOLUTION of (1-8.c):

We consider the variational problem

$$(u,w) \in V_0\colon \quad \mathsf{a}((u,w),(p,q)) = 0 \quad \forall (p,q) \in V_0\ ,$$

and show that it has the trivial solution $(u,w) \equiv 0$ only.

The trick is to pick the *special* pair of *test functions* $(p,q) := (-w,u)$, benefit from cancellation and get

$$\begin{aligned}\mathsf{a}((u,w),(-w,u)) = &= \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}(-w)\,\mathrm{d}\boldsymbol{x} - \int_\Omega c(\boldsymbol{x}) w\,(-w)\,\mathrm{d}\boldsymbol{x} + \int_\Omega u\,u\,\mathrm{d}\boldsymbol{x} + \\ &\int_{\partial\Omega} u\,u\,\mathrm{d}S + \int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, u\,\mathrm{d}\boldsymbol{x}\ , \\ &= \int_\Omega c(\boldsymbol{x}) w^2 + u^2\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} u^2\,\mathrm{d}S = 0\ ,\end{aligned}$$

which clearly implies $u \equiv 0$ and $w \equiv 0$.

HINT 1 for (1-8.d): That bilinear form is not positive definite, which can be seen by making a suitable choice of *constant* functions u and w and plugging them into both slots of the bilinear form. ⌟

SOLUTION of (1-8.d):

Choose $u = p \equiv 1$ and $w = q \equiv -1$ and compute

$$\begin{aligned} \mathsf{a}((\{x \to 1\}, \{x \to -1\}), (\{x \to 1\}, \{x \to -1\})) &= -\int_\Omega 1 \, \mathrm{d}x + \int_\Omega 1 \, \mathrm{d}x - \int_{\partial\Omega} 1 \, \mathrm{d}S \\ &= -|\Gamma| \, . \end{aligned}$$

SOLUTION of (1-9.a):

Note the following identity for vectors $\boldsymbol{a}, \boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^2$, which is a consequence of the associativity of matrix multiplication:

$$\left(\boldsymbol{a}^\top \boldsymbol{x}\right)\left(\boldsymbol{a}^\top \boldsymbol{y}\right) = \boldsymbol{x}^\top \boldsymbol{a}\boldsymbol{a}^\top \boldsymbol{y} = \left(\left(\boldsymbol{a}\boldsymbol{a}^\top\right)\boldsymbol{y}\right)^\top \boldsymbol{x} = \underbrace{\left(\boldsymbol{a}\boldsymbol{a}^\top\right)}_{\in \mathbb{R}^{2,2}} \boldsymbol{y} \cdot \boldsymbol{x} \,.$$

Thus, the variational problem can be converted into "standard form"

$$u \in H_0^1(\Omega): \quad \int_\Omega \left(\boldsymbol{a}\boldsymbol{a}^\top \operatorname{\mathbf{grad}} u(\boldsymbol{x})\right) \cdot \operatorname{\mathbf{grad}} v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega 1 \cdot v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)$$

Also note the homogeneous Dirichlet boundary conditions built into the trial and test space. Thus we get

$$\begin{aligned} -\operatorname{div}(\boldsymbol{a}\boldsymbol{a}^\top \operatorname{\mathbf{grad}} u) &= 1 \quad \text{in} \quad \Omega \,, \\ u &= 0 \quad \text{on} \quad \partial\Omega \,. \end{aligned}$$

SOLUTION of (1-9.b):

Note that there is no non-zero volume-based source term. Thus the PDE is

$$-\operatorname{div}((\boldsymbol{a}\cdot\boldsymbol{x})\,\mathbf{grad}\,u)+u=0 \quad \text{in} \quad \Omega \ . \tag{1.9.4}$$

Details can be found in [Lecture → Section 1.5.3].

To tease out the boundary conditions from (1.9.3) we apply Green's first formula and get

$$\int_\Omega \underbrace{(-\operatorname{div}((\boldsymbol{a}\cdot\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))+u(\boldsymbol{x}))}_{=0}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}+ \int_{\partial\Omega}(\boldsymbol{a}\cdot\boldsymbol{x})\,\mathbf{grad}\,u\cdot\boldsymbol{n}(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})+\int_{\partial\Omega}u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_{\partial\Omega} v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \ ,$$

which implies

$$\int_{\partial\Omega}((\boldsymbol{a}\cdot\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})+u(\boldsymbol{x})-1)\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})=0$$

for all $v\in H^1(\Omega)$. This yields the non-homogeneous Robin-type boundary condition

$$(\boldsymbol{a}\cdot\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})+u(\boldsymbol{x})=1 \quad \text{for} \quad \boldsymbol{x}\in\partial\Omega \ .$$

This agrees with the result of [Lecture → Ex. 1.8.0.6] with $\Psi(u)=u-1$.

HINT 1 for (2-1.a): Remember from [Lecture → Ex. 0.3.1.5] and [Lecture → Ex. 0.3.1.7] that for $V_0 = \mathbb{R}^2$ (2.1.1) can always be written as

$$\boldsymbol{x} \in \mathbb{R}^2: \quad \boldsymbol{y}^\top \mathbf{A}\boldsymbol{x} = \boldsymbol{y}^\top \boldsymbol{b} \quad \forall \boldsymbol{y} \in \mathbb{R}^2 \,, \tag{2.1.3}$$

with a suitable matrix $\mathbf{A} \in \mathbb{R}^{2,2}$ and vector $\boldsymbol{b} \in \mathbb{R}^2$. ⌟

HINT 2 for (2-1.a): What does Galerkin discretization mean for (2.1.3)? Consider $V_{0,h} = \operatorname{Span}\{\begin{bmatrix}0\\1\end{bmatrix}\}$, $\dim V_{0,h} = 1$. The discrete variational problem corresponding to (2.1.3) reads

$$\mu \in \mathbb{R}: \quad v \begin{bmatrix}0\\1\end{bmatrix}^\top \mathbf{A} \begin{bmatrix}0\\1\end{bmatrix} \mu = v \begin{bmatrix}0\\1\end{bmatrix}^\top \boldsymbol{b} \quad \forall v \in \mathbb{R} \,. \tag{2.1.4}$$

⌟

SOLUTION of (2-1.a):

Of course, we seek a *1-dimensional* subspace: $V_{0,h} = \operatorname{span}\{v\}$ for some $v \in \mathbb{R}^2 \setminus \{0\}$. Thus the Galerkin matrix will simply be a number that is supposed to be zero: We have to choose v such that $v^\top \mathbf{A} v = 0$. Now you have to "see" potential pairs $(\mathbf{A}, v)$:

A first choice may be a matrix that rotates a vector by $\pi/2$:

$$\mathbf{A} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad \blacktriangleright \quad v^\top \mathbf{A} v = 0 \quad \forall v \in \mathbb{R}^2 \,.$$

An alternative choice is motivated by the insight that multiplying a matrix with a coordinate vector from left and right selects a particular matrix entry:

$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad , \quad b = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad , \quad V_{0,h} = \operatorname{Span}\{ \begin{bmatrix} 0 \\ 1 \end{bmatrix} \} \,.$$

Since $\mathbf{A}$ is regular (2.1.3) will always have a unique solution, whereas (2.1.4) leads to $0 = 1$!

Remark. In fact, in the case of a symmetric bilinear form a on $\mathbb{R}^n$, $a(x, y) := y^\top \mathbf{A} x$, $\mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n}$, the discrete variational problem (2.1.2) has a unique solution for *any* $V_{0,h} \neq \{0\}$, **if and only if** $\mathbf{A}$ is **definite**, i.e., all eigenvalues of $\mathbf{A}$ have the same sign.

To see this, assume that $\mathbf{A}$ has eigenvalues $\lambda_- < 0 < \lambda_+$ with associated *orthonormal* eigenvectors $\mathbf{e}_-$ and $\mathbf{e}_+$. Then choose $\mathbf{v} := |\lambda_-|^{-\frac{1}{2}} \mathbf{e}_- + |\lambda_+|^{-\frac{1}{2}} \mathbf{e}_+$ and observe

$$\begin{aligned} \mathbf{v}^\top \mathbf{A} \mathbf{v} &= \left(|\lambda_-|^{-\frac{1}{2}} \mathbf{e}_- + |\lambda_+|^{-\frac{1}{2}} \mathbf{e}_+ \right)^\top \mathbf{A} \left(|\lambda_-|^{-\frac{1}{2}} \mathbf{e}_- + |\lambda_+|^{-\frac{1}{2}} \mathbf{e}_+ \right) \\ &= \left(|\lambda_-|^{-\frac{1}{2}} \mathbf{e}_- + |\lambda_+|^{-\frac{1}{2}} \mathbf{e}_+ \right)^\top \left(-|\lambda_-|^{\frac{1}{2}} \mathbf{e}_- + |\lambda_+|^{\frac{1}{2}} \mathbf{e}_+ \right) \\ &= -1 + 0 + 0 + 1 = 0 \,. \end{aligned}$$

SOLUTION of (2-1.b):

- You may simply cite

Theorem [Lecture → Thm. 1.2.3.44]. Existence of unique minimizer in finite dimensions

Let $J(u) := \frac{1}{2}\mathsf{a}(u,u) - \ell(u) + c$ with symmetric positive definite (→ [Lecture → Def. 1.2.3.27]) bilinear form $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$ (→ [Lecture → Def. 0.3.1.4]), linear form $\ell : V_0 \to \mathbb{R}$, $c \in \mathbb{R}$, be a quadratic functional on the vector space V_0.
If V_0 has finite dimension, then the quadratic minimization problem (→ [Lecture → Def. 1.2.3.11])

$$u_* = \operatorname*{argmin}_{u \in V_0} J(u)$$

always possesses a unique *solution.*

and the equivalence of quadratic minimization problems and linear variational problems:

Theorem [Lecture → Thm. 1.4.1.8]. Equivalence of quadratic minimization problem and linear variational problem

For a (generalized) quadratic functional $J(v) = \frac{1}{2}\mathsf{a}(v,v) - \ell(v) + c$ on a vector space V and with a symmetric positive definite bilinear form $\mathsf{a} : V \times V \to \mathbb{R}$ are equivalent:

(i) The quadratic minimization problem for J has unique minimizer $u_* \in \widehat{V}$ over the affine subspace $\widehat{V} = g + V_0$, $g \in V$.

(ii) The linear variational problem

$$u \in \widehat{V}: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,,$$

has a unique solution $u_* \in \widehat{V}$.

- See the first part of the proof of [Lecture → Thm. 2.2.1.5], where we conclude existence from uniqueness, because we deal with a linear equation in finite dimensions.
- Alternatively, an s.p.d. bilinear form will give rise to an s.p.d. Galerkin matrix $\mathbf{A}$, which will always be invertible/regular.

SOLUTION of (2-1.c):

Thanks to $V_{0,h} \subset V_0$ we can test both variational formulations with vectors $v_h \in V_{0,h}$ and then subtract them:

$$\begin{array}{rrcll} & a(u, v_h) & = & \ell(v_h) & \forall v_h \in V_{0,h} \subset V_0 \\ - & a(u_h, v_h) & = & \ell(v_h) & \forall v_h \in V_{0,h} \\ \hline & a(u - u_h, v_h) & = & 0 & \forall v_h \in V_{0,h} \, . \end{array}$$

HINT 1 for (2-1.d): First compute $J(u)$, express it via $\|u\|_a$ and $\ell(u)$! ⌟

SOLUTION of (2-1.d):

Using (2.1.1) and (2.1.2) we find

$$J(u) = \tfrac{1}{2}\mathsf{a}(u,u) - \ell(u) = -\tfrac{1}{2}\mathsf{a}(u,u) = -\tfrac{1}{2}\ell(u)\,,$$
$$J(u_h) = \tfrac{1}{2}\mathsf{a}(u_h,u_h) - \ell(u_h) = -\tfrac{1}{2}\mathsf{a}(u_h,u_h) = -\tfrac{1}{2}\ell(u_h)\,.$$

By the linearity of ℓ and Galerkin orthogonality (2.1.6):

$$J(u_h) - J(u) = \tfrac{1}{2}(\ell(u) - \ell(u_h)) = \tfrac{1}{2}\ell(u - u_h) \overset{(2.1.1)}{=} \tfrac{1}{2}\mathsf{a}(u, u - u_h)$$
$$\overset{\text{G.O.}}{=} \tfrac{1}{2}\mathsf{a}(u - u_h, u - u_h) = \tfrac{1}{2}\|u - u_h\|_{\mathsf{a}}^2\,.$$

SOLUTION of (2-10.a):

We face a task similar to what is asked in Problem 1-1.

To extract the bilinear form a and the linear functional ℓ from J from (2.10.1) we rewrite

$$J(v) = \int_\Omega \|\mathbf{grad}\, v(\boldsymbol{x})\|^2 - 2\mathbf{f}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \|\mathbf{f}(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x}\ .$$

$$\mathsf{a}(u,v) = 2\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\ , \quad u,v \in H^1_0(\Omega)\ , \tag{2.10.3}$$

$$\ell(v) = 2\int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\ , \quad v \in H^1_0(\Omega)\ . \tag{2.10.4}$$

Hence, by [Lecture → Thm. 1.4.1.8], the associated linear variational problem according to [Lecture → Def. 1.4.1.6] reads

$$u \in H^1_0(\Omega)\colon \quad \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_0(\Omega)\ . \tag{2.10.5}$$

The trial = test space is the Sobolev space $H^1_0(\Omega)$, see [Lecture → Def. 1.3.4.3]. That space is already given in the statement of the quadratic minimization problem.

Note that the bilinear form a is the weak form of $-\Delta$ with homogeneous boundary conditions.

SOLUTION of (2-10.j):

Of course, $\mathbf{f} = \mathbf{grad}\, u_h$ can be satisfied, if and only if $\mathbf{f} \in \mathbf{grad}\, \mathcal{S}^0_{1,0}(\mathcal{M})$.

SOLUTION of (2-10.k):

See the well-documented code in ➙ GitLab.

HINT 1 for (2-10.b): Recall the insights gained in [Lecture → Section 1.2.3.2] and [Lecture → Section 1.2.3.4], then appeal to [Lecture → Thm. 1.3.3.6]. ⌟

SOLUTION of (2-10.b):

- By the Poincaré-Friedrichs inequality of [Lecture → Thm. 1.3.4.17], the bilinear form is symmetric and positive definite on $H_0^1(\Omega)$.
- The right-hand-side linear form is continuous on with respect to the energy norm, since by the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)]

$$\int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \leq \|\mathbf{f}\|_{L^2(\Omega)}\, \|\mathbf{grad}\, v\|_{L^2(\Omega)} = \|\mathbf{f}\|_{L^2(\Omega)}\, \|v\|_{\mathsf{a}}\,, \quad \forall v \in H_0^1(\Omega)\,.$$

  As $\mathbf{f} \in C^0(\overline{\Omega})$, this function will certainly belong to $L^2(\Omega)$, because it is bounded.

SOLUTION of (2-10.c):

We can directly check orthogonality:

$$\int_\Omega \mathbf{f}_0(\boldsymbol{x}) \cdot \mathbf{f}_\perp(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{grad}\, u \cdot (\mathbf{f} - \mathbf{grad}\, u) \,\mathrm{d}\boldsymbol{x} = 0 \,,$$

because we can use the test function $v := u \in H_0^1(\Omega)$ in the variational formulation (2.10.5)!

HINT 1 for (2-10.d): Apply Green's first formula [Lecture → Thm. 1.5.2.7] to the linear functional on the right-hand side of the variational problem from Sub-problem (2-10.a). ⌟

SOLUTION of (2-10.d):

Following the instructions given in the hint, we can recast the right-hand-side functional as

$$\ell(v) = \int_\Omega \mathbf{f}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{f}(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad v \in H_0^1(\Omega)\,.$$

No boundary terms remain, because the test function vanishes on $\partial\Omega$.

As a consequence,

$$u \equiv 0 \quad \overset{(*)}{\Longleftrightarrow} \quad \int_\Omega \operatorname{div} \mathbf{f}(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H_0^1(\Omega) \quad \Longleftrightarrow \quad \operatorname{div} \mathbf{f} \equiv 0\,.$$

$(*)$: Since we deal with a linear problem with unique solutions, we obtain the trivial solution if and only if the right hand side is zero.

HINT 1 for (2-10.e): [Lecture → Code 2.7.4.35] and [Lecture → Code 2.7.4.37] hold the key to the solution. ⌟

HINT 2 for (2-10.e): You may just copy the implementation of the member functions of the class **lf::uscalfe::LinearFELaplaceElementMatrix**. ⌟

SOLUTION of (2-10.e):

This the minimal definition of a class that fits the concept of **ENTITY_MATRIX_PROVIDER**.

C++11 code 2.10.8: Sub-problem (2-10.e): Builder class for element matrix ⇝ GitLab

```
2  class ElementMatrixProvider {
3   public:
4    Eigen::Matrix3d Eval(const lf::mesh::Entity &entity);
5    bool isActive(const lf::mesh::Entity & /*entity*/) const { return true; }
6  };
```

The LEHRFEM++ compatible implementation of and [Lecture → Code 2.4.5.11] combined with [Lecture → Code 2.4.5.13]. See there for further explanations.

C++11 code 2.10.9: Computation of element matrix for Sub-problem (2-10.e) ⇝ GitLab

```
2  Eigen::Matrix3d ElementMatrixProvider::Eval(const lf::mesh::Entity &entity) {
3    LF_ASSERT_MSG(lf::base::RefEl::kTria() == entity.RefEl(),
4                  "Function only defined for triangular cells");
5
6    const lf::geometry::Geometry *geo_ptr = entity.Geometry();
7    Eigen::Matrix3d loc_mat;
8
9    // get area of the entity
10   const double area = lf::geometry::Volume(*geo_ptr);
11
12   const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_ptr);
13   // calculate the gradients of the basis functions.
14   // See [Lecture → Code 2.4.5.11], [Lecture → Code 2.4.5.13] for
        details.
15   Eigen::Matrix3d grad_helper;
16   grad_helper.col(0) = Eigen::Vector3d::Ones();
17   grad_helper.rightCols(2) = corners.transpose();
18   // Matrix with gradients of the local shape functions in its columns
19   const Eigen::MatrixXd grad_basis = grad_helper.inverse().bottomRows(2);
20
21   loc_mat = area * (grad_basis.transpose() * grad_basis);
22   return loc_mat;
23 }
```

SOLUTION of (2-10.f):

For a triangular cell K with vertices $\boldsymbol{a}_K^i$, $i = 1,2,3$, the element vector $\vec{\boldsymbol{\varphi}}_K$ for the finite element space $\mathcal{S}_1^0(\mathcal{M})$ and the right-hand-side functional ℓ from (2.10.4) reads

$$\vec{\boldsymbol{\varphi}}_K = |K|\mathbf{G}^\top \cdot \mathbf{f}\Big(\tfrac{1}{3}(\boldsymbol{a}_K^1 + \boldsymbol{a}_K^2 + \boldsymbol{a}_K^3)\Big)\,, \tag{2.10.10}$$

$$\mathbf{G}^\top f \begin{bmatrix} \mathbf{grad}\,\lambda_1 \cdot f \\ \mathbf{grad}\,\lambda_2 \cdot f \\ \mathbf{grad}\,\lambda_3 \cdot f \end{bmatrix},$$

$$(\vec{\boldsymbol{\varphi}}_K)_\ell = |K|\mathbf{f}(\boldsymbol{c}_K) \cdot \mathbf{grad}\,\lambda_\ell\,, \quad \boldsymbol{c}_K := \tfrac{1}{3}(\boldsymbol{a}_K^1 + \boldsymbol{a}_K^2 + \boldsymbol{a}_K^3)\,.$$

SOLUTION of (2-10.g):

Based on (2.10.10) the implementation is straightforward.

C++11 code 2.10.11: Solution code for Sub-problem (2-10.g) ⇢ GitLab

```cpp
template <typename FUNCTOR>
Eigen::Vector3d GradProjRhsProvider<FUNCTOR>::Eval(
    const lf::mesh::Entity &entity) {
  LF_ASSERT_MSG(lf::base::RefEl::kTria() == entity.RefEl(),
                "Function only defined for triangular cells");

  const lf::geometry::Geometry *geo_ptr = entity.Geometry();
  Eigen::Vector3d loc_vec;

  // get area of the entity
  const double area = lf::geometry::Volume(*geo_ptr);

  // calculate center of mass
  const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_ptr);
  const Eigen::Vector2d c = corners.rowwise().sum() / 3.;

  // value of the functor f at center of mass of the entity
  const Eigen::Vector2d func_value = f_(c);

  // calculate the gradients of the basis functions
  Eigen::Matrix3d grad_helper;
  grad_helper.col(0) = Eigen::Vector3d::Ones();
  grad_helper.rightCols(2) = corners.transpose();
  // vector of the gradients of the basis function evaluated at c
  const Eigen::MatrixXd grad_basis = grad_helper.inverse().bottomRows(2);

  loc_vec = area * (grad_basis.transpose() * func_value);
  return loc_vec;
}
```

HINT 1 for (2-10.h): According to the result of Sub-problem (2-10.a) we have to solve the linear variational problem (2.10.5), which is the weak formulation of a Dirichlet problem for the Laplacian. LEHRFEM++ supplies ample tools for dealing with homogeneous Dirichlet boundary conditions, for instance, the function `FixFlaggedSolutionComponents()`, see [Lecture → Section 2.7.6]. ⌟

SOLUTION of (2-10.h):

C++11 code 2.10.12: Function `projectionOntoGradients` for Sub-problem (2-10.h)
↪ GitLab

```
template <typename FUNCTOR>
Eigen::VectorXd projectOntoGradients(const lf::assemble::DofHandler &dofh,
                                     FUNCTOR f) {
  const lf::assemble::size_type N_dofs = dofh.NumDofs();
  Eigen::VectorXd sol_vec;

  // I. Build the (full) Galerkin matrix for the linear system.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  ProjectionOntoGradients::ElementMatrixProvider my_mat_provider;
  // co-dimension 0 because we locally assemble on cells
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, my_mat_provider, A);

  // II. Build the (full) right hand side vector
  Eigen::VectorXd phi(N_dofs);
  phi.setZero();
  ProjectionOntoGradients::GradProjRhsProvider my_vec_provider(f);
  // co-dimension 0 because we locally assemble on cells
  lf::assemble::AssembleVectorLocally(0, dofh, my_vec_provider, phi);

  // III. Enforce homogeneous dirichlet boundary conditions
  // We do this by selecting the DOFs on the boundary and setting the
  // values to zero.  Note, that we could also use this to set the
     boundary
  // to any other value (if bc were not homogeneous)

  // To select the right DOFs we need a selector:
  // * for all DOFs on the boundary return true and the value on the
     boundary
  // * for all other DOFs return false (and whatever as value)
  const double boundary_val = 0;
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(dofh.Mesh(), 2)};
  auto my_selector = [&dofh, &bd_flags, &boundary_val](unsigned int dof_idx) {
    if (bd_flags(dofh.Entity(dof_idx))) {
      return std::make_pair(true, boundary_val);
    } // interior node:  the value we return here does not matter
    return std::make_pair(false, 42.0);
  };
  // Since we know the values on the boundary we know the solution on
     these
  // DOFs and we can write the Galerkin LSE in block format and solve
     only
  // for the unknown coefficients.  This modification is done by the
     following
  // function FixFlaggedSolutionComponents().  We use the selector we
     have
  // defined above.
  // See [Lecture → Section 2.7.6] for explanations.
  lf::assemble::FixFlaggedSolutionComponents<double>(my_selector, A, phi);

  // IV. Solve the LSE using an Eigen solver
  // Convert from triplet to CRS format
  const Eigen::SparseMatrix<double> A_crs = A.makeSparse();
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
```

```
49    solver.compute(A_crs);
50    sol_vec = solver.solve(phi);
51    return sol_vec;
52  }
```

SOLUTION of (2-10.i):

C++11 code 2.10.14: Unit test for Sub-problem (2-10.i) ⇢ GitLab

```
TEST(ProjectionOntoGradients, div_free_test) {
  // Building test mesh
  auto mesh_p = lf::mesh::test_utils::GenerateHybrid2DTestMesh(3);
  // Divergence-free vector field
  const auto f = [](Eigen::Vector2d x) { return Eigen::Vector2d(-x(1), x(0)); };

  // DofHandler for the p.w.  linear Lagrangian finite element
  lf::assemble::UniformFEDofHandler dofh(mesh_p,
                                         {{lf::base::RefEl::kPoint(), 1},
                                          {lf::base::RefEl::kSegment(), 0},
                                          {lf::base::RefEl::kTria(), 0}});
  // Compute solution
  const Eigen::VectorXd sol_vec =
      ProjectionOntoGradients::projectOntoGradients(dofh, f);

  // As stated in the exercise, we would expect the solution to be zero.
  // Hence we check every entry if its (numerically) zero.
  // The GoogleTest framework provides a function we may use:
  /*  EXPECT_NEAR(value 1, value 2, max. difference) */
  const double eps = 1e-15;
  for (std::size_t i = 0; i < sol_vec.size(); ++i) {
    EXPECT_NEAR(sol_vec[i], 0.0, eps);
    // Try testing for equality, you'll see it will fail miserably!
    /* EXPECT_EQ(sol_vec[i], 0.0); */
  }
}
```

HINT 1 for (2-11.a): See [Lecture → Rem. 2.6.2.10].

SOLUTION of (2-11.a):

The dimension is 13. There is one basis function for each node (vertex).

HINT 1 for (2-11.b): All the square elements are equal, and they have side lengths $1/\sqrt{2}$. Number the nodes either clockwise or counterclockwise around the square (due to symmetry, any such numbering should yield the same matrix). There are two ways to compute their corresponding element matrices and you may choose either of them:

1. direct evaluation of the localized bilinear form a_K for pairs of local shape functions. Note that their gradients are not constant this time, but their components are still linear in one direction.
2. computation of the Galerkin matrix on the unit square, and subsequent transformation. See [Lecture $\rightarrow$ Eq. (2.6.2.3)] for the basis functions on the unit square. [Lecture $\rightarrow$ Section 2.8.3] explains transformation techniques. Your transformation $\mathbf{\Phi}_K$ in this case will simply be a scaling.

⌟

HINT 2 for (2-11.b): For validating your result: Since the local shape functions add up to the constant function $x \to 1$, the row and column sums of the element matrix must be equal to zero. ⌟

SOLUTION of (2-11.b):

Obviously, translation of a square does not affect the element matrix at all.

1. The **direct computation** of the entries of the element matrix relies on the bilinear nodal basis functions on a generic square $[0,a]^2$, which are just scaled version of those given in [Lecture → Eq. (2.6.2.3)]. Integration over the square for the evaluation of $\mathsf{a}_K(b_K^j, b_K^i)$ is easy.

2. Using the **transformation technique** we get

$$(\mathbf{A}_{\widehat{K}})_{i,j} = \int_{\widehat{K}} \mathbf{grad}\, \widehat{b}^i(\widehat{\mathbf{x}}) \cdot \mathbf{grad}\, \widehat{b}^j(\widehat{\mathbf{x}})\, \mathrm{d}\widehat{\mathbf{x}},$$

where the transformation $\mathbf{\Phi}_K : \widehat{K} \mapsto K$ is given by a simple scaling

$$\mathbf{x} = \mathbf{\Phi}_K(\widehat{\mathbf{x}}) := \begin{bmatrix} 1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} \end{bmatrix} \widehat{\mathbf{x}}\,.$$

The Jacobian of the transformation is

$$D\mathbf{\Phi}(\widehat{\mathbf{x}}) = D\mathbf{\Phi} = \begin{bmatrix} 1/\sqrt{2} & 0 \\ 0 & 1/\sqrt{2} \end{bmatrix},$$

and its determinant $|\det(D\mathbf{\Phi}(\widehat{\mathbf{x}}))| = |\det(D\mathbf{\Phi})| = 1/2$. Note that we suppressed the $\widehat{x}$ argument because the expression are constant. Due to symmetry, we only have to evaluate this integral for three different choices of i, j. Note that with the standard node numbering from [Lecture → Eq. (2.6.2.3)] we get

$$\begin{aligned}
\mathbf{grad}\, \widehat{b}^1(\widehat{x}_1, \widehat{x}_2) &= [\widehat{x}_2 - 1, \widehat{x}_1 - 1]^\top\,, \\
\mathbf{grad}\, \widehat{b}^2(\widehat{x}_1, \widehat{x}_2) &= [1 - \widehat{x}_2, -\widehat{x}_1]^\top\,, \\
\mathbf{grad}\, \widehat{b}^3(\widehat{x}_1, \widehat{x}_2) &= [\widehat{x}_2, \widehat{x}_1]^\top\,, \\
\mathbf{grad}\, \widehat{b}^4(\widehat{x}_1, \widehat{x}_2) &= [-\widehat{x}_2, 1 - \widehat{x}_1]^\top.
\end{aligned}$$

The computation is

$$\begin{aligned}
(\mathbf{A}_{\widehat{K}})_{i,i} = (\mathbf{A}_{\widehat{K}})_{3,3} &= \int_{\widehat{K}} \left(\widehat{x}_1^2 + \widehat{x}_2^2\right) \mathrm{d}\widehat{\mathbf{x}} = 2\int_0^1 \widehat{x}_1^2\, \mathrm{d}\widehat{x}_1 = \frac{2}{3}, \\
(\mathbf{A}_{\widehat{K}})_{i,i\pm 2} = (\mathbf{A}_{\widehat{K}})_{1,3} &= \int_{\widehat{K}} (\widehat{x}_1(1-\widehat{x}_1) + \widehat{x}_2(1-\widehat{x}_2))\, \mathrm{d}\widehat{\mathbf{x}} \\
&= 2\int_0^1 \widehat{x}_1(1-\widehat{x}_1)\, \mathrm{d}\widehat{x}_1 = -\frac{1}{3}, \\
(\mathbf{A}_{\widehat{K}})_{i,i\pm 1} = (\mathbf{A}_{\widehat{K}})_{i,i\pm 3} = (\mathbf{A}_{\widehat{K}})_{1,4} &= -\int_{\widehat{K}} \Big((\widehat{x}_1 - 1)^2 + \widehat{x}_2(\widehat{x}_2 - 1)\Big)\, \mathrm{d}\widehat{\mathbf{x}} \\
&= -\int_0^1 (\widehat{x}_1 - 1)^2\, \mathrm{d}\widehat{x}_1 - \int_0^1 \widehat{x}_2(\widehat{x}_2 - 1)\, \mathrm{d}\widehat{x}_2 = -\frac{1}{6}.
\end{aligned}$$

So in the end we get

$$\mathbf{A}_{\widehat{K}} = \frac{1}{6}\begin{bmatrix} 4 & -1 & -2 & -1 \\ -1 & 4 & -1 & -2 \\ -2 & -1 & 4 & -1 \\ -1 & -2 & -1 & 4 \end{bmatrix}. \tag{2.11.3}$$

Last but not least, it remains to apply the transformation (see [Lecture → Eq. (2.8.3.12)])

$$\mathbf{A}_K = \underbrace{D\boldsymbol{\Phi}^{-1}\, D\boldsymbol{\Phi}^{-T}\, |\det(D\boldsymbol{\Phi})|}_{=\mathbf{I}}\, \mathbf{A}_{\widehat{K}} = \mathbf{A}_{\widehat{K}}\,.$$

For further details we refer to [Lecture → Section 2.8.3].

Remark. A consistency check for (2.11.3) is that the row/column sums must all evaluate to zero (why?). See [Lecture → § 2.4.5.5], [Lecture → Eq. (2.4.5.7)] for an explanation.

HINT 1 for (2-11.c): Note that the triangle has side lengths $1/\sqrt{2}$, $1 - 1/\sqrt{2}$ and $\sqrt{2 - \sqrt{2}}$, and consider the remarks [Lecture → Rem. 2.4.5.9] and [Lecture → Suppl. 2.4.5.14]. Use the local node numbering inherited from the global one (i.e. vertex 1 is number 1, and so on). ⌟

HINT 2 for (2-11.c): The formula $\left[\text{Lecture} \rightarrow \text{Eq. (2.4.5.8)}\right]$ may be helpful. ⌟

SOLUTION of (2-11.c):

We use the formula [Lecture → Eq. (2.4.5.8)]. The cotangents are (pay attention to the numbering of the vertices and the angles!)

$$\begin{aligned}\cot\omega_1 &= \sqrt{2}-1,\\ \cot\omega_2 &= \frac{1}{\sqrt{2}-1},\\ \cot\omega_3 &= 0,\end{aligned}$$

giving

$$\begin{aligned}\mathbf{A}_K &= \frac{1}{2}\begin{bmatrix}\cot\omega_3+\cot\omega_2 & -\cot\omega_3 & -\cot\omega_2\\ -\cot\omega_3 & \cot\omega_3+\cot\omega_1 & -\cot\omega_1\\ -\cot\omega_2 & -\cot\omega_1 & \cot\omega_2+\cot\omega_1\end{bmatrix}\\ &= \frac{1}{2(\sqrt{2}-1)}\begin{bmatrix}1 & 0 & -1\\ 0 & (\sqrt{2}-1)^2 & -(\sqrt{2}-1)^2\\ -1 & -(\sqrt{2}-1)^2 & (\sqrt{2}-1)^2+1\end{bmatrix}.\end{aligned}$$

An alternative would be to solve [Lecture → Eq. (2.4.5.10)] for the given vertex coordinates. Then you can execute [Lecture → Code 2.4.5.13] “manually”.

HINT 1 for (2-11.d): The computations are the same as in [Lecture → § 2.4.6.9]. ⌟

SOLUTION of (2-11.d):

Note that the area of the quadrilaterals are $1/2$. The quadrature rule should then give

$$(\vec{\boldsymbol{\varphi}}_K)_i = \frac{1}{8} f(\mathbf{a}^i) \ ,$$

since the global shape functions are a cardinal basis with respect to the nodes of the mesh.

HINT 1 for (2-11.e): Do a **cell-oriented assembly** "by hand" (see [Lecture → Section 2.7.4]). For each pair of neighboring vertices i, j, walk through the elements shared by i and j, find the local element contribution from subproblems (2-11.b) or (2-11.c) and sum them up. ⌟

HINT 2 for (2-11.e): Use that the element matrices for congruent cells agree. Thus you can just use the results of (2-11.b) and (2-11.c). ⌟

HINT 3 for (2-11.e): Draw the mesh on a piece of paper and write the element matrix onto each cell:

- For one of the squares we have

$$\mathbf{A}_K = \frac{1}{6}\begin{bmatrix} 4 & -1 & -2 & -1 \\ -1 & 4 & -1 & -2 \\ -2 & -1 & 4 & -1 \\ -1 & -2 & -1 & 4 \end{bmatrix}.$$

- For a triangle K:

$$\mathbf{A}_K = \frac{1}{2(\sqrt{2}-1)}\begin{bmatrix} 1 & 0 & -1 \\ 0 & (\sqrt{2}-1)^2 & -(\sqrt{2}-1)^2 \\ -1 & -(\sqrt{2}-1)^2 & (\sqrt{2}-1)^2+1 \end{bmatrix}.$$

SOLUTION of (2-11.e):

Define $p = 1/6$ and $q = \sqrt{2} - 1$. Then,

$$\mathbf{A} = \begin{bmatrix}
\frac{1}{q} & 0 & -\frac{1}{q} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 4p+q & -p-\frac{q}{2} & 0 & 0 & -p-\frac{q}{2} & -2p & 0 & 0 & 0 & 0 & 0 & 0 \\
-\frac{1}{q} & -p-\frac{q}{2} & 8p+\frac{1}{q}(q^2+1) & -p-\frac{q}{2} & 0 & -2p & -2p & -2p & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -p-\frac{q}{2} & 4p+q & 0 & 0 & -2p & -p-\frac{q}{2} & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{q} & -\frac{1}{q} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -p-\frac{q}{2} & -2p & 0 & -\frac{1}{q} & 8p+\frac{1}{q}(q^2+1) & -2p & 0 & 0 & -p-\frac{q}{2} & -2p & 0 & 0 \\
0 & -2p & -2p & -2p & 0 & -2p & 16p & -2p & 0 & -2p & -2p & -2p & 0 \\
0 & 0 & -2p & -p-\frac{q}{2} & 0 & 0 & -2p & 8p+\frac{1}{q}(q^2+1) & -\frac{1}{q} & 0 & -2p & -p-\frac{q}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{q} & \frac{1}{q} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -p-\frac{q}{2} & -2p & 0 & 0 & 4p+q & -p-\frac{q}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & -2p & -2p & -2p & 0 & -p-\frac{q}{2} & 8p+\frac{1}{q}(q^2+1) & -p-\frac{q}{2} & -\frac{1}{q} \\
0 & 0 & 0 & 0 & 0 & 0 & -2p & -p-\frac{q}{2} & 0 & 0 & -p-\frac{q}{2} & 4p+q & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{1}{q} & 0 & \frac{1}{q}
\end{bmatrix}$$

SOLUTION of (2-11.f):

The quadrature rule gives the following formula for the triangle contributions:

$$(\vec{\boldsymbol{\varphi}}_K)_i = \frac{r}{8} f(\mathbf{a}^i),$$

where $r = 2(\sqrt{2}-1)/3$. That should give the following right-hand side vector:

$$\begin{aligned} \varphi_1 = \varphi_5 = \varphi_9 = \varphi_{13} &= 2rf(\mathbf{a}^i)/8, \\ \varphi_2 = \varphi_4 = \varphi_{10} = \varphi_{12} &= (2r+1)f(\mathbf{a}^i)/8, \\ \varphi_3 = \varphi_6 = \varphi_8 = \varphi_{11} &= (2r+2)f(\mathbf{a}^i)/8, \\ \varphi_7 &= 4f(\mathbf{a}^7)/8, \end{aligned}$$

where in each case, i will have to be replaced with the relevant node number.

SOLUTION of (2-11.g):

Let N_V, N_E, N_T, N_Q be the number of vertices, edges, triangles and quadrilaterals respectively. Then the dimension of $\mathcal{S}_2^0(\mathcal{M})$ is $N_V + N_E + N_Q = 41$.

HINT 1 for (2-11.h): In light of the supports of global shape functions, which pairs of them can interact in the bilinear form? Here, “interact” means having overlapping supports. ⌟

SOLUTION of (2-11.h):

We must be careful in order not to count non-zero entries twice, which would render the obtained bound loose. Below we refer to the local/global shape functions through the mesh entities they are associated with.

1. The number of *diagonal entries*, which are naturally non-zero, agrees with the dimension of $\mathcal{S}_2^0(\mathcal{M})$, which is derived in Sub-problem (2-11.g)

   What remains is to count non-zero *off-diagonal* entries.

2. We next determine the number of non-zero off-diagonal entries contributed by the global shape functions covering an edge. There are three for each edge, associated with its two endpoints and the midpoint, and their supports will certainly overlap.

   This yields 6 non-zero off-diagonal entries for every edge.

3. Last, we have to take into account interactions "through the interior" of cells. We have to ignore interactions "along edges" because we have already counted them.

   Triangular cell. Each vertex entertains exactly one interaction "through the interior" with the midpoint of the opposite edge. Each midpoints is coupled "through the interior" with the opposite vertex and the two other midpoints. Thus, the total number of interactions "through the interior" of a triangle K is

   $$\sharp\mathcal{V}(K) + 3 \cdot \sharp\mathcal{E}(K) = 3 + 3 \cdot 3 = 12 \,.$$

   Quadrilateral cell. Each vertex is involved in 4 interior interactions with the opposite vertex, the two midpoints of the non-adjacent edges and the center point. Each midpoint interacts "through the interior" with two vertices, the three midpoints of all the other edges and the center point, which amounts to 6 interior interactions. The center point is connected with all other 8 local shape functions. This yields

   $$4 \cdot \sharp\mathcal{V}(K) + 6 \cdot \sharp\mathcal{E}(K) + 8 = 4 \cdot 4 + 6 \cdot 4 + 8 = 48$$

   interactions "through the interior" of a quadrilateral K.

Summing up, we obtain the bound

$$\begin{aligned} \mathrm{nnz}(\mathbf{A}) &\leq (N_V + N_E + N_Q) + 6 \cdot N_E + 12 \cdot N_T + 48 N_Q \\ &\leq N_V + 7 \cdot N_E + 12 \cdot N_T + 49 \cdot N_Q \,. \end{aligned}$$

In particular, for the hybrid mesh from Fig. 11 we get 473 nonzero entries.

SOLUTION of (2-12.a):

Just read [Lecture $\rightarrow$ Def. 2.7.5.29]. Note the different spaces of polynomials used for triangles and rectangles. Be aware that LEHRFEM++ uses “degree of polynomial exactness” insteadt of “order”.

HINT 1 for (2-12.b): Use analytic formulas for the integration of polynomials over a simplex, see [Lecture → Lemma 2.7.5.5]. ⌟

SOLUTION of (2-12.b):

For triangles (TRIA), a quadrature is of order k if it can integrate exactly all *polynomials* $f \in \mathcal{P}_{k-1}(\mathbb{R}^2)$. Then, we have to test if the quadrature gets the exact result for

$$I_T(i,j) := \int_{\widehat{K}} x^i y^j \, \mathrm{d}\boldsymbol{x} \quad \forall i+j \leq k-1.$$

Note that on the unit triangle $\widehat{K}$ the coordinate functions are also baryccentric coordinate functions:

$$\lambda_2(\boldsymbol{x}) = x \quad , \quad \lambda_3(\boldsymbol{x}) = y \, , \quad \boldsymbol{x} = \begin{bmatrix} x \\ x \end{bmatrix} \in \widehat{K} \, .$$

Then we can directly apply [Lecture → Lemma 2.7.5.5] for $d = 2$:

$$\int_{\widehat{K}} x^i y^j \, \mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} \lambda_2^i \lambda_3^j \, \mathrm{d}\boldsymbol{x} = \frac{i!j!}{(i+j+2)!}, \quad i+j \leq k-1.$$

We loop over all pairs (i,j) with $i+j < k$ and test for (approximate) equality, because we are comparing results of floating point computations. A single failure of the test will make the function return with **false**.

C++11 code 2.12.2: Sub-problem (2-12.b): function `testQuadOrderTria()` ⇝ GitLab

```
2  bool testQuadOrderTria(const lf::quad::QuadRule &quad_rule,
3                         unsigned int order) {
4    bool order_isExact = true;  // return variable
5    double my_epsilon = 1e-12;
6    // Retrieve the passed quadrature rule's reference element
7    // NOLINTBEGIN(clang-analyzer-deadcode.DeadStores)
8    const lf::base::RefEl ref_element = quad_rule.RefEl();
9    // NOLINTEND(clang-analyzer-deadcode.DeadStores)
10   // Check that the passed reference element is triangular
11   assert(ref_element == lf::base::RefElType::kTria);
12   // A quadrature rule involves quadrature nodes and weights defined so that
13   // the weighted sum of the value of a function at these points approximates
14   // the integral of that function.
15   const Eigen::VectorXd weights = quad_rule.Weights();
16   const Eigen::MatrixXd points = quad_rule.Points();  // (x,y) points
17   const Eigen::VectorXd x_coords = points.row(0);
18   const Eigen::VectorXd y_coords = points.row(1);
19   // A quadrature rule over a two dimensional domain is of order k if it can
20   // integrate exactly all bivariate polynomials of order k-1. The collection of
21   // such polynomials is spanned by the set of homogeneous polynomials of order
22   // strictly less than k, i.e. by the polynomials of the form
23   /* p_IJ(x,y) = (x^I)(y^J), I+J < k: */
24   auto eval_p_IJ = [&x_coords, &y_coords](int I, int J) -> Eigen::VectorXd {
25     return x_coords.array().pow(I) * y_coords.array().pow(J);
26   }; /* evaluates p_IJ at all points (x,y) as defined by quad_rule */
27
28   /* Compare analytical value and quadrature sum for all p_IJ, I+J < k */
29   double exact_integral;  // analytical value of the integral
30   double quad_rule_sum;   // weighted sum used for approximating the integral
31   for (int I = 0; I < order; I++) {
32     for (int J = 0; J < order - I; J++) {
33       exact_integral = factorial(I) * factorial(J) / factorial(I + J + 2);
34       quad_rule_sum = eval_p_IJ(I, J).dot(weights);
```

```
35
36        // Check if the difference bewteen the results is within tolerance
37        order_isExact = fabs(exact_integral - quad_rule_sum) <=
38                        fabs(exact_integral) * my_epsilon;
39        if (!order_isExact) {
40          return order_isExact;
41        }
42      }
43    }
44    return order_isExact;
45  }
```

SOLUTION of (2-12.c):

For quadrilaterals (QUAD), a quadrature is of order k if it can integrate exactly all *tensor product polynomials* $f \in \mathcal{Q}_{k-1}(\mathbb{R}^2)$. Then, we have to test if the quadrature gets the exact result for

$$I_Q(i,j) := \int_{\hat{Q}} x^i y^j \, \mathrm{d}\boldsymbol{x} = \frac{1}{(i+1)(j+1)} \quad \forall i,j = 0 \ldots k-1 \, .$$

For the test we let i and j run from 1 to k in two nested loops. Again, approximate comparison of floating point values has to be employed.

C++11 code 2.12.3: Sub-problem (2-12.c): function `testQuadOrderQuad()` ⇝ GitLab

```
bool testQuadOrderQuad(const lf::quad::QuadRule &quad_rule,
                       unsigned int order) {
  bool order_isExact = true;  // return variable

  double my_epsilon = 1e-12;
  // Retrieve the passed quadrature rule's reference element
  // NOLINTBEGIN(clang-analyzer-deadcode.DeadStores)
  const lf::base::RefEl ref_element = quad_rule.RefEl();
  // NOLINTEND(clang-analyzer-deadcode.DeadStores)
  // Check that the passed reference element is triangular
  assert(ref_element == lf::base::RefElType::kQuad);
  // A quadrature rule consists of quadrature nodes and weights defined so that
  // the weighted sum of the value of a function at these points approximates
  // the integral of that function.
  const Eigen::VectorXd weights = quad_rule.Weights();
  const Eigen::MatrixXd points = quad_rule.Points();  // (x,y) points
  const Eigen::VectorXd x_coords = points.row(0);
  const Eigen::VectorXd y_coords = points.row(1);
  // A quadrature rule over a two dimensional domain is of order k if it can
  // integrate exactly all bivariate polynomials of order k-1. The collection of
  // such polynomials is spanned by the set of homogeneous polynomials of order
  // strictly less than k, i.e. by the polynomials of the form
  /* p_IJ(x,y) = (1-x)^I(y^J), I,J < k: */
  auto eval_p_IJ = [&x_coords, &y_coords](int I, int J) -> Eigen::VectorXd {
    return x_coords.array().pow(I) * y_coords.array().pow(J);
  }; /* evaluates p_IJ at all points (x,y) as defined by quad_rule */

  /* Compare the analytical value and the quadrature sum for all p_IJ, I+J < k
   */
  double exact_integral;  // analytical value of the integral
  double quad_rule_sum;   // weighted sum used for approximating the integral
  for (int I = 0; I < order; I++) {
    for (int J = 0; J < order; J++) {
      exact_integral = 1.0 / ((I + 1.0) * (J + 1.0));
      quad_rule_sum = eval_p_IJ(I, J).dot(weights);

      // Check if the difference bewteen the results is within tolerance
      order_isExact = fabs(exact_integral - quad_rule_sum) <=
                      fabs(exact_integral) * my_epsilon;
      if (!order_isExact) {
        return order_isExact;
      }
    }
  }
```

```
46     return order_isExact;
47 }
```

HINT 1 for (2-12.d): Note that you have to find the *maximal* degree of polynomials for which the quadrature rule still provides exact integration. To that end you have to test increasing degrees until you reach one, for which exact integration by the quadrature rule is no longer guaranteed. Start with the order that the quadrature rule itself purports to have. ⌟

SOLUTION of (2-12.d):

The implementation relying on `testQuadOrderTria()` and `testQuadOrderQuad()` is straightforward.

C++11 code 2.12.4: Sub-problem (2-12.d): function `testQuadOrderQuad()` ⇝ GitLab

```
2  unsigned int calcQuadOrder(const lf::quad::QuadRule &quad_rule) {
3    unsigned int maximal_order = quad_rule.Order();
4
5    // Retrieve the passed quadrature rule's reference element
6    const lf::base::RefEl ref_element = quad_rule.RefEl();
7
8    if (ref_element == lf::base::RefElType::kTria) {
9      assert(testQuadOrderTria(quad_rule, maximal_order));
10     while (testQuadOrderTria(quad_rule, maximal_order + 1)) {
11       maximal_order++;
12     }
13   }
14
15   if (ref_element == lf::base::RefElType::kQuad) {
16     assert(testQuadOrderQuad(quad_rule, maximal_order));
17     while (testQuadOrderQuad(quad_rule, maximal_order + 1)) {
18       maximal_order++;
19     }
20   }
21   return maximal_order;
22 }
```

SOLUTION of (2-13.a):

We start with an application of Green's formula.

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad \text{[Lecture → Eq. (13.2.3.2)]}$$

$$\int_\Omega -\operatorname{div}\big((\mathbf{I} + \boldsymbol{d}(\boldsymbol{x})\boldsymbol{d}(\boldsymbol{x})^\top)\,\mathbf{grad}\, u\big) v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n} v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) + \int_{\partial\Omega} w^2(\boldsymbol{x}) u(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega \frac{v(\boldsymbol{x})}{1 + w(\boldsymbol{x})^2} \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,,$$

Then recall [Lecture → Section 1.5.3] and [Lecture → Ex. 1.8.0.6] to understand two-step argument that follows:

1. Testing with $v \in C_0^\infty(\Omega)$ we can extract the PDE:
$$-\operatorname{div}\big((\mathbf{I} + \boldsymbol{d}(\boldsymbol{x})\boldsymbol{d}(\boldsymbol{x})^\top)\,\mathbf{grad}\, u\big) = \frac{1}{1 + w(\boldsymbol{x})^2} \quad \text{in} \quad \Omega\,.$$
2. Then, testing with general $v \in C^\infty(\overline{\Omega})$ and using this PDE we obtain the **impedance boundary conditions** [Lecture → Eq. (1.7.0.6)]
$$(\mathbf{I} + \boldsymbol{d}(\boldsymbol{x})\boldsymbol{d}(\boldsymbol{x})^\top)\,\mathbf{grad}\, u \cdot \boldsymbol{n} + w(\boldsymbol{x})^2 u = 0 \quad \text{on} \quad \partial\Omega\,.$$

This is a diffusion problem with spatially varying uniformly positive definite diffusion tensor $\boldsymbol{x} \mapsto \mathbf{I} + \boldsymbol{d}(\boldsymbol{x})\boldsymbol{d}(\boldsymbol{x}$

HINT 1 for (2-13.b): Remember that we deal with a parametric finite element method and employ the techniques and LEHRFEM++ tools presented in [Lecture → § 2.8.3.6] and [Lecture → § 2.8.3.15]. ⌟

SOLUTION of (2-13.b):

Adapting the general formula [Lecture → Eq. (2.8.3.13)] we have to evaluate

$$(\mathbf{A}_K)_{i,j} = \frac{1}{\sharp\mathcal{V}(K)} \sum_{\ell=1}^{\sharp\mathcal{V}(K)} \left(\mathbf{I} + \boldsymbol{d}(\boldsymbol{\Phi}(\widehat{\boldsymbol{m}}_\ell)\boldsymbol{d}(\boldsymbol{\Phi}(\widehat{\boldsymbol{m}}_\ell))^\top\right)(\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{m}}_\ell))^{-\top}\,\mathbf{grad}\,\widehat{b}^i(\widehat{\boldsymbol{m}}_\ell))\cdot$$
$$((\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{m}}_\ell))^{-\top}\,\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{m}}_\ell))\,|\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{m}}_\ell)|\,,$$

where $\widehat{\boldsymbol{m}}_\ell$ are the midpoints of the edges of the reference element $\widehat{K}$.

This formula implements (2.13.2) only in the case of triangular K, but not in the case of quadrilaterals. Why not?

The following codes Code 2.13.3 and Code 2.13.4 show a very "explicit" implementation of the local computations. The same computations can be accomplished with fewer loops using matrix operations. In addition, more quantities associated with the reference elements can be precomputed. This most efficient implementation is demonstrated in [Lecture → Ex. 2.8.3.29], [Lecture → Code 2.8.3.31].

C++11 code 2.13.3: Sub-problem (2-13.b): Implementation of `Eval()` member function of AnisotropicDiffusionElementMatrixProvider *for triangles* ⇝ GitLab

```
2       // I. OBTAIN COORDINATES OF THE MIDPOINTS
3       // I.i Hard-code the midpoints of the edges on the reference triangle
4       Eigen::Matrix<double, 2, 3> midpoints_ref(2, 3);
5       midpoints_ref << 0.5, 0.5, 0, 0, 0.5, 0.5;
6       // I.ii Obtain the midpoints of the parametrized triangle
7       auto midpoints_param = cell_geometry->Global(midpoints_ref);
8
9       // II. COMPUTE LOCAL INTEGRATION DATA
10      // II.i Compute the inverse of the tranposed Jacobian
11      //     (J^T)^{-1} = J*(J^T*J)^{-1}
12      // of the parametrization map at the midpoints of the reference triangle
13      const Eigen::MatrixXd JinvT(
14          cell_geometry->JacobianInverseGramian(midpoints_ref));
15      // II.ii Compute the integration element
16      //     integration_element(x) = sqrt(det(J^T*J))
17      // where J is the Jacobian of the parametrization map
18      const Eigen::VectorXd integration_element(
19          cell_geometry->IntegrationElement(midpoints_ref));
20      // II.iii Hard-code the gradients of the reference basis functions
21      Eigen::Matrix<double, 2, 3> gradients_ref(2, 3);
22      gradients_ref << -1, 1, 0, -1, 0, 1;
23
24      // III. PERFORM NUMERICAL QUADRATURE
25      element_matrix = Eigen::MatrixXd::Zero(3, 3);
26      for (int i = 0; i < 3; i++) {  // for each local degree of freedom
27        // III.i Evaluate the diffusion tensor
28        Eigen::Vector2d anisotropy_vec =
29            anisotropy_vec_field_(midpoints_param.col(i));
30        Eigen::Matrix2d diffusion_tensor =
31            Eigen::Matrix2d::Identity(2, 2) +
32            anisotropy_vec * anisotropy_vec.transpose();
33        // III. ii Compute gradients of the global basis functions
34        Eigen::MatrixXd gradients_param(2, 3);
35        gradients_param = JinvT.block(0, 2 * i, 2, 2) * gradients_ref;
```

```
      // III. iii Compute local contribution to the element matrix
      for (int j = 0; j < 3; j++) {
        for (int k = 0; k < 3; k++) {
          double integrand = (gradients_param.col(j).transpose() *
                              diffusion_tensor * gradients_param.col(k));
          element_matrix(j, k) += integration_element(i) * integrand;
        }
      }
    }
    element_matrix *= 1. / 6.;
    break;
```

C++11 code 2.13.4: Sub-problem (2-13.b): Implementation of `Eval()` member function of AnisotropicDiffusionElementMatrixProvider *for quadrilaterals* ↬ GitLab

```
    // I. OBTAIN COORDINATES OF THE MIDPOINTS
    // I.i Hard-code the midpoints of the edges on the reference quadrilateral
    Eigen::Matrix<double, 2, 4> midpoints_ref(2, 4);
    midpoints_ref << 0.5, 1, 0.5, 0, 0, 0.5, 1, 0.5;
    // I.ii Obtain the midpoints of the parametrized quadrilateral
    auto midpoints_param = cell_geometry->Global(midpoints_ref);

    // II. COMPUTE LOCAL INTEGRATION DATA
    // II.i Compute the inverse of the tranposed Jacobian of the
    //      (J^T)^{-1} = J*(J^T*J)^{-1}
    // parametrization map at the midpoints of the reference quadrilateral
    const Eigen::MatrixXd JinvT(
        cell_geometry->JacobianInverseGramian(midpoints_ref));
    // II.ii Compute the integration element
    //      integration_element(x) = sqrt(det(J^T*J))
    // where J is the Jacobian of the parametrization map
    const Eigen::VectorXd integration_element(
        cell_geometry->IntegrationElement(midpoints_ref));
    // II.iii Hard-code a matrix-valued function that returns the gradients of
    // the reference basis functions evaluated at coordinates
    auto gradients_ref =
        [](Eigen::Vector2d x) -> Eigen::Matrix<double, 2, 4> {
      Eigen::Matrix<double, 2, 4> element_matrix;
      element_matrix(0, 0) = x(1) - 1;
      element_matrix(1, 0) = x(0) - 1;
      element_matrix(0, 1) = 1 - x(1);
      element_matrix(1, 1) = -x(0);
      element_matrix(0, 2) = x(1);
      element_matrix(1, 2) = x(0);
      element_matrix(0, 3) = -x(1);
      element_matrix(1, 3) = 1 - x(0);
      return element_matrix;
    };

    // III. PERFORM NUMERICAL QUADRATURE
    element_matrix = Eigen::MatrixXd::Zero(4, 4);
    for (int i = 0; i < 4; i++) {  // for each local degree of freedom
      // III.i Evaluate the diffusion tensor
      Eigen::Vector2d anisotropy_vec =
          anisotropy_vec_field_(midpoints_param.col(i));
      Eigen::Matrix2d diffusion_tensor =
```

```
43              Eigen::Matrix2d::Identity(2, 2) +
44              anisotropy_vec * anisotropy_vec.transpose();
45          // III.ii Compute gradients of the global basis functions
46          Eigen::Matrix<double, 2, 4> gradients_param{
47              JinvT.block(0, 2 * i, 2, 2) * gradients_ref(midpoints_ref.col(i))};
48          // III.iii Compute local contribution to the element matrix
49          for (int j = 0; j < 4; j++) {
50            for (int k = 0; k < 4; k++) {
51              double integrand = (gradients_param.col(j).transpose() *
52                                  diffusion_tensor * gradients_param.col(k));
53              element_matrix(j, k) += integration_element(i) * integrand;
54            }
55          }
56        }
57        element_matrix *= 1. / 4.;
58        break;
```

Note that that the boundary term in (2.13.1) is not taken into account in Code 2.13.3 and Code 2.13.4.

Answer to the “Why not” question above: For a triangle K the mapping Φ is affine, which means that $\det D\Phi$ is constant and agrees with $2|K|$. This is not the case for a quadrilateral, for which Φ is bilinear and $\det D\Phi$ is linear, see [Lecture $\rightarrow$ Eq. (2.8.3.18)].

SOLUTION of (2-13.c):

C++11 code 2.13.5: Sub-problem (2-13.c): Implementation of `Eval()` member function of FESourceElemVecProvider *for triangles* ↪ GitLab

```
      // I. OBTAIN COORDINATES OF THE MIDPOINTS
      // I.i Hard-code the midpoints of the edges on the reference triangle
      Eigen::MatrixXd midpoints_ref(2, 3);
      midpoints_ref << 0.5, 0.5, 0, 0, 0.5, 0.5;
      // I.ii Obtain the midpoints of the parametrized triangle
      auto midpoints_param = cell_geometry->Global(midpoints_ref);

      // II. COMPUTE LOCAL INTEGRATION DATA
      // II.i Compute the integration element
      //     integration_element(x) = sqrt(det(J^T*J))
      // where J is the Jacobian of the parametrization map
      const Eigen::VectorXd integration_element(
          cell_geometry->IntegrationElement(midpoints_ref));
      // II.ii Evaluate w(x) at the midpoints
      Eigen::VectorXd w(3);
      for (int i = 0; i < 3; i++) {
        w(i) = coeff_expansion_(cell_global_idx[i]);
      }

      // III. PERFORM NUMERICAL QUADRATURE
      element_vector = Eigen::VectorXd::Zero(3);
      for (int i = 0; i < 3; i++) {  // for each local degree of freedom
        element_vector(i) +=
            integration_element(i) *
            (0.5 / (1. + std::pow(0.5 * w(i) + 0.5 * w((i + 1) % 3), 2)));
        element_vector((i + 1) % 3) +=
            integration_element(i) *
            (0.5 / (1. + std::pow(0.5 * w(i) + 0.5 * w((i + 1) % 3), 2)));
      }
      element_vector *= 1. / 6.;
      break;
```

C++11 code 2.13.6: Sub-problem (2-13.c): Implementation of `Eval()` member function of FESourceElemVecProvider *for quadrilaterals* ↪ GitLab

```
      // I. OBTAIN COORDINATES OF THE MIDPOINTS
      // I.i Hard-code the midpoints of the edges on the reference quadrilateral
      Eigen::Matrix<double, 2, 4> midpoints_ref(2, 4);
      midpoints_ref << 0.5, 1, 0.5, 0, 0, 0.5, 1, 0.5;
      // I.ii Obtain the midpoints of the parametrized quadrilateral
      auto midpoints_param = cell_geometry->Global(midpoints_ref);

      // II. COMPUTE LOCAL INTEGRATION DATA
      // II.i Compute the integration element
      //     integration_element(x) = sqrt(det(J^T*J))
      // where J is the Jacobian of the parametrization map
      const Eigen::VectorXd integration_element(
          cell_geometry->IntegrationElement(midpoints_ref));
      // II.ii Evaluate w(x) at the midpoints
```

```
16        Eigen::VectorXd w(4);
17        for (int i = 0; i < 4; i++) {
18          w(i) = coeff_expansion_(cell_global_idx[i]);
19        }
20
21        // III. PERFORM NUMERICAL QUADRATURE
22        element_vector = Eigen::VectorXd::Zero(4);
23        for (int i = 0; i < 4; i++) {  // for each local degree of freedom
24          element_vector(i) +=
25              integration_element(i) *
26              (0.5 / (1. + std::pow(0.5 * w(i) + 0.5 * w((i + 1) % 4), 2)));
27          element_vector((i + 1) % 4) +=
28              integration_element(i) *
29              (0.5 / (1. + std::pow(0.5 * w(i) + 0.5 * w((i + 1) % 4), 2)));
30        }
31        element_vector *= 1. / 4.;
32        break;
```

HINT 1 for (2-13.d): Element matrices are defined based on local shape functions, see [Lecture → Def. 2.7.4.5]. Study [Lecture → Ex. 2.5.3.7] in case you are not sure which local shape functions to use on an edge. ⌟

HINT 2 for (2-13.d): You can use [Lecture → Lemma 2.7.5.5] for $d = 1$. ⌟

SOLUTION of (2-13.d):

Denote by λ_1 and λ_2 the linear barycentric coordinate functions on the edge e as exhibited in [Lecture → Ex. 2.5.3.7]. Then we can write

$$w|_e = w_1\lambda_1 + w_2\lambda_2 \,,$$

where w_1, w_2 are the values of w in the endpoints of e. The barycentric coordinate functions λ_1, λ_2 are also the local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ on e. Therefore, the 2×2 element matrix $\mathbf{B}_e$ reads

$$\begin{aligned}
\mathbf{B}_e &= \begin{bmatrix} \int_e (w_1\lambda_1 + w_2\lambda_2)^2\lambda_1^2 \,\mathrm{d}S & \int_e (w_1\lambda_1 + w_2\lambda_2)^2\lambda_1\lambda_2 \,\mathrm{d}S \\ \int_e (w_1\lambda_1 + w_2\lambda_2)^2\lambda_1\lambda_2 \,\mathrm{d}S & \int_e (w_1\lambda_1 + w_2\lambda_2)^2\lambda_2^2 \,\mathrm{d}S \end{bmatrix} \\
&= w_1^2 \cdot \begin{bmatrix} \int_e \lambda_1^4 \,\mathrm{d}S & \int_e \lambda_1^3\lambda_2 \,\mathrm{d}S \\ \int_e \lambda_1^3\lambda_2 \,\mathrm{d}S & \int_e \lambda_1^2\lambda_2^2 \,\mathrm{d}S \end{bmatrix} + \\
&\quad 2w_1w_2 \cdot \begin{bmatrix} \int_e \lambda_1^3\lambda_2 \,\mathrm{d}S & \int_e \lambda_1^2\lambda_2^2 \,\mathrm{d}S \\ \int_e \lambda_1^2\lambda_2^2 \,\mathrm{d}S & \int_e \lambda_1^1\lambda_2^3 \,\mathrm{d}S \end{bmatrix} + \\
&\quad w_2^2 \cdot \begin{bmatrix} \int_e \lambda_1^2\lambda_2^2 \,\mathrm{d}S & \int_e \lambda_1\lambda_2^3 \,\mathrm{d}S \\ \int_e \lambda_1\lambda_2^3 \,\mathrm{d}S & \int_e \lambda_2^4 \,\mathrm{d}S \end{bmatrix} .
\end{aligned} \tag{2.13.7}$$

[Lecture → Lemma 2.7.5.5] tells us

$$\int_e \lambda_1^p\lambda_2^q \,\mathrm{d}\boldsymbol{x} = |e| \frac{p!q!}{(1+p+q)!} \,, \quad p, q \in \mathbb{N}_0 \,.$$

Thus, we can rewrite (2.13.7) as

$$\mathbf{B}_e = \frac{|e|}{120}\left(w_1^2 \begin{bmatrix} 24 & 6 \\ 6 & 4 \end{bmatrix} + 2w_1w_2 \begin{bmatrix} 6 & 4 \\ 4 & 6 \end{bmatrix} + w_2^2 \begin{bmatrix} 4 & 6 \\ 6 & 24 \end{bmatrix} \right) . \tag{2.13.8}$$

HINT 1 for (2-13.e): To identify edges on the boundary $\partial\Omega$ you can rely on the function `lf::mesh::uti`
see ➙ (documentation). ⌟

HINT 2 for (2-13.e): Of course, you have to rely on the **lf::assemble::DofHandler** object provided by `fe_space_` in order to access the values of w in the endpoints of the edge. ⌟

SOLUTION of (2-13.e):

C++11 code 2.13.9: Sub-problem (2-13.e): Constructor of ImpedanceBoundaryEdgeMatrixProvider ⇝ GitLab

```
ImpedanceBoundaryEdgeMatrixProvider::ImpedanceBoundaryEdgeMatrixProvider(
    std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space,
    Eigen::VectorXd coeff_expansion)
    : fe_space_(fe_space), coeff_expansion_(coeff_expansion) {
  auto mesh = fe_space->Mesh();
  // Obtain an array of boolean flags for the edges of the mesh: 'true'
  // indicates that the edge lies on the boundary. This predicate will
  // guarantee that the computations are carried only on the boundary edges
  bd_flags_ = std::make_shared<lf::mesh::utils::CodimMeshDataSet<bool>>(
      lf::mesh::utils::flagEntitiesOnBoundary(mesh, 1));
}
```

C++11 code 2.13.10: Sub-problem (2-13.e): Implementation of `isActive()` member function of ImpedanceBoundaryEdgeMatrixProvider ⇝ GitLab

```
bool ImpedanceBoundaryEdgeMatrixProvider::isActive(
    const lf::mesh::Entity &edge) {
  bool is_bd_edge;
  is_bd_edge = (*bd_flags_)(edge);
  return is_bd_edge;
}
```

C++11 code 2.13.11: Sub-problem (2-13.e): Implementation of `Eval()` member function of ImpedanceBoundaryEdgeMatrixProvider ⇝ GitLab

```
Eigen::MatrixXd ImpedanceBoundaryEdgeMatrixProvider::Eval(
    const lf::mesh::Entity &edge) {
  Eigen::MatrixXd element_matrix(2, 2);

  /* TOOLS AND DATA */
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_->LocGlobMap()};
  // Obtain edge data
  auto edge_global_idx = dofh.GlobalDofIndices(edge);

  // I. COMPUTE LOCAL INTEGRATION DATA
  Eigen::Vector2d w;
  // Evaluate w(x) at the degrees of freedom (endpoints of the edge)
  for (int i = 0; i < 2; i++) {
    w(i) = coeff_expansion_(edge_global_idx[i]);
  }

  // II. COMPUTE LOCAL INTEGRATION DATA
  double edge_length = lf::geometry::Volume(*edge.Geometry());
  // It is assumed here that the edge is straight!
  Eigen::MatrixXd m_1(2, 2), m_2(2, 2), m_3(2, 2);
  m_1 << 24, 6, 6, 4;
  m_2 << 6, 4, 4, 6;
```

```
  m_3 << 4, 6, 6, 24;

  // III. PERFORM NUMERICAL QUADRATURE
  element_matrix =
      (w(0) * w(0)) * m_1 + (w(0) * w(1)) * m_2 + (w(1) * w(1)) * m_3;
  element_matrix *= edge_length / 120.;

  return element_matrix;
}
```

HINT 1 for (2-13.f): In order to communicate the diffusion tensor to **lf::uscalfe::ReactionDiffusionEleme** you have to wrap it into an object of type **MeshFunction**. ⌟

HINT 2 for (2-13.f): The LEHRFEM++ class **lf::uscalfe::FeSpaceLagrangeO1** gives you complete specification of the local shape functions for linear Lagrangian finite elements, which can then be used to set up **lf::uscalfe::ReactionDiffusionElementMatrixProvider**, see [Lecture → Ex. 2.8.3.29]. ⌟

SOLUTION of (2-13.f):

C++11 code 2.13.12: Sub-problem (2-13.f): Unit test for AnisotropicDiffusionElementMatrixProvider ↪ GitLab

```
TEST(ParametricElementMatrices, TestGalerkin) {
  // use test mesh (with only affine equivalent cells!) to set up fe space
  auto mesh = lf::mesh::test_utils::GenerateHybrid2DTestMesh(5, 1);
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh);
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  const lf::base::size_type N_dofs(dofh.NumDofs());

  // compute galerkin matrix for d(x) = sin(|x|)x using the implemented class
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  auto d = [](Eigen::Vector2d x) -> Eigen::Vector2d {
    return (x * std::sin(x.norm()));
  };
  auto elmat_builder =
      ParametricElementMatrices::AnisotropicDiffusionElementMatrixProvider(d);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
  // compute galerkin matrix using ReactionDiffusionElementMatrixProvider
  lf::assemble::COOMatrix<double> B(N_dofs, N_dofs);
  auto alpha = [&d](Eigen::Vector2d x) -> Eigen::Matrix2d {
    const Eigen::Vector2d d_x{d(x)};
    return Eigen::Matrix2d::Identity(2, 2) + d_x * d_x.transpose();
  };
  lf::mesh::utils::MeshFunctionGlobal mf_alpha{alpha};
  auto zero = [](Eigen::Vector2d x) -> double { return 0.; };
  lf::mesh::utils::MeshFunctionGlobal mf_zero{zero};
  // set up quadrature rule to be able to compare
  std::map<lf::base::RefEl, lf::quad::QuadRule> quad_rules{
      {lf::base::RefEl::kTria(), lf::quad::make_TriaQR_EdgeMidpointRule()},
      {lf::base::RefEl::kQuad(), lf::quad::make_QuadQR_EdgeMidpointRule()}};

  lf::uscalfe::ReactionDiffusionElementMatrixProvider<
      double, decltype(mf_alpha), decltype(mf_zero)>
      elmat_builder_org(fe_space, mf_alpha, mf_zero, quad_rules);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder_org, B);

  auto A_crs = A.makeSparse();
  auto B_crs = B.makeSparse();
  // compare results (floating point comparison!)
  for (int i = 0; i < N_dofs; i++) {
    for (int j = 0; j < N_dofs; j++) {
      ASSERT_NEAR(A_crs.coeff(i, j), B_crs.coeff(i, j), 1E-9);
    }
  }
}
```

HINT 1 for (2-13.g): LEHRFEM++ supplies the class **lf::uscalfe::FeSpaceLagrangeO1** for lowest-order Lagrangian finite elements. ⌟

SOLUTION of (2-13.g):

C++11 code 2.13.13: Sub-problem (2-13.g): Unit test for FESourceElemVecProvider ⇝ GitLab

```
TEST(ParametricElementMatrices, TestLoad) {
  // use test mesh (with only affine equivalent cells!) to set up fe space
  auto mesh = lf::mesh::test_utils::GenerateHybrid2DTestMesh(5, 1);
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh);
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  const lf::base::size_type N_dofs(dofh.NumDofs());
  // An affine linear function that can be represented exactly
  // in the space of p.w. linear Lagrangian finite element functions
  auto w_func = [](Eigen::Vector2d x) -> double { return (3.0 * x[0] - x[1]); };
  lf::mesh::utils::MeshFunctionGlobal mf_w_func{w_func};
  // function f(x) = 1/(1 + w(x)^2)
  auto f = [&w_func](Eigen::Vector2d x) -> double {
    return 1. / (1. + std::pow(w_func(x), 2));
  };
  lf::mesh::utils::MeshFunctionGlobal mf_f{f};

  // interpolation of w on fe space: reproduces function
  auto w = lf::fe::NodalProjection<double>(*fe_space, mf_w_func);

  // use own method to assemble load vector
  auto elvec_builder =
      ParametricElementMatrices::FESourceElemVecProvider(fe_space, w);
  Eigen::Matrix<double, Eigen::Dynamic, 1> phi(N_dofs);
  phi.setZero();
  AssembleVectorLocally(0, dofh, elvec_builder, phi);
  // set up midpoint quadrature rules for all cell types
  std::map<lf::base::RefEl, lf::quad::QuadRule> quad_rules{
      {lf::base::RefEl::kTria(), lf::quad::make_TriaQR_EdgeMidpointRule()},
      {lf::base::RefEl::kQuad(), lf::quad::make_QuadQR_EdgeMidpointRule()}};

  // compute library solution
  Eigen::Matrix<double, Eigen::Dynamic, 1> phi_lib(N_dofs);
  phi_lib.setZero();
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      elvec_builder_(fe_space, mf_f, quad_rules);
  AssembleVectorLocally(0, dofh, elvec_builder_, phi_lib);

  // compare the results
  for (int i = 0; i < N_dofs; i++) {
    ASSERT_NEAR(phi(i), phi_lib(i), 1E-9);
  }
}
```

HINT 1 for (2-14.a): Recall the arguments of [Lecture → § 2.4.3.3].

SOLUTION of (2-14.a):

An affine linear function $b \in \mathcal{P}_1(\mathbb{R}^2)$ is uniquely defined by specifying its values in three non-collinear points [Lecture $\rightarrow$ § 2.4.3.3]. Since the triangle K must not be degenerate, the three midpoints of its edges cannot be collinear.

SOLUTION of (2-14.j):

For a triangular mesh every cell K is the image of the "unit triangle" $\widehat{K}$ with vertices $\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}$ under an invertible affine mapping $\mathbf{\Phi}_K : \widehat{K} \to K$, see [Lecture → § 2.7.5.12]. This provides the reference element.

As the local shape functions of $\mathcal{CR}(\mathcal{M})$ on a triangle K are linear combinations of its barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$, which remain preserved under affine pullbacks,

$$\mathbf{\Phi}_k^* \lambda_\ell = \widehat{\lambda}_\ell \,, \quad \ell = 1,2,3 \,,$$

they inherit this property. Using (2.14.3) from Sub-problem (2-14.i)

$$\mathbf{\Phi}_k^* b_K^j = \mathbf{\Phi}_K^* \underbrace{\big(1 - 2\lambda_{\mathrm{opp}(j)}\big)}_{=\widehat{b}^j} = 1 - 2\widehat{\lambda}_{\mathrm{opp}(j)} \,, \quad j = 1,2,3 \,.$$

In other word, we can find reference shape functions

$$\widehat{b}^1, \widehat{b}^2, \widehat{b}^3 \in \mathcal{P}_1(\mathbb{R}^2) \quad \text{such that} \quad \widehat{b}^j = \mathbf{\Phi}_K^* b_K^j \quad \forall K \in \mathcal{M},\ j = 1,2,3 \,.$$

SOLUTION of (2-14.k):

We write $\widehat{b}^\ell$, $\ell = 1,2,3$, for the local shape functions on the reference element. We can simply use the formula (2.14.3) found in Sub-problem (2-14.i) and that the barycentric coordinate functions on $\widehat{K}$ are given by

$$\lambda_1(\boldsymbol{x}) = 1 - x_1 - x_2 \quad , \quad \lambda_2(\boldsymbol{x}) = x_1 \quad , \quad \lambda_3(\boldsymbol{x}) = x_2 \, , \quad \boldsymbol{x} := \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} .$$

Thus, from (2.14.3), we find

$$\begin{aligned} &\widehat{b}^1(\widehat{\boldsymbol{x}}) = 1 - 2\lambda_3 = 1 - 2x_2 \, , \\ &\widehat{b}^2(\widehat{\boldsymbol{x}} = 1 - 2\lambda_1 = 2(x_1 + x_2) - 1 \, , \quad \widehat{\boldsymbol{x}} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \widehat{K} \, , \\ &\widehat{b}^3(\widehat{\boldsymbol{x}} = 1 - 2\lambda_2 = 1 - 2x_1 \, , \end{aligned} \tag{2.14.4}$$

with gradients

$$\begin{aligned} \mathbf{grad}\, \widehat{b}^1 &= -2\, \mathbf{grad}\, \lambda_3 = \begin{bmatrix} 0 \\ -2 \end{bmatrix}, \\ \mathbf{grad}\, \widehat{b}^2 &= -2\, \mathbf{grad}\, \lambda_1 = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, \\ \mathbf{grad}\, \widehat{b}^3 &= -2\, \mathbf{grad}\, \lambda_2 = \begin{bmatrix} -2 \\ 0 \end{bmatrix} . \end{aligned} \tag{2.14.5}$$

HINT 1 for (2-14.l): Use the integral formula for barycentric coordinate functions on a triangle K:

$$\int_K \lambda_1^{\alpha_1}\lambda_2^{\alpha_2}\lambda_3^{\alpha_3}\,\mathrm{d}\boldsymbol{x} = |K|\frac{\alpha_1!\alpha_2!\alpha_3!2!}{(\alpha_1+\alpha_2+\alpha_3+2)!}\,,$$

see [Lecture → Lemma 2.7.5.5].

SOLUTION of (2-14.l):

Clearly the localized bilinar form reads $\mathsf{a}_K(u,v) := \int_K u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$.

$$\begin{aligned}\mathbf{M}_K = \left[\mathsf{a}_K(b_K^i, b_K^j)\right]_{i,j=1}^3 &= \left[\mathsf{a}_K(1-2\lambda_{\mathrm{opp}(i)}, 1-2\lambda_{\mathrm{opp}(j)})\right]_{i,j=1}^3 \\ &= \left[\int_K 1\,\mathrm{d}\boldsymbol{x} - 2\int_K(\lambda_{\mathrm{opp}(i)} + \lambda_{\mathrm{opp}(j)}\,\mathrm{d}\boldsymbol{x} + 4\int_K \lambda_{\mathrm{opp}(i)}\lambda_{\mathrm{opp}(j)}\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^3 \\ &= \left(|K| - 2\cdot 2\frac{|K|}{3}\right)\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix} + \frac{|K|}{3}\begin{bmatrix}2&1&1\\1&2&1\\1&1&2\end{bmatrix} = \boxed{\tfrac{1}{3}|K|\,\mathbf{I}}\ .\end{aligned}$$

Surprisingly, the element mass matrix is diagonal, simply a scaled identity matrix!

HINT 1 for (2-14.m): Recall the ideas elaborated in [Lecture → § 1.2.3.36]. ⌟

SOLUTION of (2-14.m):

First notice that

$$\mathsf{a}_{\mathcal{M}}(v_h, v_h) = \sum_{K\in\mathcal{M}} \int_K |\mathbf{grad}\, v_h(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x} \geq 0.$$

Let us suppose that there exists a $v_h \in \mathcal{CR}_0(\mathcal{M}) \setminus \{\mathbf{0}\}$ such that $\mathsf{a}_{\mathcal{M}}(v_h, v_h) = 0$.

Then

$$\mathbf{grad}\, v_h(\boldsymbol{x}) = 0, \quad \forall K \in \mathcal{M},$$

and this implies v_h is constant $\forall K \in \mathcal{M}$. Since

$$v_h(\boldsymbol{m}) = 0, \quad \forall \boldsymbol{m} \in \mathcal{N} \cap \partial\Omega,$$

this means that $v_h \equiv 0$ for all $K \in \mathcal{M}$, which poses a contradiction as we assumed $v_h \in \mathcal{CR}_0(\mathcal{M}) \setminus \{\mathbf{0}\}$. Therefore, we conclude the desired result.

SOLUTION of (2-14.n):

Let

$$\mathbf{A}_K^L := \left[\int_K \mathbf{grad}\,\lambda_j \cdot \mathbf{grad}\,\lambda_i \,\mathrm{d}\boldsymbol{x}\right]_{i,j=1,2,3}$$
$$\mathbf{A}_K^{\mathcal{CR}} := \left[\int_K \mathbf{grad}\,b_K^j \cdot \mathbf{grad}\,b_K^i \,\mathrm{d}\boldsymbol{x}\right]_{i,j=1,2,3} .$$

From the solution of Sub-problem (2-14.i), we know that

$$b_K^j = 1 - 2\lambda_{\mathrm{opp}(j)} , \quad j = 1,\ldots,3 , \tag{2.14.11}$$

where we use the numbering of local shape functions as introduced in Fig. 14 and the definition of the index mapping opp as given in Sub-problem (2-14.i).

Combining this expression with the identity $\lambda_1 + \lambda_2 + \lambda_3 = 1$, we obtain

$$\begin{aligned} b_K^1 &= \lambda_1 + \lambda_2 - \lambda_3 , \\ b_K^2 &= \lambda_2 + \lambda_3 - \lambda_1 , \\ b_K^3 &= \lambda_1 + \lambda_3 - \lambda_2 . \end{aligned} \tag{2.14.12}$$

We want to apply the following result about the transformation of Galerkin matrices under a change of basis. Of course, element matrices are just special Galerkin matrices and so the lemma also applies to them.

Lemma [Lecture → Lemma 2.2.3.2]. Effect of change of basis on Galerkin matrix

Consider [Lecture → Eq. (2.2.1.1)] and two bases of $V_{0,h}$,

$$\mathfrak{B}_h := \{b_h^1,\ldots,b_h^N\} \quad , \quad \widetilde{\mathfrak{B}}_h := \{\widetilde{b}_h^1,\ldots,\widetilde{b}_h^N\} ,$$

related by the basis transformation matrix $\mathbf{S}$ according to

$$\widetilde{b}_h^j = \sum_{k=1}^N s_{jk} b_h^k \quad \textit{with} \quad \mathbf{S} = \left[s_{jk}\right]_{j,k=1}^N \in \mathbb{R}^{N,N} \textit{ regular.} \tag{2.14.13}$$

Then the Galerkin matrices $\mathbf{A}, \widetilde{\mathbf{A}} \in \mathbb{R}^{N,N}$, the right hand side vectors $\vec{\boldsymbol{\varphi}}, \widetilde{\vec{\boldsymbol{\varphi}}} \in \mathbb{R}^N$, and the coefficient vectors $\vec{\boldsymbol{\mu}}, \widetilde{\vec{\boldsymbol{\mu}}} \in \mathbb{R}^N$, respectively, satisfy

$$\widetilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^\top \quad , \quad \widetilde{\vec{\boldsymbol{\varphi}}} = \mathbf{S}\vec{\boldsymbol{\varphi}} \quad , \quad \widetilde{\vec{\boldsymbol{\mu}}} = \mathbf{S}^{-\top}\vec{\boldsymbol{\mu}} . \tag{2.14.14}$$

In the current setting we deal with

- the "old basis" $\mathfrak{B}_h = \{\lambda_1, \lambda_2, \lambda_3\}$,
- the "new basis" $\widetilde{\mathfrak{B}}_h := \{b_K^1, b_K^2, b_K^3\}$.

From (2.14.12) we infer that the basis transformation matrix $\mathbf{S}$ (from $\mathfrak{B}_h$ to $\widetilde{\mathfrak{B}}_h$) is given by

$$\mathbf{S} = \begin{bmatrix} 1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} .$$

Finally, from [Lecture → Lemma 2.2.3.2] we know that

$$\mathbf{A}_K^{\mathcal{CR}} = \mathbf{S}\mathbf{A}_K^L\mathbf{S}^\top \ .$$

Alternatively, one could use the fact that

$$\mathbf{grad}\, b_K^j = -2\,\mathbf{grad}\,\lambda_{\mathrm{opp}(j)}, \quad j = 1,\ldots,3 \ .$$

and define the permutation matrix $\mathbf{P}$ which performs the local index mapping described by opp as

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Then, due to the relations described above, we have

$$\mathbf{A}_K^{\mathcal{CR}} = 4\mathbf{P}\mathbf{A}_K^L\mathbf{P}^\top \ .$$

The two formulas yield the same result, because the sum of rows and columns of $\mathbf{A}_K^L$ vanishes. Hence they are linearly dependent.

HINT 1 for (2-14.o): Exploit the cardinal basis property, *cf.* [Lecture → Eq. (2.3.2.1)], of the local shape functions with respect to the set of midpoints of the edges of the triangle. ⌟

SOLUTION of (2-14.o):

By virtue of the local *cardinal basis property* of $\{b_K^1, b_K^2, b_K^3\}$ with respect to the midpoints $\boldsymbol{m}_K^\ell$, $\ell = 1,2,3$, of the edges (which are exactly the quadrature nodes in (2.14.15)), the sought element matrix $\mathbf{A}_K$ arises from $\mathbf{A}_K^{\mathcal{CR}}$ by adding a simple diagonal matrix.

$$\mathbf{A}_K = \mathbf{A}_K^{\mathcal{CR}} + \frac{|K|}{3} \begin{bmatrix} \gamma(\boldsymbol{m}_K^1) & & \\ & \gamma(\boldsymbol{m}_K^2) & \\ & & \gamma(\boldsymbol{m}_K^3) \end{bmatrix}.$$

HINT 1 for (2-14.p): As in Sub-problem (2-14.o) again exploit the cardinal basis property of the local shape functions with respect to the set of midpoints of the edges of the triangle. ⌟

SOLUTION of (2-14.p):

Using the given 3-point quadrature formula to compute $\vec{\varphi}_K$, we get for $j = 1..3$

$$(\vec{\varphi}_K)_j = \int_K f(\boldsymbol{x}) b_K^j = \tfrac{1}{3}|K| \sum_{l=1}^{3} f(\boldsymbol{m}^l) b_h^j(\boldsymbol{m}^l) = \tfrac{1}{3}|K| f(\boldsymbol{m}^l)\delta_{jl},$$

where δ_{il} is the Kronecker delta, and b_K^j, $j = 1,2,3$ the local shape functions of $\mathcal{CR}(\mathcal{M})$ from (2.14.1). Then, the resulting vector is

$$\vec{\varphi}_K = \frac{1}{3}|K| \begin{bmatrix} f(\boldsymbol{m}^1) \\ f(\boldsymbol{m}^2) \\ f(\boldsymbol{m}^3) \end{bmatrix}.$$

HINT 1 for (2-14.q): Use your solution for Sub-problem (2-14.k) to implement the local shape functions for $\mathcal{CR}(\mathcal{M})$. ⌟

SOLUTION of (2-14.q):

- The reference element is TRIA.
- The polynomial degree is 1.
- We have 3 local shape functions for each cell, and exactly 1 associated with each edge, none belonging to any other type of geometric entity.

For further explanations please consult [Lecture → § 2.8.3.25].

C++ code 2.14.16: Sub-problem (2-14.q): Member functions `RefEl()` and `Degree()` for CRReferenceFiniteElement ⇝ GitLab

```
2  lf::base::RefEl CRReferenceFiniteElement::RefEl() const {
3    lf::base::RefElType ref_el_type;
4  // TODO: task 2-14.q)
5    ref_el_type = lf::base::RefElType::kTria;
6    return lf::base::RefEl(ref_el_type);
7  }
```

```
2  unsigned int CRReferenceFiniteElement::Degree() const {
3    unsigned int degree;
4  // TODO: task 2-14.q)
5    degree = 1;
6    return degree;
7  }
```

C++ code 2.14.17: Sub-problem (2-14.q): Versions of `NumRefShapeFunctions()` for CRReferenceFiniteElement ⇝ GitLab

```
2  lf::assemble::size_type CRReferenceFiniteElement::NumRefShapeFunctions() const {
3    lf::assemble::size_type num_ref_shape_functions;
4  // TODO: task 2-14.q)
5    num_ref_shape_functions = 3;
6    return num_ref_shape_functions;
7  }
8
9  lf::assemble::size_type CRReferenceFiniteElement::NumRefShapeFunctions(
10     lf::assemble::dim_t codim) const {
11   switch (codim) {
12     // TODO: task 2-14.q)
13     case 0:
14       return 0;
15     case 1:
16       return 1;
17     case 2:
18       return 0;
19     default:
20       LF_VERIFY_MSG(false, "Codimension out of range for triangle")
21       return 0;
22   }
23 }
24
25 lf::assemble::size_type CRReferenceFiniteElement::NumRefShapeFunctions(
```

```
    lf::assemble::dim_t codim, lf::base::sub_idx_t subidx) const {
  switch (codim) {
    // TODO: task 2-14.q)
    case 0:
      LF_VERIFY_MSG((0 == subidx),
                    "Index of cell is out of range for triangle");
      return 0;
    case 1:
      LF_VERIFY_MSG((0 <= subidx && subidx < 3),
                    "Index of edge is out of range for triangle");
      return 1;
    case 2:
      LF_VERIFY_MSG((0 <= subidx && subidx < 3),
                    "Index of vertex is out of range for triangle");
      return 0;
    default:
      LF_VERIFY_MSG(false, "Codimension out of range for triangle")
      return 0;
  }
}
```

The following method is a straightforward implementation of the simple formulas (2.14.4).

C++ code 2.14.18: Sub-problem (2-14.q): Member function `EvalReferenceShapeFunctions()` for CRReferenceFiniteElement ↪ GitLab

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
CRReferenceFiniteElement::EvalReferenceShapeFunctions(
    const Eigen::MatrixXd &refcoords) const {
  // Data
  const auto num_points =
      static_cast<lf::assemble::size_type>(refcoords.cols());
  // Initialize a matrix that will store the values of the reference basis
  // functions evaluated at the coordinates passed as arguments
  Eigen::MatrixXd eval_ref_shape_functions(3, num_points);
// TODO: task 2-14.q)
  Eigen::MatrixXd ones = Eigen::VectorXd::Ones(num_points).transpose();
  // Evaluate the basis functions
  eval_ref_shape_functions.row(0) = ones - 2. * refcoords.row(1);
  eval_ref_shape_functions.row(1) = 2. * refcoords.colwise().sum() - ones;
  eval_ref_shape_functions.row(2) = ones - 2. * refcoords.row(0);
  return eval_ref_shape_functions;
}
```

SOLUTION of (2-14.r):

We simply implement (2.14.5).

C++ code 2.14.19: Sub-problem (2-14.r): `GradientsReferenceShapeFunctions()` for CRReferenceFiniteElement ↪ GitLab

```
Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>
CRReferenceFiniteElement::GradientsReferenceShapeFunctions(
    const Eigen::MatrixXd &refcoords) const {
  // Data
  const auto num_points =
      static_cast<lf::assemble::size_type>(refcoords.cols());
  // Initialize a matrix that will store the gradients of the reference basis
  // functions evaluated at the coordinates passed as arguments
  Eigen::MatrixXd grad_ref_shape_functions(3, 2 * num_points);
// TODO: task 2-14.r)
  // Evaluate the gradients
  grad_ref_shape_functions.row(0) = (Eigen::Vector2d() << 0, -2)
                                        .finished()
                                        .transpose()
                                        .replicate(1, num_points);
  grad_ref_shape_functions.row(1) =
      2. * Eigen::VectorXd::Ones(2 * num_points).transpose();
  grad_ref_shape_functions.row(2) = (Eigen::Vector2d() << -2, 0)
                                        .finished()
                                        .transpose()
                                        .replicate(1, num_points);
  return grad_ref_shape_functions;
}
```

HINT 1 for (2-14.s): For the implementation of the member function `NodalValuesToDofs()` take into account that the local shape functions for $\mathcal{RM}(\mathcal{M})$ are a cardinal basis with respect to the set of local interpolation nodes. ⌟

SOLUTION of (2-14.s):

The local evaluation nodes on the reference element are the midpoints of the edges of the unit triangle.

$$\widehat{m}^1 = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \quad , \quad \widehat{m}^2 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \quad , \quad \widehat{m}^3 = \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} .$$

C++ code 2.14.20: Sub-problem (2-14.s): `EvaluationNodes()` for CRReferenceFiniteElement, GitLab

```
Eigen::MatrixXd CRReferenceFiniteElement::EvaluationNodes() const {
  Eigen::MatrixXd eval_nodes(2, 3);
// TODO: task 2-14.s)
  eval_nodes << .5, .5, 0, 0, .5, .5;
  return eval_nodes;
}
```

By the cardinal basis property of $\widehat{b}^1, \widehat{b}^2, \widehat{b}^3$ with respect to $\widehat{m}^1, \widehat{m}^3, \widehat{m}^3$ the mapping taking nodal values to local degrees of freedom is just the *identity*.

C++ code 2.14.21: Sub-problem (2-14.s): `NodalValuesToDofs()` for CRReferenceFiniteElement GitLab

```
Eigen::Matrix<double, 1, Eigen::Dynamic>
CRReferenceFiniteElement::NodalValuesToDofs(
    const Eigen::Matrix<double, 1, Eigen::Dynamic> &nodvals) const {
  LF_VERIFY_MSG(nodvals.cols() == NumEvaluationNodes(),
                "nodvals = " << nodvals << " <-> " << NumEvaluationNodes());

  Eigen::MatrixXd coeffs;
// TODO: task 2-14.s)
  // Linear mapping is identity since the set of reference shape functions
  // forms a cardinal basis with respect to the interpolation nodes
  coeffs = nodvals;
  return coeffs;
}
```

SOLUTION of (2-14.b):

We rely on the very definition of linear independence from linear algebra. In particular, we have to show

$$\sum_{j=1}^{N} \alpha_j b_h^j(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} \in \Omega \quad \Rightarrow \quad \alpha_j = 0 \quad \forall j \,.$$

This is easy, because choosing $\boldsymbol{x} := \boldsymbol{m}_k$, (2.14.1) already implies $\alpha_k = 0$, $k = 1 \ldots, N$.

HINT 1 for (2-14.t): Maybe you have not yet noticed: the specifications for local shape functions of $\mathcal{CR}(\mathcal{M})$ on triangles are implemented in **CRReferenceFiniteElement**, see Sub-problem (2-14.q). ⌟

SOLUTION of (2-14.t):

The smart student has already realized that there is nothing to do, because **lf::uscalfe::UniformScalarFES** is essentially a wrapper class for objects of type **lf::fe::ScalarReferenceFiniteElement**.

C++ code 2.14.22: Sub-problem (2-14.t): Implementation of CRFeSpace, ↪ GitLab

```
2  class CRFeSpace : public lf::uscalfe::UniformScalarFESpace<double> {
3   public:
4    /** @brief no default constructors*/
5    CRFeSpace() = delete;
6    CRFeSpace(const CRFeSpace &) = delete;
7    CRFeSpace(CRFeSpace &&) noexcept = default;
8    CRFeSpace &operator=(const CRFeSpace &) = delete;
9    CRFeSpace &operator=(CRFeSpace &&) noexcept = default;
10
11   /** Main constructor that sets up the local-to-global index mapping
12    * by calling the constructor if its base class
13    * The parameter mesh_p is a shared pointer to underlying mesh (immutable) */
14   explicit CRFeSpace(std::shared_ptr<const lf::mesh::Mesh> mesh_p)
15       : lf::uscalfe::UniformScalarFESpace<double>(
16             std::move(mesh_p), std::make_shared<CRReferenceFiniteElement>(),
17             nullptr, nullptr) {}
18   // You might have wondered what to do, but there is nothing to do!
19   // The base class takes care of everything
20
21   ~CRFeSpace() override = default;
22 };
```

SOLUTION of (2-14.u):

The following implementation is the same as for Lagrangian finite elements, except to the special type of **lf::uscalfe::UniformScalarFESpace** object that is passed as `fe_space` argument.

C++ code 2.14.24: Sub-problem (2-14.u): Implementation of `solveCRNeumannBVP()`, ➥ GitLab

```
template <typename GAMMA_COEFF, typename F_FUNCTOR>
Eigen::VectorXd solveCRNeumannBVP(std::shared_ptr<CRFeSpace> fe_space,
                                  GAMMA_COEFF &&gamma, F_FUNCTOR &&f) {
  Eigen::VectorXd sol;
// TODO: task 2-14.u)
  // Obtain local to global index mapping for shape functions
  const lf::assemble::DofHandler &dof_handler{fe_space->LocGlobMap()};
  const size_type num_dofs = dof_handler.NumDofs();
  // Prepare coefficient and source functions as MeshFunction
  lf::mesh::utils::MeshFunctionGlobal mf_one{
      [](Eigen::Vector2d x) -> double { return 1.; }};
  lf::mesh::utils::MeshFunctionGlobal mf_gamma{gamma};
  lf::mesh::utils::MeshFunctionGlobal mf_f{f};
  // Sparse Galerkin matrix in triplet format
  lf::assemble::COOMatrix<double> A(num_dofs, num_dofs);
  // Initialize ELEMENT_MATRIX_PROVIDER object
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double, decltype(mf_one),
                                                      decltype(mf_gamma)>
      element_matrix_builder(fe_space, mf_one, mf_gamma);
  // Fill Galerkin matrix (create array of triplets)
  lf::assemble::AssembleMatrixLocally(0, dof_handler, dof_handler,
                                      element_matrix_builder, A);
  // Right-hand-side vector; do not forget to set to zero!
  Eigen::Matrix<double, Eigen::Dynamic, 1> phi(num_dofs);
  phi.setZero();
  // Initialize ELEMENT_VECTOR_PROVIDER object
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      load_vector_builder(fe_space, mf_f);
  // Fill right-hand-side vector (cell oriented assembly)
  lf::assemble::AssembleVectorLocally(0, dof_handler, load_vector_builder, phi);
  // Set up Galerkin matrix in CRS format
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
  // ... and solve the linear system of equations by Gaussian elimination
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  sol = solver.solve(phi);
  return sol;
}
```

Why can we use the same implementation as for standard Lagrangian finite elements? Well, that implementation just needs values of the reference shape functions and of their gradients. This is what we implemented in Sub-problem (2-14.q), Code 2.14.18, and Sub-problem (2-14.r), Code 2.14.19.

SOLUTION of (2-14.v):

The only new aspect compared to linear Lagrangian finite elements is that the global shape functions are associated with edges of the mesh = entities of co-dimension 1. Thus we have to retrieve boundary flags for entities of that co-dimension.

C++ code 2.14.25: Sub-problem (2-14.v): Implementation of `solveCRDirichletBVP()`
↪ GitLab

```
2   // Obtain an array of boundary flags for edges (codim-1 entities !)
3   lf::mesh::utils::CodimMeshDataSet<bool> boundary_edges{
4       lf::mesh::utils::flagEntitiesOnBoundary(fe_space->Mesh(), 1)};
5   // Enforce homogeneous boundary
6   lf::assemble::FixFlaggedSolutionComponents<double>(
7       [&boundary_edges, &dof_handler](
8           lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
9         const lf::mesh::Entity &edge{dof_handler.Entity(gdof_idx)};
10        return {boundary_edges(edge), 0.0};
11      },
12      A, phi);
13  // Set up Galerkin matrix in CRS format
14  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
15  // ... and solve the linear system of equations by Gaussian elimination
16  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
17  solver.compute(A_crs);
18  sol = solver.solve(phi);
```

HINT 1 for (2-14.w): Again, the cardinal basis property of the local/global shape functions with respect to the quadrature nodes of (2.14.15) is very convenient.

Remember the LEHRFEM++ function `lf::geometry::Volume()`. ⌟

SOLUTION of (2-14.w):

C++ code 2.14.26: Sub-problem (2-14.w): Implementation of `computeCRL2Error`, ➙ GitLab

```cpp
template <typename FUNCTION>
double computeCRL2Error(std::shared_ptr<CRFeSpace> fe_space,
                        const Eigen::VectorXd &mu, FUNCTION &&u) {
  double l2_error = 0.;

// TODO: task 2-14.w)
  // Obtain local-to-global map and current mesh object
  const lf::assemble::DofHandler &dof_handler{fe_space->LocGlobMap()};
  auto mesh_ptr = fe_space->Mesh();

  // Loop over all cells of the mesh (entities of co-dimension 0)
  for (const lf::mesh::Entity *cell : mesh_ptr->Entities(0)) {
    const lf::assemble::size_type num_nodes = cell->RefEl().NumNodes();
    LF_ASSERT_MSG(num_nodes == 3, "Only meaningful for triangles!");
    // Obtain pointer to shape information for cell
    const lf::geometry::Geometry *cell_geom{cell->Geometry()};
    // 2x3- matrix with corner coordinates in its columns
    const Eigen::MatrixXd vertices{lf::geometry::Corners(*cell_geom)};
    // clang-format off
    // 2x3-matrix of midpoint coordinates
    Eigen::MatrixXd midpoints{vertices *
      (Eigen::Matrix<double, 3, 3>(3,3) <<
                    0.5, 0.0, 0.5,
                    0.5, 0.5, 0.0,
                    0.0, 0.5, 0.5)
                .finished()};
    // clang-format on
    // Obtain global indices of local shape functions
    std::span<const lf::assemble::gdof_idx_t> cell_dof_idx(
        dof_handler.GlobalDofIndices(*cell));

    // Sum contributions of quadrature nodes
    double local_sum = 0.;
    // The CR interpolation nodes are the midpoints and so the exact
    // solution needs to be evaluated at the same points
    for (int loc_idx = 0; loc_idx < num_nodes; ++loc_idx) {
      local_sum +=
          std::pow(mu[cell_dof_idx[loc_idx]] - u(midpoints.col(loc_idx)), 2);
    }
    // Sum cell contributions
    l2_error += lf::geometry::Volume(*cell_geom) * (local_sum / num_nodes);
  }
  return std::sqrt(l2_error);
}
```

HINT 1 for (2-14.x):

- [Lecture → Code 2.7.1.12] shows how to read a mesh from file in LEHRFEM++.
- Use the sparse direct solver of `Eigen`.

⌟

SOLUTION of (2-14.x):

See Code 10.3.32 for the code.

C++ code 2.14.28: Sub-problem (2-14.x): Implementation of `L2errorCRDiscretizationDirichletBVP()` ↬ GitLab

```
double L2errorCRDiscretizationDirichletBVP(const std::string &filename) {
  double l2_error;

// TODO: task 2-14.x)
  // Right-hand-side source function
  auto f = [](Eigen::Vector2d x) -> double {
    return (2. * M_PI * M_PI + x.prod()) * std::sin(M_PI * x(0)) *
           std::sin(M_PI * x(1));
  };
  // Reaction coefficient
  auto gamma = [](Eigen::Vector2d x) -> double { return x.prod(); };
  // Analytic solution
  auto u = [](Eigen::Vector2d x) -> double {
    return std::sin(M_PI * x(0)) * std::sin(M_PI * x(1));
  };

  // Read mesh from file
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  const lf::io::GmshReader reader(std::move(mesh_factory), filename);

  // Build CR FE space
  auto fe_space = std::make_shared<CRFeSpace>(reader.mesh());

  // Solve homogeneous Dirichlet problem
  Eigen::VectorXd mu = solveCRDirichletBVP(fe_space, gamma, f);
  // Compute L2 norm of error
  l2_error = computeCRL2Error(fe_space, mu, u);
  return l2_error;
}
```

SOLUTION of (2-14.y):

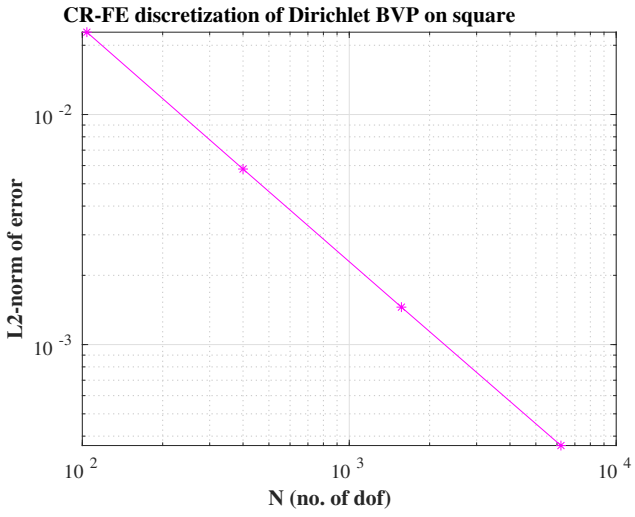
The implementation can be accessed through the course repository ↪ GitLab.

You should get the following numbers:

$N = \dim \mathcal{RC}_0(\mathcal{M})$	104	400	1568	6208
$\|u - u_h\|_{L^2(\Omega)}$	0.0227969	0.00579489	0.00145469	0.000364046

SOLUTION of (2-14.z):

The points $(N, \text{L2-error})$ approximately lie on a straught line in a doubly logarithmic plot, see Fig. 15, which hints at **algebraic covergence** according to [Lecture $\rightarrow$ Def. 3.2.2.1].
From the error table given in the solution of Subproblem (2-14.y) we also see that the error is roughly proportional to N, which indicates *rate 1*.



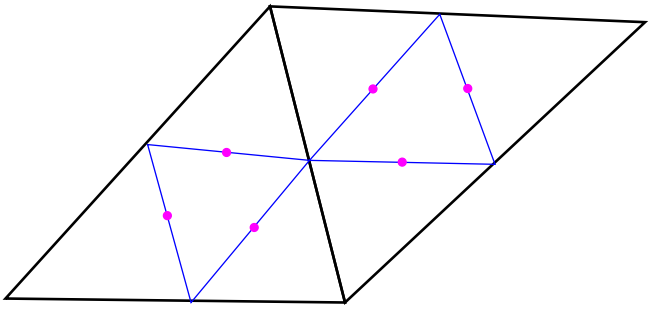
SOLUTION of (2-14.c):

Recall that a basis $\{b_h^1, \ldots, b_h^N\}$ of a space $V_{0,h}$ of functions $\Omega \mapsto \mathbb{R}$ is a cardinal basis with respect to the point set $\mathcal{N} := \{\boldsymbol{p}_1, \ldots, \boldsymbol{p}_N\}$, if $b_h^i(\boldsymbol{p}_j) = \delta_{i,j}$, possibly after renumbering of the points.

By the very definition of the functions b_h^j in (2.14.1), the set of interpolation nodes for $\mathcal{CR}(\mathcal{M})$ is the set of midpoints of edges of the mesh: $\mathcal{N} = \{\boldsymbol{m}_j,\ j = 1, \ldots, N\}$.

HINT 1 for (2-14.d): Observe that functions $v_h \in \mathcal{CR}(\mathcal{M})$ attain unique values at the midpoints of edges; they are continuous there. ⌟

SOLUTION of (2-14.d):



Functions in $\mathcal{CR}(\mathcal{M}')$ must be continuous in the midpoints of the edges of $\mathcal{M}'$. However, functions in $\mathcal{CR}(\mathcal{M})$ need not be continuous there, because the midpoints of edges of $\mathcal{M}'$ will not be midpoints of edges of $\mathcal{M}$.

◁ black lines $\hat{=}$ $\mathcal{M}$,
black + blue lines $\hat{=}$ $\mathcal{M}'$
Functions $\in \mathcal{CR}(\mathcal{M}')$ are continuous at the edge midpoints •, but not those in $\mathcal{CR}(\mathcal{M})$.

SOLUTION of (2-14.e):

In fact, the basis functions are discontinuous, see Fig. 12: $b_h^i(\boldsymbol{x}_n) = -1$ if $\boldsymbol{x}_n$ is opposite of $\boldsymbol{m}_i$ on some triangle, but $b_h^i = 0$ on all other triangles touching $\boldsymbol{x}_n$, thus we have "different values from different sides":

$$\blacktriangleright \quad \mathcal{CR}(\mathcal{M}) \not\subset C^0(\Omega) \qquad (, \text{ if } \sharp\mathcal{M} > 1).$$

Since b_h^j is piecewise smooth, the assumptions of [Lecture → Thm. 1.3.4.23] are violated:

Theorem [Lecture → Thm. 1.3.4.23]. Compatibility conditions for piecewise smooth functions in $H^1(\Omega)$

Let Ω be partitioned *into sub-domains Ω_1 and Ω_2. A function u that is continuously differentiable in both sub-domains and continuous up to their boundary, belongs to $H^1(\Omega)$,* if and only if *u is* continuous *on Ω.*

SOLUTION of (2-14.f):

The support of the function b_h^i coincides with the union of the two triangles on either side of edge i:

$$\operatorname{supp} b_h^j = \bigcup \{\overline{K} : K \in \mathcal{M}, \boldsymbol{m}_j \in \overline{K}\} .$$

Therefore, also in this case the support conditions for global shape functions of finite element spaces ($\rightarrow$ [Lecture $\rightarrow$ Section 2.5.3]) are satisfied.

SOLUTION of (2-14.g):

Yes, as it satisfies the 3 conditions introduced in [Lecture → Section 2.5.3]:

Third main ingredient of FEM: locally supported basis functions
(see [Lecture → Section 2.2] for role of bases in Galerkin discretization)

Basis functions $b_h^1, \ldots, b_h^N$ for a finite element trial/test space $V_{0,h}$ built on a mesh $\mathcal{M}$ *must* satisfy:

($\mathfrak{B}_1$) $\mathfrak{B}_h := \{b_h^1, \ldots, b_h^N\}$ is basis of $V_{0,h}$ ➢ $N = \dim V_{0,h}$,

($\mathfrak{B}_2$) each b_h^i is associated with a single geometric entity (cell/edge/face/vertex) of $\mathcal{M}$,

($\mathfrak{B}_3$) $\operatorname{supp}(b_h^i) = \bigcup\{\overline{K}\colon K \in \mathcal{M}, \boldsymbol{p} \subset \overline{K}\}$, if b_h^i associated with cell/edge/face/vertex $\boldsymbol{p}$.

1. follows by the definition of $\mathcal{CR}(\mathcal{M})$ and Sub-problem (2-14.b). These two combined imply that the set of functions defined in (2.14.1) are a basis for an N-dimensional finite element space $\mathcal{CR}(\mathcal{M}) \subset L^2(\Omega)$.
2. holds as each b_h^i is associated to a single edge.
3. As we saw in Sub-problem (2-14.f), if b_h^i is associated with the edge $\mathbf{e}$, then $\operatorname{supp}(b_h^i) = \bigcup\{\bar{K} : K \in \mathcal{M}, \mathbf{e} \subset \bar{K}\}$. Moreover, $\#\{K \in \mathcal{M} : K \subset \operatorname{supp}(b_h^i)\} = 2$.

HINT 1 for (2-14.h): Initialization of **lf::assemble::DofHandler** objects is discussed in [Lecture → § 2.7.4.16]. ⌟

SOLUTION of (2-14.h):

```
lf::assemble::UniformFEDofHandler dof_handler(
      mesh_p, {{lf::base::RefEl::kPoint(), 0},
               {lf::base::RefEl::kSegment(), 1},
               {lf::base::RefEl::kTria(), 0},
               {lf::base::RefEl::kQuad(), 0}});
```

SOLUTION of (2-14.i):

The local shape functions b_K^1, b_K^2, b_K^3 for $\mathcal{CR}(\mathcal{M})$ on a triangle K are *affine linear*,

$$\forall v_h \in \mathcal{CR}(\mathcal{M}),\ \forall K \in \mathcal{M}: \quad v_h|_K \in \mathcal{P}_1(\mathbb{R}^2)\ ,$$

and, thus, can be written as a linear combination of the barycentric coordinate functions λ_i of K.

We elaborate the derivation for b_K^1; after permutation of indices the considerations also apply to b_K^2 and b_K^3. We set

$$b_K^1 = \xi_1\lambda_1 + \xi_2\lambda_2 + \xi_3\lambda_3\ ,$$

with yet unknown coefficients $\xi_1, \xi_2, \xi_3 \in \mathbb{R}$. From (2.14.1) we obtain three equations:

$$\begin{aligned}
1 = b_K^1(\boldsymbol{m}_K^1) &= \xi_1\lambda_1(\boldsymbol{m}_K^1) + \xi_2\lambda_2(\boldsymbol{m}_K^1) + \xi_3\lambda_3(\boldsymbol{m}_K^1) = \tfrac{1}{2}\xi_1 + \tfrac{1}{2}\xi_2\ ,\\
0 = b_K^1(\boldsymbol{m}_K^2) &= \xi_1\lambda_1(\boldsymbol{m}_K^2) + \xi_2\lambda_2(\boldsymbol{m}_K^2) + \xi_3\lambda_3(\boldsymbol{m}_K^2) = \tfrac{1}{2}\xi_2 + \tfrac{1}{2}\xi_3\ ,\\
0 = b_K^1(\boldsymbol{m}_K^3) &= \xi_1\lambda_1(\boldsymbol{m}_K^3) + \xi_2\lambda_2(\boldsymbol{m}_K^3) + \xi_3\lambda_3(\boldsymbol{m}_K^3) = \tfrac{1}{2}\xi_1 + \tfrac{1}{2}\xi_3\ .
\end{aligned}$$

We have obtained a 3×3 linear system of equations for ξ_1, ξ_2, ξ_3:

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} \xi_1 \\ \xi_2 \\ \xi_3 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}\ ,$$

for which we find the solution

$$\xi_1 = 1\ ,\quad \xi_2 = 1\ ,\quad \xi_3 = -1 \quad \blacktriangleright \quad b_K^1 = \lambda_1 + \lambda_2 - \lambda_3\ .$$

This formula can be simplified based on the fact that $\lambda_1 + \lambda_2 + \lambda_3 \equiv 1$, which gives us the general formula

$$b_K^j = 1 - 2\lambda_{\mathrm{opp}(j)}\ ,\quad j = 1,2,3\ . \tag{2.14.3}$$

SOLUTION of (2-15.a):

First notice that the bilinear form

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}, \tag{2.15.4}$$

is continuous, symmetric and *positive definite* (s.p.d.) in $H^1_*(\Omega)$. Continuity and symmetry should be clear. Detailed arguments why it is positive definite are given in [Lecture → Rem. 1.8.0.14]. For s.p.d. a uniqueness of solutions of the linear variational problem is implied by the simple reasoning:

$$\left.\begin{array}{l}\mathsf{a}(u,v) = \ell(v)\,,\\ \mathsf{a}(w,v) = \ell(v)\end{array}\right. \quad \forall v \quad \Rightarrow \quad \mathsf{a}(u-w,v) = 0 \quad \forall v \quad \Rightarrow \quad \mathsf{a}(u-w,u-w) = 0 \quad \Rightarrow \quad u - w = \mathbf{0}\,.$$

Remark. From [Lecture → Thm. 1.8.0.26] we know that the energy norm induced by $\mathsf{a}(\cdot,\cdot)$ is equivalent to the norm of $H^1_*(\Omega)$, because

$$\|v\|_{L^2(\Omega)} \leq \|\mathbf{grad}\, v\|_{L^2(\Omega)} \quad \forall v \in H^1_*(\Omega)\,.$$

Moreover, as $f \in L^2(\Omega)$ and $h \in L^2(\partial\Omega)$, the right hand side is continuous with respect to the bilinear form (see [Lecture → § 1.9.0.9]). From these two facts we conclude the existence and uniqueness of solutions, for any data f, h, even if (2.15.2) is violated.

Remark. When restricting ourselves to test functions $v \in H^1_*(\Omega)$, then adding a constant to the source function f does not change the right-hand-side functional at all! Hence, we could simply add a constant to f to satisfy (2.15.2). This would be the appropriate "physical heat source" in the contexts of stationary heat conduction, see [Lecture → Section 1.6]. The next sub-problem will address this.

SOLUTION of (2-15.b):

For each $v \in H^1(\Omega)$, we define

$$\bar{v} = \frac{1}{|\Omega|} \int_\Omega v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x},$$

and $\tilde{v} := v - \bar{v}$.

This allows us rewrite $v = \tilde{v} + \bar{v} \in H^1(\Omega)$. Notice that $\tilde{v} \in H^1_*(\Omega)$. We plug this into (2.15.1) on $H^1(\Omega)$ and obtain:

$$\begin{aligned} \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, \tilde{v} \, \mathrm{d}\boldsymbol{x} &= \int_\Omega f(\tilde{v} + \bar{v}) \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h(\tilde{v} + \bar{v}) \, \mathrm{d}S \\ &= \int_\Omega f\tilde{v} \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\tilde{v} \, \mathrm{d}S + \bar{v}\Big(\int_\Omega f \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h \, \mathrm{d}S\Big). \end{aligned}$$

Use the fact that u is the solution of (2.15.1) on $H^1_*(\Omega)$, that is

$$\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, \tilde{v} \, \mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, \tilde{v} \, \mathrm{d}\boldsymbol{x} \quad \forall \tilde{v} \in H^1_*(\Omega) \,,$$

and, then, canceling the terms with $\tilde{v} \in H^1_*(\Omega)$.

$$0 = \bar{v}\Big(\int_\Omega f \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h \, \mathrm{d}S\Big).$$

We conclude that if the compatibility condition (2.15.2) is satisfied, then u also solves (2.15.1) on $H^1(\Omega)$ (although, as discussed before, it is not the only one).

So, we have to enforce (2.15.2). This can be done in many ways by changing the data. For instance, a suitable modification of f replaces it with

$$f \to f - c \quad \text{for} \quad c = \frac{1}{|\Omega|}\left(\int_\Omega f \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h \, \mathrm{d}S\right).$$

No modification of the Neumann data h is required in this case.

HINT 1 for (2-15.c): Notice that switching from $\mathcal{S}_1^0(\mathcal{M})$ to $\mathcal{S}_{1,*}^0(\mathcal{M})$ as a trial and test space amounts to dropping a single function of the standard basis of tent functions. How will this affect the Galerkin matrix and why is similar to imposing homogeneous essential boundary conditions. ⌟

HINT 2 for (2-15.c):

The LEHRFEM++ built-in function `lf::assemble::FixFlaggedSolutionComponents` ↪ (docun
comes handy for solving this sub-problem. ⌟

SOLUTION of (2-15.c):

The calls to `FixFlaggedSolutionComponents()` as implemented in Code 2.15.7 converts the Galerkin linear system of equations

$$\begin{bmatrix} a_{11} & * \\ * & \widetilde{\mathbf{A}} \end{bmatrix} \begin{bmatrix} \mu_1 \\ \widetilde{\mu} \end{bmatrix} = \begin{bmatrix} \varphi_1 \\ \widehat{\vec{\boldsymbol{\varphi}}} \end{bmatrix},$$

into

$$\begin{bmatrix} a_{11} & \mathbf{0}^\top \\ \mathbf{0} & \widetilde{\mathbf{A}} \end{bmatrix} \begin{bmatrix} \mu_1 \\ \widetilde{\mu} \end{bmatrix} = \begin{bmatrix} 0 \\ \widehat{\vec{\boldsymbol{\varphi}}} \end{bmatrix}.$$

Obviously, this has the same effect as removing the first global shape function from the basis, or, equivalently, setting its contribution to zero.

C++11 code 2.15.7: Sub-problem (2-15.c): Setting a single d.o.f. to zero ↬ GitLab

```
2  auto selector = [](lf::base::glb_idx_t idx) -> std::pair<bool, double> {
3    if (idx == 0) {
4      return {true, 0.0};  // fix first d.o.f.  to zero
5    }
6    return {false, 42.0};  // keep all others
7  };
8  lf::assemble::FixFlaggedSolutionComponents(selector, A_aux, rhs_vec);
```

SOLUTION of (2-15.d):

The kernel of the full $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrix $\mathbf{A}$ for

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x},$$

is one-dimensional and consists of constant vectors: $\mathcal{N}(\mathbf{A}) = \mathrm{Span}\{[1,1,\ldots,1]^\top\}$.

The only vector in $\mathcal{N}(\mathbf{A})$ with a vanishing first component is $\mathbf{0}$, of course. This corresponds to the fact that non-trivial constants can no longer be represented in $\mathcal{S}_1^0(\mathcal{M})$ if a single basis functions is suppressed.

For a square matrix a trivial kernel implies invertibility.

SOLUTION of (2-15.e):

Let us consider u to be the solution of (2.15.1). The second equation in (2.15.8) follows from the definition of $H^1_*(\Omega)$.

In order to tackle the first one, we define

$$\bar{v} = \frac{1}{|\Omega|}\int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x},$$

and $\tilde{v} := v - \bar{v} \in H^1_*(\Omega)$ for each $v \in H^1(\Omega)$.

This allows us rewrite $v = \tilde{v} + \bar{v} \in H^1(\Omega)$. As $\bar{v}$ is constant, and $\tilde{v} \in H^1_*(\Omega)$, we plug this into (2.15.8) and obtain:

$$\begin{aligned}\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, \tilde{v}\,\mathrm{d}\boldsymbol{x} + \lambda|\Omega|\bar{v} &= \int_\Omega f(\tilde{v}+\bar{v})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h(\tilde{v}+\bar{v})\,\mathrm{d}S \\ &= \int_\Omega f\tilde{v}\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\tilde{v}\,\mathrm{d}S + \bar{v}\Big(\int_\Omega f\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\,\mathrm{d}S\Big).\end{aligned}$$

Use the fact that u is the solution of (2.15.1) on $H^1_*(\Omega)$ and simplify the terms with $\tilde{v} \in H^1_*(\Omega)$. Then we get

$$\Big(\lambda|\Omega| - \int_\Omega f\,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} h\,\mathrm{d}S\Big)\bar{v} = 0, \forall \bar{v} \text{ with } v \in H^1(\Omega).$$

and conclude that u also solves (2.15.8) with $\lambda = \frac{1}{|\Omega|}\big(\int_\Omega f\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\,\mathrm{d}S\big)$.

Remark. Notice that $\lambda = 0$ when the compatibility condition (2.15.2) is satisfied. Test with constant functions in (2.15.8) to see this.

HINT 1 for (2-15.f): In light of (2.15.9) what is requested is the vector $\mathbf{c}$ and the extra component of $\vec{\vec{\varphi}}$. Give formulas for them. ⌟

SOLUTION of (2-15.f):

The FE Galerkin linear system for (2.15.8) is

$$\widetilde{\mathbf{A}}\vec{\widetilde{\boldsymbol{\mu}}} = \vec{\widetilde{\boldsymbol{\varphi}}}, \tag{2.15.10}$$

with the Galerkin matrix and right hand side vector augmented as follows:

$$\widetilde{\mathbf{A}} := \begin{bmatrix} \mathbf{A} & \mathbf{c} \\ \mathbf{c}^T & \mathbf{0} \end{bmatrix} \in \mathbb{R}^{N+1,N+1},$$

$$\vec{\widetilde{\boldsymbol{\mu}}} = \begin{bmatrix} \boldsymbol{\mu} \\ \lambda \end{bmatrix} \in \mathbb{R}^{N+1}, \quad \vec{\widetilde{\boldsymbol{\varphi}}} = \begin{bmatrix} \boldsymbol{\varphi} \\ 0 \end{bmatrix} \in \mathbb{R}^{N+1},$$

and where $\mathbf{c} \in \mathbb{R}^N$ has the components

$$(\mathbf{c})_j = \int_\Omega b_h^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,. \tag{2.15.11}$$

There are several ways to compute those:

1. If $\mathbf{M} \in \mathbb{R}^{N,N}$ is the Finite Element Galerkin mass matrix, that is, the Galerkin matrix for the bilinear form $(u,v) \mapsto \int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, then $\mathbf{c} = \mathbf{M1}$, $\mathbf{1}$ a vector with all components equal to 1.
2. From [Lecture → Lemma 2.7.5.5] with $d = 2$ we know

$$\int_K b_h^j = \frac{1}{3}|K| \quad \text{for} \quad K \subset \operatorname{supp} b_h^j \quad , \quad b_h^j \mathrel{\hat{=}} \text{a tent function}\,. \tag{2.15.12}$$

Hence

$$c_j = \tfrac{1}{3} \sum_{K:\mathbf{p}_j \in \overline{K}} |K|\,, \tag{2.15.13}$$

which can be efficiently computed using a *cell-oriented* accumulation of triangle areas.

SOLUTION of (2-15.g):

As we see from (2.15.13), the vector $\mathbf{c}$ can be built by cell-oriented local assembly using the element vectors $\frac{1}{3}|K|[1\,1\,1]^\top$. They are provided by **VecHelper**:

C++11 code 2.15.14: Sub-problem (2-15.g): VecHelper ⇝ GitLab

```
class VecHelper {
 public:
  explicit VecHelper() {}
  bool isActive(const lf::mesh::Entity &entity) const { return true; }
  Eigen::Vector3d Eval(const lf::mesh::Entity &entity) {
    LF_ASSERT_MSG(lf::base::RefEl::kTria() == entity.RefEl(),
                  "Function only defined for triangular cells");
    Eigen::Vector3d result;
    // Obtain shape information for the cell
    const lf::geometry::Geometry *geo_ptr = entity.Geometry();
    // Fetch area
    const double area = lf::geometry::Volume(*geo_ptr);
    // Initialize element vector |K|/3*[1,1,1]^T
    result = (area / 3.0) * Eigen::Vector3d::Ones();
    return result;
  }
};
```

To understand the following code, please study [Lecture → Ex. 2.7.4.27].

C++11 code 2.15.15: Sub-problem (2-15.g): `assembleVector_c` ⇝ GitLab

```
Eigen::VectorXd assembleVector_c(const lf::assemble::DofHandler &dofh) {
  Eigen::VectorXd c(dofh.NumDofs());
  // Do not forget to initialize vector before assembly!
  c.setZero();
  // ELEMENT_VECTOR_BUILDER object
  VecHelper my_vec_provider_c{};
  // Cell (= codim-0 entities)-oriented assembly into c
  lf::assemble::AssembleVectorLocally(0, dofh, my_vec_provider_c, c);
  return c;
}
```

HINT 1 for (2-15.h): Use your solution to Sub-problem (2-15.g) and manipulate the matrix **lf::assemble::C** object by calling its `AddToEntry()` member function. ⌟

SOLUTION of (2-15.h):

After setting up the standard $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrix in triplet format we add triplets for the extra row and column using the entries of $\mathbf{c}$:

C++11 code 2.15.16: Sub-problem (2-15.h): Excerpt of `getGalerkinLSE_augment()`
➥ GitLab

```
2   // Calculate vector c
3   Eigen::VectorXd c = assembleVector_c(dofh);
4   // Add c to the matrix
5   for (int i = 0; i < c.size(); i++) {
6     // add triplets for vector c
7     A_aux.AddToEntry(N_dofs - 1, i, c(i));
8     A_aux.AddToEntry(i, N_dofs - 1, c(i));
9   }
```

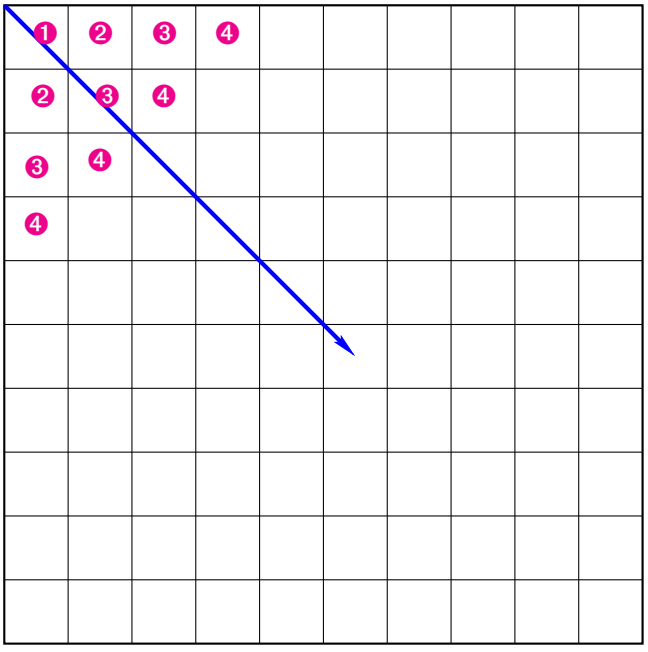
HINT 1 for (2-16.a): Remember from [Lecture → § 2.4.3.3] that an (affine) linear function $\mathbb{R}^2 \mapsto \mathbb{R}$ is already fixed by prescribing values in three non-collinear points. ⌟

HINT 2 for (2-16.a): Take into account that $W_N \subset H_0^1(\Omega)$. What does this mean for the piecewise polynomial functions $\in W_N$? ⌟

SOLUTION of (2-16.a):

Preparatory consideration: If a function $v \in Cp_1(\mathbb{R}^2)$ vanishes on two sides of a square, then it must vanish everywhere.

This is clear, because v is determined by its values in three non-collinear points [Lecture $\rightarrow$ § 2.4.3.3]; take the three corners of the square, in which v is zero.



The dimension of W_N is actually zero!

Recall that $W_N \subset H_0^1(\Omega)$ implies $v_h|_{\partial\Omega} \equiv 0$ for all $v_h \in W_N$.

❶ Start in a cell K abutting a corner. By our preparatory considerations we conclude $v_h|_K \equiv 0$.

❷ Proceed to cells adjacent to the corner cell. We already know that $v_h \in W_N$ must vanish on two of their sides and, thus, it must also vanish on them.

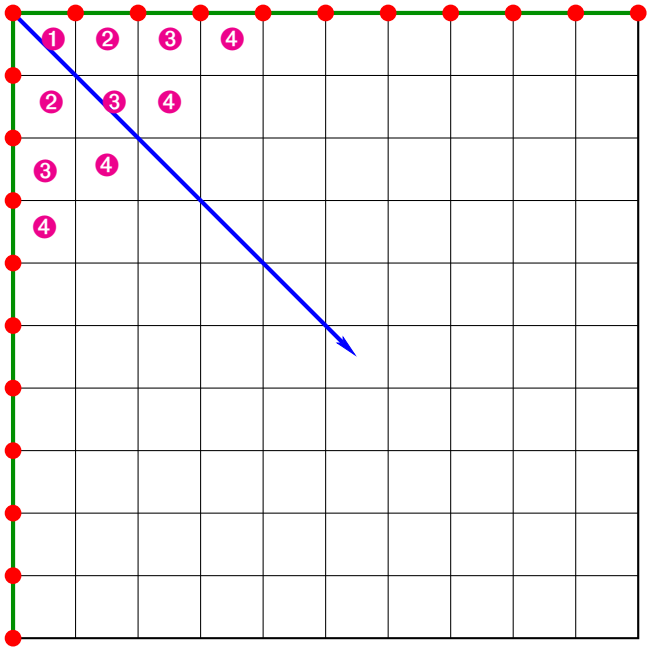
❸ Apply the same argument to the next layer of adjacent cells. This way we sweep the whole mesh and see that $v_h \equiv 0$ for any $v_h \in W_N$.

So the only function in W_N is the zero function: $\dim W_N = 0$!

HINT 1 for (2-16.b): In some mesh nodes located on $\partial\Omega$ function values can be imposed freely. Which nodes are these? ⌟

SOLUTION of (2-16.b):

Preparatory consideration: An affine linear function $v \in \mathcal{P}_1(\mathbb{R}^2)$ is already fixed by its values in three corners of a square.



1. Pick a *piecewise linear* function on the left and top side of Ω. It is determined by its values in the boundary nodes marked with ●.
2. Now adapt the argument of Sub-problem (2-16.a): When the values of $v_h \in V_N$ are set in the ●-nodes, v_h is fixed in square ❷ $\rightarrow$ fixed in squares ❷ $\rightarrow$ fixed in squares ❸ $\Rightarrow$ fixed in all cells of the mesh.
3. The functions $\in V_N$ that attain value $= 1$ in one of the ●-nodes and vanish in all others are clearly linearly independent.
4. The reasoning of Sub-problem (2-16.a) showed that a function $z_h \in V_N$ vanishing in all ●-nodes must be zero.

The functions $\in V_N$ that attain value $= 1$ in one of the ●-nodes and vanish in all others form a basis of V_N.

$\dim V_N = N_x + N_y - 1$ (= no. of ●-nodes) .

SOLUTION of (2-16.c):

This is the Lagrangian finite element space introduced in [Lecture → Section 2.6.2] and [Lecture → Ex. 2.6.2.1] and its dimension agrees with the total number $N_x \cdot N_y$ of nodes of the mesh, which serve as interpolation points in this case. Note that no boundary conditions are imposed in the case of B_N.

SOLUTION of (2-16.d):

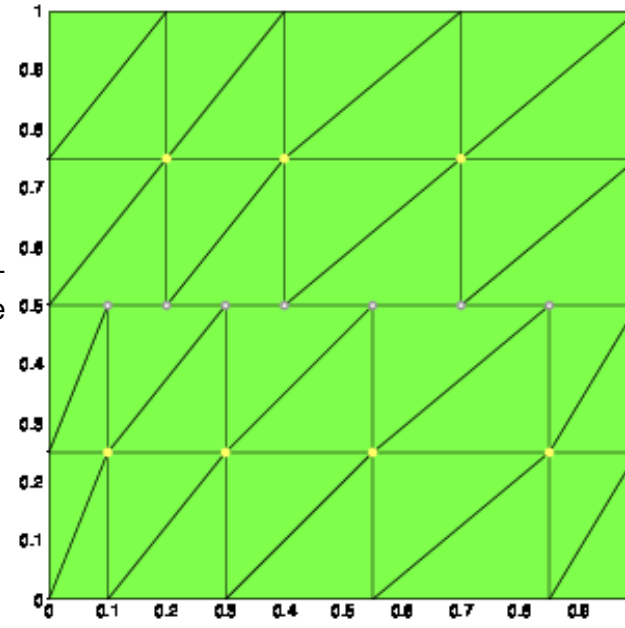
As we have zero boundary conditions, we only have degrees of freedom in the internal nodes. However, one must also neglect the dofs in the hanging nodes, since in these vertices we cannot define a basis function which is continuous in the domain and a linear polynomial when restricted to each element. Recall that a function $p \in \mathcal{P}_1$ is uniquely determined by the value at 3 non-aligned points, so the hanging nodes located at the midpoints pose a contradiction.

Actually, we can appeal to a 1D argument when we zero in on the line $y = \frac{1}{2}$: Let us assume that the interval $[0,1]$ is equipped with an equidistant (for simplicity) mesh with nodes $0 = x_0 < x_1 < \cdots < x_{n-1} < x_n = 1$ and midpoints $m_j := \frac{1}{2}(x_j + x_{j-1})$. Examine a function w that

- vanishes in $x = 0$ and $x = 1$,
- is linear on every interval $[x_{j-1}, x_j]$, $j = 1, \ldots, n$,
- is linear on every interval $[m_j, m_{j+1}]$, $j = 1, n-1$, and $[x_0, m_1]$, $[m_n, x_n]$,
- is *continuous* on $[0,1]$.

Then we conclude that $w(m_j) = \frac{1}{2}(w(x_j) + w(x_{j-1}))$ (n linear equations), $w(x_j) = \frac{1}{2}(w(m_j) + w(m_{j+1}))$ ($n-1$ linear equations), $w(x_0) = w(x_n) = 0$. This is a $(2n+1) \times (2n+1)$ linear system for the unknown values $w(x_j)$, $w(m_j)$, which has only the trivial solution $\mathbf{0}$.

In this concrete example, this leads us to 7 basis functions, as marked with yellow dots in Figure Fig. 19.



SOLUTION of (2-16.e):

For the same reasons as stated in the solution of Sub-problem (2-16.d), the functions have to be linear all along the line $y = \frac{1}{2}$.

Therefore we must neglect the dofs in the hanging nodes (hollow gray circles in Figure Fig. 19). Instead, the yellow dofs and all the boundary nodes are valid degrees of freedom. Thus, in this particular setting 24 “tent” basis functions remain.

A basis can be constructed as follows. First, for the node $(0, 0.5)$, we define the basis function along the line $y = 0.5$ as $1 - x$. We then linearly extend the function such that it is zero at $y = 0.25$ and $y = 0.75$. Everywhere outside $[0, 1] \times [0.25, 0.75]$, we define the function to be zero. For the node $(1, 0.5)$, we define the basis function along the line $y = 0.5$ as x and construct the basis function on the whole domain similarly as for the node $(0, 0.5)$. For all other nodes, we can simply use the standard tent functions.

SOLUTION of (2-17.a):

The largest pair of Sobolev spaces among $L^2(\Omega)$, $H^1(\Omega)$, and $H^2(\Omega)$ on which b is still continuous is

$$V_0 = H^1(\Omega) \quad , \quad W_0 = L^2(\Omega) \ ,$$

because by the Cauchy-Schwarz inequality we have

$$|\mathsf{b}(u,v)| \leq \|\boldsymbol{a}\|_{L^\infty(\Omega)} \int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\| |v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \leq \|\boldsymbol{a}\|_{L^\infty(\Omega)} \|\mathbf{grad}\, u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \ .$$

Since $\mathbf{grad}\, u$ occurs in the definition of b, choosing $V_0 = L^2(\Omega)$ is not possible.

SOLUTION of (2-17.b):

$$\dim V_{0,h} = \boxed{1} \cdot N_V + \boxed{0} \cdot N_e + \boxed{0} \cdot N_T + \boxed{0} \cdot N_Q \,,$$
$$\dim W_{0,h} = \boxed{0} \cdot N_V + \boxed{0} \cdot N_e + \boxed{1} \cdot N_T + \boxed{1} \cdot N_Q \,.$$

SOLUTION of (2-17.c):

- For $V_{0,h}$

```
lf::assemble::UniformFEDofHandler dh_V(
mesh_p, {{lf::base::RefEl::kPoint(),  1 },
  {lf::base::RefEl::kSegment(),  0 },
  {lf::base::RefEl::kTria(),  0 },
  {lf::base::RefEl::kQuad(),  0 }});
```

- For $W_{0,h}$

```
lf::assemble::UniformFEDofHandler dh_W(
mesh_p, {{lf::base::RefEl::kPoint(),  0 },
  {lf::base::RefEl::kSegment(),  0 },
  {lf::base::RefEl::kTria(),  1 },
  {lf::base::RefEl::kQuad(),  1 }});
```

SOLUTION of (2-17.d):

The local shape functions for $V_{0,h} = \mathcal{S}_1^0(\mathcal{M})$ are the barycentric coordinate functions

$$b_K^1(\boldsymbol{x}) = 1 - x_1 - x_2 \quad , \quad b_K^2(\boldsymbol{x}) = x_1 \quad , \quad b_K^3(\boldsymbol{x}) = x_2 \, .$$

This gives is us the constant gradients

$$\mathbf{grad}\, b_K^1(\boldsymbol{x}) = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad , \quad \mathbf{grad}\, b_K^2(\boldsymbol{x}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad , \quad \mathbf{grad}\, b_K^1(\boldsymbol{x}) = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \, .$$

The single local shape function q_K for $W_{0,h}$ is just $q_K \equiv 1$. Thus we arrive at the following formula for the entries of $\mathbf{A}_K$:

$$\begin{aligned} \mathbf{A}_K &= \left[\int_K \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \mathbf{grad}\, b_K^1 \, 1 \, \mathrm{d}\boldsymbol{x} \quad \int_K \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \mathbf{grad}\, b_K^2 \, 1 \, \mathrm{d}\boldsymbol{x} \quad \int_K \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \mathbf{grad}\, b_K^3 \, 1 \, \mathrm{d}\boldsymbol{x} \right] \\ &= \left[\int_K \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} -1 \\ -1 \end{bmatrix} \mathrm{d}\boldsymbol{x} \quad \int_K \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \end{bmatrix} \mathrm{d}\boldsymbol{x} \quad \int_K \begin{bmatrix} 1 \\ 2 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \mathrm{d}\boldsymbol{x} \right] \\ &= \begin{bmatrix} -\frac{3}{2} & \frac{1}{2} & 1 \end{bmatrix} \, , \end{aligned}$$

because $|K| = \frac{1}{2}$.

SOLUTION of (2-17.e):

Since $\boldsymbol{x} \mapsto \boldsymbol{a}(\boldsymbol{x})$ is a generic coefficient function, the use of numerical quadrature is mandatory. We use the edge-midpoint rule, available in LEHRFEM++ through the function call `make_TriaQR_EdgeMidpoi`

$$\int_K \varphi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3}\sum_{j=1}^{3}\varphi(\boldsymbol{m}_j)\,, \tag{2.17.4}$$

where $\boldsymbol{m}_j$, $j = 1,2,3$, are the midpoints of the edges of the triangle K. On the reference triangle $\widehat{K}$ the coordinates of these midpoints are

$$\widehat{\boldsymbol{m}}_1 = \begin{bmatrix} 0.5 \\ 0 \end{bmatrix} \quad , \quad \widehat{\boldsymbol{m}}_2 = \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \quad , \quad \widehat{\boldsymbol{m}}_3 = \begin{bmatrix} 0 \\ 0.5 \end{bmatrix} .$$

In the code the outer loop with index `i` runs over the three local shape functions of $V_{0.h} = \mathcal{S}_1^0(\mathcal{M})$ and the inner loop with loop index `j` performs the summation in (2.17.4).

C++ code 2.17.5: Implementation of `Eval()` member function for CDBLFElemMatProvider

```
template <typename MeshFunction>
Eigen::Matrix<double, 1, 3> CDBLFElemMatProvider<MeshFunction>::Eval(
const lf::mesh::Entity& tria) {
  // Throw error in case no triangular cell
  LF_VERIFY_MSG(tria.RefEl() == lf::base::RefEl::kTria(),
  "Unsupported cell type " << tria.RefEl());
  // Fetch constant gradients of barycentric coordinate functions
  // and the area of the triangle
  auto [grad_bary_coords, area] = getGradBaryCoords(tria);
  // Reference coordinates of quadrature nodes
  const Eigen::MatrixXd refqrnodes{
    (Eigen::MatrixXd(2, 3) << 0.5, 0.5, 0.0, 0.0, 0.5, 0.5).finished()};
  std::vector<Eigen::Vector2d> av_values{av_(tria, refqrnodes)};
  Eigen::Matrix<double, 1, 3> elmat{Eigen::Matrix<double, 1, 3>::Zero()};
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      elmat(0, i) += (grad_bary_coords.col(i)).transpose()*av_values[j];
    }
  }
  return (area / 3.0) * elmat;
}
```

HINT 1 for (2-18.a): Recall the formulas [Lecture → Eq. (1.2.3.48)] and [Lecture → Eq. (1.2.3.49)] for the gradient in polar coordinates and for integration in polar coordinates. ⌟

SOLUTION of (2-18.a):

The idea of the proof is to find a function $u \in H^1(\Omega)$ such that $\mathsf{a}(u,u) = \infty$. It goes without saying that the volume part of the bilinear form is continuous on $H^1(\Omega)$. Thus a blow-up can only be caused by the boundary terms, most likely by those involving a gradient. So lets try to find a function in $H^1(\Omega)$, Ω the unit disk, $\Omega := \{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\|_2 < 1\}$, whose gradient blows up as we approach $\partial\Omega$. The simplest choice is a function of $r := \|\boldsymbol{x}\|$ alone. So let us try

$$u(r) := (1-r)\log(1-r) + 1\,, \quad r := \|\boldsymbol{x}\| < 1\,,$$

for which we find

$$\mathbf{grad}\, u(\boldsymbol{x}) = (-1 - \log(1-r))\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|} \quad \blacktriangleright \quad \begin{aligned} \|\mathbf{grad}\, u\| &= \infty \quad \text{on} \quad \partial\Omega\,, \\ u &= 1 \quad \text{on} \quad \partial\Omega\,. \end{aligned} \tag{2.18.3}$$

First we have to verify that this u belongs to $H^1(\Omega)$. Since u is smooth in the interior of Ω this boils down to showing

$$\int_\Omega \|\mathbf{grad}\, u\|^2 \,\mathrm{d}\boldsymbol{x} + \int_\Omega u^2 \,\mathrm{d}\boldsymbol{x} < \infty\,.$$

We directly compute in polar coordinates

$$\int_\Omega \|\mathbf{grad}\, u\|^2 \,\mathrm{d}\boldsymbol{x} + \int_\Omega u^2 \,\mathrm{d}\boldsymbol{x} = 2\pi \int_0^1 \left((1+\log(1-r))^2 + (1-r)^2\log^2(1-r)\right)\mathrm{d}r < \infty\,,$$

because

$$\int_0^1 \log^2 r \,\mathrm{d}r = \Big[r(\log^2 r - 2\log r + 2)\Big]_0^1 = 2\,,$$

$$\int_0^1 \log^2 r \,\mathrm{d}r = \Big[r(\log^2 r - 1)\Big]_0^1 = 1\,,$$

and $r \mapsto r\log r$ is bounded anyway. Finally, $\mathsf{a}(u,u) = \infty$ is an immediate consequence of (2.18.3).

Closely related to this sub-problem is Problem 1-2.

HINT 1 for (2-18.b): Follow the two-step approach pursued in [Lecture → Ex. 1.5.3.11].

SOLUTION of (2-18.b):

❶ To tease out the PDE hidden in the variational problem we first test with $v \in C_0^\infty(\Omega)$, a smooth function with compact support inside Ω. In particular, such a v vanishes in a neighborhood of $\partial\Omega$ and, therefore, satisfies

$$v|_{\partial\Omega} = 0 \quad , \quad \mathbf{grad}\, v \cdot \boldsymbol{n}|_{\partial\Omega} = 0 \,.$$

This makes all boundary terms in $\mathsf{a}(\cdot,\cdot)$ and $\ell(\cdot)$ vanish and we infer

$$\mathsf{a}(u,v) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \ell(v) = 0 \quad \forall v \in C_0^\infty(\Omega) \,.$$

We integrate by parts $(*)$ applying Green's first formula

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ and functions $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ holds

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \mathrm{div}\, \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S \,. \qquad ([\text{Lecture} \to \text{Eq. (13.2.3.2)}])$$

and get

$$\int_\Omega -\Delta u\, v \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega) \quad \overset{(\star)}{\Rightarrow} \quad \Delta u = 0 \quad \text{in} \quad \Omega \,.$$

$(*)$: This is possible, because $u \in H^2(\Omega)$ means that Δu is well-defined as a function in $L^2(\Omega)$.

$(\star)$: Here we remember $\Delta u \in L^2(\Omega)$ and appeal to

Lemma [Lecture → Lemma 1.5.1.13]. fundamental lemma of the calculus of variations

Let $f \in C^0_{\mathrm{pw}}([a,b])$, $-\infty < a < b < \infty$, satisfy

$$\int_a^b f(\xi) v(\xi) \,\mathrm{d}\xi = 0 \quad \forall v \in C^\infty([a,b]), v(a) = v(b) = 0 \,.$$

Then $f \equiv 0$.

❷ Then we test with general $v \in C^\infty(\overline{\Omega})$ and apply Green's first formula again. This time the resulting boundary term cannot be dropped and the variational problem

$$u \in H^2(\Omega): \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in C^\infty(\overline{\Omega})$$

is converted into

$$\int_\Omega -\underbrace{\Delta u}_{=0} v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} (\mathbf{grad}\, u \cdot \boldsymbol{n}) v \,\mathrm{d}S - \int_{\partial\Omega} (\mathbf{grad}\, u \cdot \boldsymbol{n}) v \,\mathrm{d}S$$
$$- \int_{\partial\Omega} (\mathbf{grad}\, v \cdot \boldsymbol{n}) u \,\mathrm{d}S + c \int_{\partial\Omega} uv \,\mathrm{d}S$$

$$= c \int_{\partial\Omega} gv \,\mathrm{d}S - \int_{\partial\Omega} (\mathbf{grad}\, v \cdot \boldsymbol{n}) g \,\mathrm{d}S \quad \forall v \in C^\infty(\overline{\Omega}) \,.$$

Note that we have used the insight $\Delta u = 0$ from Step ❶. We also observe that some boundary terms cancel (marked in green) and we end up with

$$\begin{aligned} & -\int_{\partial\Omega} (\mathbf{grad}\, v \cdot \boldsymbol{n})(u-g) \,\mathrm{d}S + c \int_{\partial\Omega} (u-g) v \,\mathrm{d}S = 0 \quad \forall v \in C^\infty(\overline{\Omega}) \,, \\ \Rightarrow \quad & \int_{\partial\Omega} (cv - \mathbf{grad}\, v \cdot \boldsymbol{n})(u-g) \,\mathrm{d}S = 0 \quad \forall v \in C^\infty(\overline{\Omega}) \,. \end{aligned} \tag{2.18.5}$$

The idea to invoke a variant of

Lemma [Lecture → Lemma 1.5.3.16]. Fundamental lemma of calculus of variations on boundaries

Let $\Omega \subset \mathbb{R}^d$ be a bounded domain with piecewise smooth boundary and $\Gamma_0 \subset \partial\Omega$ a part of $\partial\Omega$ with non-zero measure. If $g \in C^0(\partial\Omega)$ satisfies

$$\int_{\Gamma_0} g(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in C^\infty(\overline{\Omega}) \,,$$

then g vanishes on Γ_0: $g|_{\Gamma_0} \equiv 0$.

The space $C^\infty(\overline{\Omega})$ gives us enough freedom to let $cv - \mathbf{grad}\, v \cdot \boldsymbol{n}$ vary over a "large enough" set of test functions in some variant of a "fundamental lemma of calculus of variations on boundaries" (The mathematical details are tricky). Thus (2.18.5) implies the Dirichlet boundary conditions

$$u = g \quad \text{on} \quad \partial\Omega \,.$$

SOLUTION of (2-18.c):

The localized bilinear form reads

$$\mathsf{a}_K(u,v) := \int_K \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_e (\mathbf{grad}\, u \cdot \boldsymbol{n}_K) v \,\mathrm{d}S - \int_e (\mathbf{grad}\, v \cdot \boldsymbol{n}_K) u \,\mathrm{d}S + c \int_e uv \,\mathrm{d}S\,,$$

where $\boldsymbol{n}_K$ is the exterior unit normal vector of K with $\boldsymbol{n}_K|_e = \frac{1}{2}\sqrt{2}\begin{bmatrix}1\\1\end{bmatrix}$. Recall that the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$ on a triangle are three barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ [Lecture → § 2.4.5.2], in this case

$$\lambda_1(\boldsymbol{x}) = 1 - x_1 - x_2 \quad , \quad \lambda_2(\boldsymbol{x}) = x_1 \quad , \quad \lambda_3(\boldsymbol{x}) = x_2\,. \tag{2.18.6}$$

Plugging them into the localized bilinear form yields the element matrix

$$\mathbf{A}_K = \left[\mathsf{a}_K(\lambda_i, \lambda_j)\right]_{i,j=1}^3 \in \mathbb{R}^{3,3}\,. \tag{2.18.7}$$

The gradients of the barycentric coordinate functions are constant:

$$\mathbf{grad}\,\lambda_1 = \begin{bmatrix}-1\\-1\end{bmatrix} \quad , \quad \mathbf{grad}\,\lambda_2 = \begin{bmatrix}1\\0\end{bmatrix} \quad , \quad \mathbf{grad}\,\lambda_3 = \begin{bmatrix}0\\1\end{bmatrix}. \tag{2.18.8}$$

We inspect the separate contributions to $\mathbf{A}_K$:

- The volume part of a_K spawns the $\mathcal{S}_1^0(\mathcal{M})$-element matrix for $-\Delta$ [Lecture → § 2.4.5.5]:

$$\int_K \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} \quad \leftrightarrow \quad \frac{1}{2}\begin{bmatrix}2 & -1 & -1\\ -1 & 1 & 0\\ -1 & 0 & 1\end{bmatrix}.$$

- For the contribution of e observe that $\lambda_1|_e \equiv 0$.

$$\int_e (\mathbf{grad}\, u \cdot \boldsymbol{n}_K) v \,\mathrm{d}S \quad \leftrightarrow \quad \frac{1}{2}\begin{bmatrix}0 & 0 & 0\\ -2 & 1 & 1\\ -2 & 1 & 1\end{bmatrix}.$$

- The other edge term contributes the transpose (not asked for):

$$\int_e (\mathbf{grad}\, v \cdot \boldsymbol{n}_K) u \,\mathrm{d}S \quad \leftrightarrow \quad \frac{1}{2}\begin{bmatrix}0 & -2 & -2\\ 0 & 1 & 1\\ 0 & 1 & 1\end{bmatrix}.$$

- The last contribution is a scaled "edge mass matrix" [Lecture → Eq. (2.7.4.44)]:

$$\int_e uv \,\mathrm{d}S \quad \leftrightarrow \quad \sqrt{2}\begin{bmatrix}0 & 0 & 0\\ 0 & 1/3 & 1/6\\ 0 & 1/6 & 1/3\end{bmatrix}.$$

Adding up all these contributions yields the final element matrix:

$$\mathbf{A}_K = \begin{bmatrix} 1 & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} + \frac{c}{3}\sqrt{2} & -1 + \frac{c}{6}\sqrt{2} \\ \frac{1}{2} & -1 + \frac{c}{6}\sqrt{2} & -\frac{1}{2} + \frac{c}{3}\sqrt{2} \end{bmatrix} .$$

Of course, $\mathbf{A}_K$ must be symmetric and, due to symmetry, swapping the rows and columns 2 and 3 must leave it unchanged.

SOLUTION of (2-18.d):

Obviously, the **ENTITY_MATRIX_PROVIDER** class **NitscheBoundaryMatProvider** is meant to compute the element contributions spawned by the boundary terms in a. Yet, the assembly performed by

```
lf::assemble::AssembleMatrixLocally(1, dofh, dofh, nitsche_bd_builder,
    A);
```

just covers edges on the boundary and, thus, can set only entries of the Galerkin matrix corresponding to global shape functions associated with boundary nodes. From Sub-problem (2-18.c) we see that also global shape functions associated with interior nodes will contribute matrix entries via the boundary terms. These contributions can never be taken into account by the given implementation.

This is the only flaw in the implementation. If there was only the boundary term $(u, v) \mapsto c \int_{\partial\Omega} uv \, \mathrm{d}S$, the given implementation could work.

HINT 1 for (2-18.e): You may copy & paste some parts from [Lecture → Code 2.7.4.37] and [Lecture → Code 2.7.4.45]. ⌟

HINT 2 for (2-18.e): In order to compute an exterior normal vector for an edge of a triangle K, note that

- the mapping $[x_1, x_2]^\top \mapsto [x_2, -x_1]^\top$ effects a rotation by $\pi/2$ in $\mathbb{R}^2$,
- and that the sign of the inner product of two vectors tells you whether they form an angle smaller or larger than $\pi/2$. This can be used to make the normal vector point into the exterior of K.

⌟

SOLUTION of (2-18.e):

As explained in Sub-problem (2-18.c) the local shape functions on the triangle K are its barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$. If $\widehat{\boldsymbol{x}} \mapsto \mathbf{F}_K + \boldsymbol{\tau}_K$, $\mathbf{F}_K \in \mathbb{R}^{2,2}$ regular, is the non-degenerate affine mapping taking the reference triangle to K, *cf.* [Lecture → Section 2.8.1], then [Lecture → Lemma 2.8.3.10] tells us that

$$\mathbf{grad}\, \lambda_i = \mathbf{F}_K^{\top} \mathbf{grad}\, \widehat{\lambda}_i \,.$$

Thus, using the gradients of the barycentric coordinate functions on the reference triangle from (2.18.8), a 2×3-matrix $\mathbf{G}$ holding the constant gradients $\mathbf{grad}\, \lambda_i$, $i = 1,2,3$ in its columns can be computed as

$$\mathbf{G} = \mathbf{F}_K^{\top} \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} . \tag{2.18.13}$$

Now, as in Sub-problem (2-18.c), we discuss the different parts of the bilinear form and their contribution to the element matrix

- Volume part $(u,v) \mapsto \int_K \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$ contributes $|K| \mathbf{G}^{\top}\mathbf{G}$, see [Lecture → Rem. 2.4.5.9].
- Consistency boundary part $(u,v) \mapsto \int_e (\mathbf{grad}\, u \cdot \boldsymbol{n}_e) v \,\mathrm{d}S$ for an edge e of K, which is located on the boundary, $e \subset \partial\Omega$ and has local number k.

  Since $\mathbf{grad}\, \lambda_i$ is constant, the contribution to the element matrix is

$$\begin{aligned} \left[\int_e (\mathbf{grad}\, \lambda_j \cdot \boldsymbol{n}_e)\, \lambda_i \,\mathrm{d}S \right]_{i,j=1}^{3} &= \left[\frac{1}{|e|} \int_e \lambda_i \,\mathrm{d}S \right]_{i=1}^{3} \left(\left[\mathbf{grad}\, \lambda_j \cdot |e| \boldsymbol{n}_e \right]_{j=1}^{3} \right)^{\top} \\ &= \left[\lambda_i(\boldsymbol{m}_e) \right]_{i=1}^{3} \left(|e| \boldsymbol{n}_e^{\top} \mathbf{G} \right) \\ &= (\mathbf{L})_{:,k} \left(|e| \boldsymbol{n}_e^{\top} \mathbf{G} \right) \quad \text{with} \quad \mathbf{L} = \begin{bmatrix} \frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} & 0 \\ 0 & \frac{1}{2} & 0 \end{bmatrix} . \end{aligned}$$

  Here $\boldsymbol{n}_e$ is the unit normal for e pointing into the exterior of K. To obtain the vector $|e|\boldsymbol{n}_e$, we start from the direction vector $\boldsymbol{d} := \boldsymbol{p}_2 - \boldsymbol{p}_1$ of e (with endpoints $\boldsymbol{p}_1, \boldsymbol{p}_2$, midpoint $\boldsymbol{m}_e$), turn it by $\frac{\pi}{2}$, $\boldsymbol{d}^{\perp} := [-d_2, d_1]^{\top}$, and check the sign of $\boldsymbol{d}^{\perp} \cdot (\boldsymbol{c} - \boldsymbol{p}_1)$, $\boldsymbol{c}$ the barycenter of K. If that quantity is positive, we have to flip $\boldsymbol{d}^{\perp}$.
- The contribution of $(u,v) \mapsto \int_e (\mathbf{grad}\, v \cdot \boldsymbol{n}_e) u \,\mathrm{d}S$ is just the transpose of what is computed in the previous item.
- The penalty term $(u,v) \mapsto c \int_e uv \,\mathrm{d}S$ for an edge e of K, which is located on the boundary, $e \subset \partial\Omega$ and has local number k, lets us add an "local edge mass matrix" to the element matrix. This contribution is

$$\frac{|e|}{6} \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \text{for} \quad k = 1 \,, \quad \frac{|e|}{6} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \quad \text{for} \quad k = 2 \quad , \quad \frac{|e|}{6} \begin{bmatrix} 2 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 2 \end{bmatrix} \quad \text{for} \quad k = 3 \,.$$

C++ code 2.18.14: Implementation of `Eval()` method for LinearFENitscheElementMatrix
➙ GitLab

```cpp
Eigen::Matrix3d LinearFENitscheElementMatrix::Eval(
    const lf::mesh::Entity &cell) const {
  LF_ASSERT_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
                "Only implemented for triangles");
  // The element matrix returned from this function
  Eigen::Matrix3d el_mat = Eigen::Matrix3d::Zero();
  // ----------------------------------------------------------------
  // I: Compute element matrix induced by volume part of bilinear form
  // ----------------------------------------------------------------
  // Fetch geometry object for current cell
  const lf::geometry::Geometry &K_geo{*(cell.Geometry())};
  LF_ASSERT_MSG(K_geo.DimGlobal() == 2, "Mesh must be planar");
  // Obtain physical coordinates of barycenter of triangle
  const Eigen::Vector2d center{K_geo.Global(c_hat_).col(0)};
  // Compute gradients of barycentric coordinate functions
  // Transformation matrix for gradients on reference triangle
  const Eigen::Matrix2d JinvT(K_geo.JacobianInverseGramian(c_hat_));
  // Transform gradients
  const Eigen::Matrix<double, 2, 3> G = JinvT * G_hat_;
  // Element matrix for linear finite elements and the Laplacian
  el_mat = lf::geometry::Volume(K_geo) * G.adjoint() * G;
  // ---------------------------------------
  // II: Boundary parts of the bilinear form
  // ---------------------------------------
  // Retrieve pointers to all edges of the triangle
  std::span<const lf::mesh::Entity *const> edges{cell.SubEntities(1)};
  LF_ASSERT_MSG(edges.size() == 3, "Triangle must have three edges!");
  // Loop over edges and check whether they are located on the bondary
  for (int k = 0; k < 3; ++k) {
    if (bd_flags_(*edges[k])) {
      // II(i): Contributions of consistency boundary parts of the bilinear form
      // Edge with local index k is an edge on the boundary
      // Fetch the coordinates of its endpoints
      const lf::geometry::Geometry &ed_geo{*(edges[k]->Geometry())};
      const Eigen::MatrixXd ed_pts{lf::geometry::Corners(ed_geo)};
      // Direction vector of the edge
      const Eigen::Vector2d dir = ed_pts.col(1) - ed_pts.col(0);
      // Rotate counterclockwise by 90 degrees
      const Eigen::Vector2d ed_normal = Eigen::Vector2d(dir(1), -dir(0));
      // For adjusting direction of normal so that it points into the exterior
      // of the domain
      const int ori = (ed_normal.dot(center - ed_pts.col(0)) > 0) ? -1 : 1;
      const Eigen::Matrix3d ed_mat =
          -L_.col(k) * ((ori * ed_normal.transpose()) * G);
      el_mat += (ed_mat + ed_mat.transpose());
      // II(ii): contributions of boundary penalty term
      const double fac = c_ * dir.norm();
      const int l = (k + 1) % 3;
      el_mat(k, k) += fac * 1.0 / 3.0;
      el_mat(k, l) += fac * 1.0 / 6.0;
      el_mat(l, k) += fac * 1.0 / 6.0;
      el_mat(l, l) += fac * 1.0 / 3.0;
    }
  }
  return el_mat;
```

```
57 }  // end Eval()
```

SOLUTION of (2-18.f):

When computing the element vector we have to respect LEHRFEM++'s numbering convention for the local shape functions as put forth in [Lecture → § 2.7.4.11]. In the case of $\mathcal{S}_1^0(\mathcal{M})$ on a triangular mesh $\mathcal{M}$ the local shape function are just the barycentric coordinate functions [Lecture → Ex. 2.5.3.6], which are associated with the vertices of a triangular cell and numbered according to the numbering of the vertices.

What complicates matters is the local definition of $\ell(v)$ on a cell K involves those edges of K that lie on the boundary of Ω:

$$\ell_K(v) = \sum_{e\in\mathcal{E}(K),e\subset\partial\Omega} c\int_e g(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) - \int_e \mathbf{grad}\, v(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,g(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,,\quad v\in C^\infty(\overline{K})\,. \tag{2.18.17}$$

This suggests that we visit all edges $e\in\mathcal{E}(K)$, $e\subset\partial\Omega$ and add their contributions to the element vector. Whether $e\subset\partial\Omega$ can easily be told by inspecting `bd_flags_(e)`.

The use of numerical quadrature is mandatory for the evaluation of the integrals in (2.18.17), because the function $\boldsymbol{x}\mapsto g(\boldsymbol{x})$ is given only in procedural form. We realize numerical quadrature by replacing $g|_e$ with its linear interpolant $\widetilde{g}$. If $e=[\boldsymbol{p}_0,\boldsymbol{p}_1]$, $\boldsymbol{p}_0,\boldsymbol{p}_1\in\partial\Omega$, then

$$\widetilde{g}(\boldsymbol{x}) = g(\boldsymbol{p}_0)(1-\widehat{x}) + g(\boldsymbol{p}_1)\widehat{x}\,,\quad \widehat{x}\mathrel{\widehat{=}}\text{local coordinate on } e\,.$$

Now, write $\boldsymbol{a}_K^0,\boldsymbol{a}_K^1,\boldsymbol{a}_K^2$ for the numbered vertices of K and consider the edge $e=[\boldsymbol{a}_K^k,\boldsymbol{a}_K^\ell]$, $k,\ell\in\{0,\ldots,2\}$ Then we have

$$c\int_e g(\boldsymbol{x})\lambda_i(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = |e|\begin{cases}\frac{1}{3}g(\boldsymbol{a}_K^k)+\frac{1}{6}g(\boldsymbol{a}_K^\ell) & \text{, if } i=k\,,\\ \frac{1}{6}g(\boldsymbol{a}_K^k)+\frac{1}{3}g(\boldsymbol{a}_K^\ell) & \text{, if } i=\ell\,,\\ 0 & \text{, if } i\notin\{k,\ell\}\,,\end{cases} \tag{2.18.18}$$

$$\int_e \mathbf{grad}\,\lambda_i(\boldsymbol{x})\cdot\boldsymbol{n}_e\,\widetilde{g}(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \tfrac{1}{2}|e|(g(\boldsymbol{a}_K^k)+g(\boldsymbol{a}_K^\ell))\,\mathbf{grad}\,\lambda_i\cdot\boldsymbol{n}_e\,. \tag{2.18.19}$$

To understand (2.18.18) see the explanations concerning the "local edge mass matrix" in Sub-problem (2-18 The formula (2.18.19) is due to the fact that $\boldsymbol{s}\mapsto\mathbf{grad}\,\lambda_i(\boldsymbol{x})$ is constant.

What is the error in Code 2.18.16? It is the use of the local numbering of the endpoints of an edge as obtained from the edge object. That local numbering has nothing to do with the local numbering of the current cell object, which is the one relevant for the element vector. When accessing the location of the endpoints of the edges we have to rely on the corner coordinates of the triangle!

The correct implementation is a follows:

C++ code 2.18.20: `Eval()` member function of class NitscheElemVecProvider: Corrected implementation ⇝ GitLab

```
2  template <typename FUNCTOR>
3  Eigen::Vector3d NitscheElemVecProvider<FUNCTOR>::Eval(
4      const lf::mesh::Entity &cell) const {
5    // Throw error in case no triangular cell
6    LF_VERIFY_MSG(cell.RefEl() == lf::base::RefEl::kTria(),
7                  "Unsupported cell type " << cell.RefEl());
8    // Fetch geometry object for current cell
9    const lf::geometry::Geometry &K_geo{*(cell.Geometry())};
10   const Eigen::MatrixXd cell_pts{lf::geometry::Corners(K_geo)};
```

```
  LF_ASSERT_MSG(K_geo.DimGlobal() == 2, "Mesh must be planar");
  // Obtain physical coordinates of barycenter of triangle
  const Eigen::Vector2d center{K_geo.Global(c_hat_).col(0)};
  // Compute gradients of barycentric coordinate functions
  // Transformation matrix for gradients on reference triangle
  const Eigen::Matrix2d JinvT(K_geo.JacobianInverseGramian(c_hat_));
  // Transform gradients
  const Eigen::Matrix<double, 2, 3> G = JinvT * G_hat_;
  // Element vector
  Eigen::Vector3d el_vec = Eigen::Vector3d::Zero();
  // Loop over edges and check whether they are
  // located on the bondary
  std::span<const lf::mesh::Entity *const> edges{cell.SubEntities(1)};
  for (int k = 0; k < 3; ++k) {
    if (bd_flags_(*edges[k])) {
      // Edge with local index k is an edge on the boundary
      // Its one endpoints has local vertex index k
      // Fetch the local index of the other end node
      const int l = (k + 1) % 3;
      // I: Contribution from consistency correction
      // Direction vector of the edge
      const Eigen::Vector2d dir = cell_pts.col(k) - cell_pts.col(l);
      // Rotate counterclockwise by 90 degrees
      const Eigen::Vector2d ed_normal = Eigen::Vector2d(dir(1), -dir(0));
      // For adjusting direction of normal
      const int ori = (ed_normal.dot(center - cell_pts.col(k)) > 0) ? -1 : 1;
      // Evaluate Dirichlet data function in the endpoints
      const double g0 = g_(cell_pts.col(k));
      const double g1 = g_(cell_pts.col(l));
      // Dirichlet data approximated as piecewise linear
      el_vec -= 0.5 * (g0 + g1) * (G.transpose() * (ori * ed_normal));
      // II: Contribution from penalty correction
      const double fac = c_ * dir.norm();
      el_vec[k] += fac * (g0 / 3.0 + g1 / 6.0);
      el_vec[l] += fac * (g1 / 3.0 + g0 / 6.0);
    }
  }
  return el_vec;
}
```

Remark. Then why did the implementation in Code 2.18.14 work, which follows the ideas of Code 2.18.16? It turns out that concerning the edge contributions to the element matrices the ordering of the endpoints of the edges does not matter!

SOLUTION of (2-19.a):

- As explained in [Lecture → Section 2.4.2] the dimension of $\mathcal{S}_1^0(\mathcal{M})$ on a triangular mesh $\mathcal{M}$ agrees with the number of nodes.

$$\dim \mathcal{S}_1^0(\mathcal{M}_M) = (M+1)^2 + M^2 = 2M^2 + 2M + 1 \ .$$

- The canonical basis functions of $\mathcal{S}_2^0(\mathcal{M}_M)$ are associated with the nodes and edges of the mesh [Lecture → Ex. 2.6.1.2], which means

$$\dim \mathcal{S}_2^0(\mathcal{M}_M) = \underbrace{(M+1)^2 + M^2}_{\sharp\text{nodes}} + \underbrace{2M(M+1) + 4M^2}_{\sharp\text{edges}} = 8M^2 + 4M + 1 \ ,$$

  because there are $2M(M+1)$ edges of the tensor-product mesh plus $4M^2$ edges inside the squares.

- In the case of V we drop all basis functions associated with nodes $\in \Gamma_D$. There are $2(M+1)$ such nodes, which implies

$$\dim V = \dim \mathcal{S}_1^0(\mathcal{M}_M) - 2(M+1) = (M+1)^2 + M^2 - 2(M+1) = 2M^2 - 1 \ .$$

- The space W emerges from $\mathcal{S}_2^0(\mathcal{M}_M)$ by dropping all canonical basis functions belonging to nodes or edges in Γ_D:

$$\dim W = \dim \mathcal{S}_2^0(\mathcal{M}_M) - 2(M+1) - 2M = 8M^2 + 4M + 1 - 4M - 2 = 8M^2 - 1 \ .$$

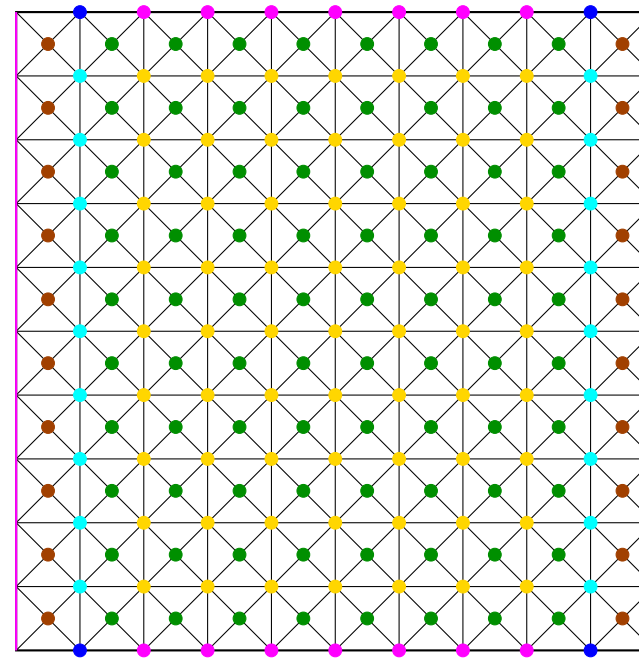
SOLUTION of (2-19.b):

Rows of the Galerkin matrix are associated with nodes of the mesh that do not lie on Γ_D. The number of non-zero entries in row i is the number of edges connecting node i with other nodes $\notin \Gamma_D$.

The figure shows the nodes $\notin \Gamma_0$ colored according to the number of "active" neighboring nodes:

- yellow nodes: eight neighbors,
- cyan nodes: seven neighbors,
- magenta nodes: five neighbors,
- green nodes: four neighbors,
- brown nodes: two neighbors.

Every row of $\mathbf{A}$ belonging to a an active node with k active neighbors has exactly k non-zero entries



- The maximal number of non-zero entries in a row of $\mathbf{A}$ is attained for rows belonging to yellow nodes, which have eight active neighbors

$$\max\{\sharp\{j \in \{1,\ldots,\dim V\} : (\mathbf{A})_{i,j} \neq 0\},\ i \in \{1,\ldots,\dim V\}\} = 9 \ .$$

- Brown nodes with only two active neighbors supply rows with a minimal number of non-zero entries

$$\min\{\sharp\{j \in \{1,\ldots,\dim V\} : (\mathbf{A})_{i,j} \neq 0\},\ i \in \{1,\ldots,\dim V\}\} = 3 \ .$$

- The number of non-zero entries of the Galerkin matrix can be read off Fig. 23 as number of active nodes plus the total number of their active neighbors. To that end we have to count the nodes of different color:
  - $\sharp\{\text{yellow nodes}\} = (M-1)(M-3)$ (eight neighbors),
  - $\sharp\{\text{cyan nodes}\} = 2(M-1)$ (seven neighbors),
  - $\sharp\{\text{magenta nodes}\} = 2(M-3)$ (five neighbors),
  - $\sharp\{\text{green nodes}\} = M(M-2)$ (four neighbors),
  - $\sharp\{\text{blue nodes}\} = 4$ (four neighbors),
  - $\sharp\{\text{brown nodes}\} = 2M$ (two neighbors)

$$\begin{aligned} \operatorname{nnz}(\mathbf{A}) &:= \sharp\{(i,j) \in \{1,\ldots,\dim V\}^2 : (\mathbf{A})_{i,j} \neq 0\} \\ &= \dim V + 8\cdot(M-1)(M-3) + 7\cdot 2(M-1) + 5\cdot 2(M-3) \\ &\quad + 4\cdot M(M-2) + 4\cdot 4 + 2\cdot 2M \\ &= 14M^2 - 12M - 5 \ . \end{aligned}$$

Alternative reasoning: If there were no essential boundary conditions, that is on the full space $\mathcal{S}_1^0(\mathcal{M}_M)$ the number of non-zero entries of the "full" Galerkin matrix would be the twice the number $6M^2+2M$ of edges of $\mathcal{M}_M$ (off-diagonal entries) plus the number $2M^2+2M+1$ of nodes of $\mathcal{M}_M$ (diagonal entries), which yields

$$\mathrm{nnz}(\text{full Galerkin matrix}) = 14M^2+6M+1\ .$$

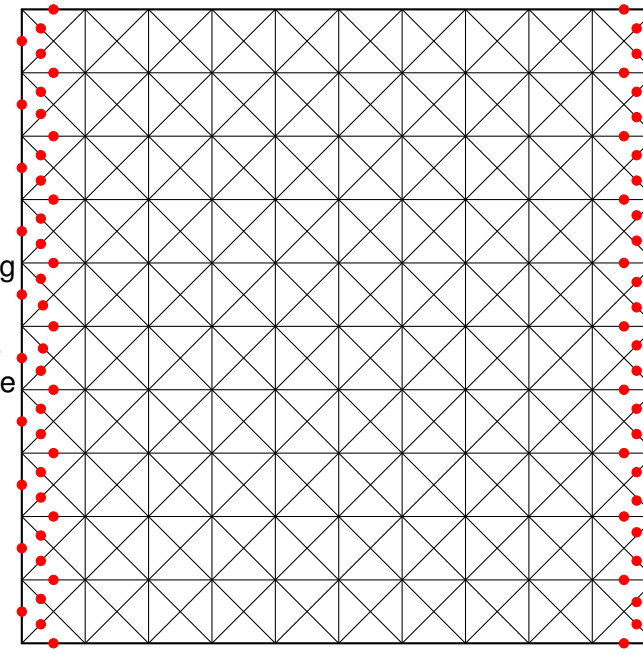
However, nodes on Γ_0 and edges connecting to those must be omitted.

There are

- $2(M+1)$ nodes on Γ_0,
- $2M$ edges on Γ_0,
- $2(M+1)$ horizontal edges emanating from nodes on Γ_0,
- $4M$ skew edges with an endpoint on Γ_0.

In the figure the edges to be dropped are marked with •. Their total number is

$$2M+2(M+1)+4M = 8M+2\ .$$



Summing up, we calculate

$$\begin{aligned}\mathrm{nnz}(\mathbf{A}) &= 2\Big(6M^2+2M-(8M+2)\Big)+\Big(2M^2+2M+1-(2M+2)\Big)\\ &= 14M^2-12M-5\ .\end{aligned}$$

SOLUTION of (2-19.c):

The entries of $\vec{\boldsymbol{\varphi}}$ are given by the formula

$$(\vec{\boldsymbol{\varphi}})_i = \int_{\partial\Omega} b_h^i \, \mathrm{d}S(\boldsymbol{x}) \;,$$

where $b_h^i \in V$ is a tent basis function associated with a node of $\mathcal{M}_M$ not located on Γ_D. Those nodes are marked with colored dots in Fig. 23.

Hence, $(\vec{\boldsymbol{\varphi}})_i \neq 0$, *if and only if* $\partial\Omega$ cuts through the interior of the support of b_h^i. This is the case only for the tent basis functions sitting on the blue and magenta nodes in Fig. 23, of which there are $2(M-1)$.

$$\sharp\{i \in \{1,\ldots,\dim V\} : \; (\vec{\boldsymbol{\varphi}})_i \neq 0\} = \boxed{2(M-1)} \;.$$

This reasoning is valid, because the tent basis functions are strictly positive in the interior of their support.

SOLUTION of (2-2.a):

From (2.2.2) and (2.2.6) we immediately deduce for $N = 4 \leftrightarrow M = 2$:

$$\mathbf{S} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 \end{bmatrix} \quad \begin{array}{l} \leftarrow \blacksquare \text{ (2.2.2a) for } j = 1 \\ \leftarrow \blacksquare \text{ (2.2.2a) for } j = 2 \\ \leftarrow \blacksquare \text{ (2.2.2b) for } j = 3 \\ \leftarrow \blacksquare \text{ (2.2.2b) for } j = 4 \end{array} .$$

We find $\mathbf{S}^\top \mathbf{S} = 2\mathbf{I}$, that is, up to scaling, the matrix $\mathbf{S}$ is orthogonal.

HINT 1 for (2-2.b): As in Sub-problem (2-2.a) compute $\mathbf{S}^\top \mathbf{S}$. ⌟

SOLUTION of (2-2.b):

From (2.2.2) and (2.2.6) we also see

$$\mathbf{S} = \left[\begin{array}{cccccccccc}
1 & 1 & 0 & 0 & \dots & & & & \dots & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & & & & \vdots \\
0 & 0 & 0 & 0 & 1 & 1 & & & & \\
\vdots & & & & & & \ddots & & & \vdots \\
 & & & & & & & 1 & 1 & 0 & 0 \\
0 & \dots & & & & & \dots & 0 & 0 & 1 & 1 \\
\hline
1 & -1 & 0 & 0 & \dots & & & & \dots & 0 \\
0 & 0 & 1 & -1 & 0 & 0 & & & & \vdots \\
0 & 0 & 0 & 0 & 1 & -1 & & & & \\
\vdots & & & & & & \ddots & & & \vdots \\
 & & & & & & & 1 & -1 & 0 & 0 \\
0 & \dots & & & & & \dots & 0 & 0 & 1 & -1
\end{array}\right]$$

← (2.2.2a) for $j = 1$
← (2.2.2a) for $j = 2$
← (2.2.2a) for $j = 3$
←
← (2.2.2a) for $j = M - 1$
← (2.2.2a) for $j = M$
← (2.2.2b) for $j = M + 1$
← (2.2.2b) for $j = M + 2$
← (2.2.2b) for $j = M + 3$
←
← (2.2.2b) for $j = N - 1$
← (2.2.2b) for $j = N$

A row permutation which can be achieved by left -multiplying $\mathbf{S}$ with an orthogonal matrix, we can transform it into a block diagonal matrix with M 2×2 diagonal blocks $\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$. Thus it becomes obvious that $\mathbf{S}^\top \mathbf{S} = 2\mathbf{I}$ as we have already seen in Sub-problem (2-2.a) for the special case $N = 4$.

HINT 1 for (2-2.c): You have to distinguish four cases according to whether i and j are in the range $\{1,\ldots,M\}$ or $\{M+1,\ldots,N\}$. This is necessary, because we clearly have two different types of "new" basis functions belonging to these index ranges. ⌟

SOLUTION of (2-2.c):

The entries of $\widetilde{\mathbf{A}}$ are given by $\left(\widetilde{\mathbf{A}}\right)_{i,j} = \mathsf{a}(\widetilde{b}_h^j, \widetilde{b}_h^i)$.

Using the defining linear combinations (2.2.2) of the new basis functions we get

$$
\begin{aligned}
1 \le i,j \le M (\blacksquare\blacksquare): \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = \mathsf{a}(b_h^{2j-1} + b_h^{2j}, b_h^{2i-1} + b_h^{2i}) \,, \\
M+1 \le i,j \le N (\blacksquare\blacksquare): \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = \mathsf{a}(b_h^{2(j-M)-1} - b_h^{2(j-M)}, b_h^{2(i-M)-1} - b_h^{2(i-M)}) \,, \\
M+1 \le i \le N, 1 \le j \le M (\blacksquare\blacksquare): \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = \mathsf{a}(b_h^{2j-1} + b_h^{2j}, b_h^{2(i-M)-1} - b_h^{2(i-M)}) \,, \\
1 \le i \le M, M+1 \le j \le N (\blacksquare\blacksquare): \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = \mathsf{a}(b_h^{2(j-M)-1} - b_h^{2(j-M)}, b_h^{2i-1} + b_h^{2i}) \,.
\end{aligned}
$$

The colored boxes, e.g. $(\blacksquare\blacksquare)$, indicate, which of the formulas (2.2.2a) or (2.2.2b) are relevant.

Then exploit the bi-linearity of a and the definition of the entries of the Galerkin matrix:

$$
\begin{aligned}
1 \le i,j \le M: \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = (\mathbf{A})_{2i,2j} + (\mathbf{A})_{2i,2j-1} + \\
& (\mathbf{A})_{2i-1,2j} + (\mathbf{A})_{2i-1,2j-1} \,, \\
M+1 \le i,j \le N: \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = (\mathbf{A})_{2(i-M)-1,2(j-M)-1} - (\mathbf{A})_{2(i-M),2(j-M)-1} - \\
& (\mathbf{A})_{2(i-M)-1,2(j-M)} + (\mathbf{A})_{2(i-M),2(j-M)} \,, \\
M+1 \le i \le N, 1 \le j \le M: \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = (\mathbf{A})_{2(i-M)-1,2j-1} - (\mathbf{A})_{2(i-M),2j-1} + \\
& (\mathbf{A})_{2(i-M)-1,2j} - (\mathbf{A})_{2(i-M),2j} \,, \\
1 \le i \le M, M+1 \le j \le N: \quad & \left(\widetilde{\mathbf{A}}\right)_{ij} = (\mathbf{A})_{2i-1,2(j-M)-1} + (\mathbf{A})_{2i,2(j-M)-1} - \\
& (\mathbf{A})_{2i-1,2(j-M)} - (\mathbf{A})_{2i,2(j-M)} \,.
\end{aligned}
\tag{2.2.7}
$$

HINT 1 for (2-2.d): Again distinguish four cases depending on whether k and ℓ are even or odd. ⌟

SOLUTION of (2-2.d):

We carefully examine the parity of the indices of the entries of $\mathbf{A}$ occurring on the right-hand-side of (2.2.7):

$$k,\ell \text{ even:} \quad (\mathbf{A})_{k,\ell} \to \left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2},\frac{\ell}{2}}, \left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2}+M,\frac{\ell}{2}+M}, -\left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2}+M,\frac{\ell}{2}}, -\left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2},\frac{\ell}{2}+M}, \tag{2.2.8}$$

$$k,\ell \text{ odd:} \quad (\mathbf{A})_{k,\ell} \to \left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2},\frac{\ell+1}{2}}, \left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2}+M,\frac{\ell+1}{2}+M}, \left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2},\frac{\ell+1}{2}+M}, \left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2}+M,\frac{\ell+1}{2}}, \tag{2.2.9}$$

$$k \text{ even, } \ell \text{ odd:} \quad (\mathbf{A})_{k,\ell} \to \left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2},\frac{\ell+1}{2}}, \left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2},\frac{\ell+1}{2}+M}, -\left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2}+M,\frac{\ell+1}{2}+M}, -\left(\widetilde{\mathbf{A}}\right)_{\frac{k}{2}+M,\frac{\ell+1}{2}}, \tag{2.2.10}$$

$$k \text{ odd, } \ell \text{ even:} \quad (\mathbf{A})_{k,\ell} \to \left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2},\frac{\ell}{2}}, -\left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2}+M,\frac{\ell}{2}+M}, \left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2}+M,\frac{\ell}{2}}, -\left(\widetilde{\mathbf{A}}\right)_{\frac{k+1}{2},\frac{\ell}{2}+M}. \tag{2.2.11}$$

SOLUTION of (2-2.e):

The implementatin is not difficult and amounts to coding (2.2.8)–(2.2.11) straightforwardly; the four different cases can be distinguished by simple **if** -statements. .

C++11 code 2.2.12: Solution code for Sub-problem (2-2.e) ⇢ GitLab

```
std::vector<Eigen::Triplet<double>> transformCOOmatrix(
    const std::vector<Eigen::Triplet<double>> &A) {
  std::vector<Eigen::Triplet<double>> A_t{};  // return value

  // First step:  find the size of the matrix by searching the maximal
  // indices.  Depends on the assumption that no zero rows/columns
      occur.
  int rows_max_idx = 0, cols_max_idx = 0;
  for (const Eigen::Triplet<double> &triplet : A) {
    rows_max_idx =
        (triplet.row() > rows_max_idx) ? triplet.row() : rows_max_idx;
    cols_max_idx =
        (triplet.col() > cols_max_idx) ? triplet.col() : cols_max_idx;
  }
  int n_cols = cols_max_idx + 1;

  // Make sure we deal with a square matrix
  assert(rows_max_idx + 1 == n_cols);
  // The matrix size must have even parity
  assert(n_cols % 2 == 0);

  int N = n_cols;  // Size of (square) matrix
  int M = N / 2;   // Half the size
  // clang-format off
  // Distribute entries of "old" matrix to new matrix
  for (const Eigen::Triplet<double> &it : A) {
    // row and column indices of current A triplet
    const int I = it.row() + 1; // k in (2.2.8)-(2.2.11)
    const int J = it.col() + 1; // ℓ in (2.2.8)-(2.2.11)

    // Distinguish different parities of indices
    if (I % 2 == 0 & J % 2 == 0) {
      // even, even, (2.2.8)
      A_t.emplace_back(I / 2 - 1, J / 2 - 1, it.value());
      A_t.emplace_back(I / 2 + M - 1, J / 2 + M - 1, it.value());
      A_t.emplace_back(I / 2 + M - 1, J / 2 - 1, -it.value());
      A_t.emplace_back(I / 2 - 1, J / 2 + M - 1, -it.value());
    } else if (I % 2 != 0 & J % 2 != 0) {
      // odd, odd, see (2.2.9)
      A_t.emplace_back((I + 1) / 2 - 1, (J + 1) / 2 - 1, it.value());
      A_t.emplace_back((I + 1) / 2 + M - 1, (J + 1) / 2 + M - 1, it.value());
      A_t.emplace_back((I + 1) / 2 - 1, (J + 1) / 2 + M - 1, it.value());
      A_t.emplace_back((I + 1) / 2 + M - 1, (J + 1) / 2 - 1, it.value());
    } else if (I % 2 == 0 & J % 2 != 0) {
      // even, odd, see (2.2.10)
      A_t.emplace_back(I / 2 - 1, (J + 1) / 2 - 1, it.value());
      A_t.emplace_back(I / 2 - 1, (J + 1) / 2 + M - 1, it.value());
      A_t.emplace_back(I / 2 + M - 1, (J + 1) / 2 + M - 1, -it.value());
      A_t.emplace_back(I / 2 + M - 1, (J + 1) / 2 - 1, -it.value());
    } else if (I % 2 != 0 & J % 2 == 0) {
```

```
51        // odd, even, see (2.2.11)
52        A_t.emplace_back((I + 1) / 2 - 1, J / 2 - 1, it.value());
53        A_t.emplace_back((I + 1) / 2 + M - 1, J / 2 + M - 1, -it.value());
54        A_t.emplace_back((I + 1) / 2 + M - 1, J / 2 - 1, it.value());
55        A_t.emplace_back((I + 1) / 2 - 1, J / 2 + M - 1, -it.value());
56      } else {
57        assert(false);
58      }
59    }
60    // clang-format on
61    return A_t;
62  }
```

The `emplace_back()` method adds an element to a vector calling its constructor with the supplied arguments.

HINT 1 for (2-2.f): Use the following theorem together with [Lecture → Lemma 2.2.3.2].

Theorem [Lecture → Thm. 0.3.1.26]. Real diagonalization of symmetric matrices

For every symmetric matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, that is, $\mathbf{A}^\top = \mathbf{A}$, we can find a diagonal matrix $D \in \mathbb{R}^{n,n}$ and an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{n,n}$ such that

$$\mathbf{Q}^\top \mathbf{A} \mathbf{Q} = \mathbf{D} \,. \tag{2.2.14}$$

⌟

SOLUTION of (2-2.f):

Let $\mathbf{A} \in \mathbb{R}^{N,N}$ be any Galerkin matrix for the s.p.d. bilinear form $a(\cdot,\cdot)$. According to the results of [Lecture → Section 2.2.3], in particular [Lecture → Rem. 2.2.3.8], $\mathbf{A}$ will be a symmetric and positive definite matrix [Lecture → Def. 0.3.1.17].

Then [Lecture → Thm. 0.3.1.26] guarantees the existence of a diagonal matrix $\mathbf{D} \in \mathbb{R}^{N,N}$ with *positive* diagonal entries, and of an orthogonal matrix $\mathbf{Q} \in \mathbb{R}^{N,N}$ such that $\mathbf{Q}^\top \mathbf{A}\mathbf{Q} = \mathbf{D}$. Then set $\mathbf{S} := \mathbf{D}^{-\frac{1}{2}}\mathbf{Q}^\top$ ($\mathbf{D}^{-\frac{1}{2}}$ is the diagonal matrix with the inverse square roots of $\mathbf{D}$ as its diagonal entries) and this regular matrix will satisfy $\mathbf{S}\mathbf{A}\mathbf{S}^\top = \mathbf{I}$. Hence, by [Lecture → Lemma 2.2.3.2] it defines a basis transformation that converts the Galerkin matrix into the identity matrix.

SOLUTION of (2-20.a):

The initialization of the d.o.f. handler object tells us that exactly one global shape function will be associated with every edge of the mesh, while two basis functions will be associated with every cell and none with the nodes. So the total number of global shape function/degrees of freedom managed by `dofh` is the number of edges of the mesh plus twice the number of cells

$$\texttt{dofh.NumDofs()} = 27 + 2 \cdot 15 = 57 \ .$$

A triangle has three edges and, thus, is inside the support of three edge-associated shape functions. Of course it also lies inside the support of its two cell-associated basis functions. So the number of *covering* local shape functions is $3 + 2 = 5$ for every triangle:

$$\texttt{dofh.NumLocalDofs(*cell_p)} = 5 \ .$$

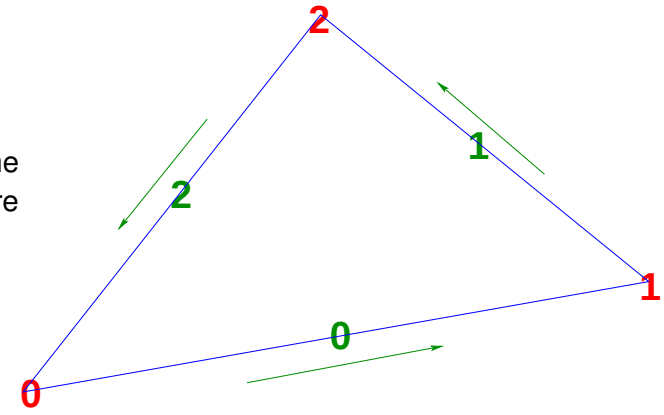
There are two shape functions *associated* with a triangle:

$$\texttt{dofh.NumInteriorDofs(*cell_p)} = 2 \ .$$

For details see [Lecture → § 2.7.4.13].

HINT 1 for (2-20.b):

LEHRFEM++'s local numbering convention for the edges of a triangular cell is displayed in the figure beside, *cf.* [Lecture → § 2.7.2.14].



⌟

HINT 2 for (2-20.b): Recall the numbering conventions of LEHRFEM++'s **lf::assemble::DofHandler** from [Lecture → Rem. 2.7.4.17]:

(I) D.o.f. associated with lower-dimensional entities are numbered first:

$$\textbf{POINT} \longrightarrow \textbf{SEGMENT} \longrightarrow \{\textbf{TRIA,QUAD}\}\,.$$

(II) The indices of d.o.f. belonging to entities of the same co-dimension increase with increasing entity indices as returned by the `Index()` function.

These rules govern both the *local and global* numbering of basis functions (local and global shape functions). ⌟

SOLUTION of (2-20.b):

The sub-entities of triangle no. 8 have the following indices

$$\begin{aligned} \text{vertices} &\leftrightarrow (7,8,3)\,, \\ \text{edges} &\leftrightarrow (18,11,10)\,, \end{aligned}$$

where the ordering reflects their local numbering.

According to [Lecture → Rem. 2.7.4.17] reproduced as hint, the basis functions associated with edges are numbered first and that numbering follows their global indices. This means that the indices of the edge-associated basis functions covering triangle 8 are $18, 11, 10$. The cell-associated basis functions are numbered after the edge-associated starting with the index 27 (= total number of edges). The two basis functions for a triangle are numbered consecutively. This means that the cell-associated basis functions for triangle 8 have the indices $27+16, 27+17$.

► `dofh.GlobalDofIndices(*cell_p)` $= \big($`18,11,10,43,44` $\big)$.

The call to `dofh.InteriorGlobalDofIndices()` just returns the global index numbers of the basis functions associated with the triangle 8:

► `dofh.InteriorGlobalDofIndices(*cell_p)` $= \big($`43,44` $\big)$.

C++ code 2.20.1: Test code

```
1 dofh.NumDofs() = 57
2 dofh.NumLocalDofs(*cell_p) = 5
3 dofh.NumInteriorDofs(*cell_p) = 2
4 dofh.GlobalDofIndices(*cell_p) = (18, 11, 10, 43, 44, )
5 dofh.InteriorGlobalDofIndices(*cell_p) = (43, 44, )
```

Console output of test code:

```
dofh.NumDofs() = 57
dofh.NumLocalDofs(*cell_p) = 5
dofh.NumInteriorDofs(*cell_p) = 2
dofh.GlobalDofIndices(*cell_p) = (18, 11, 10, 43, 44, )
dofh.InteriorGlobalDofIndices(*cell_p) = (43, 44, )
```

SOLUTION of (2-21.a):

By miraculous cancellation we have

$$\begin{aligned}
\mathbf{\Phi}\left(\begin{bmatrix}\xi\\0\end{bmatrix}\right) &= \gamma_{01}(\xi)+\gamma_{02}(1)-\gamma_{01}(0)+\xi\gamma_{12}(0)-\xi\gamma_{01}(1)-\xi\gamma_{20}(1)+\xi\gamma_{01}(0)\\
&= \gamma_{01}(\xi)\ ,\\
\mathbf{\Phi}\left(\begin{bmatrix}0\\\eta\end{bmatrix}\right) &= \gamma_{01}(0)+\gamma_{20}(1-\eta)-\gamma_{01}(0)+\eta\gamma_{12}(1)-\eta\gamma_{01}(0)-\eta\gamma_{20}(0)+\eta\gamma_{01}(0)\\
&= \gamma_{20}(1-\eta)\ ,\\
\mathbf{\Phi}\left(\begin{bmatrix}1-\tau\\\tau\end{bmatrix}\right) &= \gamma_{01}(1-\tau)+\gamma_{20}(1-\tau)-\gamma_{01}(0)+(1-\tau)\gamma_{12}(\tau)+\tau\gamma_{12}(\tau)-\\
&\quad (1-\tau)\gamma_{01}(1-\tau)-(1-\tau)\gamma_{20}(1-\tau)+\\
&\quad (1-\tau)\gamma_{01}(0)-\tau\gamma_{01}(1-\tau)-\tau\gamma_{20}(1-\tau)+\tau\gamma_{01}(0)\\
&= \gamma_{12}(\tau)\ .
\end{aligned}$$

for $0\le\xi\,\eta,\tau\le 1$. The first identity is (2.21.2) and is all that is required to solve this sub-problem.

SOLUTION of (2-21.b):

The mapping $\zeta \mapsto \begin{bmatrix} \zeta \\ 0 \end{bmatrix}$, $0 \leq \zeta \leq 1$, parameterizes the line segment $\left[\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}\right]$. Hence, the mapping $\mathbf{\Phi}$ maps this line segment to the curve given by the parameterization γ_{01}.

SOLUTION of (2-21.c):

This formula has already been given in the solution of Sub-problem (2-21.a):

$$\mathbf{\Phi}\left(\begin{bmatrix} 1-\tau \\ \tau \end{bmatrix}\right) = \gamma_{12}(\tau)\,, \quad 0 \le \tau \le 1\,.$$

HINT 1 for (2-21.d): The computations boil down to applying the product rule. Recall that the columns of the Jacobian are partial derivatives. ⌟

SOLUTION of (2-21.d):

The Jacobian is defined as

$$\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}}) = \left[\frac{\partial \boldsymbol{\Phi}}{\partial \widehat{x}_1}(\widehat{\boldsymbol{x}}), \frac{\partial \boldsymbol{\Phi}}{\partial \widehat{x}_2}(\widehat{\boldsymbol{x}})\right] \in \mathbb{R}^{2,2} .$$

Thus the colunns of the Jacobian are

$$\begin{aligned}
\left[\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})\right]_{:,1} &= \dot{\gamma}_{01}(\widehat{x}_1) + \gamma_{12}(\widehat{x}_2) - \widehat{x}_2\dot{\gamma}_{12}(1-\widehat{x}_1) - \gamma_{01}(1-\widehat{x}_2) - \gamma_{20}(1-\widehat{x}_2) \\
&\quad + \gamma_{01}(0) - \widehat{x}_2\dot{\gamma}_{01}(\widehat{x}_1) - \widehat{x}_2\dot{\gamma}_{20}(\widehat{x}_1) , \\
\left[\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})\right]_{:,2} &= -\dot{\gamma}_{20}(1-\widehat{x}_2) + \widehat{x}_1\dot{\gamma}_{12}(\widehat{x}_2) + \gamma_{12}(1-\widehat{x}_1) + \widehat{x}_1\dot{\gamma}_{01}(1-\widehat{x}_2) + \widehat{x}_1\dot{\gamma}_{20}(1-\widehat{x}_2) \\
&\quad - \gamma_{01}(\widehat{x}_1) - \gamma_{20}(\widehat{x}_1) + \gamma_{01}(0) .
\end{aligned}$$

SOLUTION of (2-21.e):

C++ code 2.21.3: Sub-problem (2-21.e): Implementation of `JacobianPhi`

```
Eigen::MatrixXd JacobianPhi(const coord_t& point, const Curve& gamma01,
                            const Curve& gamma12, const Curve& gamma20) {
  Eigen::MatrixXd J(2, 2);  // Variable for returning Jacobian
  // The formulas for the columns of the jacobian have been derived in
  // Sub-problem (2-21.d)
  J.col(0) =
      gamma01.derivative(point[0]) + gamma12(point[1]) -
      point[1] * gamma12.derivative(1. - point[0]) -
      (gamma01(1. - point[1]) + gamma20(1. - point[1]) - gamma01(0.)) -
      point[1] * (gamma01.derivative(point[0]) + gamma20.derivative(point[0]));
  J.col(1) = -gamma20.derivative(1. - point[1]) +
             point[0] * gamma12.derivative(point[1]) + gamma12(1. - point[0]) +
             point[0] * (gamma01.derivative(1. - point[1]) +
                         gamma20.derivative(1. - point[1])) -
             (gamma01(point[0]) + gamma20(point[0]) - gamma01(0.));
  return J;
}
```

HINT 1 for (2-21.f): Matrices of EIGEN are supplied with methods `determinant()`, `inverse()`, and `transpose()`. ⌟

SOLUTION of (2-21.f):

We implement [Lecture → Eq. (2.8.3.13)] with $\boldsymbol{\alpha} = \mathbf{I}$, $\widehat{b}^j = \lambda_j$ (barycentric coordinate functions, $j = 1,2,3$) and the formulas for the Jacobian $D\boldsymbol{\Phi}$ as found in Sub-problem (2-21.d). That Jacobian can be accessed via the function `JacobianPhi()` as realized in Sub-problem (2-21.e)

C++ code 2.21.4: Sub-problem (2-21.f): Implementation of `evalBlendLocMat`

```
Eigen::MatrixXd evalBlendLocMat(const Curve& gamma01, const Curve& gamma12,
                                const Curve& gamma20) {
  // Variable for returning 3 x 3 element matrix
  Eigen::MatrixXd lclMat(3, 3);
  lclMat.setZero();

  Eigen::MatrixXd xi(2, 3);  // coordinates of midpoints of curves
  xi.col(0) << 0.5, 0.;
  xi.col(1) << 0.5, 0.5;
  xi.col(2) << 0., 0.5;

  // (constant) gradients of barycentric coordinate function on the
     reference
  // element, which are the preimages of the local shape functions under
  // pullback
  Eigen::MatrixXd gradEval(3, 2);
  gradEval << -1., -1.,  // grad λ̂_1
      1., 0.,            // grad λ̂_2
      0., 1.;            // grad λ̂_3
  // Generate element matrix by adding up rank-1 matrices formed from
     gradients
  // of local shape functions at midpoints of edges.
  for (int l = 0; l < xi.cols(); ++l) {
    Eigen::Vector2d xi_l = xi.col(l);
    // Call auxiliary function implemented in Sub-problem (2-21.e)
    Eigen::MatrixXd Ji_l = JacobianPhi(xi_l, gamma01, gamma12, gamma20);
    double detJi_l = std::abs(Ji_l.determinant());
    // Transformation matrix for gradients, see [Lecture →
       Lemma 2.8.3.10]
    Eigen::MatrixXd invJT_l = Ji_l.inverse().transpose();
    // Rank-1 update of element matrix
    for (unsigned int i = 0; i < 3; i++) {
      for (unsigned int j = 0; j < 3; j++) {
        lclMat(i, j) += detJi_l *
                        (invJT_l * gradEval.row(i).transpose()).transpose() *
                        (invJT_l * gradEval.row(j).transpose());
      }
    }
  }

  // Don't forget the quadrature weight 1/6:  area
  // of the reference triangle = 1/2!
  return lclMat / 6.;
}
```

HINT 1 for (2-22.a): Read [Lecture → Rem. 1.8.0.14] and adapt the arguments. ⌟

SOLUTION of (2-22.a):

We follow the considerations from [Lecture → § 1.2.3.36].

For uniqueness, we have to prove that the bilinear form a underlying from the variational formulation of the problem is **positive definite** [Lecture → Def. 0.3.1.16], that is

$$\mathsf{a}(v,v) > 0 \quad \forall v \in H^1_\partial(\Omega) \setminus \{v = 0 \text{ in } \Omega\} \,. \tag{2.22.3}$$

Recasting the variational formulation (2.22.2) into the abstract form

$$\boxed{u \in \widehat{V}\text{:} \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0} \,, \qquad \text{[Lecture → Eq. (1.4.1.7)]}$$

we identify $\widehat{V} = V_0 = H^1_\partial(\Omega)$ and

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \, \mathrm{d}\boldsymbol{x} \quad , \quad \ell(v) := \int_\Omega f\, v \, \mathrm{d}\boldsymbol{x} \,,$$

where a is a bilinear form and ℓ is a linear form. To prove (2.22.3), we write

$$\mathsf{a}(v,v) = \int_\Omega \|\mathbf{grad}\, v\|^2 \, \mathrm{d}\boldsymbol{x} = 0 \quad \text{if and only if} \quad \mathbf{grad}\, v = 0 \,.$$

Then, $v = C$ on Ω, where C is a constant. Since $v \in H^1_\partial(\Omega)$, then $\int_{\partial\Omega} C \, \mathrm{d}S = 0$, that is $C = 0$, and a must be positive definite.

SOLUTION of (2-22.b):

The two-step approach is very similar to that elaborated in [Lecture → Ex. 1.5.3.11], where you can find more detailed explanations.

Step ❶: We first test with functions $v \in H_0^1(\Omega)$ and tease out the *PDE* by integration by parts: $-\Delta u = f$.

Step ❷: Then we test the PDE with $v \in H_\partial^1(\Omega)$, and find after integration by parts and using the PDE $-\Delta u = f$ we arrive at

$$\int_{\partial\Omega} \mathbf{grad}\, u \cdot \boldsymbol{n}\, v \,\mathrm{d}S = 0 \quad \forall v \in H_\partial^1(\Omega)\,. \tag{2.22.4}$$

Since the average of v over $\partial\Omega$ vanishes, a constant value for $\mathbf{grad}\, u \cdot \boldsymbol{n}$ satisfies this condition. If $\mathbf{grad}\, u \cdot \boldsymbol{n}$ is not constant, then (2.22.4) cannot hold, recall [Lecture → Lemma 1.5.3.16]. To flesh out the argument write

$$\varphi := \mathbf{grad}\, u \cdot \boldsymbol{n} : \partial\Omega \to \mathbb{R}\,, \quad \overline{\varphi} := \boldsymbol{x} \in \partial\Omega \to \frac{1}{|\partial\Omega|}\int_{\partial\Omega} \varphi(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \quad \varphi_0 := \varphi - \overline{\varphi}\,.$$

Then in (2.22.4) set "$v := \varphi_0$", which is possible, because $\int_{\partial\Omega} \varphi_0\,\mathrm{d}S = 0$. This implies

$$\int_{\partial\Omega} |\varphi_0(\boldsymbol{x})|^2\,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \Rightarrow \quad \varphi_0 = 0\,.$$

The quotation marks above indicate that in rigorous terms v is chosen as an arbitrarily close approximation of φ_0.

Thus we have found the *boundary condition*

$$\mathbf{grad}\, u \cdot \boldsymbol{n} \equiv \mathrm{const} \quad \text{on} \quad \partial\Omega\,.$$

This is "almost" a zero Neumann boundary condition.

SOLUTION of (2-22.c):

Since $\int_{\partial\Omega} \mu u \, \mathrm{d}S = 0 \quad \forall \mu \in \mathbb{R}$, the second equation implies $\int_{\partial\Omega} u \, \mathrm{d}S = 0$, which means $u \in H^1_\partial(\Omega)$.

Next, we test the first equation with $v \in H^1_\partial(\Omega) \subset H^1(\Omega)$, which implies $\int\limits_{\partial\Omega} \eta \, v \, \mathrm{d}\boldsymbol{x} = 0$. Therefore the solution u of the variational problem (2.22.5) solves the variational problem (2.22.2).

SOLUTION of (2-22.d):

We follow the same procedure of Sub-problem (2-22.b) for (2.22.5). Using integration by parts and the PDE $-\Delta u = f$ satisfied by u, from the first equation of (2.22.2) we infer

$$\eta \int_{\partial\Omega} v \, \mathrm{d}S = -\int_{\partial\Omega} \frac{\partial u}{\partial \boldsymbol{n}} v \, \mathrm{d}S \quad \forall v \in H^1(\Omega) \, . \tag{2.22.6}$$

From Sub-problem (2-22.b) we know that $\frac{\partial u}{\partial \boldsymbol{n}} \equiv$ const on $\partial\Omega$. Thus, from (2.22.6) we obtain $\eta = -$const , that is, the solution for η gives the negative constant value of the boundary flux.

SOLUTION of (2-22.e):

Entries of matrices and vectors arise from plugging basis functions into the (bi-)linear forms of (2.22.5):

$$
\begin{aligned}
(\mathbf{A})_{i,j} &= \int_{\Omega} \mathbf{grad}\, b_h^i(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^j(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad i,j = 1,\ldots,N\,,\\
(\mathbf{c})_i &= \int_{\partial\Omega} b_h^i(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})\,, \quad i = 1,\ldots,N\,,\\
(\vec{\boldsymbol{\varphi}})_i &= \int_{\Omega} f(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad i = 1,\ldots,N\,.
\end{aligned}
$$

HINT 1 for (2-22.f): Carry out matrix manipulations in triplet format (data type **lf::assemble::COOMatrix**) and build the sparse matrix in a final step using the `AddToEntry()` member function of **lf::assemble::CO**

⌟

SOLUTION of (2-22.f):

We assemble the augmented matrix using the block form given in (2.22.7).

C++ code 2.22.9: Sub-problem (2-22.f): Implementation of `augmentMatrix`

```
Eigen::SparseMatrix<double> augmentMatrix(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_spc,
    const Eigen::VectorXd& c) {
  const lf::assemble::DofHandler* dofh = &fe_spc->LocGlobMap();
  int N_dofs = dofh->NumDofs();
  lf::assemble::COOMatrix<double> A =
      lf::assemble::COOMatrix<double>(N_dofs + 1, N_dofs + 1);
  auto const_one = [](const Eigen::Vector2d& /*x*/) -> double { return 1.0; };
  auto const_zero = [](const Eigen::Vector2d& /*x*/) -> double { return 0.0; };
  // We first assemble the upper left block of the augmented matrix
  A = compGalerkinMatrix(*dofh, const_one, const_zero, const_zero);
  // Then we add the vector c^T to the last row and c to the last column
     of
  // the augmented matrix
  for (int i = 0; i < N_dofs; ++i) {
    A.AddToEntry(N_dofs, i, c[i]);
    A.AddToEntry(i, N_dofs, c[i]);
  }

  return A.makeSparse();
}
```

HINT 1 for (2-22.g): You may use the method `setFromTriplets` of **Eigen::SparseMatrix<double>** to create a matrix from the data returned by `compGalerkinMatrix`. ⌟

SOLUTION of (2-22.g):

From Sub-problem (2-22.f) we know that

$$(\mathbf{c})_i = \int_{\partial\Omega} b_h^i(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \quad i \in \{1,\ldots,N\}\,.$$

Recall the bilinear form, for which `compGalerkinMatrix()` assembles the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix:

$$(u,v) \mapsto \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\beta(\boldsymbol{x})\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,,$$

$u, v \in H^1(\Omega)$. Obviously, the highlighted boundary term will be relevant. It can be extracted by passing $\alpha \equiv 0$, $\gamma \equiv 0$, and $\beta \equiv 1$ to `compGalerkinMatrix()`, which makes it assemble (in triplet format) the matrix

$$\mathbf{B} \in \mathbb{R}^{N,N} \quad , \quad (\mathbf{B})_{i,j} = \int_{\partial\Omega} b_h^i(\boldsymbol{x})\,b_h^j(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,, \quad i,j = 1,\ldots,N\,. \tag{2.22.10}$$

How can we get the components of $\mathbf{c}$ from $\mathbf{B}$,

$$(\mathbf{c})_i = \int_{\partial\Omega} b_h^i(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \quad \longleftrightarrow \quad (\mathbf{B})_{i,j} = \int_{\partial\Omega} b_h^i(\boldsymbol{x})\,b_h^j(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\ ?$$

Idea: The basis $\{b_h^\ell\}_{\ell=1}^N$ forms a **partition of unity**:

$$\sum_{j=1}^N b_h^j(\boldsymbol{x}) = 1 \quad \forall \boldsymbol{x} \in \Omega\,!$$

This implies

$$(\mathbf{c})_i = \int_{\partial\Omega} b_h^i(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_{\partial\Omega} b_h^i(\boldsymbol{x})\sum_{j=1}^N b_h^j(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \sum_{j=1}^N \int_{\partial\Omega} b_h^i(\boldsymbol{x})\,b_h^j(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \sum_{j=1}^N (\mathbf{B})_{i,j}\,.$$

Remembering the rules of matrix×vector multiplication we conclude

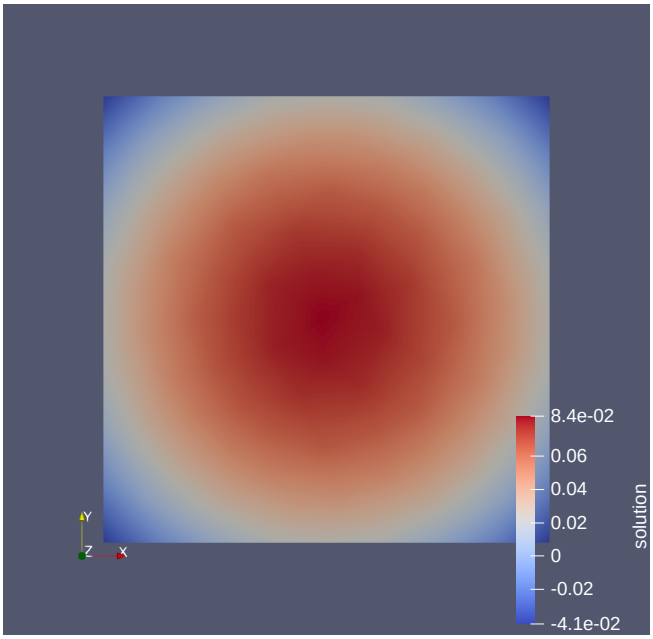
$$\mathbf{c} = \mathbf{B}\mathbf{1}\,, \quad \mathbf{1} = \begin{bmatrix}1 & 1 & \cdots & 1\end{bmatrix}^\top \in \mathbb{R}^N\,.$$

The following implementation is straightforward.

C++ code 2.22.11: Sub-problem (2-22.f): Implementation of `computeCVector`

```
2  template <typename FESPACE>
3  Eigen::VectorXd computeCVector(const FESPACE& fe_test) {
4    Eigen::VectorXd sol;
5    const lf::assemble::DofHandler* dofh = &fe_test->LocGlobMap();
6    int N_dofs = dofh->NumDofs();
7    auto const_one = [](const Eigen::Vector2d& /*x*/) -> double { return 1.0; };
8    auto const_zero = [](const Eigen::Vector2d& /*x*/) -> double { return 0.0; };
9    // We first assemble the Matrix B
10   Eigen::SparseMatrix<double> B =
11       compGalerkinMatrix(*dofh, const_zero, const_zero, const_one).makeSparse();
12   // And multiply B then with the constant vector 1 to obtain the
       CVector
```

```
13    sol = B * Eigen::VectorXd::Constant(N_dofs, 1.);
14
15    return sol;
16  }
```



◁ Surface plot of the solution, computed on the unit square with $f(\boldsymbol{x}) = 1$.

SOLUTION of (2-23.a):

- $\mathbf{M}$ is always symmetric, always positive definite and always regular. No statement can be made about sparsity.
- $\mathbf{A}$ is always symmetric, but only sometimes positive definite and regular. Again, no statement can be made about sparsity.
- $\mathbf{A}+\mathbf{M}$ is always symmetric, always positive definite and always regular. No statement can be made about sparsity.

The symmetry of the matrices is an immediate consequence of the symmetry of both bilinear forms. Since

$$\mathsf{m}(v,v) > 0 \quad \forall v \in L^2(\Omega) \setminus \{\boldsymbol{x} \mapsto 0\} \,,$$

$\mathbf{M}$ must be positive definite. It is true that

$$\mathsf{a}(v,v) \geq 0 \quad \forall v \in H^1(\Omega) \,,$$

but we have $\mathsf{a}(\{\boldsymbol{x} \mapsto 1\}, \{\boldsymbol{x} \mapsto 1\}) = 0$, so that $\mathbf{A}$ may not be positive definite. The statement about the regularity of the matrices employs the same arguments.

We do not know, whether the basis functions are locally supported. This makes it impossible to infer sparsity.

matrix	symmetric	positive definite	regular	sparse
$\mathbf{M}$	yes ✔	yes ✔	yes ✔	yes ○
	no ○	no ○	no ○	no ✔
$\mathbf{A}$	yes ✔	yes ○	yes ○	yes ○
	no ○	no ✔	no ✔	no ✔
$\mathbf{A}+\mathbf{M}$	yes ✔	yes ✔	yes ✔	yes ○
	no ○	no ○	no ○	no ✔

SOLUTION of (2-23.b):

- From [Lecture → Section 2.4.3] we know that $\dim LFE$ agrees with the number of nodes of the mesh:

$$\dim \mathcal{S}_1^0(\mathcal{M}) = \sharp\mathcal{V}(\mathcal{M}) \ .$$

- In Sub-problem (2-23.a) we have found that $\mathbf{M}$ is positive definite. Therefore this matrix can only have a trivial kernel:

$$\dim \mathcal{N}(\mathbf{M}) = 0 \ .$$

- The constant function and only that belongs to the "kernel" of a, *cf.* [Lecture → Rem. 1.8.0.14]. Therefore,

$$\mathcal{N}(\mathbf{A}) = \mathbf{1} \quad \Rightarrow \quad \dim \mathcal{N}(\mathbf{A}) = 1 \ .$$

SOLUTION of (2-23.c):

We have to carry out “cell-oriented assembly with pen and paper”, following the algorithm of [Lecture → Code 2.4.5.16] from [Lecture → Section 2.4.5.2].

Looking at [Lecture → Fig. 78], we see that the entry of $\mathbf{A}$ associated with a node of the mesh is obtained by summing diagonal entries of the element matrices of all adjacent triangles. Since all those diagonal entries $= 2$, we have

$$(\mathbf{A})_{i,i} = 2 \cdot \sharp(\text{triangles abutting node } i) \ .$$

In Fig. 29 there are nodes with $2, 3, 6$ neighboring cells, which means

$$\left\{ (\mathbf{A})_{i,i} \, , \ i = 1, \ldots, \dim \mathcal{S}_1^0(\mathcal{M}) \right\} = \{4, 6, 12\} \ .$$

SOLUTION of (2-24.a):

We refer to [Lecture → Ex. 2.8.3.29] and recall the local quadrature formula

$$\int_{\widehat{K}} f(\widehat{\boldsymbol{x}})\,\mathrm{d}\widehat{\boldsymbol{x}} \approx \sum_{\ell=1}^{P} \widehat{\omega}_\ell f(\widehat{\boldsymbol{\zeta}}_\ell)\;, \qquad \text{([Lecture → Eq. (2.7.5.23)])}$$

which gives rise to the following formula for the element matrix $\mathbf{A}_K \in \mathbb{R}^{Q,Q}$:

$$(\mathbf{A}_K)_{ij} = \sum_{\ell=1}^{P} \widehat{\omega}_\ell \Big(\boldsymbol{\alpha}(\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))(\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))^{-\top} \mathbf{grad}\, \widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell)) \cdot ((\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))^{-\top} \mathbf{grad}\, \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)) + \gamma(\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_\ell) \Big) |\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)|\;,$$

([Lecture → Eq. (2.8.3.13)])

for $i,j \in \{1,\ldots,Q\}$ and with reference shape functions $\widehat{b}^1,\ldots,\widehat{b}^Q$. A detailed derivation in given [Lecture → Section 2.8.3]. Matching [Lecture → Eq. (2.8.3.13)] with the placeholders yields

expression	A	B	C	D	E	F	G	H
$\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)$	○	○	○	○	○	○	○	✔
$\widehat{\omega}_\ell$	✔	○	○	○	○	○	○	○
$\widehat{\boldsymbol{\zeta}}_\ell$	○	○	✔	✔	✔	○	○	○
$\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_\ell)$	○	○	○	○	○	✔	○	○
$\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)$	○	✔	○	○	○	○	✔	○
$\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell)$	○	○	○	○	○	○	○	○
$(\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}}_\ell))^\top$	○	○	○	○	○	○	○	○

SOLUTION of (2-24.b):

- `lf::geometry::Geometry::Global()` realizes the mapping $\boldsymbol{\Phi} : \widehat{K} \to K$, see [Lecture → § 2.7.5.17].
- `lf::quad::QuadRule::Weights()` provides the weights $\widehat{\omega}_\ell$ of the local quadrature rule on $\widehat{K}$, refer to [Lecture → § 2.7.5.39].
- `lf::geometry::Geometry::IntegrationElement()` returns $\det \mathrm{D}\boldsymbol{\Phi}$, recall [Lecture → § 2.7.5.24].
- `lf::quad::QuadRule::Points()` gives the quadrature nodes $\widehat{\boldsymbol{\zeta}}_\ell$ on $\widehat{K}$, refer to [Lecture → § 2.7.5.39].
- `lf::geometry::Geometry::JacobianInverseGramian()` provides $\mathrm{D}\boldsymbol{\Phi}^{-\top}$, see [Lectur → § 2.8.3.14].

Member function	$\widehat{\boldsymbol{\zeta}}_\ell$	$\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}})$	$\mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}})^{-\top}$	$\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{\zeta}})$
`Geometry::Global()`	○	✔	○	○
`lf::quad::QuadRule::Weights()`	○	○	○	○
`Geometry::IntegrationElement()`	○	○	○	✔
`lf::quad::QuadRule::Points()`	✔	○	○	○
`Geometry::JacobianInverseGramian()`	○	○	✔	○

HINT 1 for (2-3.a): Refresh yourself on the contents of [Lecture → Section 1.5.1]. Use integration by parts and the "Fundamental lemma of the calculus of variations":

Lemma [Lecture → Lemma 1.5.1.13]. fundamental lemma of the calculus of variations

Let $f \in C^0_{\mathrm{pw}}([a,b])$, $-\infty < a < b < \infty$, *satisfy*

$$\int_a^b f(\xi)v(\xi)\,\mathrm{d}\xi = 0 \quad \forall v \in C^\infty([a,b]), v(a) = v(b) = 0\,.$$

Then $f \equiv 0$.

⌟

SOLUTION of (2-3.a):

Assume $u \in C^2_{0,\mathrm{pw}}([a,b])$ solves (2.3.1). Let $v \in C^1_{0,\mathrm{pw}}([a,b])$ be a test function. We integrate by parts in 1D using

$$\int_a^b f(\xi)v'(\xi)\,\mathrm{d}\xi = -\int_a^b f'(\xi)v(\xi) + \underbrace{(f(b)v(b) - f(a)v(a))}_{\text{boundary terms}} \quad \forall f, v \in C^1_{\mathrm{pw}}([a,b])\,,$$

[Lecture → Eq. (1.5.1.8)]

and, using the zero boundary conditions implicit in the trial/test space $H^1_0(\Omega)]a,b[$, arrive at:

$$\begin{aligned}\int_a^b \frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x &= \left[\frac{du}{dx}(x)v(x)\right]_a^b - \int_a^b \frac{d^2u}{dx^2}(x)\,v(x)\,\mathrm{d}x \\ &= \underbrace{\frac{du}{dx}(b)v(b) - \frac{du}{dx}(a)v(a)}_{=0 \text{ because } v(a)=v(b)=0} - \int_a^b \frac{d^2u}{dx^2}(x)\,v(x)\,\mathrm{d}x \\ &= -\int_a^b \frac{d^2u}{dx^2}(x)\,v(x)\,\mathrm{d}x.\end{aligned}$$

We insert the right hand side of (2.3.1) and see that

$$\int_a^b \frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x = -\int_a^b \frac{d^2u}{dx^2}(x)\,v(x)\,\mathrm{d}x = \int_a^b g(x)v(x)\,\mathrm{d}x.$$

That is,

$$\int_a^b \frac{d^2u}{dx^2}(x)\,v(x)\,\mathrm{d}x + \int_a^b g(x)v(x)\,\mathrm{d}x = 0 \quad \forall v \in C^1_{0,\mathrm{pw}}([a,b])\,.$$

We apply [Lecture → Lemma 1.5.1.13] to see that

$$\frac{d^2u}{dx^2}(x) + g(x) = 0\,.$$

The boundary conditions we read directly from the space, and we see that $u(a) = u(b) = 0$. Therefore, the 2-point boundary value problem is

$$\frac{d^2u}{dx^2}(x) = -g(x) \quad , \quad u(a) = u(b) = 0\,.$$

HINT 1 for (2-3.b): We have to show that the function

$$u_h = \sum_{j=1}^{M-1} u(x_j) b_h^j \in \mathcal{S}_{1,0}^0(\mathcal{M}) \,, \tag{2.3.3}$$

b_h^j the 1D tent function associated with the node x_j (local cell size $h_j := x_j - x_{j-1}$),

$$b_h^j(x) := \begin{cases} (x - x_{j-1})/h_j & \text{, if } x_{j-1} \le x \le x_j \,, \\ (x_{j+1} - x)/h_{j+1} & \text{, if } x_j \le x \le x_{j+1} \,, \\ 0 & \text{elsewhere.} \end{cases} \qquad \text{[Lecture} \rightarrow \text{Eq. (2.3.2.1)]}$$

solves the discrete variational problem assocaited with (2.3.1), that is, it satisfies

$$u \in \mathcal{S}_{1,0}^0(\mathcal{M}) : \quad \int_a^b \frac{du_h}{dx}(x) \frac{dv_h}{dx}(x) \, \mathrm{d}x = \int_a^b g(x) v_h(x) \, \mathrm{d}x \quad \forall v_h \in \mathcal{S}_{1,0}^0(\mathcal{M}) \,. \tag{2.3.4}$$

⌟

HINT 2 for (2-3.b): Understand that (2.3.4) is equivalent to showing

$$\int_{x_{j-1}}^{x_{j+1}} \frac{du_h}{dx}(x)\, \frac{db_h^j}{dx}(x)\, \mathrm{d}x = \int_{x_{j-1}}^{x_{j+1}} g(x) b_h^j(x)\, \mathrm{d}x \quad \forall j \in \{1, \ldots, M-1\}\,. \tag{2.3.5}$$

⌟

HINT 3 for (2-3.b): Split the integral in (2.3.5) into two over $[x_{j-1}, x_j]$ and x_j, x_{j+1}, respectively, and use

- the formula for the derivative of a tent function

$$\frac{db_h^j}{dx}(x) = \begin{cases} \frac{1}{h_j} & \text{, if } x_{j-1} \leq x \leq x_j \text{ ,} \\ -\frac{1}{h_{j+1}} & \text{, if } x_j < x \leq x_{j+1} \text{ ,} \\ 0 & \text{elsewhere,} \end{cases} \qquad \text{[Lecture} \rightarrow \text{Eq. (2.3.2.6)]}$$

- and the fundamental theorem of calculus [Lecture $\rightarrow$ Eq. (1.5.1.6)].

⌟

There is no escaping "tedious computations" when solving this sub-problem. You have to master integration by parts in 1D as explained in [Lecture → § 1.5.1.5].

SOLUTION of (2-3.b):

❶ Note that (2.3.2) involves (2.3.3) by the *cardinal basis property* [Lecture → Eq. (2.3.2.2)] of the tent functions.

Since b_h^j is supporterd in $[x_{j-1}, x_{j+1}]$, we have to show

$$\int_{x_{j-1}}^{x_{j+1}} \frac{du_h}{dx}(x)\, \frac{db_h^j}{dx}(x)\, \mathrm{d}x = \int_{x_{j-1}}^{x_{j+1}} g(x) b_h^j(x)\, \mathrm{d}x \quad \forall j \in \{1, \ldots, M-1\}\,. \tag{2.3.5}$$

Appealing to [Lecture → Eq. (2.3.2.6)], the left-hand side of (2.3.5) boils down to

$$\begin{aligned}\int_{x_{j-1}}^{x_{j+1}} \frac{du_h}{dx}(x)\, \frac{db_h^j}{dx}(x)\, \mathrm{d}x &= \frac{1}{h_j}\int_{x_{j-1}}^{x_j} \frac{du_h}{dx}(x)\, \mathrm{d}x - \frac{1}{h_{j+1}}\int_{x_j}^{x_{j+1}} \frac{du_h}{dx}(x)\, \mathrm{d}x \\ &= \frac{u_h(x_j) - u_h(x_{j-1})}{h_j} - \frac{u_h(x_{j+1}) - u_h(x_j)}{h_{j+1}} = \frac{u(x_j) - u(x_{j-1})}{h_j} - \frac{u(x_{j+1}) - u(x_j)}{h_{j+1}}\,,\end{aligned}$$

where we used the condition $u_h(x_j) = u(x_j)$ from the definition (2.3.3) of u_h.

From Sub-problem (2-3.a), we recall that

$$\frac{d^2u}{dx^2}(x) = u''(x) = -g(x)\,.$$

Thus, the right-hand side of (2.3.5) becomes

$$\begin{aligned}\int_{x_{j-1}}^{x_{j+1}} g(x) b_h^j(x)\, \mathrm{d}x &= -\int_{x_{j-1}}^{x_j} \frac{d^2u}{dx^2}(x)\, b_h^j(x)\, \mathrm{d}x - \int_{x_j}^{x_{j+1}} \frac{d^2u}{dx^2}(x)\, b_h^j(x)\, \mathrm{d}x \\ &\overset{\text{i.b.p.}}{=} \int_{x_{j-1}}^{x_j} \frac{du}{dx}(x)\, \frac{db_h^j}{dx}(x)\, \mathrm{d}x + \int_{x_j}^{x_{j+1}} \frac{du}{dx}(x)\, \frac{db_h^j}{dx}(x)\, \mathrm{d}x \\ &= \frac{1}{h_j}\int_{x_{j-1}}^{x_j} \frac{du}{dx}(x)\, \mathrm{d}x - \frac{1}{h_{j+1}}\int_{x_j}^{x_{j+1}} \frac{du}{dx}(x)\, \mathrm{d}x = \frac{u(x_j) - u(x_{j-1})}{h_j} - \frac{u(x_{j+1}) - u(x_j)}{h_{j+1}}\,.\end{aligned}$$

This confirms (2.3.5).

❷: An alternative approach aims to show that the vector obtained by restricting the exact solution u of (2.3.1) to the nodes of the mesh provides a solution of the linear system of equations from [Lecture → Section 2.3] arising from the Galerkin discretization of (2.3.1) by means of piecewise linear finite elements. We elaborate this for an *equidistant* mesh only, $x_j = a + \frac{b-a}{M} j$:

From Sub-problem (2-3.a), we recall that

$$\frac{d^2u}{dx^2}(x) = u''(x) = -g(x)\,.$$

We start with the linear system of equations resulting from the Galerkin discretization using tent functions on a uniform mesh with spacing h, written as $A\mu = \varphi$ where

$$A = \frac{1}{h}\begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & & \\ 0 & -1 & \ddots & \ddots & \\ \vdots & & \ddots & 2 & -1 \\ 0 & & & -1 & 2 \end{bmatrix}$$

and the exact form of the RHS is

$$\varphi_i = \int_{x_{i-1}}^{x_{i+1}} g(x) h_N^i(x)\,\mathrm{d}x$$

If $u(x)$ is the exact solution to our variational problem (and to the ODE), we check that that the system of equations is satisfied at the mesh points, that is by checking that A multiplied by the vector $u(x_i), 1 \le i \le N$ equals the RHS above.

As a tool, we make use of the "exact" Taylor expansion, written with the reminder in integral form:

$$u(x+h) = u(x) + u'(x)h + \int_x^{x+h} u''(t)(x+h-t)\,\mathrm{d}t$$
$$u(x-h) = u(x) - u'(x)h - \int_{x-h}^{x} u''(t)(x-h-t)\,\mathrm{d}t$$

Using the relation above, we have that

$$\frac{-u(x-h)+2u(x)-u(x+h)}{h} = \frac{1}{h}\int_{x-h}^{x} u''(t)(x-h-t)\,\mathrm{d}t - \frac{1}{h}\int_x^{x+h} u''(t)(x+h-t)\,\mathrm{d}t\,.$$

For the particular case of mesh node x_i, the LHS of the equation above equals the $i-th$ equation of the linear system (that is the i-th row of A multiplied by $u(x_i), 1 \le i \le N$. Using the fact that $u''(x) = -g(x)$ we can rewrite the RHS as

$$\frac{1}{h}\int_{x-h}^{x} u''(t)(x-h-t)\,\mathrm{d}t - \frac{1}{h}\int_x^{x+h} u''(t)(x+h-t)\,\mathrm{d}t = \\ -\frac{1}{h}\int_{x_{i-1}}^{x_i} g(t)(x_{i-1}-t)\,\mathrm{d}t + \frac{1}{h}\int_{x_i}^{x_{i+1}} g(t)(x_{i+1}-t)\,\mathrm{d}t\,. \quad (2.3.6)$$

On the other hand,

$$\varphi_i = \int_{x_{i-1}}^{x_{i+1}} g(x)h_N^i(x)\,\mathrm{d}x = \frac{1}{h}\int_{x_{i-1}}^{x_i} g(x)(x-x_{i-1})\,\mathrm{d}x + \frac{1}{h}\int_{x_i}^{x_{i+1}} g(x)(x_{i+1}-x)\,\mathrm{d}x = \\ = -\frac{1}{h}\int_{x_{i-1}}^{x_i} g(x)(x_{i-1}-x)\,\mathrm{d}x + \frac{1}{h}\int_{x_i}^{x_{i+1}} g(x)(x_{i+1}-x)\,\mathrm{d}x\,,$$

which proves the fact that the two are indeed equal.

HINT 1 for (2-3.c): What functions are integrated exactly by the composite trapezoidal rule? ⌟

SOLUTION of (2-3.c):

The composite trapezoidal rule is exact for $\mathcal{M}$-piecewise linear integrands. For $\mathcal{M}$-piecewise constant g the integrand gv is piecewise linear for $v \in \mathcal{S}^0_{1,0}(\mathcal{M})$. Thus the composite trapezoidal rule will yield the same value for the entries of the right-hand side vector as exact integration, which was required for the arguments of Sub-problem (2-3.b).

HINT 1 for (2-4.a): Recall [Lecture → § 1.5.1.9].

SOLUTION of (2-4.a):

Just follow the considerations of [Lecture → § 1.5.1.9] line by line. The extra term $\gamma(x)u(x)v(x)$ does not require any integration by parts. We end up with the two-point boundary value problem

$$-u'' + \gamma\, u = f \quad \text{in} \quad]0,1[\quad , \quad u(0) = u(1) = 0\,. \tag{2.4.1}$$

Note that homogeneous Dirichlet boundary conditions are enforced through the trial and test space $H_0^1(]0,1[)$.

SOLUTION of (2-4.j):

The reference implementation again invokes the functions `computeA()`, and `computeRHS()` in order to assemble the Galerkin matrix $\mathbf{A} \in \mathbb{R}^{M+1,M+1}$ and right hand side vector $\vec{\boldsymbol{\varphi}} \in \mathbb{R}^{M+1}$ for (C) on the *full* $M+1$-dimensional finite element space $\mathcal{S}_1^0(\mathcal{M})$.

Then note that the $\mathcal{S}_1^0(\mathcal{M})$ coefficient vector corresponding to the offset function (2.4.21) is $[u_0, 0, \ldots, 0, u_1]$. Hence, we can compute the $\mathcal{S}_{1,0}^0(\mathcal{M})$ coefficient vector for the linear form

$$v \mapsto \int_0^1 f(x)v(x)\,\mathrm{d}x - \mathsf{a}(u_{0,h}, v)$$

with

$$\mathsf{a}(u,v) := \int_0^1 \alpha(x)\frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x\,,$$

by the followig steps:

(i) Set

$$\vec{\boldsymbol{\nu}} := \vec{\boldsymbol{\varphi}} - u_0(\mathbf{A})_{:,1} - u_1(\mathbf{A})_{:,M+1}$$

(ii) and then use $(\vec{\boldsymbol{\nu}})_{2:M}$ as new right-hand side vector.

C++ code 2.4.25: Implementation of `solveB()` ⇝ GitLab

```
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::VectorXd solveB(const Eigen::VectorXd &mesh, FUNCTOR1 &&alpha,
                       FUNCTOR2 &&f, double u0, double u1) {
  // Nodes are indexed as 0=x_0 < x_1 < ... < x_N = 1
  unsigned N = mesh.size() - 1;
  // Initializations
  Eigen::VectorXd u(N + 1);                       // solution vec
  Eigen::SparseMatrix<double> A(N + 1, N + 1);  // laplacian galerkin mat
  // Some tool variables
  double dx_left, dx_right;  // cell widths

  // I. Build the Galerkin matrix A
  std::vector<Eigen::Triplet<double>> triplets = computeA(mesh, alpha);
  A.setFromTriplets(triplets.begin(), triplets.end());

  // II. Build the right hand side source vector
  Eigen::VectorXd rhs_vec = computeRHS(mesh, f);

  // III. Enforce dirichlet boundary conditions
  // Proceeding as in solveA, we begin by dropping the first and last rows
  // and columns of the galerkin matrix. The rhs_vec needs to be modified to
  // account for the non-homegenous Dirichlet boundary conditions. It is
  // modified using an offset function.
  Eigen::SparseMatrix<double> A_reduced = A.block(1, 1, N - 1, N - 1);
  Eigen::VectorXd rhs_vec_reduced = rhs_vec.segment(1, N - 1) -
                                    A.block(1, 0, N - 1, 1) * u0 -
                                    A.block(1, N, N - 1, 1) * u1;
```

```
29
30    // IV. Solve the LSE A*u = rhs_vec using an Eigen solver
31    Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> solver;
32    solver.compute(A_reduced);
33    if (solver.info() != Eigen::Success) {
34      throw std::runtime_error("Could not decompose the matrix");
35    }
36    u.segment(1, N - 1) = solver.solve(rhs_vec_reduced);
37
38    // The solution vector still needs to be supplemented with the known
39    // boundary values
40    u(0) = u0;  // left boundary node
41    u(N) = u1;  // right boundary node
42    return u;
43  }  // solveB
```

HINT 1 for (2-4.b): Now follow [Lecture → § 1.5.1.15] ⌟

SOLUTION of (2-4.b):

What we face is exactly [Lecture → Eq. (1.4.2.1)] and as in [Lecture → § 1.5.1.15] we get

$$-(\alpha\, u')' = f \quad \text{in} \quad]0,1[\quad , \quad u(0) = u_0,\, u(1) = u_1\ . \tag{2.4.2}$$

The trial space is an affine space taking into account the non-homogeneous Dirichlet boundary conditions.

HINT 1 for (2-4.c): Take the cue from [Lecture → § 1.5.1.9] and [Lecture → Ex. 1.5.3.11]. ⌟

SOLUTION of (2-4.c):

This is a two-point boundary value problem with *natural boundary conditions* that will turn out as homogeneous Neumann boundary conditions, because no boundary terms are present, neither in the bilinear form nor in the right-hand-side linear form. 1D integration by parts

$$\int_a^b f(\xi)v'(\xi)\,\mathrm{d}\xi = -\int_a^b f'(\xi)v(\xi) + [f(x)v(x)]_{x=a}^{x=b} \quad \forall f, v \in C^1_{\mathrm{pw}}([a,b])\ , \quad [\text{Lecture} \to \text{Eq. (1.5.1.8)}]$$

applied to (C) yields

$$-\int_0^1 \left(-\frac{d}{dx}\left(\alpha(x)\frac{du}{dx}\right)(x) + \gamma(x)u\right) v(x)\,\mathrm{d}x + \left[\alpha(x)\frac{du}{dx}(x)v(x)\right]_{x=0}^{x=1} = \int_0^1 f(x)v(x)\mathrm{d}x \quad \forall v \in H^1(]0,1[)\ .$$

As in [Lecture → Ex. 1.5.3.11] we first test with $v \in C_0^\infty([0,1])$ and, thus, by the fundamental lemma of the calculus of variations in 1D [Lecture → Lemma 1.5.1.13] obtain the differential equation

$$-(\alpha\, u')' + \gamma\, u = f \quad \text{in} \quad]0,1[\ . \tag{2.4.3}$$

Then we test with general $v \in C^\infty([0,1])$ and use the differential equation (2.4.3), which gives the boundary conditions

$$\alpha(0)\tfrac{du}{dx}(0) = \alpha(1)\tfrac{du}{dx}(1) = 0\ .$$

HINT 1 for (2-4.d): Study [Lecture → § 2.3.3.1].

HINT 2 for (2-4.d): Notice that the trapezoidal rule applied to an interval $[x_i, x_{i+1}]$ only uses two points, which are the endpoints x_i and x_{i+1}. ⌟

SOLUTION of (2-4.d):

The relevant bilinear form is

$$\mathsf{a}(u,v) = \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x) + \gamma(x)u(x)v(x)\,\mathrm{d}x\,, \quad u,v \in H_0^1(]0,1[)\,.$$

It comprises two parts, of which the first is the same as considered in [Lecture → § 2.3.3.1]. This results in the Galerkin matrix

$$\mathbf{A} = \mathbf{A}_\Delta + \mathbf{M} \in \mathbb{R}^{N,N}\,, \quad N := M-1\,,$$

where $\mathbf{A}_\Delta$ is the s.p.d. triadiagonal matrix from [Lecture → Eq. (2.3.3.2)] (numerical quadrature has no impact, because we integrate piecewise constants!)

$$\mathbf{A}_\Delta = \begin{bmatrix} \frac{1}{h_1}+\frac{1}{h_2} & -\frac{1}{h_2} & 0 & & & 0 \\ -\frac{1}{h_2} & \frac{1}{h_2}+\frac{1}{h_3} & -\frac{1}{h_3} & & & \\ 0 & \ddots & \ddots & \ddots & & \\ & & & \ddots & 0 & \\ & & \ddots & \ddots & -\frac{1}{h_{M-1}} & \\ 0 & & 0 & -\frac{1}{h_{M-1}} & \frac{1}{h_{M-1}}+\frac{1}{h_M} \end{bmatrix} \in \mathbb{R}^{N,N}\,, \quad \begin{array}{l} N := M-1\,, \\ h_\ell = |x_\ell - x_{\ell-1}|\,, \end{array} \tag{2.4.6}$$

and $\mathbf{M} \in \mathbb{R}^{N,N}$ is a so-called **mass matrix**, the $\mathcal{S}_{1,0}^0(\mathcal{M})$-Galerkin matrix for the bilinear form

$$\mathsf{m}(u,v) := \int_0^1 \gamma(x)u(x)v(x)\,\mathrm{d}x\,, \quad u,v \in H_0^1(\Omega)\,. \tag{2.4.7}$$

The integral is approximated by the trapezoidal rule [Lecture → Eq. (2.3.3.8)], which means

$$\mathsf{m}(u_h,v_h) \approx \mathsf{m}_h(u_h,v_h) := \sum_{\ell=1}^{M-1} \tfrac{1}{2}|x_{\ell+1}-x_{\ell-1}|\gamma(x_\ell)u_h(x_\ell)v_h(x_\ell)\,, \quad u_h,v_h \in \mathcal{S}_{1,0}^0(\mathcal{M})\,. \tag{2.4.8}$$

Thanks to the *cardinal basis property* [Lecture → Eq. (2.3.2.2)] for the entries of $\mathbf{M}$ we get

$$\begin{aligned} (\mathbf{M})_{i,j} = \mathsf{m}_h(b_h^j, b_h^i) &= \sum_{\ell=1}^{M-1} \tfrac{1}{2}|x_{\ell+1}-x_{\ell-1}|\gamma(x_\ell)\underbrace{b_h^j(x_\ell)}_{=\delta_{j,\ell}}\underbrace{b_h^i(x_\ell)}_{=\delta_{i,\ell}} \\ &= \begin{cases} \frac{1}{2}|x_{i+1}-x_{i-1}|\gamma(x_i) & \text{for} \quad i=j\,, \\ 0 & \text{else.} \end{cases} \end{aligned} \tag{2.4.9}$$

We have found that $\mathbf{M}$ is a diagonal matrix! Summing up we obtain a *tridiagonal* Galerkin matrix with the following entries:

$$\begin{aligned} (\mathbf{A})_{i,i} &= \frac{1}{x_i - x_{i-1}} + \frac{1}{x_{i+1}-x_i} + \gamma(x_i)\frac{x_{i+1}-x_{i-1}}{2} && i = 1,\ldots,M-1 \\ (\mathbf{A})_{i,i-1} = (\mathbf{A})_{i-1,i} &= -\frac{1}{x_i - x_{i-1}} && i = 2,\ldots,M-1\,, \end{aligned} \tag{2.4.10}$$

and all the other matrix entries vanish.

The right-hand-side linear form $\ell(v) := \int_0^1 f(x)v(x)\,\mathrm{d}x$ is exactly the same as in [Lecture → § 2.3.3.6] and, thus, we obtain the same expression [Lecture → Eq. (2.3.3.9)] for the entries of the right-hand-side vector $\vec{\boldsymbol{\varphi}}$:

$$\varphi_k := (\vec{\boldsymbol{\varphi}})_k = \int_a^b f(x)b_h^k(x)\,\mathrm{d}x \approx \tfrac{1}{2}(x_{k+1} - x_{k-1})f(x_k)\,, \quad 1 \le k \le N\,. \tag{2.4.11}$$

HINT 1 for (2-4.e): Given a vector `triplets` of triplets describing a symmetric matrix and a right-hand-side vector `b`, this is how to solve the corresponding linear system of equations in EIGEN:

```
A.setFromTriplets(triplets.begin(), triplets.end());
Eigen::SimplicialLDLT<SparseMatrix, Eigen::Lower> solver;
solver.compute(A);
if (solver.info() != Eigen::Success) {
  throw std::runtime_error("Could not decompose the matrix");
}
Eigen::VectorXd u = solver.solve(b);
```

SOLUTION of (2-4.e):

Armed with the formulas (2.4.10) and (2.4.11), this is an easy exercise in triplet-based initialization of a tridiagonal matrix.

The code of the master solution is needlessly complicated, because it separately builds different parts of the Galerkin matrix (in triplet/COO format) using the function `computeA()` and `computeM()`. The rationale was code reuse for later sub-problems.
Probably, a much easier approach to this sub-problem is the direct initialization of the entries of the tridiagonal Galerkin matrix after sufficient space has been reserved, thus circumventing the triplet format.

The following functions initializes the COO-format for the $\mathcal{S}^0_{1,0}(\mathcal{M})$-finite element Galerkin matrix corresponding to the bilinear form

$$\mathsf{a}(u,v) := \int_0^1 \alpha(x)\frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x\,, \quad u,v \in H^1(]0,1[)\,.$$

The composite midpoint rule [Lecture → Eq. (2.3.3.13)] is employed and the full finite element space including tent functions located at the endpoints is used. The formulas formulas for the entries of the matrix are given in [Lecture → § 2.3.3.10].

C++ code 2.4.12: Implementation of `computeA()` ⇝ GitLab

```
2  template <typename FUNCTOR1>
3  std::vector<Eigen::Triplet<double>> computeA(const Eigen::VectorXd &mesh,
4                                               FUNCTOR1 &&alpha) {
5    // Nodes are indexed as 0=x_0 < x_1 < ... < x_N = 1
6    unsigned N = mesh.size() - 1;
7    // Initializing the vector of triplets whose size corresponds to the
8    // number of entries in a (N+1) x (N+1) tridiagonal (band) matrix
9    std::vector<Eigen::Triplet<double>> triplets;
10   // Maximal size of vector; avoids intermediate allocations
11   triplets.reserve(3 * N + 1);
12   // Some auxiliary variables
13   double diag, off_diag;     // matrix entries
14   double dx_left, dx_right;  // cell widths
15
16   /* Computing diagonal entries */
17   // First diagonal entry (left boundary node)
18   dx_right = mesh(1) - mesh(0);
19   diag = alpha((mesh(1) + mesh(0)) / 2.) / dx_right;
20   triplets.push_back(Eigen::Triplet<double>(0, 0, diag));
21   // Last diagonal entry (right boundary node)
22   dx_left = mesh(N) - mesh(N - 1);
23   diag = alpha((mesh(N) + mesh(N - 1)) / 2.) / dx_left;
24   triplets.push_back(Eigen::Triplet<double>(N, N, diag));
25   // All diagonal entries associated to interior nodes
26   for (unsigned i = 1; i < N; ++i) {
27     dx_left = mesh(i) - mesh(i - 1);
28     dx_right = mesh(i + 1) - mesh(i);
29     diag = alpha((mesh(i - 1) + mesh(i)) / 2.) / dx_left +
30            alpha((mesh(i) + mesh(i + 1)) / 2.) / dx_right;
31     triplets.push_back(Eigen::Triplet<double>(i, i, diag));
32   }
```

```
33
34    /* Computing off-diagonal entries */
35    for (unsigned i = 0; i < N; ++i) {
36      dx_right = mesh(i + 1) - mesh(i);
37      off_diag = -alpha((mesh(i) + mesh(i + 1)) / 2.) / dx_right;
38      triplets.push_back(Eigen::Triplet<double>(i + 1, i, off_diag));
39      triplets.push_back(Eigen::Triplet<double>(i, i + 1, off_diag));
40    }
41    return triplets;
42  }  // computeA
```

The function `computeM()` initializes the triplets for the finite-element Galerkin matrix for the bilinear form from (2.4.7):

$$\mathsf{m}(u,v) := \int_0^1 \gamma(x)u(x)v(x)\,\mathrm{d}x\,, \quad u,v \in L^2(]0,1[)\,,$$

based on the finite element space $\mathcal{S}_1^0(\mathcal{M})$. Note that the full finite element space including tent basis functions associated with the endpoints is used. Numerical quadrature based on the composite trapezoidal rule [Lecture → Eq. (2.3.3.8)] is employed.

The formulas for the entries of the resulting *diagonal* Galerkin matrix $\mathbf{M}$ are given in (2.4.9).

C++ code 2.4.13: Implementation of `computeM()` ⇝ GitLab

```
2   template <typename FUNCTOR1>
3   std::vector<Eigen::Triplet<double>> computeM(const Eigen::VectorXd &mesh,
4                                                FUNCTOR1 &&gamma) {
5     // Nodes are indexed as 0=x_0 < x_1 < ... < x_N = 1
6     unsigned N = mesh.size() - 1;
7
8     // The basis tent functions spanning the space of continuous piecewise
9     // linear polynomial satisfy the cardinal basis property with respect
10    // to the nodes of the mesh. Using  the trapezoidal rule to approximate
11    // the integrals therefore lead to a diagonal (N+1) x (N+1) matrix
12    std::vector<Eigen::Triplet<double>> triplets;
13    triplets.reserve(N + 1);
14
15    // Some tool variables
16    double diag, off_diag;
17    double dx;  // cell widths
18
19    /* Computing diagonal entries */
20    // First diagonal entry (left boundary node)
21    dx = mesh(1) - mesh(0);
22    diag = gamma(mesh(0)) * 0.5 * dx;
23    triplets.push_back(Eigen::Triplet<double>(0, 0, diag));
24    // Last diagonal entry (right boundary node)
25    dx = mesh(N) - mesh(N - 1);
26    diag = gamma(mesh(N)) * 0.5 * dx;
27    triplets.push_back(Eigen::Triplet<double>(N, N, diag));
28    // All diagonal entries associated to interior nodes
29    for (unsigned i = 1; i < N; ++i) {
30      dx = mesh(i + 1) - mesh(i - 1);
31      diag = gamma(mesh(i)) * 0.5 * dx;
```

```
    triplets.push_back(Eigen::Triplet<double>(i, i, diag));
  }  // computeM

  return triplets;
}  // computeM
```

The following function `computeRHS()` implements [Lecture → Eq. (2.3.3.9)].

C++ code 2.4.14: Implementation of `computeRHS()` ⇝ GitLab

```
template <typename FUNCTOR1>
Eigen::VectorXd computeRHS(const Eigen::VectorXd &mesh, FUNCTOR1 &&f) {
  // Nodes are indexed as 0=x_0 < x_1 < ... < x_N = 1
  unsigned N = mesh.size() - 1;
  // Initializing right hand side vector
  Eigen::VectorXd rhs_vec = Eigen::VectorXd::Zero(N + 1);

  // Some tool variables
  double dx;

  /* Calculate the entries */
  // First entry (left boundary node)
  rhs_vec(0) = f(mesh(0)) * (mesh(1) - mesh(0)) * 0.5;
  // Last entry (right boundary node)
  rhs_vec(N) = f(mesh(N)) * (mesh(N) - mesh(N - 1)) * 0.5;
  // All other enties associated to interior nodes
  for (unsigned i = 1; i < N; ++i) {
    dx = mesh(i + 1) - mesh(i - 1);
    rhs_vec(i) = f(mesh(i)) * 0.5 * dx;
  }

  return rhs_vec;
}  // computeRHS
```

Note that the generic assembly functions `computeA()`, `computeM()`, and `computeRHS()` rely on the full finite element space $\mathcal{S}_1^0(\mathcal{M})$. Thus we have to drop the first and last basis function, which amounts to ignoring the first and last columns/rows and the matrices returned by the generic assembly functions.

C++ code 2.4.15: Implementation of `solveA()` ⇝ GitLab

```
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::VectorXd solveA(const Eigen::VectorXd &mesh, FUNCTOR1 &&gamma,
                       FUNCTOR2 &&f) {
  // Nodes are indexed as 0=x_0 < x_1 < ... < x_N = 1
  // Note: What is called N here, is M in the project description!
  unsigned N = mesh.size() - 1;
  // Initializations (notice initialization with zeros here)
  Eigen::VectorXd u = Eigen::VectorXd::Zero(N + 1);  // solution vec
  Eigen::SparseMatrix<double> A(N + 1, N + 1);       // laplacian galerkin mat
  Eigen::SparseMatrix<double> M(N + 1, N + 1);       // mass galerkin mat
  Eigen::SparseMatrix<double> L(N + 1, N + 1);       // full galerkin mat

  // I. Build the (full) Galerkin matrix L for the lin. sys.
  // I.i Compute the entries of the Laplace Galerkin matrix A
  // (const. ceoff. func. alpha = 1.0)
```

```
17    std::vector<Eigen::Triplet<double>> triplets_A =
18        computeA(mesh, [](double) -> double { return 1.0; });
19    // I.ii Compute the entries of the mass matrix M
20    std::vector<Eigen::Triplet<double>> triplets_M = computeM(mesh, gamma);
21    // I.iii Assemble the sparse matrices
22    A.setFromTriplets(triplets_A.begin(), triplets_A.end());
23    M.setFromTriplets(triplets_M.begin(), triplets_M.end());
24    L = A + M;  // Full Galerkin matrix of the LSE \Label[line]{SAApM}
25
26    // II. Build the right hand side source vector
27    Eigen::VectorXd rhs_vec = computeRHS(mesh, f);
28
29    // III. Enforce (zero) dirichlet boundary conditions
30    // The B.C. are u(0) = u(N) = 0. The boundary nodes are indexed by 0
31    // and N. We can therefore opt for the option of droping first and last rows
32    // and columns of L, along with the first and last entries of the rhs_vec.
33    Eigen::SparseMatrix<double> L_reduced = L.block(1, 1, N - 1, N - 1);
34    Eigen::VectorXd rhs_vec_reduced = rhs_vec.segment(1, N - 1);
35
36    // IV. Solve the LSE L*u = rhs_vec using an Eigen solver
37    Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>, Eigen::Lower> solver;
38    solver.compute(L_reduced);
39    if (solver.info() != Eigen::Success) {
40      throw std::runtime_error("Could not decompose the matrix");
41    }
42    u.segment(1, N - 1) = solver.solve(rhs_vec_reduced);
43
44    // The solution vector u was initialized with zeros, and therefore already
45    // contains the zero Dirichlet boundary data in the first and last entry
46    return u;
47  }  // solveA
```

This implementation of `solveA()` is problematic: In Code 2.4.15, **??** two matrices in EIGEN's CCS format are added. This operation could involve considerable computational cost, because it is implemented for generic sparse matrices and not aware of the tri-diagonal pattern of both $\mathbf{A}$ and $\mathbf{M}$.

HINT 1 for (2-4.f): Compared to $\mathcal{S}^0_{1,0}(\mathcal{M})$ the nodal basis for $\mathcal{S}^0_1(\mathcal{M})$ contains two extra "half-tent" functions associated with the endpoints $x = 0, 1$ of the interval, see [Lecture $\rightarrow$ Eq. (2.4.3.1)]. Thus we have to deal with $N := M + 1$ degrees of freedom now. ⌟

HINT 2 for (2-4.f): $\mathcal{S}^0_{1,0}(\mathcal{M})$-Galerkin matrices for the bilinear form

$$\mathsf{a}_\alpha(u,v) := \int_0^1 \alpha(x)\frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x$$

have been computed explicitly in [Lecture → § 2.3.3.10]. Also there the composite midpoint rule was used. ⌟

SOLUTION of (2-4.f):

Again as in Sub-problem (2-4.d) we can split the overall Galerkin matrix into two parts

$$\mathbf{A} = \mathbf{A}_\alpha + \mathbf{M}_\gamma \in \mathbb{R}^{N,N} , \quad N := M+1 ,$$

reflecting the splitting of the bilinear form. We use the quadrature approximations ($m_\ell := \frac{1}{2}(x_\ell + x_{\ell-1})$)

$$\int_0^1 \alpha(x)\frac{du}{dx}(x)\frac{dv}{dx}(x)\,dx \approx \sum_{\ell=1}^{M} |x_\ell - x_{\ell-1}|\alpha(m_\ell)\frac{du}{dx}(m_\ell)\frac{dv}{dx}(m_\ell) , \tag{2.4.16}$$

$$\begin{aligned}\int_0^1 \gamma(x)u(x)v(x)\,dx \approx\ & \tfrac{1}{2}|x_1 - x_0|\gamma(x_0)u(x_0)v(x_0) + \\ & \tfrac{1}{2}\sum_{\ell=1}^{M-1} |x_{\ell+1} - x_{\ell-1}|\gamma(x_\ell)u(x_\ell)v(x_\ell) + \\ & \tfrac{1}{2}|x_M - x_{M-1}|\gamma(x_M)u(x_M)v(x_M) .\end{aligned} \tag{2.4.17}$$

Again, $\mathbf{M}_\gamma$ turns out to be a diagonal matrix

$$\begin{aligned}\mathbf{M}_\gamma = \operatorname{diag}\Big(&\tfrac{1}{2}|x_1 - x_0|\gamma(x_0), \tfrac{1}{2}|x_2 - x_0|\gamma(x_1), \cdots , \\ &\tfrac{1}{2}|x_M - x_{M-2}|\gamma(x_{M-1}), \tfrac{1}{2}|x_M - x_{M-1}|\gamma(x_M)\Big) \in \mathbb{R}^{N,N} .\end{aligned}$$

while $\mathbf{A}$ is still symmetric and tridiagonal. Th entries of $\mathbf{A}_\alpha$ can easily be computed from [Lecture → Eq. (2.3.2.6)] and (2.4.16). As an exception resorting to C++ indexing, we end up with with the following non-zero entries of the symmetric tridiagonal matrix $\mathbf{A} \in \mathbb{R}^{M+1,M+1}$

$$\begin{aligned}(\mathbf{A})_{i,i} &= \frac{\alpha\left(\frac{x_{i-1}+x_i}{2}\right)}{x_i - x_{i-1}} + \frac{\alpha\left(\frac{x_i+x_{i+1}}{2}\right)}{x_{i+1} - x_i} + \gamma(x_i)\frac{x_{i+1} - x_{i-1}}{2}, \qquad i = 1,\ldots,M-1 \\ (\mathbf{A})_{0,0} &= \frac{\alpha\left(\frac{x_0+x_1}{2}\right)}{x_1 - x_0} + \tfrac{1}{2}|x_1 - x_0|\gamma(x_0) \\ (\mathbf{A})_{M,M} &= \frac{\alpha\left(\frac{x_M+x_{M-1}}{2}\right)}{x_M - x_{M-1}} + \tfrac{1}{2}|x_M - x_{M-1}|\gamma(x_M) \\ (\mathbf{A})_{i,i-1} = (\mathbf{A})_{i-1,i} &= -\frac{\alpha\left(\frac{x_{i-1}+x_i}{2}\right)}{x_i - x_{i-1}}, \qquad i = 1,\ldots,M .\end{aligned} \tag{2.4.18}$$

The 1D trapezoidal rule is also used to compute the entries of the right-hand-side vector:

$$\int_0^1 v(x)\,dx \approx \tfrac{1}{2}|x_1 - x_0|v(x_0) + \tfrac{1}{2}\sum_{\ell=1}^{M-1} |x_{\ell+1} - x_{\ell-1}|v(x_\ell) + \tfrac{1}{2}|x_M - x_{M-1}|v(x_M) .$$

Thus the cardinal basis property of the functions q_N^j, $j = 0, \ldots, M$, leads to the formulas

$$\begin{aligned}(\vec{\boldsymbol{\varphi}})_0 &= \tfrac{1}{2}|x_1 - x_0| \,, \\ (\vec{\boldsymbol{\varphi}})_k &= |x_{k+1} - x_{k-1}| \,, \quad k = 1, \ldots, M-1, \\ (\vec{\boldsymbol{\varphi}})_M &= \tfrac{1}{2}|x_M - x_{M-1}| \,.\end{aligned} \tag{2.4.19}$$

SOLUTION of (2-4.g):

It is straightforward how to set up a the triplet format of a tridiagonal matrix with entries according to (2.4.18). Then use one of EIGEN's built-in sparse solvers, see the hint for Sub-problem (2-4.e).

The reference implementation relies on the auxiliary functions `computeA()`, `computeM()`, and `comput`

C++ code 2.4.20: Implementation of `solveC()` ⇝ GitLab

```
2  template <typename FUNCTOR1, typename FUNCTOR2>
3  Eigen::VectorXd solveC(const Eigen::VectorXd &mesh, FUNCTOR1 &&alpha,
4                         FUNCTOR2 &&gamma) {
5    // Nodes are indexed as 0=x_0 < x_1 < ... < x_N = 1
6    unsigned N = mesh.size() - 1;
7    // Initializations (notice initialization with zeros here)
8    Eigen::VectorXd u(N + 1);                        // solution vec
9    Eigen::SparseMatrix<double> A(N + 1, N + 1);  // laplacian galerkin mat
10   Eigen::SparseMatrix<double> M(N + 1, N + 1);  // mass galerkin mat
11   Eigen::SparseMatrix<double> L(N + 1, N + 1);  // full galerkin mat
12
13   // I. Build the (full) Galerkin matrix L for the lin. sys.
14   // I.i Compute the entries of the Laplace Galerkin matrix A
15   std::vector<Eigen::Triplet<double>> triplets_A = computeA(mesh, alpha);
16   // I.ii Compute the entries of the mass matrix M
17   std::vector<Eigen::Triplet<double>> triplets_M = computeM(mesh, gamma);
18   // I.iii Assemble the sparse matrices
19   A.setFromTriplets(triplets_A.begin(), triplets_A.end());
20   M.setFromTriplets(triplets_M.begin(), triplets_M.end());
21   L = A + M;  // Full Galerkin matrix of the LSE
22
23   // II. Build the right hand side source vector
24   Eigen::VectorXd rhs_vec =
25       computeRHS(mesh, [](double) -> double { return 1.0; });
26
27   // IV. Solve the LSE A*u = rhs_vec using an Eigen solver
28   Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> solver;
29   solver.compute(L);
30   if (solver.info() != Eigen::Success) {
31     throw std::runtime_error("Could not decompose the matrix");
32   }
33   u = solver.solve(rhs_vec);
34
35   return u;
36 }  // solveC
```

SOLUTION of (2-4.i):

We have already worked out the details in Sub-problem (2-4.d) and Sub-problem (2-4.f). Refer to the solutions of these sub-problems.

The Galerkin matrix for (2.4.22) has size $(M-1)\times(M-1)$, is symmetric and triadiagonal with the following non-zero entries

$$\begin{aligned} (\mathbf{A})_{i,i} &= \frac{\alpha\left(\frac{x_{i-1}+x_i}{2}\right)}{x_i - x_{i-1}} + \frac{\alpha\left(\frac{x_i+x_{i+1}}{2}\right)}{x_{i+1}-x_i} \qquad i = 1,\ldots,M-1 \\ (\mathbf{A})_{i,i-1} = (\mathbf{A})_{i-1,i} &= -\frac{\alpha\left(\frac{x_{i-1}+x_i}{2}\right)}{x_i - x_{i-1}} \qquad i = 2,\ldots,M-1\,. \end{aligned} \tag{2.4.23}$$

The only new aspect coming into play is a modification of the first and last entry of the right-hand-side vector compared to the Dirichlet problem with homogeneous boundary conditions, *cf.* the discussion in [Lecture → Rem. 2.3.3.15]. In fact what we add to $(\vec{\boldsymbol{\varphi}})_1$ and $(\vec{\boldsymbol{\varphi}})_{M-1}$ are the entries $(\mathbf{A})_{0,0}$ and $(\mathbf{A})_{M,M}$ of the Galerkin matrix from Sub-problem (2-4.f) multiplied with u_0 and u_1, respectively. Hence, we end up with the following right-hand-side vector for (2.4.22)

$$\left(\widetilde{\vec{\boldsymbol{\varphi}}}\right)_k = \tfrac{1}{2}|x_{k+1}-x_{k-1}|f(x_k) + \begin{cases} u_0 \cdot \dfrac{\alpha\left(\frac{x_0+x_1}{2}\right)}{x_1-x_0} & \text{for} \quad k=1\,, \\ u_1 \cdot \dfrac{\alpha\left(\frac{x_M+x_{M-1}}{2}\right)}{x_M-x_{M-1}} & \text{for } k=M-1\,, \\ 0 & \text{else.} \end{cases} \tag{2.4.24}$$

HINT 1 for (2-5.a): If you are in doubt refresh yourself about the contents of [Lecture → Ex. 1.8.0.10].
⌟

SOLUTION of (2-5.a):

As usual we use Green's first formula in order to whisk the gradient from the test function v onto $\mathbf{grad}\, u$ as a divergence:

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \tag{2.5.2}$$

This yields

$$-\int_\Omega \underbrace{\operatorname{div}(\mathbf{grad}\, u)(\boldsymbol{x})}_{=\Delta u(\boldsymbol{x})}\, v(\boldsymbol{x}) + u(\boldsymbol{x})\, v(\boldsymbol{x})\ \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\ \mathrm{d}\boldsymbol{x}$$

for all $v \in H^1(\Omega)$.

Then we test with functions compactly supported in Ω: $v \in C^\infty_0(\Omega)$, which yields the PDE $-\Delta u + u = f$ after invoking the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4]. Afterwards, we test with general v that do not necessarily vanish on $\partial\Omega$, and get homogeneous Neumann boundary conditions. For details study carefully [Lecture → Ex. 1.5.3.11].

The resulting boundary value problem in strong form reads:

$$-\Delta u + u = f \quad \text{in} \quad \Omega \quad , \quad \mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega\,. \tag{2.5.3}$$

This is a homogeneous Neumann problem for the negative Laplacian plus a so-called reaction term. More details about its derivation from (2.5.1) can be found in [Lecture → Ex. 1.5.3.11].

HINT 1 for (2-5.b): Here, in analogy to [Lecture → Eq. (2.4.5.6)] we have

$$\mathbf{M}_K := \left[\int_K \lambda_i(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^{3}, \tag{2.5.5}$$

where $\lambda_1, \lambda_2, \lambda_3$ are the barycentric coordinate functions for the triangle K as defined in [Lecture → § 2.4.5.2]. ⌟

HINT 2 for (2-5.b): Compute the entries of the element matrix by analytic evaluation of the two-dimensional integrals. In order to avoid cumbersome computations, you may rely on the general formula from [Lecture → Lemma 2.7.5.5] that gives the integral of products of barycentric coordinate functions over triangles.

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1!\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, . \tag{2.5.6}$$

SOLUTION of (2-5.b):

Appealing to

$$\mathbf{M}_K := \left[\int_K \lambda_i(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^{3} \tag{2.5.5}$$

and [Lecture → Lemma 2.7.5.5] for $d = 2$, we find for the element matrix the general formula:

$$\mathbf{M}_K = \frac{|K|}{12}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}, \tag{2.5.7}$$

where $|K|$ stands for the area of the triangle K. Note that the bilinear form m is symmetric, $\mathsf{m}(u,v) = \mathsf{m}(v,u)$ for all functions u, v, which must lead to a symmetric element matrix. It is advisable to verify this symmetry when validating a code.

SOLUTION of (2-5.c):

We rely on (2.5.7). The area $|K|$can be computed by the usual determinant formula, see [Lecture → Code 2.4.5.13]. See Code 2.5.9 for the code.

C++ code 2.5.9: Sub-problem (2-5.c): Implementation of `ElementMatrix_Mass_LFE` ↪ GitLab

```
2 Eigen::Matrix3d ElementMatrix_Mass_LFE(const TriGeo_t &V) {
3   Eigen::Matrix3d element_matrix;
4   element_matrix << 2.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 2.0;
5   element_matrix *= getArea(V) / 12.0;
6   return element_matrix;
7 }
```

C++ code 2.5.10: Sub-problem (2-5.c): Auxiliary function `getArea()` ↪ GitLab

```
2 double getArea(const TriGeo_t &triangle) {
3   return std::abs(
4       0.5 *
5       ((triangle(0, 1) - triangle(0, 0)) * (triangle(1, 2) - triangle(1, 1)) -
6        (triangle(0, 2) - triangle(0, 1)) * (triangle(1, 1) - triangle(1, 0))));
7 }
```

Remember EIGEN's << operator for the initialization of dense matrices. You need not rely on auxiliary functions like `getArea()`. That had been introduced to be reused in other parts of the code.

SOLUTION of (2-5.d):

The square for the L^2-error is approximately computed as

$$\|u_h - u\|^2_{L^2(\Omega)} \approx \sum_{K\in\mathcal{M}} \frac{1}{3}|K|\Big((u_h-u)^2(\boldsymbol{a}_K^1) + (u_h-u)^2(\boldsymbol{a}_K^2) + (u_h-u)^2(\boldsymbol{a}_K^3)\Big)\,,$$

where $\boldsymbol{a}_K^i$, $i = 1,2,3$, are the vertices of the triangle K. Thanks to the cardinal basis property [Lecture → Eq. (2.4.3.2)] of the tent functions, the values $u_h(\boldsymbol{a}_K^i)$ agree with coefficients stored in uFEM and the "d.o.f. mapper" [Lecture → Eq. (2.4.5.21)] tells the right index. For profound explanations refer to [Lecture → § 2.4.5.20].

See Code 2.5.13 for the code that implements a cell-oriented algorithm in the spirit of finite element assembly, *cf.* [Lecture → Code 2.4.5.23].

C++ code 2.5.13: Sub-problem (2-5.d): Implementation of `L2Error()` ⇝ GitLab

```
2  template <typename FUNCTOR>
3  double L2Error(const TriaMesh2D &mesh, const Eigen::VectorXd &uFEM,
4                 FUNCTOR &&exact) {
5    double l2error_squared = 0.0;
6    // loop over all triangles
7    for (int i = 0; i < mesh._elements.rows(); ++i) {
8      TriGeo_t triangle = mesh.getVtCoords(i);
9
10     // loop over all three vertices of the triangle
11     Eigen::Vector3d error_at_vertices;
12     for (int k = 0; k < 3; ++k) {
13       error_at_vertices(k) =
14           exact(triangle.col(k)) - uFEM(mesh._elements(i, k));
15     }
16     // Add squared error per triangle
17     l2error_squared +=
18         getArea(triangle) / 3.0 * error_at_vertices.squaredNorm();
19   }
20   return std::sqrt(l2error_squared);
21 }
```

The computational effort is proportional to the number $\sharp\mathcal{M}$ of cells of the mesh for large meshes.

HINT 1 for (2-5.e): The gradient of a function $u_h \in \mathcal{S}_1^0(\mathcal{M})$ is piecewise constant. On each cell it can be computed as a linear combination of the gradients of barycentric coordinate functions. Those can be computed by means of the function `gradbarycoordinates()` from [Lecture → Code 2.4.5.11]. ⌟

SOLUTION of (2-5.e):

Again we rely on cell-oriented summation, because (see solution of Sub-problem (2-5.d) for notations.)

$$|u_h - u|^2_{H^1(\Omega)} \approx \sum_{K\in\mathcal{M}} \tfrac{1}{3}|K| \Big(\Big\| \mathbf{grad}\, u_h(a_K^1) - \mathbf{grad}\, u(a_K^1) \Big\|^2 + \Big\| \mathbf{grad}\, u_h(a_K^2) - \mathbf{grad}\, u(a_K^2) \Big\|^2 + \Big\| \mathbf{grad}\, u_h(a_K^3) - \mathbf{grad}\, u(a_K^3) \Big\|^2 \Big) .$$

Since $u_h \in \mathcal{S}_1^0(\mathcal{M})$, $\mathbf{grad}\, u_h$ is $\mathcal{M}$-piecewise constant! So, in fact the gradients $\mathbf{grad}\, u_h(a_K^\ell)$ are the same vector $\mathbf{grad}\, u_h|_K$ for any $\ell = 1,2,3$. This vector can be computed according to

$$\mathbf{grad}\, u_h|_K = \sum_{\ell=1}^{3} u_h(a_K^\ell)\mathbf{grad}\, \lambda_\ell ,$$

where $\lambda_1, \lambda_2, \lambda_3$ are the barycentric coordinate functions for K. A call to `gradbarycoordinates()` gives the constant vectors $\mathbf{grad}\, \lambda_\ell$, $\ell = 1,2,3$, for a triangle .

Note that the values $u_h(a_K^\ell)$ occur as basis expansion coefficients for u_h. See Code 2.5.14 for the implementation.

C++ code 2.5.14: Sub-problem (2-5.e): Implementation of `H1SError` ➙ GitLab

```
2  double H1Serror(
3      const TriaMesh2D &mesh, const Eigen::VectorXd &uFEM,
4      const std::function<Eigen::Vector2d(const Eigen::Vector2d &)> exact) {
5    double H1Serror_squared = 0.0;
6
7    // loop over all triangles
8    for (int i = 0; i < mesh._elements.rows(); ++i) {
9      const TriGeo_t triangle = mesh.getVtCoords(i);
10
11     // loop over all three vertices of the triangle
12     Eigen::Vector3d values_at_vertices;
13     for (int k = 0; k < 3; ++k) {
14       values_at_vertices(k) = uFEM[mesh._elements(i, k)];
15     }
16
17     // gradient of FEM approximation (same for all 3 vertices!)
18     Eigen::Vector2d gradient_FEM =
19         gradbarycoordinates(triangle) * values_at_vertices;
20
21     // loop over all three vertices of the triangle
22     Eigen::Vector3d error_at_vertices;
23     for (int k = 0; k < 3; ++k) {
24       Eigen::Vector2d gradient_exact = exact(triangle.col(k));
25       error_at_vertices(k) = (gradient_FEM - gradient_exact).squaredNorm();
26     }
27
28     // Add squared error per triangle
29     H1Serror_squared += getArea(triangle) / 3.0 * error_at_vertices.sum();
30   }
31
```

```
32    return std::sqrt(H1Serror_squared);
33  }
```

HINT 1 for (2-5.f): Given the exact solution, you can use (2.5.1) to obtain the right-hand side f after having determined the associated partial differential equation, as has been done in [Lecture → Section 1.5.3].

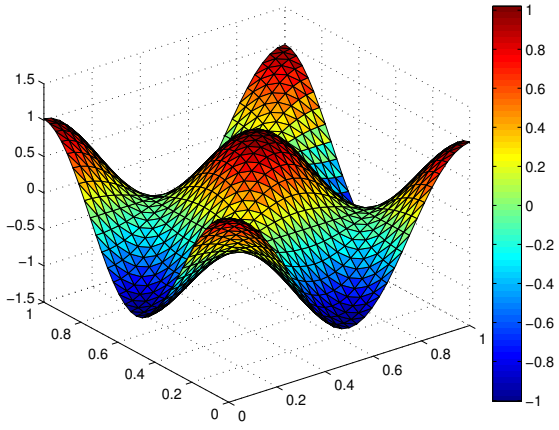
HINT 2 for (2-5.f): The supplied function `ElementMatrix_LaplMass_LFE()` and `assemLoad_LF` from [Lecture → Code 2.4.6.8] can be used. The former function calls both `ElementMatrix_Lapl_LF` similar to `assembleGalMatLFE` from [Lecture → Code 2.4.5.24], and `ElementMatrix_Mass_LFE(` from Sub-problem (2-5.c). ⌟

HINT 3 for (2-5.f): Using the mesh `Square4.txt` supplied in the `meshes` folder, the L^2-norm error should be around 0.0031 and the H^1-seminorm error around 0.665. ⌟

SOLUTION of (2-5.f):

See Code 10.3.32 for the code and Fig. 193 for the plot.

The visualization of the mesh and of the solution is done by PYTHON scripts residing in the `scripts` folder ↪ GitLab.



C++ code 2.5.16: Sub-problem (2-5.f): Implementation of `solve()` ↪ GitLab

```
2  std::tuple<Eigen::VectorXd, double, double> Solve(
3      const SimpleLinearFiniteElements::TriaMesh2D &mesh) {
4    const double pi = 3.1415926535897;
5
6    // define the source function f
7    auto f = [pi](const Eigen::Vector2d &x) {
8      return (1.0 + 8.0 * pi * pi) * std::cos(2.0 * pi * x(0)) *
9             std::cos(2.0 * pi * x(1));
10   };
11   // the exact solution of the linear variational problem
12   auto uExact = [pi](const Eigen::Vector2d &x) {
13     return std::cos(2 * pi * x(0)) * std::cos(2 * pi * x(1));
14   };
15
16   Eigen::VectorXd U;
17   double l2error;
18   double h1error;
19
20   // the gradient of uExact that can be easily analytically computed
21   auto gradUExact = [pi](const Eigen::Vector2d &x) {
22     Eigen::Vector2d gradient;
23     gradient << -2 * pi * std::sin(2 * pi * x(0)) * std::cos(2 * pi * x(1)),
24         -2 * pi * std::cos(2 * pi * x(0)) * std::sin(2 * pi * x(1));
25     return gradient;
26   };
27   // Compute the galerkin matrix A and load vector L
28   Eigen::SparseMatrix<double> A = SimpleLinearFiniteElements::assembleGalMatLFE(
29       mesh, SimpleLinearFiniteElements::getElementMatrix);
30
31   Eigen::VectorXd L =
32       SimpleLinearFiniteElements::assemLoad_LFE(mesh, localLoadLFE, f);
33
34   // solve the LSE using the sparse LU solver of Eigen
35   Eigen::SparseLU<Eigen::SparseMatrix<double>, Eigen::COLAMDOrdering<int>>
36       solver;
37   solver.analyzePattern(A);
```

```
38    solver.factorize(A);
39    U = solver.solve(L);
40
41    // compute both the L2 and the H1s error
42    l2error = SimpleLinearFiniteElements::L2Error(mesh, U, uExact);
43    h1error = SimpleLinearFiniteElements::H1Serror(mesh, U, gradUExact);
44    return std::make_tuple(U, l2error, h1error);
45  }
```

The driver code demonstrates how to farm out visualization to an external PYTHON script via the `system()` kernel function call.

C++ code 2.5.17: Sub-problem (2-5.f): Plotting mesh and solution ↪ GitLab

```
2   #define MESH "Square4"
3
4   int main() {
5     std::cout << "NumPDE homework problem on triangular linear FEM for 2D "
6                  "reaction-diffusion BVP "
7               << std::endl;
8     std::string meshfile = "meshes/" MESH ".txt";
9
10    SimpleLinearFiniteElements::TriaMesh2D square_mesh(meshfile);
11    std::cout << "Mesh loaded from " << meshfile << std::endl;
12    std::cout << "Mesh info: " << square_mesh._nodecoords.rows() << " vertices, "
13              << square_mesh._elements.rows() << " elements" << std::endl;
14
15    // print both H1 and L2 errors and plot Mesh
16    std::tuple<Eigen::VectorXd, double, double> solution =
17        SimpleLinearFiniteElements::Solve(square_mesh);
18
19    std::cout << "L2-error:  " << std::get<1>(solution) << std::endl;
20    std::cout << "H1s-error: " << std::get<2>(solution) << std::endl;
21
22    // plot MESH
23    std::string meshplot = MESH ".eps";
24    systemcall::execute(
25        ("python3 -B scripts/plot_mesh.py " + meshfile + " " + meshplot).c_str());
26    std::cout << "Generated " + meshplot << std::endl;
27
28    // path and name of output files
29    std::string meshfile_solution = MESH "_solution.txt";
30    std::string meshplot_solution = MESH "_solution.eps";
31
32    // generate 3d mesh file from solution
33    square_mesh.SaveMesh3D(meshfile_solution, std::get<0>(solution));
34    std::cout << "Generated " + meshfile_solution << std::endl;
35
36    // plot the 3d mesh file
37    systemcall::execute(("python3 -B scripts/plot_surf.py " + meshfile_solution +
38                         " " + meshplot_solution)
39                            .c_str());
40    std::cout << "Generated " + meshplot_solution << std::endl;
41    return 0;
42  }
```

HINT 1 for (2-6.a): Take into account that we are dealing with a *hybrid* 2D mesh. ⌟

SOLUTION of (2-6.a):

The rows of $\mathbf{G}$ are in one-on-one relationship with the edges of the mesh, whereas the columns are related to the nodes of the mesh. A row for a particular edge has a non-zero entry exactly for those nodes that are endpoints of the edge. Hence $\mathbf{G}$ has exactly 2 non-zero entries per row.

The rows of $\mathbf{D}$ are associated with cells and have non-zero entries in exactly those columns, which belong to the edges of the cell. Therefore the rows of $\mathbf{D}$ have either 3 (for a TRIA) or 4 (for a QUAD) non-zero entries. -

SOLUTION of (2-6.b):

Recall the explanations for Sub-problem (2-6.a). They tell us:

- The number of non-zero entries in a column of $\mathbf{G}$ tells the number of edges adjacent to the node to which that column is associated.
- The number of non-zero entries in a column of $\mathbf{D}$ is the number of cells adjacent to the associated edge.

SOLUTION of (2-6.c):

For the mesh from Fig. 7 we have $N_0 = 2$, $N_1 = 6$, and $N_2 = 5$. The black arrows indicates which is the first and second endpoint of an edge. This gives the edge-node incidence matrix:

$$\mathbf{G} = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

To determine the cell-edge incidence matrix, look up the induced orientations in Fig. 5 and Fig. 6 and mark them in Fig. 7. Thus you can read off

$$\mathbf{D} = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & -1 & 0 & 1 \end{bmatrix}.$$

SOLUTION of (2-6.d):

Code 2.6.9 shows the straightforward implementation based on an outer loop traversing all edges. Refer to the copious comments in the code for further explanations.

C++11 code 2.6.4: Computation of edge-node incidence matrix G ↪ GitLab

```
2  Eigen::SparseMatrix<int> computeEdgeVertexIncidenceMatrix(
3      const lf::mesh::Mesh &mesh) {
4    // Store edge-vertex incidence matrix here
5    Eigen::SparseMatrix<int, Eigen::RowMajor> G;
6
7    // Mesh::NumEntities(unsigned codim) returns the number of elements
8    // with given codimension.  Codim(Edge) = 1, Codim(Node) = 2.
9    const lf::mesh::Mesh::size_type numEdges = mesh.NumEntities(1),
10                                   numNodes = mesh.NumEntities(2);
11   // Following the demo for the reserve()-initialising the sparse matrix
       given
12   // in the exercise sheet.  From (2.1a) we know that G has exactly 2
       entries
13   // per row.
14   G = Eigen::SparseMatrix<int, Eigen::RowMajor>(numEdges, numNodes);
15   G.reserve(Eigen::VectorXi::Constant(numEdges, 2));
16
17   // To compute G efficiently we iterate over all edges and check the
       index
18   // of the nodes at its end.  This is the efficient way to do the
       assembly,
19   // introduced as "distribute scheme" in class.  We cannot iterative
       over
20   // vertices, because LehrFEM++ does not allow to visit the edges
21   // adjacent to a vertex
22   for (const lf::mesh::Entity *edge : mesh.Entities(1)) {
23     // Get index of this edge
24     lf::mesh::Mesh::size_type edgeIdx = mesh.Index(*edge);
25     // Get the nodes and their indices.
26     // Note, that seen from the edges the nodes have codim 1, not 2,
27     // hence we call SubEntities(1).  This is a relative codimension!
28     auto nodes = edge->SubEntities(1);
29     lf::mesh::Mesh::size_type firstNodeIdx = mesh.Index(*nodes[0]);
30     lf::mesh::Mesh::size_type lastNodeIdx = mesh.Index(*nodes[1]);
31     // Add the matrix entries according to the definition
32     G.coeffRef(edgeIdx, firstNodeIdx) += 1;
33     G.coeffRef(edgeIdx, lastNodeIdx) -= 1;
34   }
35
36   return G;
37 }
```

HINT 1 for (2-6.e): If you find it too cumbersome to test for the induced orientation of edges by just relying on calls to `SubEntities()`, LEHRFEM++ mak make your life easier with the member function

```
lf::nonstd::span<const Orientation>
   lf::mesh::Entity::RelativeOrientations() const;
```

which, for a cell, returns an array of **lf::mesh::Orientation** "mismatch flags" between the intrinsic and induced orientations of its edges. The function `lf::mesh::to_sign()` converts an orientation flag to an integer $\in \{-1, 1\}$, ⌟

SOLUTION of (2-6.e):

Code 2.6.5 runs over all cells and then sets the entries of $\mathbf{D}$ according to what is returned by `relativeOri` for all the edges of a cell.

C++11 code 2.6.5: Computation of cell-edge incidence matrix $\mathbf{D}$ ↪ GitLab

```
Eigen::SparseMatrix<int> computeCellEdgeIncidenceMatrix(
    const lf::mesh::Mesh &mesh) {
  // Store cell-edge incidence matrix here
  Eigen::SparseMatrix<int, Eigen::RowMajor> D;

  // Mesh::NumEntities(unsigned codim) returns the number of elements
  // with given codimension.  Codim(Edge) = 0, Codim(Node) = 1.
  const lf::mesh::Mesh::size_type numCells = mesh.NumEntities(0),
                                  numEdges = mesh.NumEntities(1);
  // Following the demo for the reserve()-initialising the sparse matrix
     given
  // in the exercise sheet.  From (2.1a) we know that D has at most 4
     entries
  // per row.
  D = Eigen::SparseMatrix<int, Eigen::RowMajor>(numCells, numEdges);
  D.reserve(Eigen::VectorXi::Constant(numCells, 4));

  // To compute D efficiently we iterate over all cells and check the
  // orientations (+1 or -1, same as in the definition of the matrix D)
  // of its edges.  For this we may use RelativeOrientations().
  for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
    // Get cell index
    lf::mesh::Mesh::size_type cellIdx = mesh.Index(*cell);
    // Get edges and their orientations (these already the entries for
       D!)
    auto edges = cell->SubEntities(1);
    auto edgeOrientations = cell->RelativeOrientations();

    // Iterate over both and add to D
    auto edgeIt = edges.begin();
    auto orntIt = edgeOrientations.begin();
    for (; edgeIt != edges.end() && orntIt != edgeOrientations.end();
         ++edgeIt, ++orntIt) {
      // Get the edge index and add its orientation to D
      lf::mesh::Mesh::size_type edgeIdx = mesh.Index(**edgeIt);
      D.coeffRef(cellIdx, edgeIdx) += lf::mesh::to_sign(*orntIt);
    }
  }

  return D;
}
```

HINT 1 for (2-6.f): For the proof you need to consider only a single triangular/quadrilateral cell. Then introduce the numbers d_ℓ, where $d_\ell = 1$, if the intrinsic orientation of the ℓ-th edge matches its induced orientation, $d_\ell = -1$ otherwise. ⌟

SOLUTION of (2-6.f):

We study a single entry of $\mathbf{DG}$:

$$(\mathbf{DG})_{j,i} = (\mathbf{D})_{j,:} \cdot (\mathbf{G})_{:,i} \,, \quad j \in \{1,\ldots,N_0\} \,, \quad i \in \{1,\ldots,N_2\} \,. \tag{2.6.8}$$

If the node $\boldsymbol{x}^i \in \mathcal{V}(\mathcal{M})$ is not a vertex of the cell K_j, then non-zero entries will not meet in the dot product in (2.6.8). In this case $(\mathbf{DG})_{j,i} = 0$.

Otherwise, let $e_k, e_\ell \in \mathcal{E}(\mathcal{M})$ be the two edges which are

- both adjacent to $\boldsymbol{x}^i$,
- and both belong to ∂K_j.

Let $d_k, d_\ell \in \{-1, 1\}$ denote the induced orientations of e_k and e_ℓ with respect to K. Without loss of generality assume that $\boldsymbol{x}^i$ is the first vertex of K_j and that $d_k = d_\ell = 1$, that is the edges are oriented as in Fig. 5 and Fig. 6. Then one points awway from $\boldsymbol{x}^i$ and the other towwards $\boldsymbol{x}^i$, which means that $(\mathbf{G})_{\ell,i}$ and $(\mathbf{G})_{k,i}$ have opposite signs. Hence,

$$\begin{aligned}(\mathbf{DG})_{j,i} &= (\mathbf{D})_{j,k}(\mathbf{G})_{k,i} + (\mathbf{D})_{j,\ell}(\mathbf{G})_{\ell,i} = \underbrace{d_k}_{=1}(\mathbf{G})_{k,i} + \underbrace{d_\ell}_{=1}(\mathbf{G})_{\ell,i} \\ &= (\mathbf{G})_{k,i} + (\mathbf{G})_{\ell,i} = 0 \,.\end{aligned}$$

SOLUTION of (2-6.g):

C++11 code 2.6.9: Computation of edge-node incidence matrix G ↪ GitLab

```
bool testZeroIncidenceMatrixProduct(const lf::mesh::Mesh &mesh) {
  bool isZero = false;

  Eigen::SparseMatrix<int> G = computeEdgeVertexIncidenceMatrix(mesh),
                           D = computeCellEdgeIncidenceMatrix(mesh);

  Eigen::SparseMatrix<int> O = D * G;
  // Possibility 1:
  // Not prone to roundoff errors, since O is an integer matrix!
  isZero = O.norm() == 0;
  // Possibility 2:  But this doesn't use the fact that O is sparse.
  // isZero = Eigen::MatrixXi(O).isZero(0);
  return isZero;
}
```

Note that roundoff errors do not have to be taken into account, because we rely on matrices with integer entries.

In the file `boundarylength.cc` write a LEHRFEM++-based C++ function

```
double volumeOfDomain(const std::shared_ptr<lf::mesh::Mesh> mesh_p);
```

that returns the area of the domain covered by a 2D hybrid mesh passed through the `mesh_p` pointer argument. You may use the function `lf::geometry::Volume()` ↪ GITHUB.

SOLUTION of (2-7.a):

Of course, the implementation will employ a loop over all cells (entities of co-dimension 0) of the mesh and then add up their areas.

C++ code 2.7.1: Sub-problem (2-7.a): Implementation of `volumeOfDomain()` ⇢ GitLab

```
2  double volumeOfDomain(const std::shared_ptr<lf::mesh::Mesh> mesh_p) {
3    double volume = 0.0;
4    // iterate over all cells (co-dimension = 0)
5    for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
6      const lf::geometry::Geometry *geo_p = cell->Geometry();
7      volume += lf::geometry::Volume(*geo_p);
8    }
9
10   return volume;
11 }
```

Note that the loop variable `cell` is a pointer, as already remarked in [Lecture → Ex. 2.7.2.6].

Study the description and the implementation of the LEHRFEM++ function (➙ GITHUB)

```
CodimMeshDataSet<bool>
lf::mesh::utils::flagEntitiesOnBoundary(
  const std::shared_ptr<const lf::mesh::Mesh> &mesh_p,
  lf::base::dim_t codim);
```

and explain how edges on the boundary are identified in this function. Explanations pertinent to the **MeshDataSet** concept of LEHRFEM++ can be found in [Lecture → Suppl. 2.7.2.8] and also in the documentation of **lf::mesh::utils::CodimMeshDataSet**.

C++ code 2.7.2: (Skeleton) Implementation of the LEHRFEM++ function `flagEntitiesOnBoundary()`

```
1  CodimMeshDataSet<bool> flagEntitiesOnBoundary(
2      const std::shared_ptr<const Mesh>& mesh_p, lf::base::dim_t codim) {
3    // count cells adjacent to entities of co-dimension 1
4    CodimMeshDataSet<lf::base::size_type> no_adjacent_cells{
5        CountNumSuperEntities(mesh_p, 1, 1)};
6    // flag array
7    CodimMeshDataSet<bool> bd_flags{mesh_p, codim, false};
8    // relative codimension with respect to faces (entities of
       co-dimension 1)
9    const lf::base::dim_t rel_codim = codim - 1;
10   // Run through faces and flag sub-entities
11   for (const lf::mesh::Entity *edge_p : mesh_p->Entities(1)) {
12     if (no_adjacent_cells(*edge_p) == 1) {
13       // Boundary face detected!
14       // Traverse all sub-entities of a specific relative co-dimension
15       for (const lf::mesh::Entity* subent : edge->SubEntities(rel_codim)) {
16         bd_flags(*subent) = true;
17       }
18     }
19   }
20   return bd_flags;
21 }
```

SOLUTION of (2-7.b):

The function `flagEntitiesOnBoundary()` calls `countNoSuperEntities()`, which counts the number of adjacent entities of the next lower co-dimension. In the case of edges (co-dimension 1), these are the adjacent cells (co-dimension 0).

C++ code 2.7.3: (Skeleton) Implementation of the LEHRFEM++ function `countNoSuperEntities()`

```
CodimMeshDataSet<lf::base::size_type> CountNumSuperEntities(
    const std::shared_ptr<const Mesh>& mesh_p, lf::base::dim_t codim_sub,
    lf::base::dim_t codim_super) {
  // Declare and initialize the data set
  CodimMeshDataSet<lf::base::size_type> sup_ent_cnt{mesh_p, codim_sub, 0};
  // Gap in co-dimensions
  const lf::base::dim_t super_codim = codim_sub - codim_super;
  // Run through all super entities
  for (const lf::mesh::Entity* e : mesh_p->Entities(super_codim)) {
    // Traverse all sub-entities of a specific relative co-dimension
    for (const lf::mesh::Entity* subent : e->SubEntities(codim_super)) {
      // Write access to an entry in the data set
      sup_ent_cnt(*subent) += 1;
    }
  }
  return sup_ent_cnt;
}
```

The criterion for identifying a boundary edge is that it has exactly one adjacent cell.

Implement a function (in the file `boundarylength.cc`)

```
double lengthOfBoundary(
  const std::shared_ptr<lf::mesh::Mesh> mesh_p)
```

that calculates the length of the boundary of the domain covered by the mesh passed through the `mesh_p` pointer.

HINT 1 for (2-7.c): Of course you can use the boolean mesh data set as it is returned by the LEHRFEM++ function `flagEntitiesOnBoundary()`. ⌟

SOLUTION of (2-7.c):

The natural idea is to loop over all edges, but skipping those not on the boundary.

C++ code 2.7.4: Implementation of a `lengthOfBoundary()` ↪ GitLab

```
2  double lengthOfBoundary(const std::shared_ptr<lf::mesh::Mesh> mesh_p) {
3    double length = 0.0;
4    // Obtain an array of boolean flags for the vertices of the mesh:
       'true'
5    // indicates that the vertex lies on the boundary.
6    auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
7
8    // iterate over all edges (co-dimension = 1)
9    for (const lf::mesh::Entity *cell : mesh_p->Entities(1)) {
10     // check if edge is part of the boundary
11     if (bd_flags(*cell)) {
12       const lf::geometry::Geometry *geo_p = cell->Geometry();
13       length += lf::geometry::Volume(*geo_p);
14     }
15   }
16
17   return length;
18 }
```

Based on LEHRFEM++ implement an LEHRFEM++-based C++ function

```
std::pair<double, double> measureDomain(std::string filename);
```

that reads a 2D (hybrid) mesh from the `.msh`-file [Lecture → Ex. 2.7.1.5] with the name `filename` and returns both the volume and the length of the boundary of the domain covered by the mesh.

Use this function to enhance the `main()` routine in the file `boundarylength_main.cc` so that the resulting executable solves the tasks described in the introduction to this problem: It should read the name of the `.msh`-file from the command line and print the area of Ω and the length of $\partial\Omega$.

HINT 1 for (2-7.d): There is a LEHRFEM++ demo that teaches how to read a mesh from an `.msh` file ➙ GITHUB, [Lecture → Code 2.7.1.12]. The key class is **lf::io::GmshReader**. ⌟

SOLUTION of (2-7.d):

The right approach is to copy a few lines from the LEHRFEM++ demo or **??**.

C++ code 2.7.5: Implementation of `measureDomain()` ⇝ GitLab

```
std::pair<double, double> measureDomain(std::string filename) {
  double volume, length;

  // Load mesh into a Lehrfem++ object
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  lf::io::GmshReader reader(std::move(mesh_factory), filename);
  std::shared_ptr<lf::mesh::Mesh> mesh_p = reader.mesh();

  // call the functions we already implemented
  length = LengthOfBoundary::lengthOfBoundary(mesh_p);
  volume = LengthOfBoundary::volumeOfDomain(mesh_p);

  return {volume, length};
}
```

This is how you can read a command line argument and pass it to the function:

C++ code 2.7.6: `main()` driver function for this problem ⇝ GitLab

```
int main(int argc, char *argv[]) {
  if (argc > 1) {
    std::string file_name(argv[1]);
    std::pair<double, double> result = measureDomain(file_name);

    std::cout << "The area of the domain is: " << result.first << std::endl;
    std::cout << "The length of the boundary is: " << result.second
              << std::endl;
  } else {
    std::cout << "Correct usage: Call this executable and supply the name of a "
                 "msh-file as a command line argument "
              << std::endl;
  }
}
```

If you call the code with the path of to the mesh file `hex4.msh` ⇝ GITHUB the output should be:

```
The area of the domain is: 2.59808
The length of the boundary is: 6
```

SOLUTION of (2-8.a):

A simple example from EIGEN's documentation

```
VectorXd x(n), b(n);
SparseMatrix<double> A;
SparseLU<SparseMatrix<double>, COLAMDOrdering<int> >   solver;
// fill A and b;
// Compute the ordering permutation vector from the structural pattern
   of A
solver.analyzePattern(A);
// Compute the numerical factorization
solver.factorize(A);
//Use the factors to solve the linear system
x = solver.solve(b);
```

Note that the calls to `analyzePattern()` and `factorize()` are not really required. Only if a linear system with the same coefficient matrix but different right-hand sides has to be solved, a separate call to `factorize()` is recommended, in order to do the factorization only once.

SOLUTION of (2-8.j):

C++11 code 2.8.15: Sub-problem (2-8.j): Implementation of `Eval()` for MyLinearLoadVector ↪ GitLab

```
Eigen::Vector4d computeLoadVector(
    const Eigen::MatrixXd &vertices,
    std::function<double(const Eigen::Vector2d &)> f) {
  // Number of nodes of the element:  triangles = 3, rectangles = 4
  const int num_nodes = vertices.cols();
  // Vector for returning element vector
  Eigen::Vector4d elem_vec = Eigen::Vector4d::Zero();
  // Area of the cell
  double area;
  // Midpoints of edges in the reference cell
  Eigen::MatrixXd midpoints(2, num_nodes);
  switch (num_nodes) {
    case 3: {
      // Compute cell area for triangles
      area = 0.5 * ((vertices(0, 1) - vertices(0, 0)) *
                        (vertices(1, 2) - vertices(1, 0)) -
                    (vertices(1, 1) - vertices(1, 0)) *
                        (vertices(0, 2) - vertices(0, 0)));
      // clang-format off
      midpoints << vertices(0, 0) + vertices(0, 1),
     vertices(0, 1) + vertices(0, 2),
     vertices(0, 2) + vertices(0, 0),
     vertices(1, 0) + vertices(1, 1),
     vertices(1, 1) + vertices(1, 2),
     vertices(1, 2) + vertices(1, 0);
      // clang-format on
      break;
    }
    case 4: {
      // Compute cell area for rectangles
      area =
          (vertices(0, 1) - vertices(0, 0)) * (vertices(1, 3) - vertices(1, 0));
      // clang-format off
      midpoints << vertices(0, 0) + vertices(0, 1),
     vertices(0, 1) + vertices(0, 2),
     vertices(0, 2) + vertices(0, 3),
     vertices(0, 3) + vertices(0, 0),
     vertices(1, 0) + vertices(1, 1),
     vertices(1, 1) + vertices(1, 2),
     vertices(1, 2) + vertices(1, 3),
     vertices(1, 3) + vertices(1, 0);
      // clang-format on
      break;
    }
    default: {
      LF_ASSERT_MSG(false, "Illegal entity type!");
      break;
    }
  }                  // end switch
  midpoints *= 0.5;  // The factor 1/2
  // Evaluate f(x) at the quadrature points, i.e.  the midpoints of the
```

```
    edges
  Eigen::VectorXd fvals = Eigen::VectorXd::Zero(4);
  for (int i = 0; i < num_nodes; ++i) {
    fvals(i) = f(midpoints.col(i));
  }
  // Midpoint quadrature for all nodes of the element
  for (int k = 0; k < num_nodes; k++) {
    // Contribution from one end of an edge
    elem_vec[k] += 0.5 * fvals(k);
    // Contribution from the other end of the edge
    elem_vec[(k + 1) % num_nodes] += 0.5 * fvals(k);
  }
  // Rescale with quadrature weights
  elem_vec *= (area / num_nodes);
  return elem_vec;
}
```

SOLUTION of (2-8.c):

This is a minimal implementation. It uses a 2D hybrid mesh from LEHRFEM++'s test suit of meshes.

C++11 code 2.8.1: Sub-problem (2-8.c): Implementation of `solve()` ↪ GitLab

```
2  template <lf::assemble::EntityMatrixProvider ELMAT_PROVIDER,
3            lf::assemble::EntityMatrixProvider ELVEC_PROVIDER>
4  Eigen::VectorXd solve(ELMAT_PROVIDER &elmat_provider,
5                        ELVEC_PROVIDER &elvec_provider) {
6    // Use one of LehrFEM++'s default meshes.  Try different meshes by
       changing the
7    // function index parameter.  See the documentation of that function
       for
8    // details ablut the available meshes
9    std::shared_ptr<const lf::mesh::Mesh> mesh_p =
10       lf::mesh::test_utils::GenerateHybrid2DTestMesh(8, 1.0 / 3.0);
11   // We use a linear Lagrangian FE space
12   auto fe_space =
13       std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
14   // Reference to current mesh, obtained from the FE space
15   const lf::mesh::Mesh &mesh{*(fe_space->Mesh())};
16   // Obtain local->global index mapping for current finite element space
17   const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
18   // Dimension of finite element space'
19   const lf::base::size_type N_dofs(dofh.NumDofs());
20
21   // Matrix in triplet format holding Galerkin matrix, zero initially.
22   lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
23   // Invoke assembly on cells (co-dimension = 0).  The element matrix
       provider is
24   // passed as an argument
25   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider, A);
26   Eigen::SparseMatrix<double> A_crs = A.makeSparse();
27
28   // Right-hand side vector; has to be set to zero initially
29   Eigen::Matrix<double, Eigen::Dynamic, 1> phi(N_dofs);
30   phi.setZero();
31   // Invoke assembly on cells (codim == 0).  The element vector provider
       is
32   // passed as an argument
33   AssembleVectorLocally(0, dofh, elvec_provider, phi);
34
35   // Define solution vector
36   Eigen::VectorXd sol_vec = Eigen::VectorXd::Zero(N_dofs);
37
38   // Solve linear system using Eigen's sparse direct elimination
39   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
40   solver.compute(A_crs);
41   // Make sure LU-decomposition could be computed
42   if (solver.info() != Eigen::Success) {
43     throw std::runtime_error("Could not decompose the matrix");
44   }
45   // Backward substitution
46   sol_vec = solver.solve(phi);
```

Note that it is advisable to check the return status of EIGEN's sparse solver, because in rare cases it may

fail to perform the elimination step.

HINT 1 for (2-8.d): Note that the constructor of **lf::uscalfe::LinearFELocalLoadVector** expects an **MeshFunction** object as argument. Study [Lecture → Suppl. 2.8.3.32] to learn how to wrap a functor object into a **MeshFunction** object. ⌟

SOLUTION of (2-8.d):

C++11 code 2.8.3: Sub-problem (2-8.d): Implementation of `solvePoissonBVP()` ⇝ GitLab

```
2  Eigen::VectorXd solvePoissonBVP() {
3    // Convert the globally defined function f to a LehrFEM++ mesh
       function object
4    lf::mesh::utils::MeshFunctionGlobal mf_f{f};
5
6    // The basis expansion coefficient vector for the finite-element
       solution
7    Eigen::VectorXd solution = Eigen::VectorXd::Zero(1);
8
9    // Define the element matrix and element vector builders and solve the
       system
10   lf::uscalfe::LinearFELaplaceElementMatrix elmat_builder;
11   lf::uscalfe::LinearFELocalLoadVector<double, decltype(mf_f)> elvec_builder(
12       mf_f);
13
14   std::cout << "====================" << std::endl;
15   std::cout << "solvePoissonBVP" << std::endl;
16   // Call the function that solves the Galerkin linear system of
       equations
17   solution = solve(elmat_builder, elvec_builder);
18
19   return solution;
20 }
```

The function `f()` is defined as a regular global C++ function. It is wrapped into a **MeshFunction** [Lecture → Suppl. 2.8.3.32] object so that it can be accessed by LEHRFEM++ built-in **PROVIDER** classes. If you find this confusing, just read `mf_f` as `f`.

HINT 1 for (2-8.e): [Lecture → Ex. 1.8.0.10] and [Lecture → Rem. 1.8.0.14] can give you a crucial hint. ⌟

SOLUTION of (2-8.e):

The code terminates with an error:

```
==================
solvePoissonBVP
Error of solver: Absolute error = 31.6288, relative error = 1.24895
Norms of FE solution: L2-norm = 9.58266e+16, H1-seminorm = 25.229
```

Let $\mathfrak{B} := \{b_h^i\}_{i=1}^N$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$, the tent function basis of $\mathcal{S}_1^0(\mathcal{M})$, a tent function for *every* nodes of the mesh, recall [Lecture → Section 2.4.3]. Also remember that the local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ on triangular meshes are the barycentric coordinate functions of the triangles. For any triangle K, its barycentric coordinate functions λ_i^K, $i = 1,2,3$, sum to 1:

$$\lambda_1^K + \lambda_2^K + \lambda_3^K \equiv 1 \,,$$

Hence, the sum of the three tent functions associated with the vertices of a triangle is the constant function 1 on that triangle. Arguing like this for all triangles of the mesh $\mathcal{M}$, we conclude the **partition of unity property**

$$\sum_{j=1}^N b_h^j(\boldsymbol{x}) = 1 \quad \forall \boldsymbol{x} \in \Omega \,. \tag{2.8.4}$$

As we learned in [Lecture → Ex. 1.8.0.10], the bilinear form underlying the variational formulation of the Neumann BVP (2.8.2) is

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,, \quad u,v \in H^1(\Omega) \,. \tag{2.8.5}$$

Since the gradient of any constant function vanishes, we find

$$\mathsf{a}(\{\boldsymbol{x} \mapsto 1\}, v) = 0 \quad \forall v \in H^1(\Omega) \,. \tag{2.8.6}$$

Combining this with (2.8.4) yields

$$\mathsf{a}(\sum_{j=1}^N b_h^j, v) = 0 \quad \forall v \in H^1(\Omega) \,,$$

$$\blacktriangleright \quad \sum_{j=1}^N \mathsf{a}(b_h^j, b_h^i) = 0 \quad \forall i = 1,\ldots,N \,,$$

$$\blacktriangleright \quad \sum_{j=1}^N (\mathbf{A})_{i,j} = 0 \quad \text{for} \quad \mathbf{A} = \left[\mathsf{a}(b_h^j, b_h^i)\right]_{i,j=1}^N \in \mathbb{R}^{N,N} \,.$$

Thus, the Galerkin matrix $\mathbf{A}$ for the Neumann problem is singular, because the vector $\mathbf{1} := [1,1,\ldots,1]^\top$ belongs to its kernel, $\mathbf{A1} = \mathbf{0}$, which makes EIGEN's direct elimination solver break down. No solution can be computed.

The deeper reason is the the boundary data and the right-hand-side source functions for Neumann problems have to satisfy the **compatibility condition** [Lecture → Eq. (1.8.0.13)], which is violated for the provided right-hand-side function f.

SOLUTION of (2-8.f):

The weak formulation of (2.8.7) is derived as that for the pure Neumann problem in [Lecture → Ex. 1.8.0.10] The extra zero-order term is affected by integration by parts and just gives rise to an additive contribution $(u,v) \mapsto \int_\Omega u\,v\,\mathrm{d}\boldsymbol{x}$ to the bilinear form.

Thus, the bilinear form for the weak formulation of (2.8.7) reads

$$\mathsf{a}(u,v) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v + u\,v\,\mathrm{d}\boldsymbol{x}\,. \tag{2.8.8}$$

This bilinear form is s.p.d., because $\mathsf{a}(v,v) = \|v\|^2_{H^1(\Omega)}$ [Lecture → Def. 1.3.4.8], and, hence, all Galerkin matrices will the s.p.d., too. In particular, they will all be regular.

HINT 1 for (2-8.g): Compute the entries of the element matrix by analytic evaluation of the two-dimensional integrals. In order to avoid cumbersome computations, you may rely on the general formula from

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1!\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, . \tag{2.8.10}$$

[Lecture → Lemma 2.7.5.5] that gives the integral of products of barycentric coordinate functions over triangles. ⌟

SOLUTION of (2-8.g):

Appealing to [Lecture → Lemma 2.7.5.5] for $d = 2$ and $\alpha_1 + \alpha_2 + \alpha_3 = 2$ we find for the element matrix the general formula:

$$\mathbf{M}_K = \frac{|K|}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} , \tag{2.8.11}$$

where $|K|$ stands for the area of the triangle K.

The same question is also asked in Sub-problem (2-5.b).

SOLUTION of (2-8.h):

The local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ on the unit square are given in [Lecture → Eq. (2.6.2.3)]. By simple scaling those for a general rectangle can be obtained.

The integration over the rectangle can be split into two one-dimensional integrations, which yield

$$\mathbf{M}_K = \frac{|K|}{36}\begin{bmatrix} 4 & 2 & 1 & 2 \\ 2 & 4 & 2 & 1 \\ 1 & 2 & 4 & 2 \\ 2 & 1 & 2 & 4 \end{bmatrix} . \tag{2.8.12}$$

Remark. Then fact that the shape of the cells enters into the element mass matrices only through the area $|K|$ is a consequence of a more general truth that is explained in [Lecture → Rem. 2.8.1.10].

HINT 1 for (2-8.i): As we see from the formula (2.8.8), the bilinear form arising from (2.8.7) is the sum of

- the bilinear form $(u, v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \, \mathrm{d}x$ for $-\Delta$,
- the the bilinear form $(u, v) \mapsto \int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x}$.

Therefore, also the element matrices can be written as sums of element matrices belonging to these two bilinear forms. These have been computed in Sub-problem (2-8.g) and Sub-problem (2-8.h). ⌟

SOLUTION of (2-8.i):

C++11 code 2.8.13: Sub-problem (2-8.i): Implementation of `Eval()` for MyLinearFEElementMatrix ↪ GitLab

```
Eigen::Matrix<double, 4, 4> MyLinearFEElementMatrix::Eval(
    const lf::mesh::Entity &cell) {
  // Topological type of the cell
  const lf::base::RefEl ref_el{cell.RefEl()};

  // Obtain the vertex coordinates of the cell, which completely
  // describe its shape.
  const lf::geometry::Geometry *geo_ptr = cell.Geometry();
  // Matrix storing corner coordinates in its columns
  auto vertices = geo_ptr->Global(ref_el.NodeCoords());
  // Matrix for returning element matrix
  Eigen::Matrix<double, 4, 4> elem_mat;
  // Initialize matrix containing the mass part of the element matrix
  Eigen::Matrix<double, 4, 4> mass_elem_mat;
  // Retrieve laplace part of element matrix from LehrFEM's built-in
     laplace
  // element matrix builder
  lf::uscalfe::LinearFELaplaceElementMatrix laplace_elmat_builder;
  auto laplace_elem_mat = laplace_elmat_builder.Eval(cell);
  // Computations differ depending on the type of the cell
  switch (ref_el) {
    case lf::base::RefEl::kTria(): {
      double area = 0.5 * ((vertices(0, 1) - vertices(0, 0)) *
                           (vertices(1, 2) - vertices(1, 0)) -
                       (vertices(1, 1) - vertices(1, 0)) *
                           (vertices(0, 2) - vertices(0, 0)));
      // clang-format off
      mass_elem_mat << 2.0, 1.0, 1.0, 0.0,
     1.0, 2.0, 1.0, 0.0,
     1.0, 1.0, 2.0, 0.0,
     0.0, 0.0, 0.0, 0.0;
      // clang-format on
      mass_elem_mat *= area / 12.0;
      break;
    }
    case lf::base::RefEl::kQuad(): {
      double area =
          (vertices(0, 1) - vertices(0, 0)) * (vertices(1, 3) - vertices(1, 0));
      // clang-format off
      mass_elem_mat << 4.0, 2.0, 1.0, 2.0,
     2.0, 4.0, 2.0, 1.0,
     1.0, 2.0, 4.0, 2.0,
     2.0, 1.0, 2.0, 4.0;
      // clang-format on
      mass_elem_mat *= area / 36.0;
      break;
    }
    default: {
      LF_ASSERT_MSG(false, "Illegal cell type");
    }
  }  // end switch
```

```
52    elem_mat = laplace_elem_mat + mass_elem_mat;
53    return elem_mat;
54  }
```

SOLUTION of (2-9.a):

Refer to [Lecture → Section 2.5.3] and [Lecture → Def. 2.5.3.4].

- Global shape functions are the elements of the basis of the finite element trial and test space used in step II of a finite-element Galerkin discretization. They are associated with geometric entities of the mesh $\mathcal{M}$ and are *locally supported*. Global shape functions are defined on the whole computational domain Ω. Usually, global shape function will be $\mathcal{M}$-piecewise smooth, but not smooth everywhere.
- Local shape functions for a particular cell of the mesh are the (non-trivial) restrictions of global shape functions to that cell. Thus, local shape functions are defined only on a particular cell of the mesh and not beyond. Usually, local shape functions are smooth.

Plugging local shape functions into localized bilinear forms we obtain element matrices.

SOLUTION of (2-9.b):

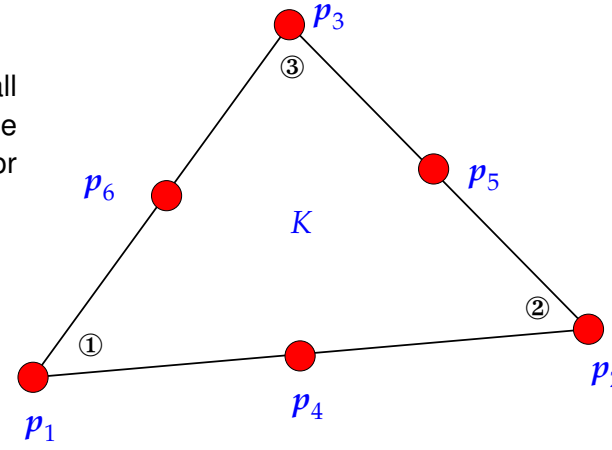
The local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ on a triangle $K \in \mathcal{M}$ span the space $\mathcal{P}_1(\mathbb{R}^2)$, whereas the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ are a basis for $\mathcal{P}_2(\mathbb{R}^2)$. Since clearly, $\mathcal{P}_1(\mathbb{R}^2) \subset \mathcal{P}_2(\mathbb{R}^2)$, the task obviously has a unique solution.

As usual, we write λ_ℓ, $\ell = 1, 2, 3$, for the barycentric coordinate functions, which are the local shape of $\mathcal{S}_1^0(\mathcal{M})$ on every triangle. For the local shape functions $b_K^1, \ldots, b_K^6$ of $\mathcal{S}_2^0(\mathcal{M})$ we adopt the numbering of [Lecture → Eq. (2.6.1.6)] and find

$$\begin{aligned} \lambda_1 &= b_K^1 + \tfrac{1}{2} b_K^4 + \tfrac{1}{2} b_K^6 \,, \\ \lambda_2 &= b_K^2 + \tfrac{1}{2} b_K^5 + \tfrac{1}{2} b_K^4 \,, \\ \lambda_3 &= b_K^3 + \tfrac{1}{2} b_K^6 + \tfrac{1}{2} b_K^5 \,. \end{aligned} \tag{2.9.1}$$

How do we get these linear combinations? Recall the *cardinal basis property* of $\{b_K^1, \ldots, b_K^6\}$ w.r.t. the local interpolation nodes $\boldsymbol{p}_\ell$ as given in Fig. 8. For *any* $q \in \mathcal{P}_2(\mathbb{R}^2)$ we have

$$\mathbf{q} = \sum_{\ell=1}^{6} q(\boldsymbol{p}_\ell)\, b_K^\ell \,.$$



So all we need to do is to evaluate the barycentric coordinate functions at the interpolation nodes $\boldsymbol{p}_\ell$. The values of the barycentric coordinate functions at the interpolation nodes do not require explaining.

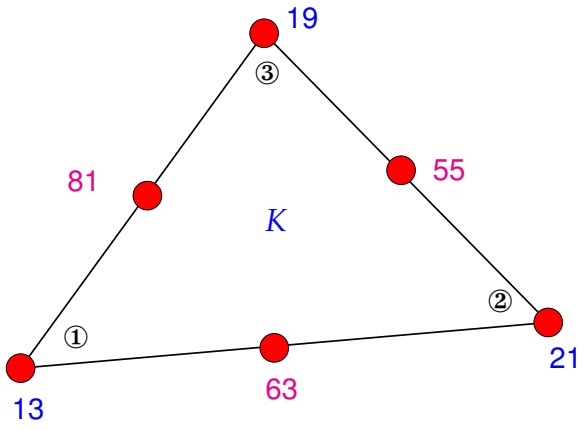
SOLUTION of (2-9.c):

The member functions of **lf::assemble::DofHandler** are explained in [Lecture → § 2.7.4.13].

- `size_type lf::assemble::DofHandler::NumLocalDofs(const lf::mesh::Ent & ) const;`
- `... lf::assemble::DofHandler::GlobalDofIndices(const lf::mesh::Enti & ) const;`
- `size_type lf::assemble::DofHandler::NumInteriorDofs(const lf::mesh: & ) const;`
- `... lf::assemble::DofHandler::InteriorGlobalDofIndices(const lf::me & ) const`

For a single triangle $K \in \mathcal{M}$ the figure beside shows in blue the indices of global shape functions associated with nodes, in magenta the indices of those global shape functions that belong to edges.

The local numbers of the vertices of the triangles are given as ①, ②, ③.



Describe the output of all member functions of **lf::assemble::DofHandler** (as listed in [Lecture → § 2.7.4.13 that take an **lf::mesh::Entity** object as an argument, when that object refers to the above triangle K.

SOLUTION of (2-9.d):

This function can be implemented in two ways:

1. In a loop running from 0 to $\texttt{NoDofs}() - 1$, one fetches the mesh entity to which the current d.o.f. belongs via `Dofhandler::Entity()`, then queries the co-dimension of that entity and increments the respective counter.
2. Loop over all mesh entities, get the indices of the global shape functions associated with them via `DofHandler::InteriorGlobalDofIndices()`, and count them. This number is also available through `DofHandler::NumInteriorDofs()`.

The following code realizes the second way.

C++ code 2.9.2: Sub-problem (2-9.d): Implementation of `countEntityDofs()` ↪ GitLab

```
2  std::array<std::size_t, 3> countEntityDofs(
3      const lf::assemble::DofHandler &dofhandler) {
4    std::array<std::size_t, 3> entityDofs;
5    // Idea:  iterate over entities in the mesh and get interior number of
       dofs for
6    // each
7    std::shared_ptr<const lf::mesh::Mesh> mesh = dofhandler.Mesh();
8    for (std::size_t codim = 0; codim <= 2; ++codim) {
9      entityDofs[codim] = 0;
10     for (const auto *el : mesh->Entities(codim)) {
11       if (el->RefEl() == lf::base::RefEl::kQuad()) {
12         throw "Only triangular meshes are allowed!";
13       }
14       entityDofs[codim] += dofhandler.NumInteriorDofs(*el);
15     }
16   }
17   return entityDofs;
18 }
```

SOLUTION of (2-9.e):

In a loop over all entities of co-dimension 1 and 2 in the given mesh, one checks the boundary flag for each entity. If the flag is **true**, we increment by the number of d.o.f.s associated with this boundary entity.

C++ code 2.9.3: Sub-problem (2-9.e): Implementation of `countBoundaryDofs()` ↪ GitLab

```
2  std::size_t countBoundaryDofs(const lf::assemble::DofHandler &dofhandler) {
3    std::shared_ptr<const lf::mesh::Mesh> mesh = dofhandler.Mesh();
4    // given an entity, bd_flags(entity) == true, if the entity is on the
5    // boundary
6    lf::mesh::utils::AllCodimMeshDataSet<bool> bd_flags(
7        lf::mesh::utils::flagEntitiesOnBoundary(mesh));
8    std::size_t no_dofs_on_bd = 0;
9    // Edges and nodes can be on the boundary
10   for (const auto *edge : mesh->Entities(1)) {
11     if (bd_flags(*edge)) {
12       no_dofs_on_bd += dofhandler.NumInteriorDofs(*edge);
13     }
14   }
15   for (const auto *node : mesh->Entities(2)) {
16     if (bd_flags(*node)) {
17       no_dofs_on_bd += dofhandler.NumInteriorDofs(*node);
18     }
19   }
20   return no_dofs_on_bd;
21 }
```

Note that the code calls `NumInteriorDofs()`, which tells the number of global shape functions associated with a mesh entity.

We need not iterate over entities of co-dimension 0 (mesh cells), because those cannot be located on the boundary.

HINT 1 for (2-9.f): The integral of a barycentric coordinate function over any triangle K is equal to $\frac{1}{3}|K|$. ⌟

SOLUTION of (2-9.f):

Run over all cells, get the indices of the tent functions covering them via `DofHandler::GlobalDofInd` sum the corresponding entries of the coefficient vector, multiply with $\frac{1}{3}|K|$ and add the result to the global summation variable.

C++ code 2.9.4: Sub-problem (2-9.f): Implementation of `IntegrateLinearFEFunction` ↪ GitLab

```
2  double integrateLinearFEFunction(
3      const lf::assemble::DofHandler& dofhandler,
4      const Eigen::VectorXd& mu) {
5    double I = 0;
6    std::shared_ptr<const lf::mesh::Mesh> mesh = dofhandler.Mesh();
7    for (const auto *cell : mesh->Entities(0)) {
8      if (cell->RefEl() == lf::base::RefEl::kQuad()) {
9          throw "Only triangular meshes are allowed!";
10     }
11     // check if we the FE space is really S_1^0
12     if (dofhandler.NumLocalDofs(*cell) != 3) {
13       throw "Not a S_1^0 FE space on a triangular mesh!";
14     }
15     // Retrieve area of the triangle by calling
16     // a LehrFEM++ utility function
17     const double area = lf::geometry::Volume(*(cell->Geometry()));
18     // iterate over dofs
19     auto int_dofs = dofhandler.GlobalDofIndices(*cell);
20     for (auto dof_idx_p = int_dofs.begin(); dof_idx_p < int_dofs.end();
21          ++dof_idx_p) {
22       // local integral of the basis function associated with this dof:
23       // in linear Lagrangian FE, the integral over the basis functions
           over
24       // a triangle K is:  1/3*vol(K)
25       const double I_bary = 1.0 / 3.0 * area;
26       // multiply by the value at the dof to get local contribution
27       I += I_bary * mu(*dof_idx_p);
28     }
29   }
30   return I;
31 }
```

SOLUTION of (2-9.g):

From [Lecture → Lemma 2.7.5.5] with $d = 2$ and the barycentric coordinate representations from [Lecture → Eq. (2.6.1.6)] we get:

$$\int_K b_K^\ell \,\mathrm{d}\boldsymbol{x} = \begin{cases} 0 & \text{for} \quad \ell = 1,2,3\,, \\ \frac{1}{3}|K| & \text{for} \quad \ell = 4,5,6\,. \end{cases}$$

Hence, for $v_h \in \mathcal{S}_2^0(\mathcal{M})$

$$\int_\Omega v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \sum_{K\in\mathcal{M}} \tfrac{1}{3}|K|\Big(v_h(\boldsymbol{p}_K^4) + v_h(\boldsymbol{p}_K^5) + v_h(\boldsymbol{p}_K^6)\Big)\,,$$

where we wrote $\boldsymbol{p}_K^\ell$, $\ell = 1,\ldots,6$, for the local interpolation nodes [Lecture → Eq. (2.6.1.5)] belonging to the triangle K. Note that the numbers $v_h(\boldsymbol{p}_K^5)$ can directly be retrieved from the coefficient vector mu, owing to the cardinal basis property of the global shape functions w.r.t. the set of global interpolation nodes, see [Lecture → Eq. (2.6.1.4)].

C++ code 2.9.5: Sub-problem (2-9.g): Implementation of `integrateQuadraticFEFunction()` ↬ GitLab

```
2  double integrateQuadraticFEFunction(const lf::assemble::DofHandler &dofhandler,
3                                      const Eigen::VectorXd &mu) {
4    double l = 0;
5    std::shared_ptr<const lf::mesh::Mesh> mesh = dofhandler.Mesh();
6    for (const auto *cell : mesh->Entities(0)) {
7      // check if we the FE space is really S_2^0
8      if (dofhandler.NumLocalDofs(*cell) != 6) {
9        throw "Not a S_2^0 FE space!";
10     }
11     const double weight = 1.0 / 3.0 * lf::geometry::Volume(*(cell->Geometry()));
12     // iterate over dofs
13     auto int_dofs = dofhandler.GlobalDofIndices(*cell);
14     // The integrated basis functions associated with the nodes are 0:
15     // ∫_K b_K^j dx = 0 for j = 1,2,3.  Skip!
16     for (int l = 3; l < 6; ++l) {
17       // The integrated basis functions associated with the edges are:
18       // ∫_K b_K^j dx = |K|/3 for j = 4,5,6
19       // multiply by the value at the dof to get local contribution
20       l += (weight * mu(int_dofs[l]));
21     }
22   }
23   return l;
24 }
```

HINT 1 for (2-9.h): In Sub-problem (2-9.b) you determined how to write the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$ as a linear combination of those of $\mathcal{S}_2^0(\mathcal{M})$. This formula can now be used. ⌟

SOLUTION of (2-9.h):

Note that this task has nothing to do with assembly, we do not build any matrix. Nevertheless, a cell-oriented implementation is the simplest option:

- Loop over all cells of the mesh
- For each cell $K \in \mathcal{M}$ obtain the global indices of both the local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ ($\rightarrow$ indices i_1, i_2, i_3) and of the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ ($\rightarrow$ indices $j_1, \ldots, j_6$). A call of `DofHandler::GlobalDofIndices()` for both **lf::assemble::DofHandler** object yields these index arrays.
- Then based on (2.9.1) *set*

$$
\begin{aligned}
(\vec{\eta})_{j_\ell} &= \sum_{j=1}^{3} (\vec{\mu})_{i_j} \underbrace{\lambda_j(\boldsymbol{p}_K^\ell)}_{=\delta_{j,\ell}} &&= (\vec{\mu})_{i_\ell}\,, \quad \ell = 1,2,3\\
(\vec{\eta})_{j_4} &= \sum_{j=1}^{3} (\vec{\mu})_{i_j} \underbrace{\lambda_j(\boldsymbol{p}_K^4)}_{\in\{0,\frac{1}{2}\}} &&= \frac{1}{2}\Big((\vec{\mu})_{i_1} + (\vec{\mu})_{i_2}\Big)\,,\\
(\vec{\eta})_{j_5} &= \sum_{j=1}^{3} (\vec{\mu})_{i_j} \underbrace{\lambda_j(\boldsymbol{p}_K^5)}_{\in\{0,\frac{1}{2}\}} &&= \frac{1}{2}\Big((\vec{\mu})_{i_2} + (\vec{\mu})_{i_3}\Big)\,,\\
(\vec{\eta})_{j_6} &= \sum_{j=1}^{3} (\vec{\mu})_{i_j} \underbrace{\lambda_j(\boldsymbol{p}_K^6)}_{\in\{0,\frac{1}{2}\}} &&= \frac{1}{2}\Big((\vec{\mu})_{i_3} + (\vec{\mu})_{i_1}\Big)\,.
\end{aligned}
$$

Note that we assign the same value to entries of $\vec{\eta}$ possibly multiple times, but this does not do any harm (as long as you *do not sum*, which is not appropriate here).

C++ code 2.9.6: Sub-problem (2-9.h): Implementation of `convertDOFsLinearQuadratic()` ↬ GitLab

```
2  Eigen::VectorXd convertDOFsLinearQuadratic(
3      const lf::assemble::DofHandler &dofh_Linear_FE,
4      const lf::assemble::DofHandler &dofh_Quadratic_FE,
5      const Eigen::VectorXd &mu) {
6    if (dofh_Linear_FE.Mesh() != dofh_Quadratic_FE.Mesh()) {
7      throw "Underlying meshes must be the same for both DOF handlers!";
8    }
9    std::shared_ptr<const lf::mesh::Mesh> mesh =
10       dofh_Linear_FE.Mesh();  // get the mesh
11   // coefficient vector for returning the result
12   Eigen::VectorXd zeta(dofh_Quadratic_FE.NumDofs());
13   // Play safe:  always set zero if you're not sure to set every entry
       later
14   // on for us this shouldn't be a problem, but just to be sure
15   zeta.setZero();
16   for (const auto *cell : mesh->Entities(0)) {
17     // check if the spaces are actually linear and quadratic
18     if (dofh_Linear_FE.NumLocalDofs(*cell) != 3 ||
19         dofh_Quadratic_FE.NumLocalDofs(*cell) != 6) {
```

```
      throw std::runtime_error(
          "dofh_Linear_FE must have 3 dofs per cell and dofh_Quadratic_FE 6!");
    }
    // get the global dof indices of the linear and quadratic FE spaces, note
    // that the vectors obey the LehrFEM++ numbering, which we will make use of
    // lin_dofs will have size 3 for the 3 dofs on the nodes and
    // quad_dofs will have size 6, the first 3 entries being the nodes and
    // the last 3 the edges
    std::span<const lf::assemble::gdof_idx_t> lin_dofs =
        dofh_Linear_FE.GlobalDofIndices(*cell);
    std::span<const lf::assemble::gdof_idx_t> quad_dofs =
        dofh_Quadratic_FE.GlobalDofIndices(*cell);
    for (std::size_t l = 0; l <= 2; ++l) {
      // Let p_1,p_2,p_3 be the nodes of the triangle and p_4,p_5,p_6 the
      // edge midpoints.  Let λ_1,λ_2,λ_3 be the local basis
      // functions of S_1^0 and b_1,..,b_6 for S_2^0.  The values of
      // λ_i in the interpolation nodes p_1,..,p_6 are the
      // coefficients of writing λ_i as lin.  comb of b_1,..b_6.
      // From 2-9.a we know λ_l(p_l) = 1; l = 1,2,3.
      // Hence we simply copy the coefficient of λ_l to b_l
      // for l = 1,2,3.
      zeta(quad_dofs[l]) = mu(lin_dofs[l]);
      // And λ_l(p_l + 3) = 0.5, λ_{(l+1)mod3}(p_l + 3) = 0.5;
      // l = 1,2,3.  Hence we copy 0.5 of the respective coefficients!
      zeta(quad_dofs[l + 3]) =
          0.5 * (mu(lin_dofs[l]) + mu(lin_dofs[(l + 1) % 3]));
    }
  }
  return zeta;
}
```

We can rephrase the task as:

> Compute the $\mathcal{S}_2^0(\mathcal{M})$ basis expansion coefficients (using the standard basis of $\mathcal{S}_2^0(\mathcal{M})$ as introduced in [Lecture → Ex. 2.6.1.2]) of $\mathcal{S}_2^0(\mathcal{M})$-interpolant of a function $f_h \in \mathcal{S}_1^0(\mathcal{M})$ (based on its nodal tent function basis).

We know that the nodal basis of $\mathcal{S}_2^0(\mathcal{M})$ is a *cardinal basis* with respect to the following set of interpolation nodes

$$\mathcal{V}(\mathcal{M}) \cup \{\text{midpoints } \boldsymbol{m}_e : \ e \in \mathcal{E}(\mathcal{M})\}\,, \tag{2.9.7}$$

where $\mathcal{V}(\mathcal{M})$ is the set of nodes of the mesh $\mathcal{M}$, and $\mathcal{E}(\mathcal{M})$ is the set of edges of $\mathcal{M}$. Therefore, the basis expansion coefficients of $v_h \in \mathcal{S}_2^0(\mathcal{M})$ agree with the function values in those points.

For $f_h \in \mathcal{S}_1^0(\mathcal{M})$ we have that the value

- $f_h(\boldsymbol{x})$, $\boldsymbol{x} \in \mathcal{V}(\mathcal{M})$, is directly available as the basis expansion coefficient of the tent function sitting at $\boldsymbol{x}$, and
- $f_h(\boldsymbol{m}_e)$, $e \in \mathcal{E}(\mathcal{M})$, can be computed as the *average* of the values of f_h in the endpoints of the edge e,

$$f_h(\boldsymbol{m}_e) = \tfrac{1}{2}(f_h(\boldsymbol{p}_1) + f_h(\boldsymbol{p}_2))\,, \quad \boldsymbol{p}_1, \boldsymbol{p}_2 \hat{=} \text{ endpoints of } e\,.$$

Summing up, an alternative implementation could involve the following steps:

- First traverses all nodes of the mesh and copy the entries of the coefficient vectors associated with a node, and
- then visit all edges of the mesh, fetch the global index number of the $\mathcal{S}_2^0(\mathcal{M})$-d.o.f. associated with the, the global index numbers of the $\mathcal{S}_1^0(\mathcal{M})$-d.o.f. associated with the endpoints of the edge, and set the $\mathcal{S}_2^0(\mathcal{M})$-d.o.f. to the average of the two $\mathcal{S}_1^0(\mathcal{M})$-d.o.f.s.

This is realized in Code 2.9.8.

C++ code 2.9.8: Sub-problem (2-9.h): Alternative implementation of `convertDOFsLinearQuadratic()` ↬ GitLab

```
Eigen::VectorXd convertDOFsLinearQuadratic_alt(
    const lf::assemble::DofHandler &dofh_Linear_FE,
    const lf::assemble::DofHandler &dofh_Quadratic_FE,
    const Eigen::VectorXd &mu) {
  LF_ASSERT_MSG(dofh_Linear_FE.Mesh() == dofh_Quadratic_FE.Mesh(),
                "Underlying meshes must be the same for both DOF handlers!");
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh_Linear_FE.Mesh();
  // coefficient vector for returning the result
  Eigen::VectorXd zeta(dofh_Quadratic_FE.NumDofs());
  // Visit all nodes of the mesh and copy dof values
  for (const lf::mesh::Entity *node : mesh_p->Entities(2)) {
    // Obtain global index numbers of the GSFs at the node for both
       finite
    // element spaces
    std::span<const lf::assemble::gdof_idx_t> lin_dofs =
        dofh_Linear_FE.InteriorGlobalDofIndices(*node);
    std::span<const lf::assemble::gdof_idx_t> quad_dofs =
        dofh_Quadratic_FE.InteriorGlobalDofIndices(*node);
    LF_ASSERT_MSG(lin_dofs.size() == 1, "One dof per node expected");
    LF_ASSERT_MSG(quad_dofs.size() == 1, "One dof per node expected");
    // Just copy the coefficient
    zeta[quad_dofs[0]] = mu[lin_dofs[0]];
  }
  // Run through all edges of the mesh
  for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
    // Obtain pointers to endpoints of edge
    std::span<const lf::mesh::Entity *const> endpoints{edge->SubEntities(1)};
    LF_ASSERT_MSG(endpoints.size() == 2, "Edge must have two endpoints");
    // Obtain indices of GSFs for linear FE spaces associated with
       endpoints
    std::span<const lf::assemble::gdof_idx_t> lindof_p0 =
        dofh_Linear_FE.InteriorGlobalDofIndices(*endpoints[0]);
    LF_ASSERT_MSG(lindof_p0.size() == 1, "Onle one dof per vertex allowed");
    std::span<const lf::assemble::gdof_idx_t> lindof_p1 =
        dofh_Linear_FE.InteriorGlobalDofIndices(*endpoints[1]);
    LF_ASSERT_MSG(lindof_p1.size() == 1, "Onle one dof per vertex allowed");
    std::span<const lf::assemble::gdof_idx_t> quad_dofs =
        dofh_Quadratic_FE.InteriorGlobalDofIndices(*edge);
    LF_ASSERT_MSG(quad_dofs.size() == 1,
                  "Only one dof associated with an edge");
    // Set value of edge dof to average of adjacent vertex dofs
    zeta[quad_dofs[0]] = 0.5 * (mu[lindof_p0[0]] + mu[lindof_p1[0]]);
  }
  return zeta;
```

```
44 }
```

HINT 1 for (3-1.a): Recall the definition of continuity of a linear functional from

Definition [Lecture → Def. 1.2.3.42]. Continuity of a linear and bilinear form

Consider a normed vector space V_0 with norm $\|\cdot\|$. A linear form $\ell : V_0 \to \mathbb{R}$ (→ **??**) is continuous or bounded on V_0, if

$$\exists C > 0: \quad |\ell(v)| \leq C\|v\| \quad \forall v \in V_0 \, .$$

A bilinear form $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$ (→ [Lecture → Def. 0.3.1.4]) on V_0 is continuous, if

$$\exists K > 0: \quad |\mathsf{a}(u,v)| \leq K\|u\|\|v\| \quad \forall u, v \in V_0 \, .$$

and then use [Lecture → Thm. 1.9.0.10]. ⌟

SOLUTION of (3-1.a):

By using the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)] in $L^2(\partial\Omega)$,

$$\left|\int_{\partial\Omega} u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right| \leq \left(\int_{\partial\Omega} |u(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2}\left(\int_{\partial\Omega} |v(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2} = \|u\|_{L^2(\partial\Omega)}\|v\|_{L^2(\partial\Omega)}\,,$$

the multiplicative trace inequality,

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

$$\exists C = C(\Omega) > 0\colon \quad \|u\|^2_{L^2(\partial\Omega)} \leq C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega)\,.$$

and the fact that $\|w\|_{L^2(\partial\Omega)} \leq C_1$, we obtain

$$\begin{aligned} |F(v)| &\leq \int_{\partial\Omega} |w(\boldsymbol{x})v(\boldsymbol{x})|\,\mathrm{d}S(\boldsymbol{x}) \leq \|w\|_{L^2(\partial\Omega)}\|v\|_{L^2(\partial\Omega)} \\ &\leq C_1C_2\|v\|^{1/2}_{L^2(\Omega)}\|v\|^{1/2}_{H^1(\Omega)} \leq C\|v\|_{H^1(\Omega)}\,. \end{aligned}$$

Note that we have to use the full $H^1(\Omega)$-norm as introduced in [Lecture → Def. 1.3.4.8] and not the seminorm $|\cdot|_{H^1(\Omega)}$! This proves the continuity of F w.r.t. $H^1(\Omega)$.

SOLUTION of (3-1.b):

For a linear variational problem with s.p.d. bilinear form, the energy norm of a finite element function v_h described by a coefficient vector $\vec{\eta}$ is

$$\|v_h\|_a = \mathsf{a}(v_h, v_h)^{1/2} = \left(\vec{\eta}^\top \mathbf{A}\vec{\eta}\right)^{1/2},$$

where $\mathbf{A}$ is the Galerkin matrix. Now, note that the norm $|v_h|_{H^1(\Omega)}$ agrees with the energy norm induced by the left-hand side bilinear form from (3.1.2) in the case $\alpha \equiv 1$, $\gamma \equiv 0$, and $\beta \equiv 0$.

C++ code 3.1.4: Sub-problem (3-1.b): Implementation of `compH1seminorm` ↪ GitLab

```
2  double compH1seminorm(const lf::assemble::DofHandler &dofh,
3                        const Eigen::VectorXd &u) {
4    double result = 0.0;
5
6    // Compute Stiffness matrix with alpha := 1, beta := 0, gamma := 0
7    auto const_one = [](Eigen::Vector2d x) -> double { return 1.0; };
8    auto const_zero = [](Eigen::Vector2d x) -> double { return 0.0; };
9    auto A = compGalerkinMatrix(dofh, const_one, const_zero, const_zero);
10
11   // Compute seminorm using the Stiffness matrix
12   result = u.transpose() * A * u;
13   result = std::sqrt(result);
14   return result;
15 }
```

HINT 1 for (3-1.c): As coefficient $\beta(\boldsymbol{x})$ pass the weight function $w(\boldsymbol{x})$. Also note that the constant function $\boldsymbol{x} \mapsto 1$ is contained in the finite element space $\mathcal{S}_1^0(\mathcal{M})$. ⌟

SOLUTION of (3-1.c):

Start from the trivial observation that

$$F(v_h) := \int_{\partial\Omega} w(\boldsymbol{x}) v_h(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = \int_{\partial\Omega} w(\boldsymbol{x})\, 1\, v_h(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = \mathsf{a}(\{\boldsymbol{x} \mapsto 1\}, v_h) \,,$$

where a stands for the bilinear form on the left-hand side of (3.1.2) with the particular choice $\alpha, \gamma \equiv 0$, and $\beta := w$.

The function $\{\boldsymbol{x} \mapsto 1\} \in \mathcal{S}_1^0(\mathcal{M})$ has a basis representation through the coefficient vector $\mathbf{1}$, whose every component is equal to 1: if $\{b_h^1, \ldots, b_h^N\}$ is the tent-function nodal basis of $\mathcal{S}_1^0(\mathcal{M})$ then

$$\sum_{\ell=1}^{1} 1 \cdot b_h^\ell(\boldsymbol{x}) = 1 \quad \forall \boldsymbol{x} \in \Omega \,.$$

Hence, if $\vec{\boldsymbol{\eta}} \in \mathbb{R}^N$ is the coefficient vector of $v_h \in \mathcal{S}_1^0(\mathcal{M})$ we find

$$F(v_h) = \mathbf{1}^\top \mathbf{A} \vec{\boldsymbol{\eta}} \,,$$

where $\mathbf{A} \in \mathbb{R}^{N,N}$ is the Galerkin matrix for a.

C++ code 3.1.5: Sub-problem (3-1.c): Implementation of `compBoundaryFunctional()`
➥ GitLab

```
2  template <typename FUNCTION>
3  double compBoundaryFunctional(const lf::assemble::DofHandler &dofh,
4                                const Eigen::VectorXd &u, FUNCTION &&w) {
5    double result = 0.0;
6    // constant zero function
7    auto const_zero = [](Eigen::Vector2d x) -> double { return 0.0; };
8    // Generate Galerkin matrix for boundary term alone, unsing the weight
9    // function as coefficient. Note the use of std::forward to preserve rvalue
10   // types.
11   auto A = compGalerkinMatrix(dofh, const_zero, const_zero,
12                               std::forward<FUNCTION>(w));
13   // A vector with all components set to one
14   Eigen::VectorXd ones = Eigen::VectorXd::Ones(dofh.NumDofs());
15   // The trick: evaluate the functional as 1^T*A*u
16   result = ones.transpose() * A * u;
17   return result;
18 }
```

SOLUTION of (3-1.d):

Using the multiplicative trace inequality, we have

$$\|u-u_h\|^2_{L^2(\partial\Omega)} \le C\,\|u-u_h\|_{L^2(\Omega)}\|u-u_h\|_{H^1(\Omega)}\,.$$

$$\|u-u_h\|_{H^1(\Omega)} \le C \inf_{v_h\in V_{0,N}} \|u-v_h\|_{H^1(\Omega)}\,.$$

$$\|u-u_h\|_{L^2(\Omega)} \le Ch_{\mathcal{M}}\cdot \inf_{v_h\in V_{0,N}} \|u-v_h\|_{H^1(\Omega)}\,.$$

$$\inf_{v_h\in V_{0,N}} \|u-v_h\|_{H^1(\Omega)} \le Ch_{\mathcal{M}}|u|_{H^2(\Omega)}\,.$$

$$\|u-u_h\|^2_{L^2(\partial\Omega)} \le Ch^3_{\mathcal{M}} \quad\Rightarrow\quad \|u-u_h\|_{L^2(\partial\Omega)} = O\left(h^{\frac{3}{2}}_{\mathcal{M}}\right) \quad \text{for} \quad h_{\mathcal{M}}\to 0\,.$$

The highlighted estimate completely answers the problem.

HINT 1 for (3-1.e): Abstract duality estimates are stated in [Lecture → Thm. 3.6.1.7]. ⌟

SOLUTION of (3-1.e):

We denote by $g_F \in H^1(\Omega)$ the dual solution of (3.1.6), which satisfies

$$\mathsf{a}(v, g_F) := \int_\Omega \mathbf{grad}\, v \cdot \mathbf{grad}\, g_F + v\, g_F \, \mathrm{d}\boldsymbol{x} = F(v) := \int_{\partial\Omega} w\, v \, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \,. \tag{3.1.9}$$

Using integration by parts for $v \in H^1_0(\Omega)$, we have

$$\int_\Omega (-\Delta g_F + g_F) v \, \mathrm{d}\boldsymbol{x} = 0 \,,$$

which leads to the PDE

$$-\Delta g_F + g_F = 0 \,. \tag{3.1.10}$$

Testing (**??**) with $v \in H^1(\Omega)$ and using (**??**), we get the following Neumann boundary condition:

$$\mathbf{grad}\, g_F \cdot \boldsymbol{n} = w \,.$$

The boundary value problem satisfied by the dual solution is then

$$\begin{array}{rll} -\Delta g_F + g_F &= 0 & \text{in } \Omega, \\ \mathbf{grad}\, g_F \cdot \boldsymbol{n} &= w & \text{on } \partial\Omega \,. \end{array} \tag{3.1.11}$$

SOLUTION of (3-1.f):

from Sub-problem (3-1.a) we know that $F : H^1(\Omega) \to \mathbb{R}$ is continuous. Using the duality estimates of [Lecture → Thm. 3.6.1.7], Cea's lemma, [Lecture → Thm. 1.8.0.20], and [Lecture → Thm. 3.3.5.6], $V_{0,h} = \mathcal{S}_1^0(\mathcal{M})$,

$$\begin{aligned} |F(u) - F(u_h)| &\leq \|u - u_h\|_{\mathsf{a}} \cdot \inf_{v_h \in V_{0,N}} \|g_F - v_h\|_{\mathsf{a}} \\ &= \inf_{v_h \in V_{0,h}} \|u - v_h\|_{H^1(\Omega)} \inf_{v_h \in V_{0,h}} \|g_F - v_h\|_{H^1(\Omega)} \\ &\leq C\, h_{\mathcal{M}} \inf_{v_h \in V_{0,h}} \|g_F - v_h\|_{H^1(\Omega)} \leq C\, h_{\mathcal{M}} \inf_{v_h \in V_{0,N}} \|g_F - v_h\|_{H^1(\Omega)} \,. \end{aligned}$$

Since g_F is a solution to (**??**) and $w \in C^1_{\mathsf{pw}}(\partial\Omega)$, by Thm. 3.1.7 we conclude $g_F \in H^2(\Omega)$. Then, by [Lecture → Thm. 3.3.5.6],

$$\inf_{v_h \in V_{0,h}} \|g_F - v_h\|_{H^1(\Omega)} \leq C\, h_{\mathcal{M}} \|g_F\|_{H^2(\Omega)} \,.$$

$$|F(u) - F(u_h)| \leq C\, h_{\mathcal{M}}^2 \|g_F\|_{H^2(\Omega)} \|u\|_{H^2(\Omega)} = O(h_{\mathcal{M}}^2) \quad \text{for} \quad h_{\mathcal{M}} \to 0 \,. \tag{3.1.12}$$

Supplement. We could also try to exploit the estimate of Sub-problem (3-1.d), because the Cauchy-Schwarz inequality in $L^2(\partial\Omega)$ gives

$$|F(u) - F(u_h)| \leq \|w\|_{L^2(\partial\Omega)} \|u - u_h\|_{L^2(\partial\Omega)} = O(h_{\mathcal{M}}^{\frac{3}{2}}) \,.$$

This approach loses $O(h_{\mathcal{M}}^{\frac{1}{2}})$ compared to (**??**)!

SOLUTION of (3-1.g):

C++11 code 3.1.13: Sub-problem (3-1.g): C++ function `approxBoundaryFunctionalValues()` ↪ GitLab

```
std::vector<std::pair<unsigned int, double>> approxBoundaryFunctionalValues(
    unsigned int L) {
  std::vector<std::pair<unsigned int, double>> result;
  auto meshes = generateTestMeshSequence(L - 1);
  int num_meshes = meshes->NumLevels();
  for (int level = 0; level < num_meshes; ++level) {
    auto mesh_p = meshes->getMesh(level);

    // Set up global FE space; lowest order Lagrangian finite elements
    auto fe_space =
        std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);

    // Obtain local->global index mapping for current finite element
       space
    const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
    const lf::base::size_type N_dofs(dofh.NumDofs());

    // compute galerkin matrix with alpha = 1.0, gamma = 1.0, beta = 0.0
    auto const_one = [](Eigen::Vector2d x) -> double { return 1.0; };
    auto const_zero = [](Eigen::Vector2d x) -> double { return 0.0; };
    auto A = compGalerkinMatrix(dofh, const_one, const_one, const_zero);

    // compute load vector for f(x) = x.norm()
    auto f = [](Eigen::Vector2d x) -> double { return x.norm(); };
    lf::mesh::utils::MeshFunctionGlobal mf_f{f};
    lf::uscalfe::ScalarLoadElementVectorProvider elvec_builder(fe_space, mf_f);
    Eigen::VectorXd phi(N_dofs);
    phi.setZero();
    AssembleVectorLocally(0, dofh, elvec_builder, phi);

    // solve system of equations
    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
    solver.compute(A);
    Eigen::VectorXd u = solver.solve(phi);

    // set up weight function
    auto w = [](Eigen::Vector2d x) -> double { return 1.0; };
    double functional_value = compBoundaryFunctional(dofh, u, w);

    result.push_back({N_dofs, functional_value});
  }
  return result;
}
```

C++11 code 3.1.14: Sub-problem (3-1.g): C++ function `generateTestMeshSequence()` ↪ GitLab

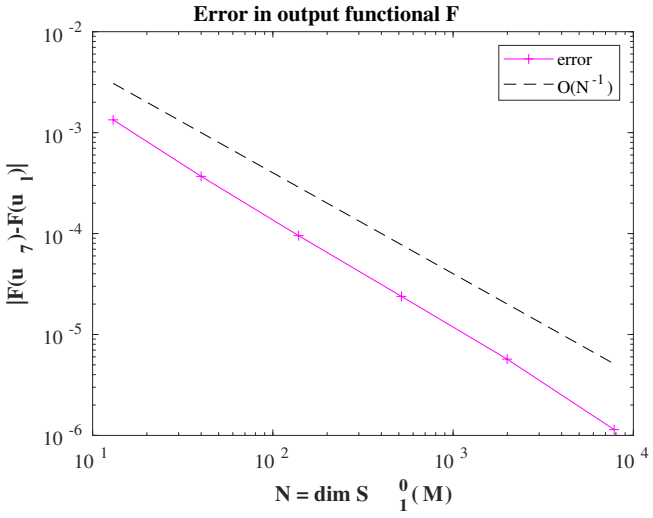
```
std::shared_ptr<lf::refinement::MeshHierarchy> generateTestMeshSequence(
    unsigned int L) {
  auto mesh = lf::mesh::test_utils::GenerateHybrid2DTestMesh(3, 1.0 / 3.0);
```

```
    std::shared_ptr<lf::refinement::MeshHierarchy> meshes =
        lf::refinement::GenerateMeshHierarchyByUniformRefinemnt(mesh, L);
    return meshes;
}
```

SOLUTION of (3-1.h):

We obtain the following deviations from the reference value that we accept as bona-fide output error.

N	$\lvert F(u_l) - F(u_7)\rvert$
13	1.10634e-04
40	8.19722e-05
139	2.73702e-05
517	7.52150e-06
1993	1.86477e-06
7825	3.78886e-07



We can read off asymptotic *algebraic convergence* with *rate* 1 (even slightly better) in $N := \dim \mathcal{S}_1^0(\mathcal{M})$ for $N \to \infty$. The matches the theoretical estimate from Sub-problem (3-1.f), because $O(N^{-1}) = O(h_{\mathcal{M}}^2)$ in the case of uniform regular refinement.

C++11 code 3.1.15: Sub-problem (3-1.h): Extended `main()` function ↪ GitLab

```
int main() {
  // read in mesh and set up finite element space
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  lf::io::GmshReader reader(std::move(mesh_factory), "meshes/square.msh");
  auto mesh = reader.mesh();
  // obtain dofh for lagrangian finite element space
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh);
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  // Solve test problem
  Eigen::VectorXd mu = AvgValBoundary::solveTestProblem(dofh);
  // compute H1 seminorm of the solution
  double h1s_norm = AvgValBoundary::compH1seminorm(dofh, mu);
  // compute boundary functional
  auto w = [](Eigen::Vector2d x) -> double { return 1.0; };
  double boundary_functional =
      AvgValBoundary::compBoundaryFunctional(dofh, mu, w);

  std::cout << "H1s-norm: " << h1s_norm << "\n";
  std::cout << "F: " << boundary_functional << "\n";

  auto results = AvgValBoundary::approxBoundaryFunctionalValues(7);
  double ground_truth = results[6].second;
  std::cout << "N_Dofs     Error\n";
  for (int i = 0; i < 6; i++) {
    std::cout << results[i].first << "   "
              << std::abs(results[i].second - ground_truth) << "\n";
  }
  return 0;
```

```
}
```

C++11 code 3.1.16: Sub-problem (3-1.h): C++ function `solveTestProblem()` ⇢ GitLab

```
Eigen::VectorXd solveTestProblem(const lf::assemble::DofHandler &dofh) {
  // Obtain Galerkin matrix for alpha = beta = gamma := 1.0
  auto const_one = [](Eigen::Vector2d x) -> double { return 1.0; };
  auto A = compGalerkinMatrix(dofh, const_one, const_one, const_one);

  // Set up load vector
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(dofh.Mesh());
  lf::mesh::utils::MeshFunctionConstant mf_identity{1.0};
  lf::uscalfe::ScalarLoadElementVectorProvider elvec_builder(fe_space,
                                                             mf_identity);
  Eigen::VectorXd phi(dofh.NumDofs());
  phi.setZero();
  AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // Solve system of linear equations
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A);
  Eigen::VectorXd mu = solver.solve(phi);

  return mu;
}
```

HINT 1 for (3-10.a): Don't forget to specify the test and trial spaces! ⌟

SOLUTION of (3-10.a):

First we observe that (3.10.1) features inhomogeneous Dirichlet boundary conditions on Γ_D and Neumann boundary conditions on Γ_S and Γ_N. Next we have to fix the configuration/trial space V (where the solution u lives) and the test space V_0 (where the test functions v live):

$$\begin{aligned} V &= \left\{ u \in H^1(\Omega) : u_{|\Gamma_D} = 1 \right\}, \\ V_0 &= \left\{ u \in H^1(\Omega) : u_{|\Gamma_D} = 0 \right\}. \end{aligned} \tag{3.10.7}$$

The definition of V takes into account the inhomogeneous Dirichlet boundary condition along Γ_D. Next we find the variational formulation of (3.10.1) by multiplying the PDE by a test function $v \in V_0$ and integrating over the domain Ω:

$$\begin{aligned} -\int_\Omega \operatorname{div}(\alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ &\quad - \int_{\partial\Omega} \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &= \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ &\quad - \int_{\Gamma_S} \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &\quad - \int_{\Gamma_N} \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &\quad - \int_{\Gamma_D} \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &= \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ &\quad + \int_{\Gamma_S} u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &\quad - \int_{\Gamma_N} 0\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &\quad - \int_{\Gamma_D} \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,0\;\mathrm{d}S(\boldsymbol{x}), \end{aligned}$$

where in the first equality we apply Green's first formula [Lecture → Thm. 1.5.2.7], in the second equality we split the boundary of Ω into the different parts and in the third equality we have used the Neumann boundary conditions and the fact that $v(\boldsymbol{x})$ vanishes on Γ_D.

With the above, we can write the variational formulation of (3.10.1) as

$$\text{find } u \in V \text{ such that } \mathsf{a}(u,v) = 0 \text{ for all } v \in V_0,$$

where the bilinear form a is given by

$$\mathsf{a}(u,v) := \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\Gamma_S} u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}).$$

SOLUTION of (3-10.j):

A proposed solution code for the function `geoThermBdElim` is shown in **??**. Elimination targets the coefficients associated with basis functions numbered from 0 through n, because those belong to vertices on Γ_D according to the rules for lexicographic numbering.

C++ code 3.10.21: Sub-problem (3-10.j): Implementation of `geoThermBdElim` → GitLab

```
2  void geoThermBdElim(unsigned int n, std::vector<Eigen::Triplet<double>> &A) {
3    // Identify Triplets on Boundary with Dirichlet Condition
4    for (auto &a : A) {
5      if (a.row() < n + 1) {
6        a = Eigen::Triplet(a.row(), a.col(), 0.0);
7      }
8    }
9
10   // Set to identity on Dirchlet Boundary part of the boundary Gamma
11   for (int i = 0; i < n + 1; i++) {
12     A.push_back(Eigen::Triplet(i, i, 1.0));
13   }
14 }
```

SOLUTION of (3-10.k):

C++ code 3.10.22: Sub-problem (3-10.k): Implementation of `geoThermSolve` ↪ GitLab

```cpp
template <typename FUNCTOR1, typename FUNCTOR2>
Eigen::VectorXd geoThermSolve(unsigned int n, FUNCTOR1 &&alpha,
                              FUNCTOR2 &&Psi) {
  // Total Number of dofs
  int N_dofs = (n + 1) * (n + 1);

  // Basis expansion coefficient vector mu
  Eigen::VectorXd mu = Eigen::VectorXd::Zero(N_dofs);

  // Get the assembled Galerkin matrix A in triplets format
  std::vector<Eigen::Triplet<double>> triplets =
      assembleGeoTherm(n, alpha, Psi);
  // Eliminate Dirichlet boundary conditions in Galerkin matrix A
  geoThermBdElim(n, triplets);

  // Obtain Sparse Galerkin matrix
  Eigen::SparseMatrix<double> A(N_dofs, N_dofs);
  A.setFromTriplets(triplets.begin(), triplets.end());

  // Set the right-hand side vector to zero
  Eigen::VectorXd b = Eigen::VectorXd::Zero(N_dofs);
  // Set the right-hand side vector to one accounting for the Dirichlet
     boundary
  for (int j = 0; j < n + 1; j++) {
    b(j) = 1.0;
  }

  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A);
  if (solver.info() != Eigen::Success) {
    throw std::runtime_error("Could not decompose the matrix");
  }
  mu = solver.solve(b);

  return mu;
}
```

SOLUTION of (3-10.l):

A proposed solution code for the function `geoThermSurfInt()` is shown in **??**. Recall the local→global index map from Sub-problem (3-10.h) and the fact that the basis functions on Γ_S are the last in lexicographic ordering.

C++ code 3.10.24: Sub-problem (3-10.l): Implementation of `geoThermSurfInt` ⇢ GitLab

```
2  template <typename FUNCTOR>
3  double geoThermSurfInt(unsigned int n, FUNCTOR &&Psi,
4                         const Eigen::VectorXd &mu) {
5    // Mesh width
6    double h = 1.0 / n;
7
8    // Surface integral of geothermal problem
9    double SurfInt = 0.0;
10
11   for (int j = 0; j < n; j++) {
12     // Compute length of edfe
13     double dl2 = h * h + (Psi((j + 1) * h) - Psi(j * h)) *
14                          (Psi((j + 1) * h) - Psi(j * h));
15     double dl = std::sqrt(dl2);
16
17     // Trapezoidal rule
18     int i = j + (n + 1) * n;
19     int ip = j + 1 + (n + 1) * n;
20     SurfInt += 0.5 * dl * (mu(i) + mu(ip));
21   }
22
23   return SurfInt;
24 }
```

SOLUTION of (3-10.m):

Two variational crimes are committed:

1. Polygonal (straight lines!) boundary approximation of Γ_S dictated by $\Psi(x_1)$.
2. The integrals of the bilinear form are compute approximately by quadrature.

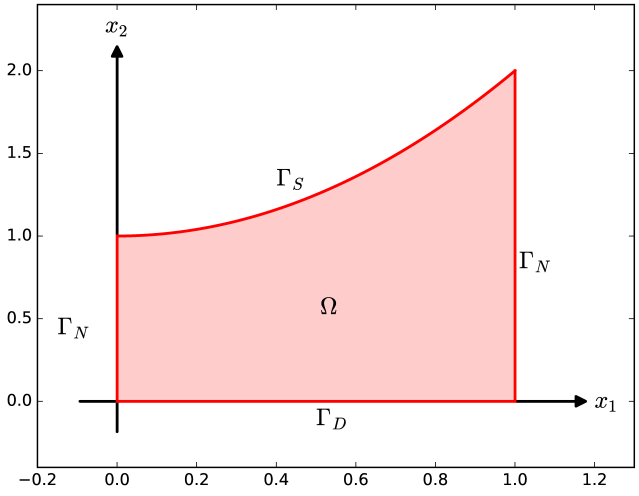
HINT 1 for (3-10.n): Recall the guidelines from [Lecture → Section 3.5] concerning "admissible variational crimes". ⌟

SOLUTION of (3-10.n):

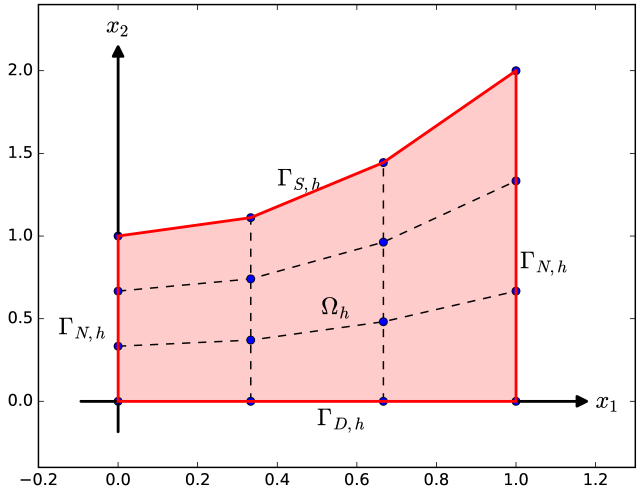
From [Lecture → Exp. 3.5.1.2] and [Lecture → Section 3.5.2] we know that the committed variational crimes are admissible if bilinear Lagrangian finite elements and the trapezoidal quadrature rule are used, i.e. they do not alter the (optimal) asymptotic behavior of $|u - u_h|_{H^1(\Omega)} = O(N^{-\frac{1}{2}})$.

SOLUTION of (3-10.b):

The domain Ω and its mesh $\mathcal{M}$ for $\Psi(x_1) = x_1^2 + 1$ are shown in **??**. Note that all the cells have straight edges, *cf.* [Lecture → Section 2.8.2], but they are no parallelograms.



Domain for 3-10.



Mesh for 3-10.

HINT 1 for (3-10.c): Recall the information from [Lecture → § 2.8.2.2] and, especially, [Lecture → Eq. (2.8.2.5)].

SOLUTION of (3-10.c):

Now, from the lecture [Lecture → § 2.8.2.2] we know that the following bilinear transformation does the job:

$$\boldsymbol{\Phi}_{j,l}(\hat{\boldsymbol{x}}) = (1-\hat{x}_1)(1-\hat{x}_2)\mathbf{a}^1 + \hat{x}_1(1-\hat{x}_2)\mathbf{a}^2 + \hat{x}_1\hat{x}_2\mathbf{a}^3 + (1-\hat{x}_1)\hat{x}_2\mathbf{a}_4 \,, \tag{3.10.9}$$

where $[\hat{x}_1, \hat{x}_2]^T \in \hat{K}$. More concrete formulas can be obtained by plugging in the coordinates given in (3.10.5), but keeping the formula (**??**) has advantages.

HINT 1 for (3-10.d): The inverse of a regular 2×2 matrix is simply

$$\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad \Rightarrow \quad \mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} . \tag{3.10.11}$$

⌟

SOLUTION of (3-10.d):

We have to evaluate the following expression

$$\mathbf{J}_{j,l}(\hat{\boldsymbol{x}}) = D\boldsymbol{\Phi}_{j,l}(\hat{\boldsymbol{x}}) = \begin{bmatrix} \frac{\partial \Phi^1_{j,l}}{\partial \hat{x}_1} & \frac{\partial \Phi^1_{j,l}}{\partial \hat{x}_2} \\ \frac{\partial \Phi^2_{j,l}}{\partial \hat{x}_1} & \frac{\partial \Phi^2_{j,l}}{\partial \hat{x}_2} \end{bmatrix} , \tag{3.10.12}$$

where $\Phi^{1,2}_{j,l}$ refers to the first and second component of the mapping $\boldsymbol{\Phi}_{j,l}$, respectively. Writing $\boldsymbol{a}^\ell = \begin{bmatrix} a^\ell_1 \\ a^\ell_2 \end{bmatrix}$, this can easily be evaluated to

$$\begin{aligned} \mathbf{J}_{j,l}(\hat{\boldsymbol{x}}) &= D\boldsymbol{\Phi}_{j,l}(\hat{\boldsymbol{x}}) \\ &= \begin{bmatrix} -(1-\hat{x}_2)a^1_1 + (1-\hat{x}_2)a^2_1 + \hat{x}_2 a^3_1 - \hat{x}_2 a^4_1 & -(1-\hat{x}_1)a^1_1 - \hat{x}_1 a^2_1 + \hat{x}_1 a^3_1 + (1-\hat{x}_1)a^4_1 \\ -(1-\hat{x}_2)a^1_2 + (1-\hat{x}_2)a^2_2 + \hat{x}_2 a^3_2 - \hat{x}_2 a^4_2 & -(1-\hat{x}_1)a^1_2 - \hat{x}_1 a^2_2 + \hat{x}_1 a^3_2 + (1-\hat{x}_1)a^4_2 \end{bmatrix} \\ &= \begin{bmatrix} (a^2_1 - a^1_1) + (a^1_1 - a^2_1 + a^3_1 - a^4_1)\hat{x}_2 & (a^4_1 - a^1_1) + (a^1_1 - a^2_1 + a^3_1 - a^4_1)\hat{x}_1 \\ (a^2_2 - a^1_2) + (a^1_2 - a^2_2 + a^3_2 - a^4_2)\hat{x}_2 & (a^4_2 - a^1_2) + (a^1_2 - a^2_2 + a^3_2 - a^4_2)\hat{x}_1 \end{bmatrix} \\ &= \begin{bmatrix} h & 0 \\ h[\Psi((j+1)h) - \Psi(jh)](l+\hat{x}_2) & h\Psi(jh) + h[\Psi((j+1)h) - \Psi(jh)]\hat{x}_1 \end{bmatrix} . \end{aligned}$$

Obviously

$$\det \mathbf{J}_{j,l}(\hat{\boldsymbol{x}}) = h^2(\Psi(jh) + [\Psi((j+1)h) - \Psi(jh)]\hat{x}_1) . \tag{3.10.13}$$

The inverse Jacobian transposed follows then directly from the formula (**??**) given as a hint.

$$\begin{aligned} \mathbf{J}^{-\top}_{j,l}(\hat{\boldsymbol{x}}) &= D\boldsymbol{\Phi}^{-\top}_{j,l}(\hat{\boldsymbol{x}}) \\ &= \frac{1}{h^2\Psi(jh) + h^2[\Psi((j+1)h) - \Psi(jh)]\hat{x}_1)} \\ &\quad \begin{bmatrix} h\Psi(jh) + h[\Psi((j+1)h) - \Psi(jh)]\hat{x}_1 & -h[\Psi((j+1)h) - \Psi(jh)](l+\hat{x}_2) \\ 0 & h \end{bmatrix} . \end{aligned} \tag{3.10.14}$$

Details can also be found in [Lecture → Ex. 2.8.3.17]. All the formulas may also be expressed in terms of the vertex position vectors $\mathbf{a}^i$ from (3.10.5).

SOLUTION of (3-10.e):

A solution is proposed in **??** and **??**. The implement the formulas (**??**) and (**??**).

C++ code 3.10.15: Sub-problem (3-10.d): Implementation of `integrationElement` ↪ GitLab

```
2  template <typename FUNCTOR>
3  double integrationElement(unsigned int n, unsigned int j, unsigned int l,
4                            FUNCTOR &&Psi, Eigen::Vector2d xhat) {
5    // Mesh width
6    double h = 1.0 / n;
7    double detJ = 0.0;
8
9    detJ = h * h * Psi(j * h) + h * h * (Psi((j + 1) * h) - Psi(j * h)) * xhat(0);
10   detJ = std::abs(detJ);
11
12   return detJ;
13 }
```

C++ code 3.10.16: Sub-problem (3-10.d): Implementation of `jacobianInverseTransposed` ↪ GitLab

```
2  template <typename FUNCTOR>
3  Eigen::Matrix2d jacobianInverseTransposed(unsigned int n, unsigned int j,
4                                            unsigned int l, FUNCTOR &&Psi,
5                                            Eigen::Vector2d xhat) {
6    // Mesh width
7    double h = 1.0 / n;
8    // Inverse Jacobian transposed
9    Eigen::Matrix2d invJT;
10
11   invJT(0, 0) = h * Psi(j * h) + h * (Psi((j + 1) * h) - Psi(j * h)) * xhat(0);
12   invJT(0, 1) = -h * (Psi((j + 1) * h) - Psi(j * h)) * (l + xhat(1));
13   invJT(1, 0) = 0.0;
14   invJT(1, 1) = h;
15
16   double detJ = integrationElement(n, j, l, Psi, xhat);
17   invJT /= detJ;
18
19   return invJT;
20 }
```

HINT 1 for (3-10.f): The formula [Lecture → Eq. (2.8.3.13)] could be useful. ⌟

SOLUTION of (3-10.f):

The proposed solution code displayed in **??** uses formula [Lecture → Eq. (2.8.3.13)] to compute the volume contributions to the element matrix.

The boundary contributions can be computed as explained in **??**, because the restrictions of bi-linear local shape functions to the edges of a cell are plain affine linear functions. Every edge carries two affine linear local shape functions.

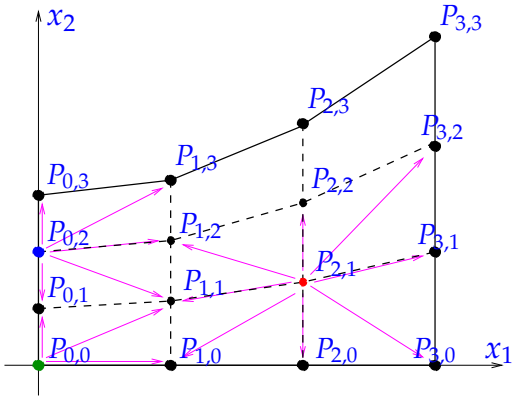
C++ code 3.10.17: Sub-problem (3-10.f): Implementation of `geoThermElemMat` ⇝ GitLab

```
2  template <typename FUNCTOR1, typename FUNCTOR2>
3  Eigen::MatrixXd geoThermElemMat(unsigned int n, unsigned int j, unsigned int l,
4                                  FUNCTOR1 &&alpha, FUNCTOR2 &&Psi) {
5    // Mesh width
6    double h = 1.0 / n;
7
8    // Quadrature nodes and weights on reference element
9    Eigen::MatrixXd xq(2, 4);
10   xq << 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0, 1.0;
11   Eigen::Vector4d wq;
12   wq << 0.25, 0.25, 0.25, 0.25;
13
14   // Volume contributions to element matrix A
15   Eigen::MatrixXd A = Eigen::MatrixXd::Zero(4, 4);
16
17   for (int iq = 0; iq < 4; iq++) {
18     // iq-th quadrature node from xq
19     Eigen::Vector2d xq_iq = xq.col(iq);
20     // Inverse Jacobian transposed at iq-th quadrature node
21     Eigen::Matrix2d invJT_iq = jacobianInverseTransposed(n, j, l, Psi, xq_iq);
22     // determinant at iq-th quadrature node
23     double detJ_iq = integrationElement(n, j, l, Psi, xq_iq);
24
25     // Reference element gradients of basis functions at iq-th node
26     Eigen::MatrixXd bhat_grad_iq(2, 4);
27     bhat_grad_iq = bhats_grad(xq_iq);
28
29     // Introduce g for a more compact notation, see lecture notes
       (3.7.36)
30     Eigen::MatrixXd g_iq(4, 2);
31     g_iq.row(0) = bhat_grad_iq.col(0).transpose() * invJT_iq.transpose();
32     g_iq.row(1) = bhat_grad_iq.col(1).transpose() * invJT_iq.transpose();
33     g_iq.row(2) = bhat_grad_iq.col(2).transpose() * invJT_iq.transpose();
34     g_iq.row(3) = bhat_grad_iq.col(3).transpose() * invJT_iq.transpose();
35
36     // Global coordinates of iq-th quadrature node for alpha
37     Eigen::Vector2d xq_gl;
38     xq_gl << (j + xq_iq(0)) * h, (l + xq_iq(1)) * h * Psi((j + xq_iq(0)) * h);
39
40     // Weighted contributions to element matrix
41     A += wq(iq) * alpha(xq_gl) * detJ_iq * g_iq * g_iq.transpose();
42   }
43
44   // For Neumann Boundary Conditions:  Surface Contributions
45   if (l == n - 1) {
```

```
46      double dl2 = h * h + (l + 1) * (l + 1) * h * h *
47                           (Psi((j + 1) * h) - Psi(j * h)) *
48                           (Psi((j + 1) * h) - Psi(j * h));
49      double dl = std::sqrt(dl2);
50
51      Eigen::Vector4d bhat2 = bhats(xq.col(2));
52      Eigen::Vector4d bhat3 = bhats(xq.col(3));
53
54      A(2, 2) += 0.5 * dl * (bhat2(2) * bhat2(2) + bhat3(2) * bhat3(2));
55      A(2, 3) += 0.5 * dl * (bhat2(2) * bhat2(3) + bhat3(2) * bhat3(3));
56      A(3, 2) += 0.5 * dl * (bhat2(3) * bhat2(2) + bhat3(3) * bhat3(2));
57      A(3, 3) += 0.5 * dl * (bhat2(3) * bhat2(3) + bhat3(3) * bhat3(3));
58    }
59
60    return A;
61  }
```

SOLUTION of (3-10.g):

Row and columns of the global Galerkin matrix are associated with nodes of the mesh. Since we cannot rule out that all entries of the element matrices are non-zero and none do cancel, we conclude that $\{\mathbf{A}\}_{i,k} \neq 0$ is possible only if, node i and node k are both vertices of quadrilateral.



◁ $n = 3$: Magenta arrows $\longrightarrow$ indicate, if two nodes of the mesh are "connected" in the sense that the entry of the global Galerkin matrix corresponding to the pair can be non-zero.

Hence, an interior node is connected to 9 other nodes, whereas a node next to a corner is connected to only three other nodes. ▷

Thus the number of non-zero entries of rows of $\mathbf{A}$ is between 3 and 9.

In detail, ignoring Dirichlet boundary conditions, $(\mathbf{A})_{i,k}$ will be zero, if

- node i is in the interior of Ω and

$$j \notin \{i-1, i, i+1, i-n-2, i-n-1, i-n, i+n, i+n+1, i+n+2\} ,$$

- node i is in the interior of an edge of $\partial\Omega$ and

$$j \notin \{i \pm 1, i, i-n-1 \pm 1, i+n+1 \pm 1, i-n-1, i+n+1\} ,$$

  where $\pm$ is $+$ for the left edge, and $-$ for the right edge,

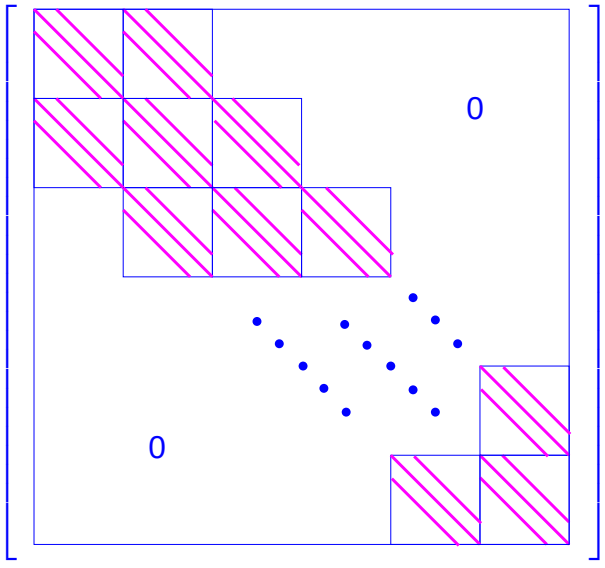
- node i coincides with a corner of Ω, and

$$j \notin \{i, i+\sigma_1, i+\sigma_2(n+1), i+\sigma_2(n+1)+\sigma_1\} , \quad \sigma_1, \sigma_2 \in \{-1, 1\} ,$$

  where $\sigma_1 = +1$ for corners on the left edge, and $\sigma_2 = +1$ for corners on the bottom edge.

Before the elimination of degrees of freedom corresponding to nodes on the Dirichlet boundary Γ_D the $(n+1)^2 \times (n+1)^2$-Galerkin matrix has the following structure:

$\mathbf{A} =$



◁ Structure of global Galerkin matrix $\mathbf{A}$.

The structure also reveals that

- rows of the matrix related to truly interior nodes, the red node ● in **??**, feature 9 non-zero entries
- rows of the matrix belonging to corner nodes, the green node ● in **??**, have 3 non-zero entries
- rows of the matrix for nodes at the boundary, the blue node ● in **??**, have 6 non-zero entries.

SOLUTION of (3-10.h):

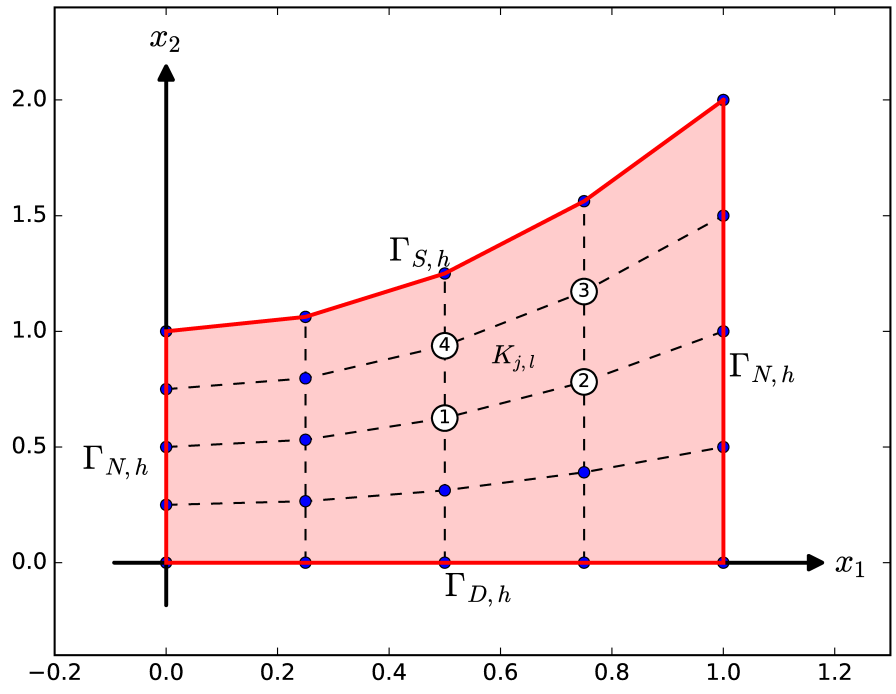
Assuming the ordering of Sub-problem (3-10.g), the four shape functions associated with the corners of $K_{j,l}$ (see **??**) map to the following global index numbers:

$$
\begin{aligned}
1 &\to j + (n+1)l \\
2 &\to (j+1) + (n+1)l \\
3 &\to (j+1) + (n+1)(l+1) \\
4 &\to j + (n+1)(l+1).
\end{aligned}
$$

A proposed solution code for the function `geoThermLocalToGlobal` is shown in **??**.

C++ code 3.10.19: Sub-problem (3-10.h): Implementation of `geoThermLocalToGlobal`
➥ GitLab

```
int geoThermLocalToGlobal(unsigned int n, unsigned int j, unsigned int l,
                          unsigned int local_dof) {
  // Map local indices of basis functions to global indices
  int global_dof;

  switch (local_dof) {
    case 0:
      global_dof = j + (n + 1) * l;
      break;

    case 1:
      global_dof = j + 1 + (n + 1) * l;
      break;

    case 2:
      global_dof = j + 1 + (n + 1) * (l + 1);
      break;

    case 3:
      global_dof = j + (n + 1) * (l + 1);
      break;

    default:
      global_dof = 66;
      break;
  }

  return global_dof;
}
```



◁ Local local shape functions associated with the corners of $K_{j,l}$.

SOLUTION of (3-10.i):

A proposed solution code for the function `assemGeoThermMat` is shown in **??**. If closely follows the assembly policy developed in [Lecture → Section 2.7.4.2].

C++ code 3.10.20: Sub-problem (3-10.i): Implementation of `assemGeoThermMat` ↪ GitLab

```
2  template <typename FUNCTOR1, typename FUNCTOR2>
3  std::vector<Eigen::Triplet<double>> assembleGeoTherm(unsigned int n,
4                                                       FUNCTOR1 &&alpha,
5                                                       FUNCTOR2 &&Psi) {
6    // Reserve triplets for Galerkin Matrix A
7    std::vector<Eigen::Triplet<double>> triplets;
8    triplets.reserve(4 * 4 * n * n);
9
10   // Loop over elements
11   for (int j = 0; j < n; j++) {
12     for (int l = 0; l < n; l++) {
13       // Obtain element matrix
14       Eigen::MatrixXd A_jl = geoThermElemMat(n, j, l, alpha, Psi);
15
16       // Assemble global matrix
17       for (int iloc = 0; iloc < 4; iloc++) {
18         for (int jloc = 0; jloc < 4; jloc++) {
19           // Glonal indices from local dofs
20           int iglo = geoThermLocalToGlobal(n, j, l, iloc);
21           int jglo = geoThermLocalToGlobal(n, j, l, jloc);
22
23           triplets.push_back(Eigen::Triplet(iglo, jglo, A_jl(iloc, jloc)));
24         }
25       }
26     }
27   }
28
29   return triplets;
30 }
```

HINT 1 for (3-11.a): Implement `PSL()` first, then copy your code for `PSL()` to `PDL()` and modify it appropriately. ⌟

SOLUTION of (3-11.a):

See **??** and **??**.

C++ code 3.11.6: Sub-problem (3-11.a): Implementation of `PSL()`

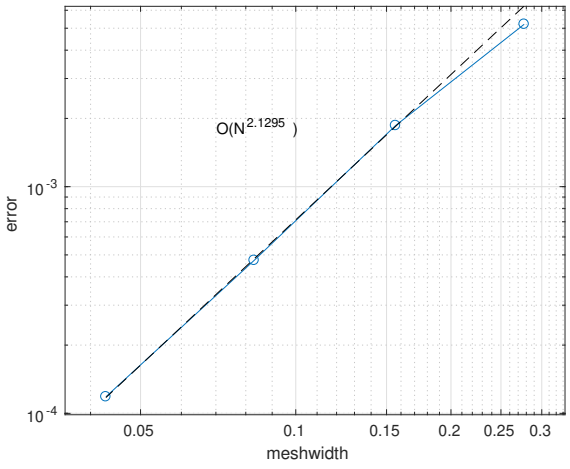
C++ code 3.11.7: Sub-problem (3-11.a): Implementation of `PDL()`

HINT 1 for (3-11.b): For the implementation use the supplied lambda functions `u()` and `gradu()` to evaluate respectively u and $\mathbf{grad}\, u$ in the point $\boldsymbol{x}$. ⌟

SOLUTION of (3-11.b):

See **??** and **??**.

C++ code 3.11.9: Sub-problem (3-11.c): Implementation of `pointEval()`



◁ Convergence of deviations for formula (3.11.8).

That (3.11.8) does not hold exactly is due to quadrature errors. This accounts for the observed algebraic convergence with rate 2 as mesh width $h \to 0$.

SOLUTION of (3-11.c):

Since $\Psi_{\boldsymbol{x}}$ is zero in a ball around $\boldsymbol{x}$, where $G_{\boldsymbol{x}}$ has the singularity, the product $G_{\boldsymbol{x}}\Psi_{\boldsymbol{x}}$ is smooth. Thus, we can apply Green's formula

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,.$$

with $\mathbf{j} := \mathbf{grad}\, u$, $v := G_{\boldsymbol{x}}\Psi_{\boldsymbol{x}}$, and get:

$$\begin{aligned} P_{\mathrm{SL}}(\mathbf{grad}\, u \cdot \boldsymbol{n}) &= \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{y}) G_{\boldsymbol{x}}(\boldsymbol{y}) \underbrace{\Psi_{\boldsymbol{x}}(\boldsymbol{y})}_{=1} \cdot \boldsymbol{n}\, \mathrm{d}S(\boldsymbol{y}) \\ &= \int_\Omega \mathbf{grad}\, u(\boldsymbol{y}) \cdot \mathbf{grad}(G_x\Psi_x)(\boldsymbol{y}) + \underbrace{\operatorname{div} \mathbf{grad}\, u(\boldsymbol{y})}_{=0} (G_x\Psi_x)(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y}\,, \end{aligned} \tag{3.11.11}$$

because $\Delta u = 0$.

HINT 1 for (3-11.d): Recall the formula $\Delta(vw) = v\Delta w + 2\,\mathbf{grad}\, w \cdot \mathbf{grad}\, v + w\Delta v$ and use that $\Delta G_{\boldsymbol{x}}(\boldsymbol{y}) = 0$ for $\boldsymbol{y} \neq \boldsymbol{x}$. ⌟

SOLUTION of (3-11.d):

Remembering that $\operatorname{div}\mathbf{grad} = \Delta$, we get by applying Green's formula *twice*:

$$\begin{aligned} P_{\mathrm{SL}}(\mathbf{grad}\, u \cdot \boldsymbol{n}) &= \int_{\partial\Omega} (\mathbf{grad}\, u(\boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y}))\, (G_{\boldsymbol{x}}\Psi_{\boldsymbol{x}})(\boldsymbol{y})\, \mathrm{d}S(\boldsymbol{y}) \\ &= \int_{\Omega} \mathbf{grad}\, u(\boldsymbol{y}) \cdot \mathbf{grad}\, u \cdot \mathbf{grad}(G_{\boldsymbol{x}}\Psi_{\boldsymbol{x}})(\boldsymbol{y})\, \mathrm{d}\boldsymbol{y} \\ &= -\int_{\Omega} u(\boldsymbol{y})\Delta(G_{\boldsymbol{x}}\Psi_{\boldsymbol{x}})(\boldsymbol{y})\, d\boldsymbol{y} + \int_{\partial\Omega} u(\boldsymbol{y})\, \mathbf{grad}(G_{\boldsymbol{x}}\Psi_{\boldsymbol{x}})(\boldsymbol{y}) \cdot \boldsymbol{n}\, dS = \\ &= -\int_{\Omega} u(\boldsymbol{y})(\mathbf{grad}\, G_{\boldsymbol{x}}(\boldsymbol{y}) \cdot \mathbf{grad}\, \Psi_{\boldsymbol{x}}(\boldsymbol{y}) + G_{\boldsymbol{x}}(\boldsymbol{y})\Delta\Psi_{\boldsymbol{x}}(\boldsymbol{y}))\, d\boldsymbol{y} + \\ &\quad \int_{\partial\Omega} u(\boldsymbol{y})\, \mathbf{grad}\, G_{\boldsymbol{x}}(\boldsymbol{y})\Psi_{\boldsymbol{x}}(\boldsymbol{y}) \cdot \boldsymbol{n}\, dS + \int_{\partial\Omega} u(\boldsymbol{y})G_{\boldsymbol{x}}(\boldsymbol{y}) \underbrace{\mathbf{grad}\, \Psi_{\boldsymbol{x}}(\boldsymbol{y})}_{=0}\, dS \end{aligned} \tag{3.11.13}$$

where for the last step we applied the formula given in the Hint and used that $\Delta G_{\boldsymbol{x}}(\boldsymbol{y}) = 0$ for $\boldsymbol{y} \neq \boldsymbol{x}$.

Now, because of the assumption (**P2**) on $\Psi_{\boldsymbol{x}}$, the last integral $\int_{\partial\Omega}$ in the last equality vanishes and we get (3.11.12).

SOLUTION of (3-11.e):

We observe that

$$\Phi(y) := 2\,\mathbf{grad}\, G_x(y) \cdot \mathbf{grad}\, \Psi_x(y) + G_x(y)\Delta\Psi_x(y)$$

is continuous on $\bar{\Omega}$, since $\Psi_x, \mathbf{grad}\, \Psi_x \equiv 0$ in the vicinity of x by (**P3**). As a consequence Φ is bounded. Therefore, $\Phi \in L^2(\Omega)$ and we can apply Cauchy-Schwarz inequality [Lecture $\rightarrow$ Eq. (1.3.4.15)]:

$$|J^*(u)| \leq \|u\|_{L^2(\Omega)} \|\Phi\|_{L^2(\Omega)} \,. \tag{3.11.15}$$

In $H^1_0(\Omega)$, the Poincaré inequality [Lecture $\rightarrow$ Thm. 1.3.4.17] holds, i.e. $\|u\|_{L^2(\Omega)} \leq C_P |u|_{H^1(\Omega)}$, giving the continuity of J^* with respect to the energy norm, which agrees with the $|u|_{H^1(\Omega)}$-seminorm for the Laplacian.

$$|J^*(u)| \leq C_P |u|_{H^1(\Omega)} \|\Phi\|_{L^2(\Omega)} = \bar{C} |u|_{H^1(\Omega)} \quad \forall u \in H^1_0(\Omega) \,, \tag{3.11.16}$$

for some constant $\bar{C}$.

HINT 1 for (3-11.f): For the implementation you can use the two already supplied functions `function Gval = G(x,y)` and `function Ggradval = Ggrad(x,y)` that return $G_x(y)$ and $\mathbf{grad}\, G_x(y)$ respectively.

⌟

SOLUTION of (3-11.f):

See **??**.

C++ code 3.11.18: Sub-problem (3-11.f): Implementation of `Jstar`

SOLUTION of (3-11.g):

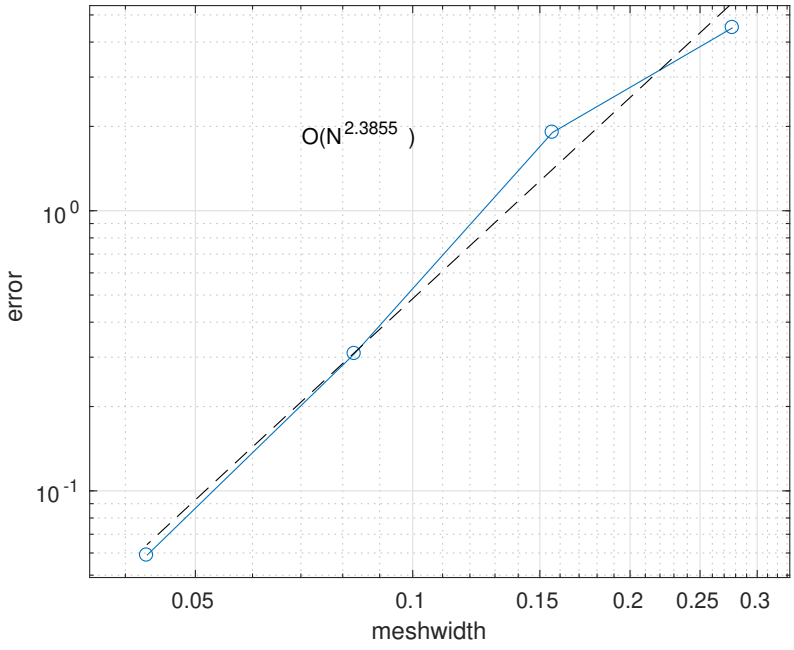
See **??**.

C++ code 3.11.19: Sub-problem (3-11.g): Implementation of `Solve_LaplDir_LFE()`

SOLUTION of (3-11.h):

See **??** and **??**. The observed convergence rate is 1.8013 with respect to h.

C++ code 3.11.20: Sub-problem (3-11.h): Convergence test of `stab_point_eval`



◁ Convergence plot for Sub-prob

HINT 1 for (3-12.a): Note that for constant Dirichlet data the gradient of the potential is orthogonal to the boundary, because the gradient cannot have any tangential component. ⌟

SOLUTION of (3-12.a):

Using the information form the hint, it is clear that on the boundary the gradient is parallel to the unit normal vector field $\boldsymbol{n}$:

$$\mathbf{grad}\, u(\boldsymbol{x}) = \big(\mathbf{grad}\, u(\boldsymbol{x}\cdot\boldsymbol{n}(\boldsymbol{x})\big)\, \boldsymbol{n}(\boldsymbol{x})\,, \quad \boldsymbol{x}\in\partial\Omega\,.$$

HINT 1 for (3-12.j): Examine [Lecture → § 3.6.2.4]. Try $u(\boldsymbol{x}) = \left(\|\boldsymbol{x}\| - \frac{4}{15}\right)^{\alpha}$ for some suitably chosen α. ⌟

SOLUTION of (3-12.j):

The gradient of the function

$$u(\boldsymbol{x}) := \left(\|\boldsymbol{x}\| - \frac{4}{15} \right)^{\alpha}, \quad \|\boldsymbol{x}\| > \frac{4}{15},$$

is

$$\mathbf{grad}\, u(\boldsymbol{x}) = \alpha \left(r - \frac{4}{15} \right)^{\alpha-1} \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|}, \quad r := \|\boldsymbol{x}\|,$$

where r is the distance to the origin. The integral is over the inner boundary integrates, we have $r = 4/15$ there, and so the gradient will be infinite for $\alpha < 1$.

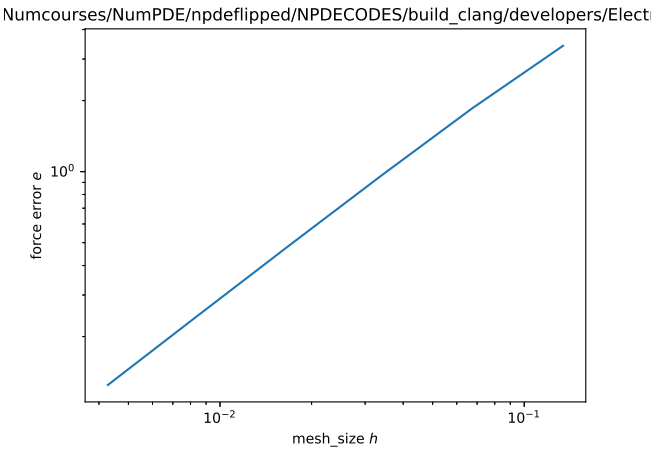
The function u is in $H^1(\Omega)$ if $\|u\|_{H^1(\Omega)} < \infty$, in particular

$$|u|^2_{H^1(\Omega)} = \int_0^{2\pi} \int_{4/15}^{r^{\text{out}}(\phi)} \mathbf{grad}\, u(r) \cdot \mathbf{grad}\, u(r) r \, dr d\phi = \alpha^2 \int_0^{2\pi} \int_{4/15}^{r^{\text{out}}(\phi)} \left(r - \frac{4}{15} \right)^{2\alpha-1} dr d\phi,$$

which fulfills $|u|^2_{H^1(\Omega)} < \infty$ for $2\alpha - 1 \geq -1$, i.e., $\alpha \geq 0$. Thus we have a counter-example for $0 \leq \alpha < 1$.

SOLUTION of (3-12.k):

For details of how to determine kind and rate of convergence refer to [Lecture → § 3.2.2.5].



h	$\|\boldsymbol{F}(u_h) - \boldsymbol{F}(u)\|$	est. rate
0.13410	3.41384	
0.06745	1.85473	0.88786
0.03401	0.96490	0.95422
0.01707	0.49185	0.97791
0.00855	0.24828	0.98916
0.00428	0.12472	0.99463

◁ Doubly logarithmic plot of h vs. $\|\boldsymbol{F}(u_h) - \boldsymbol{F}(u)\|$

From both the error table and the doubly logarithmic plot we read off algebraic convergence with rate 1.

HINT 1 for (3-12.l): Please recall the following result of elliptic regularity theory:

Theorem [Lecture → Thm. 3.4.0.2]. Smooth elliptic lifting theorem

If $\partial\Omega$ is C^∞-smooth, ie. possesses a local parameterization by C^∞-functions, and $\sigma \in C^\infty(\overline{\Omega})$, then, for any $k \in \mathbb{N}$,

$$\left.\begin{array}{l} u \in H^1_0(\Omega) \quad \text{and} \quad -\operatorname{div}(\sigma \operatorname{\mathbf{grad}} u) \in H^k(\Omega) \\ u \in H^1(\Omega)\,,\ -\operatorname{div}(\sigma \operatorname{\mathbf{grad}} u) \in H^k(\Omega) \text{ and } \operatorname{\mathbf{grad}} u \cdot \mathbf{n} = 0 \text{ on } \partial\Omega \end{array}\right\} \quad \Rightarrow \quad u \in H^{k+2}(\Omega)\,.$$

In addition, for such u there is $C = C(k,\Omega,\sigma)$ such that

$$\|u\|_{H^{k+2}(\Omega)} \leq C \|\operatorname{div}(\sigma \operatorname{\mathbf{grad}} u)\|_{H^k(\Omega)}\,.$$

SOLUTION of (3-12.l):

These functions are not solutions to (3.12.1). Solutions u should enjoy regularity of at least $u \in H^2(\Omega)$, which is sufficient to ensure that the gradient belongs to $H^1(\Omega)$ and the force functional is finite.

HINT 1 for (3-12.m): If u is constant on Γ_1, $\mathbf{grad}\, u$ will be parallell to $\boldsymbol{n}$ there, see also Sub-problem (3-12.

SOLUTION of (3-12.m):

Assume that $\mathbf{grad}\, u = c\boldsymbol{n}$ for some c. Then, on one hand, we have $\mathbf{grad}\, u \cdot \boldsymbol{n} = c$, and

$$\frac{1}{2}(\mathbf{grad}\, u \cdot \boldsymbol{n})\, \mathbf{grad}\, u = \frac{1}{2}c^2\boldsymbol{n}\ .$$

On the other hand, we have $\| \mathbf{grad}\, u\|^2 = c^2$, so we have

$$\boldsymbol{T}(u) \cdot \boldsymbol{n} = c^2\boldsymbol{n} \cdot \boldsymbol{n}^T \cdot \boldsymbol{n} - \frac{1}{2}c^2\boldsymbol{n} = \frac{1}{2}c^2\boldsymbol{n}\ ,$$

because $\boldsymbol{n}^T \cdot \boldsymbol{n} = \|\boldsymbol{n}\|^2 = 1$.

SOLUTION of (3-12.n):

The row i of $\boldsymbol{T}$ is

$$\boldsymbol{T}_i = \frac{\partial u}{\partial x_i} \operatorname{\mathbf{grad}} u^T - \frac{1}{2}\Big[\|\operatorname{\mathbf{grad}} u\|^2\Big]_i \,,$$

where $[s]_i$ means a vector with zeros, except for the value s in position i. Thus,

$$\operatorname{div} \boldsymbol{T}_i = \frac{\partial u}{\partial x_i}\big(\operatorname{div}\operatorname{\mathbf{grad}} u\big)^\top + \frac{\partial^2 u}{\partial x_1 \partial x_i}\frac{\partial u}{\partial x_1} + \frac{\partial^2 u}{\partial x_2 \partial x_i}\frac{\partial u}{\partial x_2} - \frac{1}{2}\frac{\partial}{\partial x_i}\|\operatorname{\mathbf{grad}} u\|^2 \,.$$

The first term will vanish due to (3.12.1). The rest will cancel out after computing the last term.

HINT 1 for (3-12.o): First, show that

$$\boldsymbol{F}(u) = \int_{\partial\Omega} \Psi(\boldsymbol{x})\boldsymbol{T}(u)(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \, \mathrm{d}S$$

and then use Gauss' theorem (the divergence theorem, [Lecture → Thm. 1.5.2.4]). Be inspired by the derivation of [Lecture → Eq. (3.6.2.6)]. ⌟

SOLUTION of (3-12.o):

We start with the identity from Sub-problem (3-12.m), then use $\Psi|_{\Gamma_1} = 1$ followed by an application of the divergence theorem. Finally we use the product rule and the result of Sub-problem (3-12.n)/

$$\begin{aligned}
\boldsymbol{F}(u) &= \frac{1}{2}\int_{\Gamma_1} (\mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}))\, \mathbf{grad}\, u(\boldsymbol{x})\, \mathrm{d}S \\
&= \int_{\Gamma_1} \boldsymbol{T}(u)(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S = \int_{\partial\Omega} (\Psi(\boldsymbol{x})\boldsymbol{T}(u)(\boldsymbol{x})) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S \\
&= \int_{\Omega} \mathrm{div}(\Psi \boldsymbol{T})\, \mathrm{d}\boldsymbol{x} \\
&= \int_{\Omega} (\boldsymbol{T}(u)(\boldsymbol{x}) \cdot \mathbf{grad}\, \Psi(\boldsymbol{x}) + \Psi(\boldsymbol{x})\, \mathrm{div}\, \boldsymbol{T}(u)(\boldsymbol{x}))\, \mathrm{d}\boldsymbol{x} \\
&= \int_{\Omega} \boldsymbol{T}(u)(\boldsymbol{x})\, \mathbf{grad}\, \Psi(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \tilde{\boldsymbol{F}}(u).
\end{aligned}$$

SOLUTION of (3-12.p):

Exploit that the cut-off function is supposed to be smooth up to the boudnary:

$$
\begin{aligned}
|\tilde{\boldsymbol{F}}(v)| &= \left| \int_\Omega \left(\mathbf{grad}\, v \cdot \mathbf{grad}\, v^T - \frac{1}{2} \| \mathbf{grad}\, v \|^2 \boldsymbol{I} \right) \mathbf{grad}\, \Psi \, \mathrm{d}\boldsymbol{x} \right| \\
&\leq \| \mathbf{grad}\, \Psi \|_{L^\infty} \int_\Omega \frac{3}{2} \| \mathbf{grad}\, v \|^2 \, \mathrm{d}\boldsymbol{x} \leq C(\Psi) |v|_{H^1}^2 \,,
\end{aligned}
$$

where $C(\Psi) = \frac{3}{2} \| \mathbf{grad}\, \Psi \|_{L^\infty}$.

HINT 1 for (3-12.q): Use $\Psi = u^{\mathrm{ex}}$ (the exact solution from (**??**)) and the three-point local quadrature rule that relies on the midpoints of the edges of a triangle. ⌟

SOLUTION of (3-12.q):

The implementation relies on the following steps inside a loop over all cells:

1. Compute the gradient of the finite element solution, which is constant on efvery cell K.
2. Based on $\mathbf{grad}\, u_h|_K$ build the Maxwell stress tensor as defined in (3.12.15).
3. Compute $\mathbf{grad}\, u^{\mathrm{ex}}$ at the midpoints of the edges of the current cell K. To evaluate $\mathbf{grad}\, u^{\mathrm{ex}}(x_k)$, note that

$$\mathbf{grad} \log \|\boldsymbol{x}\| = \frac{1}{\|\boldsymbol{x}\|} \mathbf{grad}\, \|\boldsymbol{x}\| = \frac{\boldsymbol{x}}{\|\boldsymbol{x}\|^2} .$$

   We use a lambda function to implement that computation.

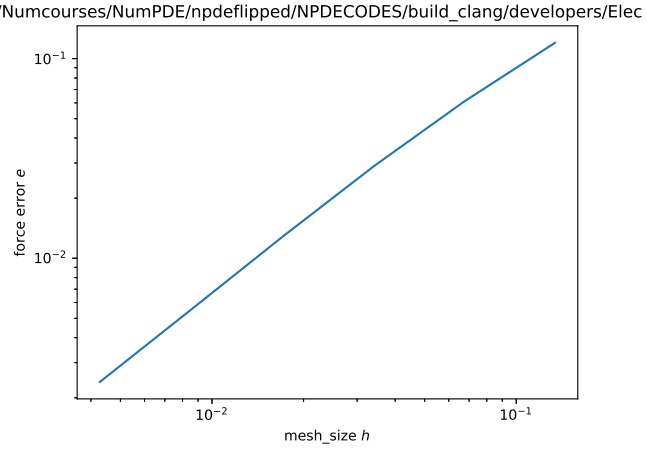
C++ code 3.12.19: Sub-problem (3-12.q): Evaluate the force functional $\tilde{F}(u)$
➙ GitLab

```
Eigen::Vector2d computeForceDomainFunctional(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
    Eigen::VectorXd approx_sol) {
  Eigen::Vector2d approx_force;  // to return

  // Pointer to current mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};

  Eigen::Vector2d a(-16.0 / 15.0, 0.0);
  Eigen::Vector2d b(-1.0 / 15.0, 0.0);
  auto grad_uExact = [&a, &b](Eigen::VectorXd x) -> Eigen::Vector2d {
    return ((x - a) / (x - a).squaredNorm() - (x - b) / (x - b).squaredNorm()) /
           std::log(2.0);
  };

  // INTEGRATION TOOLS
  Eigen::MatrixXd stress_tensor(2, 2);      // maxwell stress tensor
  Eigen::MatrixXd mid_pts(2, 3);            // midpoints of the edges
  Eigen::Matrix<double, 2, 3> elgrad_mat;   // gradients of basis elements
  Eigen::Vector2d loc_grad_approx_sol;  // gradient expansion coeff FE solution
  Eigen::MatrixXd Id = Eigen::Matrix<double, 2, 2>::Identity();

  // PERFORMING INTEGRATION
  approx_force.setZero();
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    auto endpoints = lf::geometry::Corners(*(cell->Geometry()));
    mid_pts.col(0) = 0.5 * (endpoints.col(0) + endpoints.col(1));
    mid_pts.col(1) = 0.5 * (endpoints.col(1) + endpoints.col(2));
    mid_pts.col(2) = 0.5 * (endpoints.col(2) + endpoints.col(1));

    double area = lf::geometry::Volume(*(cell->Geometry()));

    // II COMPUTING GRADIENT OF THE FE APPROXIMATE SOLUTION
    // II.i Obtain global FE indices of the vertices
    const auto dof_idx_vec = dofh.GlobalDofIndices(*cell);
    // II.ii Obtain the gradients of the barycentric coordinate functions
```

```
40      elgrad_mat = gradbarycoordinates(*cell);
41      // II.iii Compute the gradient of the passed coefficient vector
42      loc_grad_approx_sol = elgrad_mat.col(0) * approx_sol(dof_idx_vec[0]) +
43                            elgrad_mat.col(1) * approx_sol(dof_idx_vec[1]) +
44                            elgrad_mat.col(2) * approx_sol(dof_idx_vec[2]);
45
46      // III ASSEMBLING MAXWELL STRESS TENSOR
47      stress_tensor = loc_grad_approx_sol * loc_grad_approx_sol.transpose() -
48                      0.5 * loc_grad_approx_sol.squaredNorm() * Id;
49
50      // IV SUMMING LOCAL CONTRIBUTION
51      approx_force += (area / 3.0) * stress_tensor *
52                      (grad_uExact(mid_pts.col(0)) + grad_uExact(mid_pts.col(1)) +
53                       grad_uExact(mid_pts.col(2)));
54    }
55    return approx_force;
56  } // computeForceDomainFunctional
```

SOLUTION of (3-12.r):

Consult [Lecture → § 3.2.2.5] for information about how to tease out information about asymptotic convergence from data.



h	$\left\|\widetilde{\boldsymbol{F}}(u_h) - \boldsymbol{F}(u)\right\|$	est. rate
0.13410	0.12001	
0.06745	0.06097	0.98541
0.03401	0.02887	1.09153
0.01707	0.01283	1.17703
0.00855	0.00554	1.21494
0.00428	0.00240	1.20847

◁ Doubly logarithmic plot of h vs. $\left\|\widetilde{\boldsymbol{F}}(u_h) - \boldsymbol{F}(u)\right\|$.

We observe algebraic convergence with a better rate of ~ 1.2 compared to that for the boundary-based force functional, see Sub-problem (3-12.k). Also note that the actual error for $\widetilde{\boldsymbol{F}}$ is considerably smaller than for $\boldsymbol{F}$, which reflects far smaller constants in the asymptotic error law.

SOLUTION of (3-12.b):

In mathematical notation the function stored in `uExact` is

$$u(\boldsymbol{x}) = \frac{1}{\log 2}(\log \|\boldsymbol{x} - \boldsymbol{a}\| - \log \|\boldsymbol{x} - \boldsymbol{b}\|) - 1 \,, \quad \boldsymbol{x} \in \mathbb{R}^2 \setminus \{\boldsymbol{a}, \boldsymbol{b}\} \,, \tag{3.12.6}$$

where $\boldsymbol{a} = (-16/15, 0)$ and $\boldsymbol{b} = (-1/15, 0)$ are the positions of the point charges.

A function on the form $\log \|\boldsymbol{x} - \boldsymbol{a}\|$ can be written as $\log r$ for a suitable choice of polar coordinates (with origin in $\boldsymbol{a}$). The Laplacian in polar coordinates is

$$\Delta u = \frac{1}{r}\frac{\partial}{\partial r}(r\frac{\partial u}{\partial r}) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \varphi^2} \,. \qquad \text{[Lecture} \rightarrow \text{Eq. (3.4.0.5)]}$$

Hence, straightforward calculus shows that $r \mapsto \log r$ is a harmonic function in $\mathbb{R}^2 \setminus \{\mathbf{0}\}$. As a consequence, the function from (**??**) is also harmonic: $\Delta u = 0$. Then nsert $\boldsymbol{x} \in \partial\Omega$ to verify the boundary conditions.

HINT 1 for (3-12.c): Exact computation is tedious. Therefore, you may use "overkill" numerical quadrature (e.g. based on the trapezoidal rule) to obtain a good approximation for the exact value of the force functional. To this end, you may either complete the function `computeExactForce()` supplied in file `electrostaticforce.cc` or write your own PYTHON function serving that purpose (in which case one can simply rely on `scipy.integrate.quad`). ⌟

HINT 2 for (3-12.c): You should obtain the force value $\boldsymbol{F} = [13.0776, 0]^\top$. ⌟

SOLUTION of (3-12.c):

Thanks to symmetry the x_2-compoent of the force must vanish and we need only compute the x_1-component.

We parameterize the inner circle Γ_1:

$$\boldsymbol{x}(\tau) = \boldsymbol{a} + \begin{bmatrix} r_1 \cos\tau \\ r_1 \sin\tau \end{bmatrix}, \quad \tau \in [0, 2\pi[\quad \blacktriangleright \quad \boldsymbol{n}(\tau) = \begin{bmatrix} \cos\tau \\ \sin\tau \end{bmatrix}. \tag{3.12.7}$$

The gradient of the exact solution u from (**??**) is

$$\mathbf{grad}\, u(\boldsymbol{x}) = \frac{1}{\log 2}\left(\frac{\boldsymbol{x}-\boldsymbol{a}}{\|\boldsymbol{x}-\boldsymbol{a}\|^2} - \frac{\boldsymbol{x}-\boldsymbol{b}}{\|\boldsymbol{x}-\boldsymbol{b}\|^2} \right), \boldsymbol{x} \in \Omega\,.$$

Then by (3.12.4) we have to evaluate

$$F_1 := r_i \int_0^{2\pi} |\,\mathbf{grad}\, u(\boldsymbol{x}(\tau)) \cdot \boldsymbol{n}(\tau)|^2 \cos(\tau)\, \mathrm{d}\tau\,.$$

You may write a short PYTHON or C++ program for evaluating this integral approximately, using, for instance, the equidistant trapezoidal rule, which will converge exponentially in this case. See **??** for an example.

The resulting force acts, unsurprisingly, only in the x-direction, and has a value there of 13.0776.

C++ code 3.12.8: Sub-problem (3-12.c): Implementation of `computeExactForce()`
➥ GitLab

```
2  Eigen::Vector2d computeExactForce() {
3    Eigen::Vector2d force;  // return vector
4    // Real overkill quadrature!
5    unsigned int N = 1e6;  // nb. of quadrature points
6    // Data
7    double r = 4.0 / 15.0;  // radius of the circle
8    Eigen::Vector2d a(-16.0 / 15.0, 0.0);
9    Eigen::Vector2d b(-1.0 / 15.0, 0.0);
10   // Auxiliary variables
11   Eigen::Vector2d x;     // euclidean coordinates
12   Eigen::Vector2d n;     // normal vector at the coordinates
13   Eigen::Vector2d grad;  // gradient of the exact solution
14   double angle;          // interior angle
15
16   // Compute the force using the trapezoidal rule
17   force.setZero();  // initially zero
18   for (unsigned int i = 0; i < N; i++) {
19     // I. Find the euclidean coordinates of integration node
20     angle = i * (2 * M_PI) / N;
21     x(0) = r * std::cos(angle);
22     x(1) = r * std::sin(angle);
23     // II. Evaluate the gradient at the coordinates
24     grad = ((x - a) / (x - a).squaredNorm() - (x - b) / (x - b).squaredNorm()) /
25            std::log(2.0);
26     // III. Evaluate the normal vector at the coordinates
27     n = -x / x.norm();
```

```
28      // IV. Evaluate full integrand
29      force += grad.dot(n) * grad;
30    }
31    // V. Scale the result of the summation
32    force *= r * M_PI / N;
33    return force;
34  }  // end computeExactForce
```

HINT 1 for (3-12.d): If the norm of the point is not larger than $4/15$, the point belongs to an edge on the inner boundary. ⌟

SOLUTION of (3-12.d):

See **??**.

C++ code 3.12.9: Sub-problem (3-12.d): Implementation of the lambda function ⇢ GitLab

```
2   auto mf_bd_values =
3       lf::mesh::utils::MeshFunctionGlobal([](Eigen::Vector2d x) -> double {
4         if (x.norm() < 0.27) {
5           return 1.0;
6         }
7         return 0.0;
8       });
```

HINT 1 for (3-12.e): The solution of a pure Dirichlet problem for $-\Delta$ is explained in [Lecture → Ex. 2.7.6.19]. You can rely on **lf::uscalfe::LinearFELaplaceElementMatrix** and `InitEssentialCondi` already available in the LEHRFEM++ library. ⌟

SOLUTION of (3-12.e):

See **??**. Refer to the LEHRFEM++ documentation for detailed explanations.

C++ code 3.12.10: Sub-problem (3-12.e): Assembling left- and right-hand side of FE linear system

```
  // I : ASSEMBLY
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Right hand side vector, must be initialized with 0!
  Eigen::Matrix<double, Eigen::Dynamic, 1> phi(N_dofs);
  phi.setZero();
  // Computing volume matrix for negative Laplace operator
  // Initialize object taking care of local mass (volume) computations.
  lf::uscalfe::LinearFELaplaceElementMatrix elmat_builder{};
  // Invoke assembly on cells (co-dimension = 0 as first argument)
  // Information about the mesh and the local-to-global map is passed through
  // a Dofhandler object, argument 'dofh'. This function call adds triplets to
  // the internal COO-format representation of the sparse matrix A.
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);

  // II. IMPOSING ESSENTIAL BOUNDARY CONDITIONS

  // Obtain arrays of boolean flags for the edges and nodes of the mesh, 'true'
  // indicates that the edge or node lies on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
  // Determine the fixed dofs on the boundary and their values
  auto flag_values{lf::fe::InitEssentialConditionFromFunction(
      *fe_space_p, bd_flags, mf_bd_values)};
  // II.ii Eliminate Dirichlet dofs from the linear system
  lf::assemble::FixFlaggedSolutionCompAlt<double>(
      [&flag_values](lf::assemble::glb_idx_t dof_idx) {
        return flag_values[dof_idx];
      },
      A, phi);
```

SOLUTION of (3-12.f):

We simply loop over all edges (co-dimension 1 entities) and fetch their lengths.

C++ code 3.12.11: Sub-problem (3-12.f): Computing the mesh width ↪ GitLab

```
2  double getMeshSize(const std::shared_ptr<const lf::mesh::Mesh> &mesh_p) {
3    double mesh_size = 0.0;
4
5    // Find maximal edge length
6    double edge_length;
7    // Loop over all edges of the mesh
8    for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
9      // Compute the length of the edge
10     auto endpoints = lf::geometry::Corners(*(edge->Geometry()));
11     edge_length = (endpoints.col(0) - endpoints.col(1)).norm();
12     if (mesh_size < edge_length) {
13       mesh_size = edge_length;
14     }
15   }
16   return mesh_size;
17 };  // getMeshSize
```

HINT 1 for (3-12.g): Look at [Lecture → Ex. 3.2.3.4] to learn how to compute error norms using the tools provided by the LEHRFEM++ library. ⌟

HINT 2 for (3-12.g): Types of convergence are defined in [Lecture → Def. 3.2.2.1]. The computation of convergence rates is explained in [Lecture → § 3.2.2.5]. ⌟

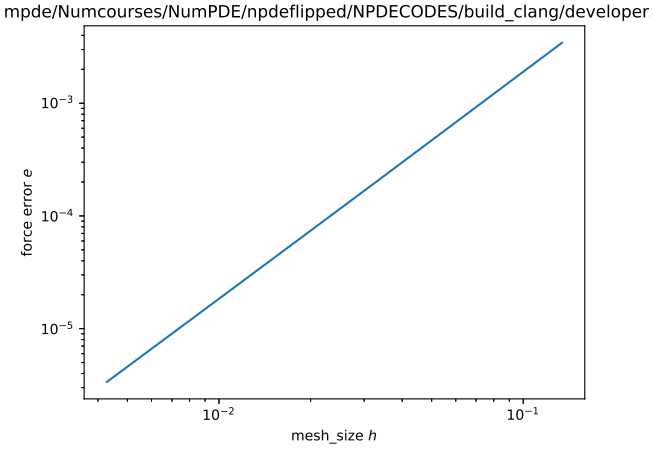
SOLUTION of (3-12.g):

Using **MeshFunction** the computation of the $L^2(\Omega)$-norm of the error can be done in two lines.

C++ code 3.12.12: Sub-problem (3-12.g): Computing the $L^2(\Omega)$-norm of the finite-element discretization error ➙ GitLab

```
2   // COMPUTE L2 ERROR FOR THE POISSON DIRICHLET PROBLEM
3   auto mf_approx_sol = lf::fe::MeshFunctionFE(fe_space_p, approx_sol);
4   errorsL2PoissonBVP[i] = std::sqrt(lf::fe::IntegrateMeshFunction(
5       *mesh_p, lf::uscalfe::squaredNorm(mf_uExact - mf_approx_sol), 2));
```

There are other ways to perform this computation, see [Lecture → Ex. 3.2.3.4] and, in particular, [Lecture → Code 3.2.3.5].



h	$\|u - u_h\|_{L^2(\Omega)}$	estimated rate
0.13410	3.44140e-03	
0.06745	8.58576e-04	2.02044
0.03401	2.15133e-04	2.02098
0.01707	5.38544e-05	2.00992
0.00855	1.34705e-05	2.00509
0.00428	3.36823e-06	2.00265

The convergence is algebraic, and the reported rate is close to 2, in terms of the meshwidth. We expect this from the results of [Lecture → Section 3.6.3], because the boundary of Ω is smooth such that we can reap the benefits of full elliptic regularity.

SOLUTION of (3-12.h):

It is enough to observe that $\boldsymbol{F}(2u) = 4\boldsymbol{F}(u)$. If $\boldsymbol{F}$ was linear, we would have $\boldsymbol{F}(2u) = 2\boldsymbol{F}(u)$.

HINT 1 for (3-12.i): This subproblem requires you to compute normals of edges on the (inner) boundary. You should take extra care that they point outwards with respect to the cell.

The integrals can be evaluated exactly, so there is no need for quadrature. ⌟

SOLUTION of (3-12.i):

To evaluate the formula

$$F(u_h) = \frac{1}{2}\int_{\Gamma_1} (\mathbf{grad}\, u_h(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}))\, \mathbf{grad}\, u_h(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})\,, \tag{3.12.13}$$

we note that gradients of piecewise linear functions are constant vectors on each part $\partial K \cap \Gamma_1$ of a mesh cell K (in the code we evaluate $\mathbf{grad}\, u_h(\boldsymbol{x})$ at the arbitrary point $\boldsymbol{x} = \mathbf{0}$). For a triangle K the normal $\boldsymbol{n}$ is also constant on $\partial K \cap \Gamma_1$ and the whole integrand of (**??**) is independent of $\boldsymbol{x}$. This is implemented in **??**. Note that we do transform the gradient from the reference triangle $\hat{K}$ to the real triangle K.

C++ code 3.12.14: Sub-problem (3-12.i): Evaluate the force functional $F(u_h)$ ↪ GitLab

```
Eigen::Vector2d computeForceBoundaryFunctional(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
    Eigen::VectorXd approx_sol) {
  Eigen::Vector2d approx_force;  // to return

  // MESH AND FINITE ELEMENTS SPACE DATA
  // Pointer to current mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
  // Obtain arrays of boolean flags for the edges and nodes of the mesh, 'true'
  // indicates that the edge or node lies on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
  // This predicate returns 'true' if the edge belongs to the interior boundary
  auto interior_bd_flags = [&bd_flags](const lf::mesh::Entity &edge) -> bool {
    if (bd_flags(edge)) {
      auto endpoints = lf::geometry::Corners(*(edge.Geometry()));
      if (endpoints.col(0).norm() < 0.27) {
        return true;
      }
    }
    return false;
  };

  // INTEGRATION TOOLS
  double edge_length;
  Eigen::VectorXd normal_vec;
  Eigen::Matrix2d rotation_mat;
  rotation_mat << 0, 1, -1, 0;              // rotates a 2d vec by 90 deg.
  Eigen::Matrix<double, 2, 3> elgrad_mat;  // gradients of basis elements
  Eigen::Vector2d loc_grad_approx_sol;  // gradient expansion coeff FE solution

  // PERFORMING INTEGRATION
  approx_force.setZero();
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    for (const lf::mesh::Entity *edge : cell->SubEntities(1)) {
      if (interior_bd_flags(*edge)) {
        auto endpoints = lf::geometry::Corners(*(edge->Geometry()));
        edge_length = (endpoints.col(1) - endpoints.col(0)).norm();

        // I COMPUTING NORMAL VECTOR
        // I.i Finding a unit vector perpendicular to the edge
        normal_vec =
```

```
45              rotation_mat * (endpoints.col(1) - endpoints.col(0)).normalized();
46          // I.ii The normal vector must be pointing outward
47          if (normal_vec.dot(endpoints.col(0)) > 0) {
48            normal_vec *= -1.0;
49          }
50
51          // II COMPUTING GRADIENT OF THE FE APPROXIMATE SOLUTION
52          // II.i Obtain global FE indices of the vertices
53          const auto dof_idx_vec = dofh.GlobalDofIndices(*cell);
54          // II.ii Obtain the gradients of the barycentric coordinate functions
55          elgrad_mat = gradbarycoordinates(*cell);
56          // II.iii Compute the gradient of the passed coefficient vector
57          loc_grad_approx_sol = elgrad_mat.col(0) * approx_sol(dof_idx_vec[0]) +
58                                elgrad_mat.col(1) * approx_sol(dof_idx_vec[1]) +
59                                elgrad_mat.col(2) * approx_sol(dof_idx_vec[2]);
60
61          // III SUMMING LOCAL CONTRIBUTION
62          approx_force += edge_length * loc_grad_approx_sol.dot(normal_vec) *
63                          loc_grad_approx_sol;
64        }
65      }
66    }
67    approx_force *= 0.5;
68    return approx_force;
69  }  // computeForceFunctional
```

SOLUTION of (3-13.a):

We apply the arguments of [Lecture → Section 1.5.3] and Green's first formula [Lecture → Thm. 1.5.2.7] to the left-hand-side bilinear form a of (3.13.1). To do so we have to assume $u \in C^2_{\mathrm{pw}}(\overline{\Omega})$, *cf.* [Lecture → Ass. 1.5.3.1]:

$$\int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \\ -\int_\Omega \mathrm{div}(\sigma\,\mathbf{grad}\,u)\,v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S = 0 \quad \forall v \in V_0\,, \tag{3.13.3}$$

where $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ is the exterior unit normal vector field.

First, we test with $v \in C_0^\infty(\Omega)$ (smooth, compactly supported in Ω) and recover the PDE

$$-\mathrm{div}(\sigma\,\mathbf{grad}\,u) = 0 \quad \text{in} \quad \Omega\,. \tag{3.13.4}$$

Next, we test with $v \in V_0$ and use the PDE (**??**), which yields

$$\int_{\partial\Omega}\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in V_0\,.$$

Since v can attain arbitrary values on Γ_{-1}, we conclude homogeneous *Neumann boundary conditions* on the insulated boundary part:

$$\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \text{on} \quad \Gamma_{-1}\,. \tag{3.13.5}$$

Dirichlet boundary conditions on the contacts are directly specified in (3.13.1):

$$u = 0 \quad \text{on} \quad \Gamma_0 \quad , \quad u = 1 \quad \text{on} \quad \Gamma_1\,.$$

HINT 1 for (3-13.j): The following auxiliary function is provided and can be used:

```
Eigen::Matrix<double, 2, 3>
  GradsBaryCoords(Eigen::Matrix<double, 2, 3> vertices);
```

It takes the vertex coordinates of a triangle as the columns of a 2×3-matrix and returns the (constant) gradients of the barycentric coordinate functions in the columns of another 2×3-matrix. This function can be called like

```
auto grad_bary_coords =
   GradsBaryCoords(
     lf::geometry::Corners(*(cell->Geometry())));
```

where `cell` is a pointer to a mesh entity of type `TRIA`. ⌟

SOLUTION of (3-13.j):

The standard approach involves a loop over all (triangular) cells of the mesh adding up the contributions from every cell to the integral. Since we use a one-point quadrature rule, those cell contributions boil down to

$$s_K := |K|\, \sigma(\boldsymbol{c}_K)\, \mathbf{grad}\, u_h|_K \cdot \mathbf{grad}\, \Psi(\boldsymbol{c}_K)\,, \quad K \in \mathcal{M}\,, \quad \boldsymbol{c}_K \triangleq \text{center of gravity of } K.$$

Here we have used that $\mathbf{grad}\, u_h$ is $\mathcal{M}$-piecewise constant.

C++ code 3.13.12: Implementation of `stabFlux()` ⇝ GitLab

```
template <typename SIGMAFUNCTION, typename PSIGRAD>
double stabFlux(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    const Eigen::VectorXd &sol_vec, SIGMAFUNCTION &&sigma, PSIGRAD &&gradpsi) {
  // Underlying FE mesh
  const lf::mesh::Mesh &mesh{*(fe_space->Mesh())};
  // Local-to-Global map for local/global shape function indices
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  // Reference coordinates of "midpoint" of a triangle
  const Eigen::MatrixXd zeta_ref{
      (Eigen::Matrix<double, 2, 1>() << 1.0 / 3.0, 1.0 / 3.0).finished()};
  // Summation variable
  double s = 0.0;
  // Loop over all cells
  for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Not implemented for " << *cell);
    // Obtain geometry information for entity
    const lf::geometry::Geometry &geo{*cell->Geometry()};
    // Compute the gradients of the barycentric coordinate functions
    const Eigen::Matrix<double, 2, 3> grad_bary_coords{
        GradsBaryCoords(lf::geometry::Corners(geo))};
    // Compute the area of the triangle
    const double area = lf::geometry::Volume(geo);
    // DofHandler must provide three degrees of freedom per cell for piecewise
    // linear Lagrangian finite elements on triangles
    LF_ASSERT_MSG(dofh.NumLocalDofs(*cell) == 3,
                  "contactFlux: 3 dofs per triangle mandatory!");
    // Fetch indices of global shape functions associated with triangle
    const auto glob_dof_idx{dofh.GlobalDofIndices(*cell)};
    // Compute (constant) local gradient of the finite element solution
    const Eigen::Vector2d local_gradient{
        grad_bary_coords * (Eigen::Vector3d() << sol_vec[glob_dof_idx[0]],
                            sol_vec[glob_dof_idx[1]], sol_vec[glob_dof_idx[2]])
                               .finished()};
    // Find physical/world coordinates of "midpoint" of triangle
    const Eigen::MatrixXd zeta{geo.Global(zeta_ref)};
    // Add contribution of triangle
    const auto quadnode{zeta.col(0)};
    s += area * (gradpsi(quadnode)).dot(sigma(quadnode) * local_gradient);
  }
  return s;
}
```

A very elegant implementation can use LEHRFEM++'s **MeshFunction** utility classes:

C++ code 3.13.13: MeshFunction-based implementation of `stabFlux()` ⇝ GitLab

```
template <typename SIGMAFUNCTION, typename PSIGRAD>
double stabFluxMF(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    const Eigen::VectorXd &sol_vec, SIGMAFUNCTION &&sigma, PSIGRAD &&gradpsi) {
  std::shared_ptr<const lf::mesh::Mesh> mesh_p{fe_space->Mesh()};
  // Coefficient function and weight function
  const lf::mesh::utils::MeshFunctionGlobal mf_sigma(sigma);
  const lf::mesh::utils::MeshFunctionGlobal mf_gradpsi(gradpsi);
  // Build a MeshFunction representing the gradient of the finite element
  // solution
  const lf::fe::MeshFunctionGradFE mf_grad(fe_space, sol_vec);
  // Mesh function representing the integrand
  const auto mf_itg{lf::mesh::utils::transpose(mf_sigma * mf_grad) *
                    mf_gradpsi};
  const double s = lf::fe::IntegrateMeshFunction(
      *mesh_p, mf_itg, [](const lf::mesh::Entity &e) {
        return lf::quad::make_QuadRule(e.RefEl(), 2);
      })(0, 0);
  return s;
}  // end stabFluxMF
```

Of course also the facilitites of **lf::fe::ScalarReferenceFiniteElement** can be used:

C++ code 3.13.14: lf::fe::ScalarReferenceFiniteElement-based implementation of `stabFlux()` ⇝ GitLab

```
template <typename SIGMAFUNCTION, typename PSIGRAD>
double stabFluxTRF(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    const Eigen::VectorXd &sol_vec, SIGMAFUNCTION &&sigma, PSIGRAD &&gradpsi) {
  // Underlying FE mesh
  const lf::mesh::Mesh &mesh{*(fe_space->Mesh())};
  // Local-to-Global map for local/global shape function indices
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  // Reference coordinates of "midpoint" of a triangle (center of gravity)
  const Eigen::MatrixXd zeta_ref{
      (Eigen::Matrix<double, 2, 1>() << 1.0 / 3.0, 1.0 / 3.0).finished()};
  // Obtain gradients of reference shape functions at center of gravity
  const lf::fe::ScalarReferenceFiniteElement<double> &ref_lsf{
      *fe_space->ShapeFunctionLayout(lf::base::RefEl::kTria())};
  LF_ASSERT_MSG(
      ref_lsf.NumRefShapeFunctions() == 3,
      "There must be 3 reference shape functions associated with vertices");
  const Eigen::MatrixXd ref_lsf_grads{
      ref_lsf.GradientsReferenceShapeFunctions(zeta_ref)};
  LF_ASSERT_MSG(ref_lsf_grads.rows() == 3, "Three gradients required!");
  LF_ASSERT_MSG(ref_lsf_grads.cols() == 2, "Two components expected!");
  // Summation variable
  double s = 0.0;
  // Loop over all cells
  for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Not implemented for " << *cell);
```

```
29      // Obtain geometry information for entity
30      const lf::geometry::Geometry &geo{*cell->Geometry()};
31      // Fetch the transformation matrix for gradients
32      const Eigen::MatrixXd JinvT{geo.JacobianInverseGramian(zeta_ref)};
33      LF_ASSERT_MSG(
34          (JinvT.rows() == 2) && (JinvT.cols() == 2),
35          "JinvT is " << JinvT.rows() << " x " << JinvT.cols() << "-matrix!");
36      // Compute the gradients of the local shape functions by transformation
37      const auto grad_lsf{JinvT * ref_lsf_grads.transpose()};
38      // Compute the area of the triangle
39      const double area = lf::geometry::Volume(geo);
40      // DofHandler must provide three degrees of freedom per cell for piecewise
41      // linear Lagrangian finite elements on triangles
42      LF_ASSERT_MSG(dofh.NumLocalDofs(*cell) == 3,
43                    "contactFlux: 3 dofs per triangle mandatory!");
44      // Fetch indices of global shape functions associated with triangle
45      const auto glob_dof_idx{dofh.GlobalDofIndices(*cell)};
46      // Compute (constant) local gradient of the finite element solution
47      const Eigen::Vector2d local_gradient{
48          grad_lsf * (Eigen::Vector3d() << sol_vec[glob_dof_idx[0]],
49                      sol_vec[glob_dof_idx[1]], sol_vec[glob_dof_idx[2]])
50                         .finished()};
51      // Find physical/world coordinates of "midpoint" of triangle
52      const Eigen::MatrixXd zeta{geo.Global(zeta_ref)};
53      // Add contribution of triangle
54      const auto quadnode{zeta.col(0)};
55      s += area * (gradpsi(quadnode)).dot(sigma(quadnode) * local_gradient);
56    }
57    return s;
58  }
59  /* SAM_LISTING_END_G */
60
61  }  // namespace dmxbc
62
63  #endif
```

SOLUTION of (3-13.k):

- $I(u_\ell)$ in column (I),
- $I^*(u_\ell)$ in column (III),
- $I(u_\ell) - I(u)$ in column (II),
- $I^*(u_\ell) - I(u)$ in column (IV)

because in contrast to $u \mapsto I(u)$ the functional $u \mapsto I^*(u)$ is continuous with respect to the energy norm underlying (3.13.1), which is equivalent to the $H^1(\Omega)$-norm. This was established in Sub-problem (3-13.i). Therefore, the duality estimate of [Lecture $\rightarrow$ Thm. 3.6.1.7] applies and will yield faster convergence compared to the unbounded functional $u \mapsto I(u)$.

For both functionals the data are compatible with *algebraic convergence* in the meshwidth and we conjecture

$$I(u_\ell) - I(u) = O(h_\ell) \quad , \quad I^*(u_\ell) - I(u) = O(h_\ell^2) \quad \text{for} \quad h_\ell \to 0 \, .$$

SOLUTION of (3-13.l):

The problem statement tells us that $\boldsymbol{\sigma}(\boldsymbol{x}) = \sigma(\boldsymbol{x})\mathbf{I}_2$ with $\sigma \in C^1(\overline{\Omega})$. In this concrete case the dual solution satisfies the variational problem

$$g_F \in V_0: \quad \int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\, g_F \cdot \mathbf{grad}\, v\,\mathrm{d}\boldsymbol{x} = \int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\, v \cdot \mathbf{grad}\,\Psi\,\mathrm{d}\boldsymbol{x} \quad \forall v \in V_0\ .$$

In order to tease out the strong form, we apply the first Green formula to the right-hand-side functional and get

$$g_F \in V_0: \quad \int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\, g_F \cdot \mathbf{grad}\, v\,\mathrm{d}\boldsymbol{x} = -\int_\Omega v\ \mathrm{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,\Psi)\,\mathrm{d}\boldsymbol{x} \quad \forall v \in V_0\ .$$

The function Ψ defined by the above lambda function is

$$\Psi(\boldsymbol{x}) = \begin{cases} 1 & \text{for } x_1 \geq 3\ , \\ \cos^2((3-x_1)\pi/2) & \text{for } 2 < x_1 < 3\ , \\ 0 & \text{for} x_1 \leq 2\ . \end{cases}$$

This is just a function of x_1!

Note that *no boundary terms occur*, because, thanks to the special choice of Ψ

$$\mathbf{grad}\,\Psi(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} \in \partial\Omega\ .$$

Thus, as in [Lecture → Section 1.5.3] we deduce the following boundary value problem with mixed Dirichlet and (homogeneous) Neumann boundary conditions:

$$\mathrm{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\, g_F) = \mathrm{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,\Psi) \quad \text{in} \quad \Omega\ ,$$

$$g_F = 0 \quad \text{on} \quad \Gamma_0 \cup \Gamma_1\ ,$$

$$\sigma(\boldsymbol{x})\,\mathbf{grad}\, g_F \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \Gamma_{-1}\ .$$

$\mathbf{grad}\,\Psi$ could be evaluated explicitly, but this is not required.

SOLUTION of (3-13.b):

This sub-problem addresses the **offset function trick** discussed in [Lecture → § 1.4.1.9]:

- u_0 must be a function $\in H^1(\Omega)$ satisfying the Dirichlet boundary conditions: $u_0|_{\Gamma_0} = 0$, $u_0|_{\Gamma_1} = 1$. A simple choice is the continuous piecewise linear function

$$u_0(\boldsymbol{x}) = \begin{cases} 0 & \text{for} \quad x_1 < 3\,, \\ x_1 - 3 & \text{for} \quad x_1 > 3\,, \end{cases} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{3.13.6}$$

  However, there is an infinite stock of admissible functions u_0.

- Once we have chosen u_0, the right-hand-side functional is

$$\ell(v) := -\mathsf{a}(u_0, v) = -\int_\Omega \sigma(\boldsymbol{x})\, \mathbf{grad}\, u_0 \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}\,, \quad v \in V_0 \,.$$

  For the special choice of u_0 from (**??**) this means

$$\ell(v) = -\int_{\Omega \cap \{x_1 > 3\}} \sigma(\boldsymbol{x}) \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}\,.$$

SOLUTION of (3-13.c):

See [Lecture → Section 3.5.2]. Since the mesh contains only triangular cells with straight edges, the circular parts of the boundary cannot be represented exactly. Hence the domain of integration in the evaluation of the bilinear form is changed to a polygonal approximation of the computational domain Ω. This results in a perturbation of the bilinear form.

Remark. From [Lecture → Section 3.5.2] we know that this is an admissible variational crime, since we use merely piecewise linear Lagrangian finite elements.

SOLUTION of (3-13.d):

According to the definition of V_0 from (3.13.2) functions in $\mathcal{S}^0_{1,C}(\mathcal{M}) := \mathcal{S}^0_1(\mathcal{M}) \cap V_0$ have to vanish on both Γ_0 and Γ_1. We know that the dimension of $\mathcal{S}^0_1(\mathcal{M})$ agrees with the number of nodes of the mesh, because every node has exactly one tent basis functions associated with it.

Demanding that the finite-element functions vanish on $\Gamma_0 \cup \Gamma_1$ amounts to dropping all tent functions belonging to nodes $\in \overline{\Gamma}_0 \cup \overline{\Gamma}_1$. Inspecting the mesh plot we see that there are six such nodes. Therefore

$$\dim \mathcal{S}^0_{1,C}(\mathcal{M}) = \dim \mathcal{S}^0_1(\mathcal{M}) - 6 = 55 - 6 = 49 \ .$$

HINT 1 for (3-13.e): You can rely on the LEHRFEM++ utility class **lf::uscalfe::ReactionDiffusionElemer** and the convenience functions `lf::assemble::FixFlaggedSolutionComponents()` or `lf::a`

SOLUTION of (3-13.e):

The main challenge is to enforce the Dirichlet boundary conditions following the approach explained in [Lecture → § 2.7.6.13]. This entails the construction of an appropriate **SELECTOR** object based on the information in `nodeflags`.

Note that the following code is more general than asked for, because it is designed to deal with an arbitrary number of contacts.

C++ code 3.13.8: Implementation of `solveMixedBVP()` ⇝ GitLab

```
template <typename SIGMAFUNCTOR>
Eigen::VectorXd solveMixedBVP(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    lf::mesh::utils::CodimMeshDataSet<int> &nodeflags,
    std::vector<double> &&voltvals, SIGMAFUNCTOR &&sigma) {
  // Mesh functions for coefficients
  lf::mesh::utils::MeshFunctionGlobal mf_sigma{sigma};
  lf::mesh::utils::MeshFunctionConstant<double> mf_gamma{0.0};
  // Reference to current mesh
  const lf::mesh::Mesh &mesh{*(fe_space->Mesh())};
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  // Dimension of finite element space = number of nodes of the mesh
  const lf::base::size_type N_dofs(dofh.NumDofs());
  LF_ASSERT_MSG(N_dofs == mesh.NumEntities(2),
                " N_dofs must agree with number of nodes");
  std::cout << "Solving BVP with " << N_dofs << " FE dofs" << std::endl;
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Right-hand side vector
  Eigen::VectorXd phi(N_dofs);
  // Assembly of Galerkin matrix and right-hand side vector plus
  // special treatment of dofs at contacts
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<
      double, decltype(mf_sigma), decltype(mf_gamma)>
      elmat_builder(fe_space, mf_sigma, mf_gamma);
  // Cell-oriented assembly
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
  // Right-hand side vector is zero
  phi.setZero();
  // Number of Dirichlet boundary parts
  const int NContacts = voltvals.size();
  // Selector functor for Dirichlet data
  auto selector = [&](lf::assemble::gdof_idx_t idx) -> std::pair<bool, double> {
    const lf::mesh::Entity &node{dofh.Entity(idx)};
    const int ids = nodeflags(node);
    if ((ids >= 0) && (ids < NContacts)) return {true, voltvals[ids]};
    return {false, 42.0};
  };
  // Eliminate Dirichlet dofs from linear system
  lf::assemble::FixFlaggedSolutionComponents<double>(selector, A, phi);
  // Assembly completed: Convert COO matrix A into CRS format using Eigen's
  // internal conversion routines.
  const Eigen::SparseMatrix<double> A_crs = A.makeSparse();

```

```
47    // Solve linear system using Eigen's sparse direct elimination
48    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
49    solver.compute(A_crs);  // LU decomposition
50    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
51    Eigen::VectorXd sol_vec = solver.solve(phi);
52    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
53    return sol_vec;
54  }  // end solveMixedBVP
```

SOLUTION of (3-13.f):

By (**??**), (**??**), and Gauss' divergence theorem:

$$\begin{aligned} 0 = \int_\Omega \operatorname{div}(\sigma \,\mathbf{grad}\, u)\,\mathrm{d}x &= \int_{\partial\Omega} \sigma(x)\,\mathbf{grad}\, u(x)\cdot \boldsymbol{n}(x)\,\mathrm{d}S(x) \\ &= \int_{\Gamma_0\cup\Gamma_1} \sigma(x)\,\mathbf{grad}\, u(x)\cdot \boldsymbol{n}(x)\,\mathrm{d}S(x) \\ &= \int_{\Gamma_0} \sigma(x)\,\mathbf{grad}\, u(x)\cdot \boldsymbol{n}(x)\,\mathrm{d}S(x) + \int_{\Gamma_1} \sigma(x)\,\mathbf{grad}\, u(x)\cdot \boldsymbol{n}(x)\,\mathrm{d}S(x)\,, \end{aligned}$$

because the Neumann boundary part is not crossed by any flux.

SOLUTION of (3-13.g):

It is important to realize that the computation of $\mathbf{grad}\, u_h \cdot \boldsymbol{n}|_e$ on an edge $e \subset \partial\Omega$ has to access nodal values/degrees of freedom of u_h *off that edge*. Therefore the computation has to rely on looping over cells.

The following implementation makes use of LEHRFEM++'s **MeshFunction** facilities instead of invoking `getTriangleGradLambdaNormals()`.

C++ code 3.13.10: Alternative Implementation of `contactFlux()` ⇝ GitLab

```
2  template <typename SIGMAFUNCTION>
3  double contactFluxMF(
4      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
5      const Eigen::VectorXd &sol_vec, SIGMAFUNCTION &&sigma,
6      const lf::mesh::utils::CodimMeshDataSet<int> &edgeids, int contact_id = 0) {
7    // The underlying finite element mesh
8    const lf::mesh::Mesh &mesh{*(fe_space->Mesh())};
9    // Variable for summing boundary flux
10   double s = 0.0;
11   // Counter for edges on selected contact
12   unsigned int ed_cnt = 0;
13   // Compute exterior edge-weighted normals
14   lf::mesh::utils::CodimMeshDataSet<Eigen::Vector2d> normals{
15       exteriorEdgeWeightedNormals(fe_space->Mesh())};
16   // Build a MeshFunction representing the gradient of the finite element
17   // solution
18   const lf::fe::MeshFunctionGradFE mf_grad(fe_space, sol_vec);
19   // Reference coordinates of edge midpoints of a triangle
20   const Eigen::MatrixXd mp_refc{
21       (Eigen::Matrix<double, 2, 3>() << 0.5, 0.5, 0.0, 0.0, 0.5, 0.5)
22           .finished()};
23   // Loop over all cells
24   for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
25     const lf::base::RefEl ref_el_type{cell->RefEl()};
26     LF_ASSERT_MSG(ref_el_type == lf::base::RefEl::kTria(),
27                   "contactFlux: implemented for triangles only");
28     // Obtain array of edge pointers (sub-entities of co-dimension 1)
29     std::span<const lf::mesh::Entity *const> sub_ent_range{
30         cell->SubEntities(1)};
31     // Must be three edges
32     LF_ASSERT_MSG(sub_ent_range.size() == 3, "Triangle must have three edges!");
33     // Obtain gradient values at midpoints of edges
34     const std::vector<Eigen::VectorXd> grad_at_mp{mf_grad(*cell, mp_refc)};
35     // Visit edges, check flags, and add contribution to flux integral
36     for (lf::base::sub_idx_t j = 0; j < ref_el_type.NumSubEntities(1); ++j) {
37       const lf::mesh::Entity &edge{*sub_ent_range[j]};
38       LF_ASSERT_MSG(edge.RefEl() == lf::base::RefEl::kSegment(),
39                     "Not an edge!");
40       if (edgeids(edge) == contact_id) {
41         // Exterior normal vector
42         LF_ASSERT_MSG(normals.DefinedOn(edge),
43                       "Normal vector not available for " << edge);
44         const Eigen::Vector2d n{normals(edge)};
45         LF_ASSERT_MSG(n.norm() > 0, " zero normal vector ?");
46         const auto sigma_val{sigma((cell->Geometry())->Global(mp_refc.col(j)))};
47         s += n.dot(sigma_val * grad_at_mp[j]);
```

```
48            ed_cnt++;
49          }
50        }  // end loop over edges
51      }    // end loop over cells
52      std::cout << "Summed flux for " << ed_cnt << " edges." << std::endl;
53      return s;
54  }  // end contactFluxMF
```

If you use `getTriangleGradLambdaNormals()`, the **MeshFunction** object is required.

SOLUTION of (3-13.h):

The reasoning is borrowed from [Lecture → Section 3.6.2]: We apply Green's first formula [Lecture → Thm. 1.5.2.7] to $I^*(u)$ and obtain

$$\begin{aligned} I^*(u) &= \int_\Omega \underbrace{-\operatorname{div}(\sigma(x)\,\mathbf{grad}\,u(x))}_{=0}\,\Psi(x)\,\mathrm{d}x + \int_{\partial\Omega} \underbrace{\sigma(x)\,\mathbf{grad}\,u(x)\cdot n(x)}_{=0 \text{ on } \Gamma_{-1}}\,\Psi(x)\,\mathrm{d}S(x) \\ &= \int\limits_{\Gamma_0\cup\Gamma_1} \sigma(x)\,\mathbf{grad}\,u(x)\cdot n(x)\,\Psi(x)\,\mathrm{d}S(x)\ . \end{aligned}$$

Recalling (3.13.9) we recover $I(u)$, if

$$\Psi|_{\Gamma_0} = 0 \quad \text{and} \quad \Psi|_{\Gamma_1} = 1\ .$$

SOLUTION of (3-13.i):

By [Lecture → Def. 1.2.3.42] we have to show that there is a constant $C > 0$ such that

$$I^*(u) \leq C\|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega) .$$

Since $\boldsymbol{\sigma} \in (C^0(\overline{\Omega}))^{2,2}$, this matrix-valued function is certainly bounded:

$$\max_{\boldsymbol{x}\in\Omega} \|\boldsymbol{\sigma}(\boldsymbol{x})\|_2 < \infty .$$

Thus we can pull the maximum of the matrix norm of $\boldsymbol{\sigma}$ out of the integral. Then we apply the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)]:

$$\begin{aligned} I^*(u) &\leq \max_{\boldsymbol{x}\in\Omega} \|\boldsymbol{\sigma}(\boldsymbol{x})\|_2 \int_\Omega \|\mathbf{grad}\, u\|_2 \, \|\mathbf{grad}\, \Psi\|_2 \, \mathrm{d}\boldsymbol{x} \\ &\leq \underbrace{\max_{\boldsymbol{x}\in\Omega} \|\boldsymbol{\sigma}(\boldsymbol{x})\|_2 \|\mathbf{grad}\, \Psi\|_{L^2(\Omega)}}_{=:C} \|\mathbf{grad}\, u\|_{L^2(\Omega)} \quad \forall u \in H^1(\Omega) . \end{aligned}$$

HINT 1 for (3-14.a): You can write $\widehat{\psi}^{\ell}$ as a linear combination of barycentric coordinate functions on $\widehat{K}$ and then use the following lemma.

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1! \alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, . \tag{3.14.2}$$

SOLUTION of (3-14.a):

We first pursue considerations valid for a general triangle K. We write

$$\psi_K^\ell = \sum_{k=1}^3 \xi_{k,l}\lambda_k \,.$$

From the condition (3.14.1) we obtain the linear system of equations

$$\int_K \psi_K^\ell(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \sum_{k=1}^3 \xi_{k,l}\int_K \lambda_k(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \delta_{\ell,j} := \begin{cases} 1 & \text{, if } \ell = j\,, \\ 0 & \text{, if } \ell \neq j\,, \end{cases} \qquad \ell,j \in \{1,2,3\}\,. \quad (3.14.3)$$

By means of [Lecture → Lemma 2.7.5.5] we compute

$$\left[\int_K \lambda_k(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{k,j=1}^3 = \frac{1}{12}|K|\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} =: \mathbf{M}_K\,.$$

Be smart and realize that both all diagonal and all off-diagonal entries have the same value! Defining $\mathbf{X} := [\xi_{k,\ell}]_{k,\ell=1}^3 \in \mathbb{R}^{3,3}$ (**??**) becomes

$$\mathbf{M}_K\mathbf{X} = \mathbf{I} \quad \Leftrightarrow \quad \mathbf{X} = \mathbf{M}_K^{-1} = \frac{3}{|K|}\begin{bmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{bmatrix}\,.$$

It is advisable to compute the inverse $\mathbf{M}_K^{-1}$ with PYTHON! This means

$$\begin{aligned} \psi_K^1 &= \frac{3}{|K|}(3\lambda_1 - \lambda_2 - \lambda_3)\,, \\ \psi_K^2 &= \frac{3}{|K|}(-\lambda_1 + 3\lambda_2 - \lambda_3)\,, \\ \psi_K^3 &= \frac{3}{|K|}(-\lambda_1 - \lambda_2 + 3\lambda_3)\,. \end{aligned} \qquad (3.14.4)$$

For the reference triangle we have $|K| = \frac{1}{2}$ and

$$\lambda_1(\boldsymbol{x}) = 1 - x_1 - x_2\,, \quad \lambda_2(\boldsymbol{x}) = x_1\,, \quad \lambda_3(\boldsymbol{x}) = x_2\,, \quad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\,.$$

Combining the formulas, we find

$$\widehat{\psi}^1 = 6(3 - 4x_1 - 4x_2)\,, \quad \widehat{\psi}^2 = 6(-1 + 4x_1)\,, \quad \widehat{\psi}^3 = 6(-1 + 4x_2)\,.$$

Alternatively, we can also start from the trial expressions

$$\widehat{\psi}^\ell(\boldsymbol{x}) = \alpha_\ell x_1 + \beta_\ell x_2 + \gamma_\ell\,, \quad \alpha_\ell,\beta_\ell,\gamma_\ell \in \mathbb{R}\,, \quad \ell \in \{1,2,3\}\,,$$

and derive a linear system of equations for the coefficients.

Supplement: The following computation offers a partial validation of the obtained formulas:

$$\int_{\widehat{K}} (\widehat{\psi}^1(\boldsymbol{x}) + \widehat{\psi}^3(\boldsymbol{x}) + \widehat{\psi}^3(\boldsymbol{x}))\lambda_j\,\mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} 6\lambda_j\,\mathrm{d}\boldsymbol{x} = 1\,.$$

SOLUTION of (3-14.b):

This result can be shown in several different ways: First, it is clear from the general formula (**??**), because $\boldsymbol{\Phi}_K^* \lambda_j = \widehat{\lambda}_j$ for the barycentric coordinate functions, see [Lecture → Eq. (2.8.1.15)].

Alternatively, we can directly verify the condition (3.14.1) for ψ_K^ℓ from (3.14.5), which maps the ℓ-th vertex of $\widehat{K}$ to the ℓ-th vertex of K. First note that (3.14.5) means

$$\psi_K^\ell(\boldsymbol{x}) = \frac{|\widehat{K}|}{|K|}\widehat{\psi}^\ell(\boldsymbol{\Phi}_K^{-1}(\boldsymbol{x}))\,, \quad \boldsymbol{x} \in K \qquad \Leftrightarrow \qquad \psi_K^\ell(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})) = \frac{|\widehat{K}|}{|K|}\widehat{\psi}^\ell(\widehat{\boldsymbol{x}})\,, \quad \widehat{\boldsymbol{x}} \in \widehat{K}\,.$$

Then we *transform the integral* to $\widehat{K}$, which is particularly easy for the affine mapping $\boldsymbol{\Phi}_K$, whose Jacobian is constant with determinant $= |K|/|\widehat{K}|$:

$$\int_K \psi_K^\ell(\boldsymbol{x})\lambda_j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{|K|}{|\widehat{K}|}\int_{\widehat{K}} \psi_K^\ell(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\widehat{\lambda}_j(\widehat{\boldsymbol{x}})\,\mathrm{d}\widehat{\boldsymbol{x}} = \int_{\widehat{K}} \widehat{\psi}^\ell(\widehat{\boldsymbol{x}})\widehat{\lambda}_j(\widehat{\boldsymbol{x}})\,\mathrm{d}\widehat{\boldsymbol{x}} = \delta_{\ell,j}\,.$$

Here we used the fact that $\widehat{\lambda}_j = \boldsymbol{\Phi}_K^* \lambda_j$, where $\widehat{\lambda}_j$, $j = 1,2,3$, are the barycentric coordinate functions on $\widehat{K}$, see [Lecture → Eq. (2.8.1.15)].

HINT 1 for (3-14.c): We need only check (3.14.8) for $v_h \in \mathfrak{B}_h$ and the "$=$" in the nodes of the mesh. ⌟

SOLUTION of (3-14.c):

Obviously, Q_h is a linear mapping. By linearity it is enough to verify (3.14.8) for a basis of $\mathcal{S}_1^0(\mathcal{M})$. Of course, we rely on the tent-function basis and we have to show

$$Q_h(b_h^q) = b_h^q \quad \forall b_h^q \in \mathfrak{B}_h \,. \tag{3.14.9}$$

Moreover, we exploit that a function in $\mathcal{S}_1^0(\mathcal{M})$ is uniquely determined by its values on the nodes of the mesh. Thus we have to show that

$$\big(Q_h(b_h^q)\big)(z) = \begin{cases} 1 & \text{, if } z = q \,, \\ 0 & \text{, if } x \neq q \,, \end{cases} \qquad \forall q, z \in \mathcal{N}(\mathcal{M}) \quad [\hat{=} \text{ nodes of } \mathcal{M}] \,. \tag{3.14.10}$$

By the cardinal basis property of $\mathfrak{B}_h = \{b_h^q\}_{q \in \mathcal{N}(\mathcal{M})}$, which means, $b_h^q(z) = \delta_{q,z}$ (Kronecker symbol), we have

$$\big(Q_h(b_h^q)\big)(q) = \int_{K_q} b_h^q(x)\, \psi_{K_q}^q(x)\, \mathrm{d}x \,.$$

Key is to remember that $b_h^q\big|_{K_q}$ is a local shape function on K_q and, thus, agrees with the barycentric coordinate function belonging to vertex q of K_q. By the property (3.14.1) the integral evaluates to 1, which implies $\big(Q_h(b_h^q)\big)(q) = 1$.

For $z \in \mathcal{N}(\mathcal{M})$, $z \neq q$, we can distinguish two cases:

1. If q is not a vertex of K_z, then $b_h^q\big|_{K_z} \equiv 0$ and the integral vanishes.
2. If q is a vertex of K_z, then $b_h^q\big|_{K_z}$ is a barycentric coordinate function of K_z. However, it belongs to a vertex different from that of $\psi_{K_z}^z$ and by (3.14.1) the integral must vanish, too.

We conclude that $\big(Q_h(b_h^q)\big)(z) = 0$.

HINT 1 for (3-14.d): The formula (3.14.5) is very useful. ⌟

HINT 2 for (3-14.d): Your final formula can contain the affine maps $\mathbf{\Phi}_K$, $K \in \mathcal{M}$, and the functions $\widehat{\psi}^j$, $j = 1, 2, 3$. ⌟

SOLUTION of (3-14.d):

By [Lecture → Eq. (2.7.5.23)] the mapped quadrature rule on K reads

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{|\widehat{K}|}\sum_{\ell=1}^{P}\widehat{\omega}_\ell f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\;,\quad f\in C^0(\overline{K})\;.$$

Thus, using (3.14.5), $\boldsymbol{\Phi}_K^*\psi_K^\ell = \frac{|\widehat{K}|}{|K|}\widehat{\psi}^\ell$, (3.14.7) becomes

$$\begin{aligned}\widetilde{\mathrm{Q}}_h(v) &= \sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})}\frac{|K_{\boldsymbol{p}}|}{|\widehat{K}|}\sum_{\ell=1}^{P}\widehat{\omega}_\ell v(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\psi_{K_{\boldsymbol{p}}}^{\boldsymbol{p}}(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\,b_h^{\boldsymbol{p}}\\ &= \sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})}\frac{|K_{\boldsymbol{p}}|}{|\widehat{K}|}\sum_{\ell=1}^{P}\widehat{\omega}_\ell v(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\frac{|\widehat{K}|}{|K_{\boldsymbol{p}}|}\widehat{\psi}^j(\widehat{\boldsymbol{\zeta}}_\ell)\,b_h^{\boldsymbol{p}}\\ &= \sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})}\sum_{\ell=1}^{P}\widehat{\omega}_\ell\, v(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_\ell))\,\widehat{\psi}^j(\widehat{\boldsymbol{\zeta}}_\ell)\,b_h^{\boldsymbol{p}}\;,\end{aligned}\tag{3.14.11}$$

where $j\in\{1,2,3\}$ stands for the local vertex number of the node $\boldsymbol{p}$ in $K_{\boldsymbol{p}}$: $\psi_{K_{\boldsymbol{p}}}^j = \psi_{K_{\boldsymbol{p}}}^{\boldsymbol{p}}$.

Note that the values $\widehat{\psi}^j(\widehat{\boldsymbol{\zeta}}_\ell)$ do not depend on $\boldsymbol{p}$ and can be precomputed for all j and ℓ.

SOLUTION of (3-14.e):

Since the functions ψ_K^ℓ are polynomials of degree 1, for $v \in \mathcal{S}_3^0(\mathcal{M})$, which is a polynomial of degree 3 on each cell, the integrand in the definition (3.14.7) is a polynomial of degree 4, which has to be integrated exactly. Therefore we have to issue the command `make_QuadRule(lf::base::RefEl::kTria(),` , which will return a quadrature rule meeting the specifications with a minimal number of nodes.

HINT 1 for (3-14.f): You are advised to rely on the function `findKp()`. ⌟

SOLUTION of (3-14.f):

Of course, in the beginning we use function `findKp()` from **??** to obtain the mapping $p \mapsto K_p$ and the local number of p in K_p.

Next, we invoke `lf::quad::make_QuadRule()` as in Sub-problem (3-14.e) and obtain $P \in \mathbb{N}$ weights $\widehat{\omega}_\ell$ and nodes $\widehat{\zeta}_\ell$ of a quadrature rule on $\widehat{K}$, see Sub-problem (3-14.d). We *precompute* the values $\widehat{\psi}^j(\widehat{\zeta}_\ell)$, $j = 1,2,3$, $\ell = 1,\ldots,P$.

Then implement (**??**) by traversing all nodes $p \in \mathcal{N}(\mathcal{M})$, visiting K_p, fetching $|K_p|$, and performing local quadrature on K_p. Note that, using the notations of (**??**), $v(\Phi_K(\widehat{\zeta}_\ell)) = \texttt{v_mf}(K_p, \widehat{\zeta}_\ell)$.

C++ code 3.14.13: Implementation of quasi-interpolation operator onto $\mathcal{S}_1^0(\mathcal{M})$ ➙ GitLab

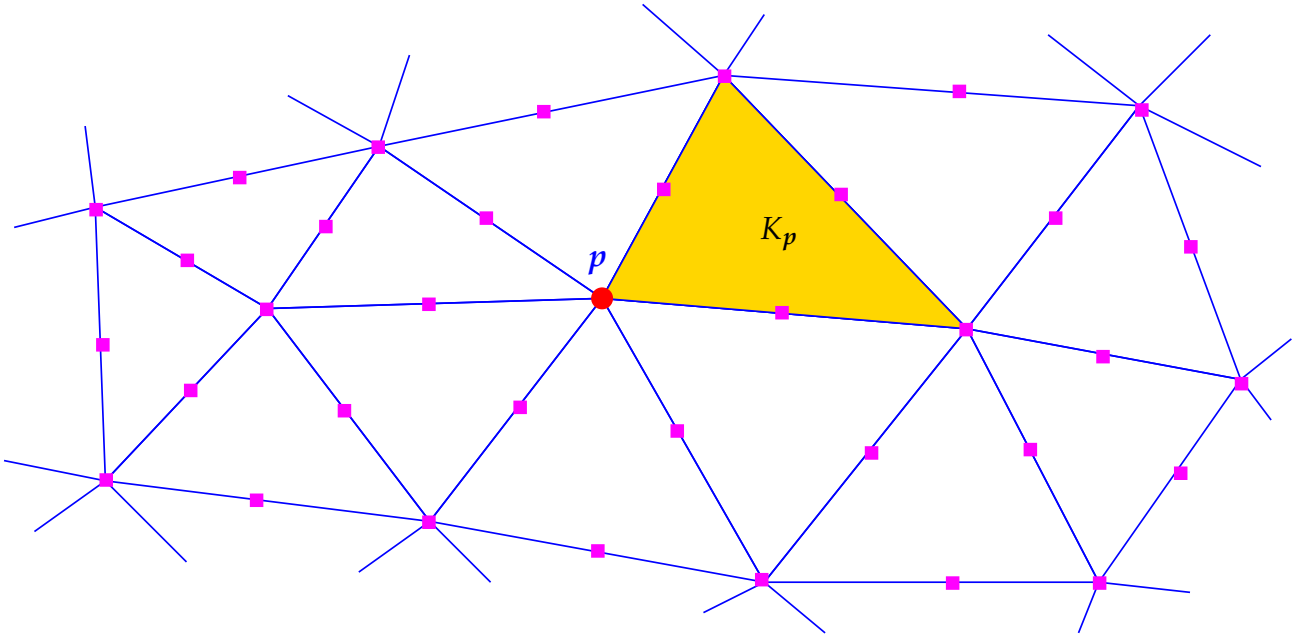
```
template <typename MESHFUNCTION>
Eigen::VectorXd quasiInterpolate(
    const lf::uscalfe::FeSpaceLagrangeO1<double> &fe_space,
    MESHFUNCTION &&v_mf) {
  // Get quadrature points and weights on the reference triangle for a
  // quadrature rule that integrates quadratic polynomials exactly
  lf::quad::QuadRule quadrule =
      lf::quad::make_QuadRule(lf::base::RefEl::kTria(), 2);
  const int P = quadrule.NumPoints();
  Eigen::MatrixXd zeta_hat = quadrule.Points();
  Eigen::VectorXd w_hat = quadrule.Weights();
  // Precomputation:  Cache values of (ψ̂^1, ψ̂^2, ψ̂^3)
  // at reference quadrature nodes.  Use the formulas (??).
  Eigen::Matrix<double, 3, 2> A;
  A << -24.0, -24.0, 24.0, 0.0, 0.0, 24.0;
  Eigen::Vector3d b{18.0, -6.0, -6.0};
  auto psi_hat = [A, b](Eigen::Vector2d x) -> Eigen::Vector3d {
    return A * x + b;
  };
  Eigen::MatrixXd psi_hat_values(3, P);
  for (int l = 0; l < P; ++l) {
    psi_hat_values.col(l) = psi_hat(zeta_hat.col(l));
  }
  // Retrieve the map p ↦ (K_p, j)
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space.Mesh();
  lf::mesh::utils::CodimMeshDataSet<
      std::pair<const lf::mesh::Entity *, unsigned int>>
      Kp_mesh_data_set = findKp(mesh_p);

  const lf::assemble::DofHandler &dofh{fe_space.LocGlobMap()};
  const int N = dofh.NumDofs();
  // Vector for returning the basis expansion coefficients of the
  // quasi-interpolant
  Eigen::VectorXd coefficients(N);
  // Compute projection onto each basis function
  for (lf::assemble::gdof_idx_t i = 0; i < N; ++i) {
    // Get p, K_p and the local index j of p in K_p
    const lf::mesh::Entity &point = dofh.Entity(i);
    std::pair<const lf::mesh::Entity *, unsigned int> Kp_localindex =
        Kp_mesh_data_set(point);
    const lf::mesh::Entity *Kp = Kp_localindex.first;
```

```
43      unsigned int j = Kp_localindex.second;
44      // Evaluate v on quadrature points in K_p
45      const std::vector<double> v_zeta = v_mf(*Kp, zeta_hat);
46      // Compute integral on current triangle Kp
47      double sum = 0.0;
48      for (int l = 0; l < P; ++l) {
49        sum += w_hat(l) * v_zeta[l] * psi_hat_values(j, l);
50      }
51      coefficients(i) = sum;
52    }
53    return coefficients;
54  }
```

HINT 1 for (3-14.g): Do not attempt to compute the entries of $\mathbf{R}$. “Support arguments” suffice. ⌟

SOLUTION of (3-14.g):

The rows of the matrix $\mathbf{R}$ correspond to the basis functions of $\mathcal{S}_1^0(\mathcal{M})$ and, thus, can be associated with nodes of the mesh. The nodal basis functions of $\mathcal{S}_2^0(\mathcal{M})$ are associated with either the nodes of the mesh or the edges, see [Lecture → Ex. 2.6.1.2]. Their "locations" are indicated by □ in **??**.



Single out a node $p \in \mathcal{N}(\mathcal{M})$ (• in **??**) and consider the corresponding row of $\mathbf{R}$. From the definition of Q_h it is clear that only those basis functions of $\mathcal{S}_2^0(\mathcal{M})$ contribute to the value $(Q_h(v_h))(p)$, whose supports overlap with K_p. These are the six basis functions belonging to the vertices and edges of K_p.

Hence, every row of $\mathbf{R}$ has at most 6 non-zero entries .

HINT 1 for (3-14.h): In your proof you can appeal to the following result without further justification:

Theorem 3.14.15. Equivalence of continuous and discrete L^2-norms

With a constant $C_2 > 0$ depending only on the shape regularity measure $\rho_{\mathcal{M}}$ of $\mathcal{M}$ holds

$$\frac{1}{C_2}\|v_h\|^2_{L^2(\Omega)} \leq \sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})} |K_{\boldsymbol{p}}| \left(v_h(\boldsymbol{p})\right)^2 \leq C_2\|v_h\|^2_{L^2(\Omega)} \quad \forall v_h \in \mathcal{S}^0_1(\mathcal{M}) \,,$$

where $K_{\boldsymbol{p}}$ is the largest cell of $\mathcal{M}$ adjacent to $\boldsymbol{p}$.

HINT 2 for (3-14.h): The formula (3.14.5) could be important for the proof. ⌟

SOLUTION of (3-14.h):

Using **??** we get

$$\begin{aligned}\|Q_h(v)\|_{L^2(\Omega)}^2 &\leq C_2 \sum_{p\in\mathcal{N}(\mathcal{M})} |K_p| \left| \int_{K_p} v(x)\psi_{K_p}^p(x)\,\mathrm{d}x \right|^2 \\ &\leq C_2 \sum_{p\in\mathcal{N}(\mathcal{M})} |K_p| \int_{K_p} |v(x)|^2\,\mathrm{d}x \int_{K_p} \left|\psi_{K_p}^p(x)\right|^2\,\mathrm{d}x\,,\end{aligned}$$

where we used the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)] in the second step.

To estimate $\left\|\psi_{K_p}^p\right\|_{L^2(K_p)}$ we rely on (3.14.5) and, by *transforming the integral to the reference triangle* $\widehat{K}$, find that (j is the number of the vertex p in K_p)

$$\int_{K_p} \left|\psi_{K_p}^p(x)\right|^2\,\mathrm{d}x = \frac{|K_p|}{|\widehat{K}|} \int_{\widehat{K}} |K_p|^{-2}|\widehat{K}|^2 \left|\widehat{\psi}^j(\widehat{x})\right|^2\,\mathrm{d}\widehat{x} = |K_p|^{-1}\,\widehat{C}_j\,,$$

where $\widehat{C}_j > 0$, $j = 1,2,3$, are three fixed numbers ("universal constants"). Combining the two estimates and canceling $|K_p|$ we end up with

$$\|Q_h(v)\|_{L^2(\Omega)}^2 \leq C_2 \max_{j=1,2,3} \widehat{C}_j \sum_{p\in\mathcal{N}(\mathcal{M})} \int_{K_p} |v(x)|^2\,\mathrm{d}x\,.$$

The final argument is, that in this sum, *a cell of the mesh occurs as domain of integration most three times*, which means

$$\|Q_h(v)\|_{L^2(\Omega)}^2 \leq 3C_2 \max_{j=1,2,3} \widehat{C}_j \sum_{K\in\mathcal{M}} \int_K |v(x)|^2\,\mathrm{d}x = \underbrace{3C_2 \max_{j=1,2,3} \widehat{C}_j}_{=:C^2} \|v\|_{L^2(\Omega)}^2\,.$$

HINT 1 for (3-14.i): Note that the two functions u_1 and u_2 have different smoothness properties. ⌟

SOLUTION of (3-14.i):

Observe that from row to the next row of the table the meshwidth is halved.

$\|f_A - Q_k(f_A)\|_{L^2(\Omega)}$:	Algebraic convergence with rate < 2
$\|f_A - Q_k(f_A)\|_{H^1(\Omega)}$:	Algebraic convergence with rate < 1
$\|f_B - Q_k(f_B)\|_{L^2(\Omega)}$:	Algebraic convergence with rate ≈ 2
$\|f_B - Q_k(f_B)\|_{H^1(\Omega)}$:	Algebraic convergence with rate ≈ 1

The function u_1 has a kink and belongs to $C^0_{\mathrm{pw}}(\overline{\Omega}) \subset H^1(\Omega)$, but not to $H^2(\Omega)$. Conversely, we have $u_2 \in H^2(\Omega)$. *For the smoother function we expect faster convergence* of the quasi-interpolation error, which makes us guess $f_A \leftrightarrow u_1$ and $f_B \leftrightarrow u_2$, mainly based on the behavior of the $H^1(\Omega)$-norm.

Supplement: If $u \in H^2(\Omega)$ we conclude from [Lecture → Cor. 3.3.3.4], the $L^2(\Omega)$-continuity (3.14.14) of Q_h, and the projector property established in Sub-problem (3-14.c) that

$$\begin{aligned}\|u - Q_h u\|_{L^2(\Omega)} &= \inf_{v_h \in \mathcal{S}^0_1(\mathcal{M})} \|(u - v_h) - Q_h(u - v_h)\|_{L^2(\Omega)} \\ &\overset{(3.14.14)}{\leq} (1 + C) \inf_{v_h \in \mathcal{S}^0_1(\mathcal{M})} \|u - v_h\|_{L^2(\Omega)} \leq C h^2_{\mathcal{M}} \|u\|_{H^2(\Omega)} \,.\end{aligned}$$

It can also be shown that on sequences of meshes created by uniform regular refinement the quasi-interpolation operator Q_h is even continuous in $H^1(\Omega)$:

$$\|Q_h(v)\|_{H^1(\Omega)} \leq C \|v\|_{H^1(\Omega)} \quad \forall v \in L^2(\Omega) \,, \tag{3.14.16}$$

with $C > 0$ depending only on the quasi-uniformity measure of the mesh. This permits us to repeat the above argument and appeal to [Lecture → Thm. 3.3.5.6]:

$$\begin{aligned}\|u - Q_h u\|_{H^1(\Omega)} &= \inf_{v_h \in \mathcal{S}^0_1(\mathcal{M})} \|(u - v_h) - Q_h(u - v_h)\|_{H^1(\Omega)} \\ &\overset{(??)}{\leq} (1 + C) \inf_{v_h \in \mathcal{S}^0_1(\mathcal{M})} \|u - v_h\|_{H^1(\Omega)} \leq C h_{\mathcal{M}} \|u\|_{H^2(\Omega)} \,.\end{aligned}$$

SOLUTION of (3-15.a):

First, recall that all the meshes $\mathcal{M}_\ell$ have the same shape-regularity measure. Then, we rely on a result presented in class:

Theorem [Lecture → Thm. 3.3.2.21]. Error estimate for piecewise linear interpolation

For any $u \in C^2(\bar{\Omega})$ and 2D piecewise linear interpolation $\mathsf{I}_1 : C^0(\overline{\Omega}) \to \mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh, holds

$$\|u - \mathsf{I}_1 u\|_{L^2(\Omega)} \leq \sqrt{\tfrac{3}{8}}\, h_{\mathcal{M}}^2 \left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)} ,$$
$$\|\mathbf{grad}(u - \mathsf{I}_1 u)\|_{L^2(\Omega)} \leq \sqrt{\tfrac{3}{24}}\, \rho_{\mathcal{M}} h_{\mathcal{M}} \left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)} .$$

where $h_{\mathcal{M}}$ denotes the mesh width (→ [Lecture → Def. 3.2.1.4]) and $\rho_{\mathcal{M}}$ the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$.

or

Corollary [Lecture → Cor. 3.3.3.4]. Error estimate for piecewise linear interpolation in 2D

Under the assumptions/with notations of [Lecture → Thm. 3.3.2.21]

$$\begin{aligned} \|u - \mathsf{I}_1 u\|_{L^2(\Omega)} &\leq \sqrt{\tfrac{3}{8}}\, h_{\mathcal{M}}^2 |u|_{H^2(\Omega)} , \\ |u - \mathsf{I}_1 u|_{H^1(\Omega)} &\leq \sqrt{\tfrac{3}{24}}\, \rho_{\mathcal{M}}\, h_{\mathcal{M}} |u|_{H^2(\Omega)} , \end{aligned} \qquad \forall u \in H^2(\Omega) .$$

The first estimate from this theorem/corollary amounts to

$$\|u - \mathsf{I}_1 u\|_{L^2(\Omega)} \leq C h_{\mathcal{M}}^2 |u|_{H^2(\Omega)} \quad \forall u \in H^2(\Omega) ,$$

and shows that

- $q \in \{0, 1, 2\} \subset \mathbb{N}_0$,
- $p \in \{2, 3, 4, \ldots\} \subset \mathbb{N}_0$,

The most important case is $q = 2$, $p = 2$, which is asserted in the theorem, but all the other values for the parameters are also possible.

Note that $p < 2$ is not possible, because the nodal interpolation operators are not defined on the spaces $L^2(\Omega) = H^0(\Omega)$ and $H^1(\Omega)$, because in two dimensions both these spaces contain unbounded functions, recall [Lecture → Ex. 1.2.3.45]. Conversely, $p \geq 2$ is admissible, because according to [Lecture → Thm. 3.3.3.8] the spaces $H^p(\Omega)$ contain only continuous and bounded functions.

SOLUTION of (3-15.b):

Note that $|u-u_\ell|_{H^1(\Omega)}$ is the energy norm of the finite-element discretization error. Thus we have by the optimality of Galerkin solutions

$$|u-u_\ell|_{H^1(\Omega)} = \inf_{v_h \in \mathcal{S}^0_2(\mathcal{M}_\ell)} |u-v_h|_{H^1(\Omega)} \,.$$

This suggests that we appeal to the following key result from the course:

Theorem [Lecture → Thm. 3.3.5.6]. Best approximation error estimates for Lagrangian finite elements

Let $\Omega \subset \mathbb{R}^d$, $d = 1,2,3$, be a bounded polygonal/polyhedral domain equipped with a mesh $\mathcal{M}$ consisting of simplices or parallelepipeds. Then, for each $k \in \mathbb{N}$, there is a constant $C > 0$ depending only on k and the shape regularity measure $\rho_{\mathcal{M}}$ such that

$$\inf_{v_h \in \mathcal{S}^0_p(\mathcal{M})} \|u-v_h\|_{H^1(\Omega)} \le C \left(\frac{h_{\mathcal{M}}}{p}\right)^{\min\{p+1,k\}-1} \|u\|_{H^k(\Omega)} \quad \forall u \in H^k(\Omega) \,.$$

[Lecture → Eq. (3.3.5.7)]

The exact solution $\boldsymbol{x} \to \exp(-\|\boldsymbol{x}\|^2)$ belongs to $C^\infty(\overline{\Omega})$ and, therefore, to every Sobolev space $H^m(\Omega)$. Thus smoothness of the exact solution does not limit the rate of asypmtotic convergence and we can apply [Lecture → Thm. 3.3.5.6] with $p=2$ and $k=3$:

$$|u-u_\ell|_{H^1(\Omega)} = O(h_\ell^2) \quad \text{for} \quad \ell \to \infty \,. \tag{3.15.2}$$

As in [Lecture → § 3.3.5.12] we make the assumption that the asymptotic estimates are sharp

$$\epsilon_\ell := |u-u_\ell|_{H^1(\Omega)} \approx C h_\ell^2 \quad \forall \ell \in \mathbb{N}_0 \,.$$

In order to achieve

$$\frac{\epsilon_\ell}{\epsilon_m} \approx \frac{C h_\ell^2}{C h_m^2} \overset{!}{=} 1000$$

we have to aim for $h_m \approx \frac{1}{\sqrt{1000}} h_\ell \approx \frac{1}{31.6} h_\ell$.

A single step of Regular refinement reduces the mesh width by a factor of 2. Thus we need 5 refinement steps to end up with a mesh whose mesh width is $\le \frac{1}{31.6}\cdot$ the mesh width of $\mathcal{M}_\ell$.

Alternatively, from (**??**) we conclude that halving the meshwidth h_ℓ, which is achieved by uniform refinement, will reduce the error by a factor of 4. So the number of required refinement steps is the smallest power of 4 that exceeds 1000: $4^r \overset{!}{\ge} 1000$, with the solution $r = 5$.

SOLUTION of (3-16.a):

The reference triangle $\widehat{K}$ has vertices $\widehat{\boldsymbol{a}}^1 := \left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]$, $\widehat{\boldsymbol{a}}^2 := \left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$, and $\widehat{\boldsymbol{a}}^3 := \left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$. The local shape functions on $\widehat{K}$ are linear and, therefore, of the form

$$\widehat{b}^\ell = \boldsymbol{c}^\ell \cdot \widehat{\boldsymbol{x}} + \tau^\ell \,, \quad \boldsymbol{c}^\ell \in \mathbb{R}^2, \; \tau^\ell \in \mathbb{R} \,, \quad \ell = 1,2,3 \,.$$

According to (3.16.1) the cofficients $\boldsymbol{c}^\ell$ and τ^ℓ can be found from the interpolation conditions

$$\widehat{b}^\ell(\boldsymbol{m}_k) = \delta_{\ell,k} \,, \quad l,k \in \{1,\ldots,3\} \,,$$
$$\boldsymbol{m}_1 = \begin{bmatrix}\frac{1}{2}\\0\end{bmatrix} \quad , \quad \boldsymbol{m}_2 = \begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\end{bmatrix} \quad , \quad \boldsymbol{m}_3 = \begin{bmatrix}0\\\frac{1}{2}\end{bmatrix} \quad .$$

Skipping the details, we find

$$\widehat{b}^1(\widehat{\boldsymbol{x}}) = 1 - 2\widehat{x}_2 \,, \quad \widehat{b}^2(\widehat{\boldsymbol{x}}) = 2(\widehat{x}_1 + \widehat{x}_2) - 1 \,, \quad \widehat{b}^3(\widehat{\boldsymbol{x}}) = 1 - 2\widehat{x}_1 \quad .$$

SOLUTION of (3-16.b):

There is a single global basis function associated with every edge.

```
lf::assemble::UniformFEDofHandler dh_CR(
mesh_p, {{lf::base::RefEl::kPoint(), 0},
  {lf::base::RefEl::kSegment(), 1},
  {lf::base::RefEl::kTria(), 0},
  {lf::base::RefEl::kQuad(), 0}});
```

SOLUTION of (3-16.c):

- Every edge of $\mathcal{M}$ contributes a single diagonal entry.
- Non-zero off-diagonal entries are due to pairs of basis functions with overlapping supports. For every triangle we find three such pairs, each contributing two potentially non-zero off-diagonal entries.

$$\mathrm{nnz}(\mathbf{A}) \leq 6 \cdot \sharp\mathcal{M} + 1 \cdot \sharp\mathcal{E} \ .$$

For a triangular mesh we have $\sharp\mathcal{M} \leq \frac{2}{3}\sharp\mathcal{E}$. This can be seen by counting the triangles as neighbors of edges. Each triangle is counted three times and for every edge we count at most two triangles. From this estimate we deduce the slightly less sharp but acceptable boundary

$$\mathrm{nnz}(\mathbf{A}) \leq 5 \cdot \sharp\mathcal{E} \ .$$

SOLUTION of (3-16.d):

The key observation is that the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$ and $\mathcal{CR}(\mathcal{M})$ can be linearly transformed into each other *independently of* K. Fixing a triangle $K \in \mathcal{M}$ we know that the local shape functions of $\mathcal{S}_1^0(\mathcal{M})$ are the barycentric coordinate function $\lambda_1, \lambda_2, \lambda_3$. For the three local shape functions of $\mathcal{CR}(\mathcal{M})$ we write b_K^1, b_K^2, b_K^3. Both sets are bases of the same space $\mathcal{P}_1(\mathbb{R}^2)$, which means that they can be transformed into each other by a bijective linear mapping.

From the condition $b_K^i(\boldsymbol{m}_j) = \delta_{i,j}$, $i,j = 1,2,3$, $\boldsymbol{m}_k$ the midpoint of edge k, we readily deduce

$$\begin{aligned} b_K^1 &= 1 - 2\lambda_3 = \lambda_1 + \lambda_2 - \lambda_3 \,, \\ b_K^2 &= 1 - 2\lambda_1 = \lambda_2 + \lambda_3 - \lambda_1 \,, \\ b_K^3 &= 1 - 2\lambda_2 = \lambda_1 + \lambda_3 - \lambda_2 \,. \end{aligned} \tag{3.16.4}$$

The associated local basis transformation matrix $\mathbf{S}_K \in \mathbb{R}^{3,3}$ is

$$\mathbf{S}_K = \begin{bmatrix} 1 & 1 & -1 \\ -1 & 1 & 1 \\ 1 & -1 & 1 \end{bmatrix} .$$

Then we can apply

Lemma [Lecture → Lemma 2.2.3.2]. Effect of change of basis on Galerkin matrix

Consider [Lecture → Eq. (2.2.1.1)] and two bases of $V_{0,h}$,

$$\mathfrak{B}_h := \{b_h^1, \ldots, b_h^N\} \quad , \quad \widetilde{\mathfrak{B}}_h := \{\widetilde{b}_h^1, \ldots, \widetilde{b}_h^N\} \,,$$

related by the basis transformation matrix $\mathbf{S}$ according to

$$\widetilde{b}_h^j = \sum_{k=1}^N s_{jk} b_h^k \quad \textit{with} \quad \mathbf{S} = \left[s_{jk}\right]_{j,k=1}^N \in \mathbb{R}^{N,N} \textit{ regular.} \tag{3.16.5}$$

Then the Galerkin matrices $\mathbf{A}, \widetilde{\mathbf{A}} \in \mathbb{R}^{N,N}$, the right hand side vectors $\vec{\boldsymbol{\varphi}}, \widetilde{\vec{\boldsymbol{\varphi}}} \in \mathbb{R}^N$, and the coefficient vectors $\vec{\boldsymbol{\mu}}, \widetilde{\vec{\boldsymbol{\mu}}} \in \mathbb{R}^N$, respectively, satisfy

$$\widetilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^\top \quad , \quad \widetilde{\vec{\boldsymbol{\varphi}}} = \mathbf{S}\vec{\boldsymbol{\varphi}} \quad , \quad \widetilde{\vec{\boldsymbol{\mu}}} = \mathbf{S}^{-\top}\vec{\boldsymbol{\mu}} \,. \tag{3.16.6}$$

and find

$$\mathbf{A}_K^{\mathcal{CR}} = \mathbf{S}\mathbf{A}_K^{\mathcal{S}}\mathbf{S}^\top \,,$$

that is $\mathbf{S}_K := \mathbf{S}$, $\mathbf{T}_K := \mathbf{S}^\top$.

HINT 1 for (3-16.e): Exploit the cardinal-basis property of $\{b_h^1, \ldots, b_h^N\}$. ⌟

SOLUTION of (3-16.e):

Due to the cardinal-basis property of $\{b_h^1, \ldots, b_h^N\}$ with respect to the set $\mathcal{N} = \{\boldsymbol{m}_1, \ldots, \boldsymbol{m}_N\}$ of interpolation nodes we find $\nu_j = v_h(\boldsymbol{m}_j)$ for $v_h := \sum_{k=1}^{M} \nu_k q_h^k$. The value of v_h at the midpoint of an edge is just the arithmetic average of its (nodal) values at the endpoints of the edge, which are directly available through the basis expansion coefficients, since also the standard nodal basis of $\mathcal{S}_1^0(\mathcal{M})$ is a cardinal basis.

C++ code 3.16.8: Implementation of `LinLagrDOFsToCRDOFs()`

```
Eigen::VectorXd LinLagrDOFsToCRDOFs(const lf::assemble::DofHandler &dh_linfe,
                                    const lf::assemble::DofHandler &dh_cr,
                                    const Eigen::VectorXd &mu) {
  using gdof_idx_t = lf::assemble::gdof_idx_t;
  LF_ASSERT_MSG(dh_linfe.Mesh() == dh_cr.Mesh(),
                "DofHandlers must be based on the same mesh");
  LF_ASSERT_MSG(dh_linfe.NumDofs() == mu.size(), "Vector length mismath");
  // Underlying mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p{dh_linfe.Mesh()};
  const lf::mesh::Mesh &mesh{*mesh_p};
  LF_ASSERT_MSG((dh_cr.NumDofs() == mesh.NumEntities(1)) &&
                    (dh_linfe.NumDofs() == mesh.NumEntities(2)),
                "#DOFs do not match #entities");
  // The coefficients of with respect to the nodal basis of the C.-R. FE space
  // are simply the values of the function in the midpoints of edges.For a
  // piecewise linear continuous function these are available as simple average
  // of nodal values in the endpoints of the edge.  Vector for storing result
  Eigen::VectorXd nu(dh_cr.NumDofs());
  for (const lf::mesh::Entity *e : mesh.Entities(1)) {
    // Obtain global numbers of Lagrangian DOFs associated with the endpoints of
    // the edge, that is, of those DOFs covering the edge.
    nonstd::span<const gdof_idx_t> lf_dof_idx{dh_linfe.GlobalDofIndices(*e)};
    LF_ASSERT_MSG(lf_dof_idx.size() == 2,
                  "Exactly two basis functions must cover an edge");
    // Obtain global number of C.-R. basis function associated with the edge
    nonstd::span<const gdof_idx_t> cr_dof_idx{dh_cr.GlobalDofIndices(*e)};
    LF_ASSERT_MSG(cr_dof_idx.size() == 1,
                  "A single C.-R. basis function may belong to an edge!");
    // Value of p.w.  linear continuous FE function at the midpoint of the edge
    const double mp_val = (mu[lf_dof_idx[0]] + mu[lf_dof_idx[1]]) / 2.0;
    // Store value as appropriate coefficient
    nu[cr_dof_idx[0]] = mp_val;
  }
  return nu;
}
```

SOLUTION of (3-16.f):

Plugging the basis expansions

$$\mathsf{P}w_h = \sum_{k=1}^{M} \mu_k q_h^k \quad , \quad w_h = \sum_{j=1}^{N} \nu_j b_h^j$$

into (3.16.9) and testing with $v_h := q_h^\ell$ gives the following linear system of equations

$$\mathbf{X}\vec{\boldsymbol{\mu}} = \mathbf{Y}\vec{\boldsymbol{\nu}} \quad , \qquad \begin{aligned} \mathbf{X} &:= \left[\int_\Omega q_h^\ell(\boldsymbol{x}) q_h^j(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\right]_{\ell,j=1}^{M} \in \mathbb{R}^{M,M} \,, \\ \mathbf{Y} &:= \left[\int_\Omega b_j^\ell(\boldsymbol{x}) q_h^j(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\right]_{\substack{j=1,\ldots,N \\ \ell=1,\ldots,M}} \in \mathbb{R}^{M,N} \,, \end{aligned} \tag{3.16.10}$$

from which we can extract the formula $\vec{\boldsymbol{\mu}} = \mathbf{X}^{-1}\mathbf{Y}\vec{\boldsymbol{\nu}}$.

SOLUTION of (3-16.g):

The key insight is that we have to exploit the inclusion $\mathcal{S}_1^0(\mathcal{M}_h) \subset \mathcal{CR}(\mathcal{M}_h)$, which involves

$$\inf_{v_h \in \mathcal{CR}(\mathcal{M}_h)} \|u - v_h\|_{L^2(\Omega)} \leq \inf_{v_h \in \mathcal{S}_1^0(\mathcal{M}_h)} \|u - v_h\|_{L^2(\Omega)} ,$$
$$\inf_{v_h \in \mathcal{CR}(\mathcal{M}_h)} \sum_{K \in \mathcal{M}} |u - v_h|^2_{H^1(K)} \leq \inf_{v_h \in \mathcal{S}_1^0(\mathcal{M}_h)} |u - v_h|^2_{H^1(\Omega)} .$$

This makes it possible to appeal to the estimates for best approximation error of

Theorem [Lecture → Thm. 3.3.5.6]. Best approximation error estimates for Lagrangian finite elements

Let $\Omega \subset \mathbb{R}^d$, $d = 1,2,3$, be a bounded polygonal/polyhedral domain equipped with a mesh $\mathcal{M}$ consisting of simplices or parallelepipeds. Then, for each $k \in \mathbb{N}$, there is a constant $C > 0$ depending only on k and the shape regularity measure $\rho_{\mathcal{M}}$ such that

$$\inf_{v_h \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_h\|_{H^1(\Omega)} \leq C \left(\frac{h_{\mathcal{M}}}{p} \right)^{\min\{p+1,k\}-1} \|u\|_{H^k(\Omega)} \quad \forall u \in H^k(\Omega) .$$

[Lecture → Eq. (3.3.5.7)]

with $p = 1$, $k = 3$. We conclude

$$\inf_{v_h \in \mathcal{CR}(\mathcal{M}_h)} \|u - v_h\|_{L^2(\Omega)} = O\left(h^2\right) \quad \text{for} \quad h \to 0 ,$$
$$\inf_{v_h \in \mathcal{CR}(\mathcal{M}_h)} \sum_{K \in \mathcal{M}} |u - v_h|^2_{H^1(K)} = O\left(h^2\right) \quad \text{for} \quad h \to 0 .$$

SOLUTION of (3-17.a):

We face a very standard Dirichlet boundary value problem and according to [Lecture → Ex. 1.8.0.2] its variational formulation is:

$$u \in H_0^1(\Omega): \quad \int_\Omega \alpha(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\ .$$

SOLUTION of (3-17.b):

Since we have assumed $f \equiv 0$ and $\alpha \equiv 1$, we conclude

$$Z_h := \{v_h \in \mathcal{S}_1^0(\mathcal{M}) : \ [\![\mathbf{grad}\, v_h]\!]_e = 0 \ \forall e \in \mathcal{E}\} \ ,$$

which means that the (piecewise constant) gradients of functions in Z_h are continuous. Since $\mathcal{S}_1^0(\mathcal{M}) \subset C^0$ we infer $Z_h \subset C^1(\overline{\Omega})$. However, a continuously differentiable piecewise linear function must be a global linear function, a polynomial of degree 1 on Ω:

$$Z_h = \mathcal{P}_1(\mathbb{R}^2) \ .$$

HINT 1 for (3-17.c): What does the Laplacian do to a global linear function? ⌟

SOLUTION of (3-17.c):

Linear functions belong to the nullspace of Δ! Hence, the expression for η_K simplifies to

$$\eta_K = \mu_K^2 \|f\|_{L^2(K)}^2 = \frac{h_K^2}{\alpha_K} \int_K f(\boldsymbol{x})^2 \,\mathrm{d}\boldsymbol{x} \,.$$

Since f is given in procedural form we have to resort to a quadrature formula for the approximate evaluation of the integral. The requirement that the evaluation is exact for $f \in \mathcal{S}_1^0(\mathcal{M})$ means that we have to perform exact integration, whenever $f^2 \in \mathcal{P}_2(\mathbb{R}^2)$ on K. This is accomplished by using a quadrature rule of order ≥ 3 . The simplest instance is the edge-midpoint rule, in LEHRFEM++ available through

```
const lf::quad::QuadRule qr{lf::quad::make_TriaQR_EdgeMidpointRule()};
```

Its quadrature nodes are the midpoints of the edges of a triangle K and the weights are $\frac{1}{6}|K|$. Any other quadrature rule that is exact for polynomials of degree 2 is also eligible. For the use of **lf::quad::QuadRule** objects refer to [Lecture → Code 2.7.5.42].

To obtain the weights μ_K we have to

- determine the size h_K of a triangle, that is, the length of its longest edge. To that end we can either query the coordinates of the vertices of the triangle through the `lf::geometry::Corners()` function or visit the edges and get their endpoint coordinates.
- access the piecewise constant coefficient function α for a point inside K, which amounts to evaluating the **MeshFunction** for any point inside the reference triangle, e.g.,

  ```
  const Eigen::MatrixXd dummy =
  Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
  const double alpha_K = mf_alpha_(*K, dummy)[0];
  ```

  where `K` is a pointer to the current **lf::mesh::Entity** object for the triangle K.

C++ code 3.17.7: Implementation of `volumeResiduals()` → GitLab

```
2  lf::mesh::utils::CodimMeshDataSet<double> volumeResiduals(
3      const dataDiscreteBVP &disc_bvp, const Eigen::VectorXd & /*u_vec*/) {
4    // Get pointer to underlying mesh
5    std::shared_ptr<const lf::mesh::Mesh> mesh_p =
6        disc_bvp.pwlinfespace_p_->Mesh();
7    // Mesh data set for returning the volume residuals
8    lf::mesh::utils::CodimMeshDataSet<double> vol_res(mesh_p, 0, 0.0);
9
10   // Edge midpoint quadrature rule for triangles, which is exact for
         quadratic
11   // polynomials and allows the exact computation of the volume residual
12   // contributions for piecewise linear source functions.
13   const lf::quad::QuadRule qr{lf::quad::make_TriaQR_EdgeMidpointRule()};
14   const Eigen::VectorXd qw{qr.Weights()};  // quadrature weigth vector
15
16   // Run over all cells of the mesh
17   for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
18     LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
19                   "Implemented for triangles only");
20     // Obtain information about the shape of the cell
```

```
21      const lf::geometry::Geometry &geo{*(cell->Geometry())};
22      // Ddetermine size of triangle (length of longest edge)
23      const Eigen::MatrixXd corners{lf::geometry::Corners(geo)};
24      const double h0 = (corners.col(1) - corners.col(0)).norm();
25      const double h1 = (corners.col(2) - corners.col(1)).norm();
26      const double h2 = (corners.col(0) - corners.col(2)).norm();
27      const double h_K = std::max({h0, h1, h2});
28
29      // I: Locally integrate the square of the source function
30      // Gramian determinants
31      const Eigen::VectorXd gram_dets{geo.IntegrationElement(qr.Points())};
32      // Fetch values of source function in quadrature nodes.  Note that
            local
33      // reference coordinates have to be passed to the evaluation
            operator of the
34      // MeshFunction.
35      const std::vector<double> f_vals{disc_bvp.mf_f_(*cell, qr.Points())};
36      // Evaluation of quadrature formula
37      double f_sq_int = 0.0;
38      for (lf::base::size_type l = 0; l < qr.NumPoints(); ++l) {
39        f_sq_int += qw[l] * f_vals[l] * f_vals[l] * gram_dets[l];
40      }
41      // II: Fetch prefactor depending on diffusion coefficient
42      // We need a "dummy point" in the reference element in order to call
            the
43      // evaluation operator for the MeshFunction.
44      const Eigen::MatrixXd dummy = Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
45      const double alpha_K = disc_bvp.mf_alpha_(*cell, dummy)[0];
46      LF_ASSERT_MSG(alpha_K > 0,
47                    "Diffusion coefficients must be strictly positive!");
48      const double mu_K = h_K * h_K / alpha_K;
49      // Store cell contribution
50      vol_res(*cell) = f_sq_int * mu_K;
51    }
52    return vol_res;
53  }
```

HINT 1 for (3-17.d): Note that thanks to Ass. 3.17.2 $\boldsymbol{s} \mapsto \alpha(\boldsymbol{x})\,\mathbf{grad}\,u_h(\boldsymbol{x})$ is an $\mathcal{M}$-piecewise constant function. ⌟

HINT 2 for (3-17.d): To compute a jump you have to find out whether the edge normal $\boldsymbol{n}_e$ points into or out of a triangle K. Check the sign of $\boldsymbol{n}_{+e} \cdot (\boldsymbol{a}_e - \boldsymbol{p})$, where $\boldsymbol{a}_e$ is the vertex of K opposite to e and $\boldsymbol{p}$ is one of the endpoints of e. According to LEHRFEM++'s numbering convention the vertex opposite to the j-th edge, $j = 0, 1, 2$, is the `(j+2)%3`-th vertex, where `%` is the integer division remainder operator in C++. ⌟

SOLUTION of (3-17.d):

Some preliminary considerations: To begin with we note that the direction of the edge normal $\boldsymbol{n}_e$ does not matter: replacing $\boldsymbol{n}_e \to -\boldsymbol{n}_e$ does not affect the value of ρ_e.

In the formula for ρ_e the vector $\boldsymbol{n}_e$ was normalized, $\|\boldsymbol{n}_e\| = 1$, but this is not necessary. Writing $\boldsymbol{d}_e$ for the "direction vector" of the edge $e \in \mathcal{E}_0$ and $|e| = h_e$ for its length, we have

$$\boldsymbol{n}_e = \frac{1}{|e|}\widetilde{\boldsymbol{n}}_e\,, \quad \widetilde{\boldsymbol{n}}_e := \boldsymbol{d}_e^{\perp}\,,$$

where $\perp$ designates a planar rotation by 90 degrees

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\perp} = \begin{bmatrix} y \\ -x \end{bmatrix}\,, \quad x, y \in \mathbb{R}\,.$$

Using the fact that $\boldsymbol{x} \mapsto \alpha(\boldsymbol{x})\,\mathbf{grad}\,u_h(\boldsymbol{x})$ is a piecewise constant function, we can express

$$\rho_e = \frac{|e|}{\max\{\alpha_e^+, \alpha_e^-\}}\frac{1}{|e|}\|\widetilde{n}_e \cdot [\![\alpha\,\mathbf{grad}\,u_h]\!]_e\|^2 = \frac{1}{\max\{\alpha_e^+, \alpha_e^-\}}\|\widetilde{n}_e \cdot [\![\alpha\,\mathbf{grad}\,u_h]\!]_e\|^2\,.$$

We can identify several subtasks involved in the evaluation of these terms:

- We have to retrieve the direction vectors of the edges and rotate them by 90 degrees.
- We have to get the vector $\mathbf{grad}\,u_h|_K$ for every cell $K \in \mathcal{M}$.
- We have to collect the values $\max\{\alpha_e^+, \alpha_e^-\}$ for every edge $e \in \mathcal{E}_0$.
- We have to compute the jumps $[\![\alpha\ \mathbf{grad}\,u_h]\!]_e$ across every interior edge e.

Difficulties are compounded by the fact that in LEHRFEM++ one cannot access the cells adjacent to an edge. Hence, it is advisable to tackle the last two subtasks in a *cell-oriented* manner:

- To find $\max\{\alpha_e^+, \alpha_e^-\}$ we distribute the α-values from the cells to a **lf::mesh::utils::CodimMeshData** object associatyed with $\mathcal{E}$, and form the maximum in the process.
- To get $[\![\alpha\ \mathbf{grad}\,u_h]\!]_e$ we sum the vectors $\alpha\ \mathbf{grad}\,u_h|_K$ in the edges of K *after weighting with the relative orientation of the edge normal vector*. This will turn summation into subtraction, because the relative orientations of the normal of an interior edge with respect to its two adjacent cells will necessarily be opposite.

C++ code 3.17.8: Implementation of `edgeResiduals()` ➙ GitLab

```
2  lf::mesh::utils::CodimMeshDataSet<double> edgeResiduals(
3      const dataDiscreteBVP &disc_bvp, const Eigen::VectorXd &u_vec) {
4    // Get pointer to underlying mesh
5    std::shared_ptr<const lf::mesh::Mesh> mesh_p =
6        disc_bvp.pwlinfespace_p_->Mesh();
7    // Mesh data set for returning the volume residuals
8    lf::mesh::utils::CodimMeshDataSet<double> edge_res(mesh_p, 1, 0.0);
9    // Step I: Create an auxiliary MeshDataSet storing non-normalized (!)
         edge
10   // normals = edge direction vectors turned by 90 degrees
11   lf::mesh::utils::CodimMeshDataSet<Eigen::Vector2d> edge_normals(mesh_p, 1);
12   lf::mesh::utils::CodimMeshDataSet<Eigen::Vector2d> edge_startpt(mesh_p, 1);
```

```
  for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
    // Obtain information about the shape of the edge
    const lf::geometry::Geometry &geo{*(edge->Geometry())};
    const Eigen::MatrixXd corners{lf::geometry::Corners(geo)};
    // Starting point of the edge
    edge_startpt(*edge) = corners.col(0);
    // Direction vector of the edge
    const Eigen::Vector2d dir = corners.col(1) - corners.col(0);
    // Rotate counterclockwise by 90 degrees an d store
    edge_normals(*edge) = Eigen::Vector2d(dir(1), -dir(0));
  }
  // Step II: Compute the gradient of the finite element solution, which
  // LehrFEM++ can do for us.
  const lf::fe::MeshFunctionGradFE grad_uh(disc_bvp.pwlinfespace_p_, u_vec);

  // Step III: Visit the triangles of the mesh, fetch the constant(!)
     gradient
  // of the finite-element solution, multiply it with the edge normals
     and the
  // coefficient and sum the results for every edge not loacted on the
  // bounmdary.  This summation is done in an auxiliary MeshDataSet.
  lf::mesh::utils::CodimMeshDataSet<double> alpha_max(mesh_p, 1, 0.0);
  lf::mesh::utils::CodimMeshDataSet<double> edge_flux_jump(mesh_p, 1, 0.0);
  lf::mesh::utils::CodimMeshDataSet<bool> bd_ed_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Implemented for triangles only");
    // Retrieve constant diffusion coefficient
    const Eigen::MatrixXd dummy = Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
    const double alpha_K = disc_bvp.mf_alpha_(*cell, dummy)[0];
    LF_ASSERT_MSG(alpha_K > 0,
                  "Diffusion coefficients must be strictly positive!");
    // Fetch scaled gradient of the finite element solution
    const Eigen::Vector2d nablau_K = (grad_uh(*cell, dummy)[0]) * alpha_K;
    /* Debugging output
    std::cout << "cell " << mesh_p->Index(*cell)
              << ": alpha*grad u_h = " << nablau_K.transpose() << std::endl;
    */
    // Obtain shape of cell
    const lf::geometry::Geometry &cell_geo{*(cell->Geometry())};
    const Eigen::MatrixXd cell_vert{lf::geometry::Corners(cell_geo)};
    // Visit all three edges of the triangle
    std::span<const lf::mesh::Entity *const> edges{cell->SubEntities(1)};
    LF_ASSERT_MSG(edges.size() == 3, "Triangle must have three edges!");
    for (int l = 0; l < 3; ++l) {
      if (!bd_ed_flags(*edges[l])) {
        // Determine orientation of edge normal
        const lf::mesh::Entity &edge = *edges[l];
        const Eigen::Vector2d &normal{edge_normals(edge)};
        const int ori =
            (normal.dot(cell_vert.col((l + 2) % 3) - edge_startpt(edge)) > 0)
                ? 1
                : -1;
        edge_flux_jump(edge) += (ori * nablau_K.dot(normal));
        /* Debugging output
        std::cout << "edge " << mesh_p->Index(edge) << ", r.ori = " << ori
                  << ": edge jump = " << edge_flux_jump(edge) << std::endl;
```

```
69          */
70        }
71        if (alpha_K > alpha_max(*edges[l])) {
72          alpha_max(*edges[l]) = alpha_K;
73        }
74      }
75    }
76    // Now we have all required information in the auxiliary MeshDataSets
77    // and we traverse the edges again and compute the scaled jump norms.
78    for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
79      const double ed_flux = edge_flux_jump(*edge);
80      edge_res(*edge) = (ed_flux * ed_flux) / alpha_max(*edge);
81    }
82    return edge_res;
83  }
```

SOLUTION of (3-18.a):

1. $\mathcal{S}^0_{1,0}(\mathcal{M}_\ell)$ is spanned by basis functions associated with the *interior* nodes of $\mathcal{M}_\ell$:

$$\dim \mathcal{S}^0_{1,0}(\mathcal{M}_\ell) = N^\ell_V - N^\ell_{\partial,V} \ .$$

2. $\mathcal{S}^0_{2,0}(\mathcal{M}_\ell)$ is spanned by basis functions associated with the *interior* nodes and the interior edges of $\mathcal{M}_\ell$, see [Lecture $\rightarrow$ Ex. 2.6.1.2]:

$$\dim \mathcal{S}^0_{2,0}(\mathcal{M}_\ell) = N^\ell_V + N^\ell_E - N^\ell_{\partial,V} - N^\ell_{\partial,E} \ .$$

SOLUTION of (3-18.b):

Since $\boldsymbol{\alpha}$ is supposed to be uniformly positive definite and bounded it follows by the Poincaré-Friedrichs inequality from [Lecture → Thm. 1.3.4.17] that $|\widetilde{u}_\ell - \widetilde{u}_\ell|_{H^1(\Omega)}$ can be bounded from below and above by the energy norm of the Galerkin discretization error. Thus, we conclude from Cea's lemma [Lecture → Thm. 3.1.3.7] and [Lecture → Thm. 3.3.5.6] that

$$|\widetilde{u}_\ell - u|_{H^1(\Omega)} \leq Ch_\ell^1 \|u\|_{H^2(\Omega)} ,$$

with a constant $C > 0$ independent of ℓ, which means, by the triangle inequality

$$|\widetilde{u}_\ell - \widetilde{u}_{\ell-1}|_{H^1(\Omega)} \leq |\widetilde{u}_\ell - u|_{H^1(\Omega)} + |\widetilde{u}_{\ell-1} - u|_{H^1(\Omega)} \leq Ch_\ell^1 \|u\|_{H^2(\Omega)} ,$$

because $h_{\ell-1} = 2h_\ell$. We infer

$$|\widetilde{u}_\ell - \widetilde{u}_{\ell-1}|_{H^1(\Omega)} = O(h_\ell) \quad \text{for} \quad h_\ell \to 0 .$$

Note that only if we knew $u \in H^3(\Omega)$ we could conclude $|\widetilde{u}_\ell - \widetilde{u}_{\ell-1}|_{H^1(\Omega)} = O\big(h_\ell^3\big)$, but this information is not available.

SOLUTION of (3-18.c):

The relationships (3.18.2) and (3.18.3) are the well-known Galerkin orthogonalities introduced in [Lecture → Section 3.1.3]:

$$\mathsf{a}(\widetilde{u}_\ell - u, v_h) = 0 \quad \forall v_h \in \mathcal{S}^0_{2,0}(\mathcal{M}_\ell) \ ,$$

$$\mathsf{a}(u - u_\ell, v_h) = 0 \quad \forall v_h \in \mathcal{S}^0_{1,0}(\mathcal{M}_\ell) \ .$$

From these (3.18.4) follows immediately by adding and subtracting u, because $\mathcal{S}^0_{1,0}(\mathcal{M}_\ell) \subset \mathcal{S}^0_{2,0}(\mathcal{M}_\ell)$:

$$\mathsf{a}(\widetilde{u}_\ell - u_\ell, v_h) = 0 \quad \forall v_h \in \mathcal{S}^0_{1,0}(\mathcal{M}_\ell) \ .$$

SOLUTION of (3-18.d):

❶ The bilinear from a is an inner product on $H_0^1(\Omega)$. Therefore we can employ the Cauchy-Schwarz inequality along with Galerkin orthogonality:

$$\begin{aligned}\|\widetilde{u}_\ell - u_\ell\|_a^2 &= \mathsf{a}(\widetilde{u}_\ell - u_\ell, \widetilde{u}_\ell - u + u - u_\ell)\\ &= \underbrace{\mathsf{a}(\widetilde{u}_\ell - u_\ell, \widetilde{u}_\ell - u)}_{=0} + \mathsf{a}(\widetilde{u}_\ell - u_\ell, u - u_\ell)\\ &\leq \|\widetilde{u}_\ell - u_\ell\|_a \, \|u - u_\ell\|_a \,,\end{aligned}$$

because $\widetilde{u} - u_\ell \in \mathcal{S}^0_{2,0}(\mathcal{M}_\ell)$. Canceling one factor $\|\widetilde{u}_\ell - u_\ell\|_a$ yields $C_2 = 1$. One could also conclude this from $\mathcal{S}^0_{1,0}(\mathcal{M}_\ell) \subset \mathcal{S}^0_{2,0}(\mathcal{M}_\ell)$ and the optimality of Galerkin solutions w.r.t. the energy norm.

❷ The other bound is immediate from the triangle inequality

$$\begin{aligned}\|u_\ell - u\|_a &\leq \|\widetilde{u}_\ell - u\|_a + \|\widetilde{u}_\ell - u_\ell\|_a\\ &\leq \gamma \|u_\ell - u\|_a + \|\widetilde{u}_\ell - u_\ell\|_a \,,\end{aligned}$$

which implies $C_1 = 1 - \gamma$. Here it is important that $\gamma < 1$.

SOLUTION of (3-18.e):

In LEHRFEM++ both finite element spaces are equipped with a cardinal basis with respect to sets of interpolation nodes:

- $\mathcal{S}_1^0(\mathcal{M})$: interpolation nodes are all nodes of the mesh
- $\mathcal{S}_2^0(\mathcal{M})$: interpolation nodes are the nodes of the mesh and the midpoints of all edges [Lecture → Ex. 2.6.1.2].

Hence the algorithm for transforming the basis representations is the following:

1. Just copy the values of $u_h \in \mathcal{S}_1^0(\mathcal{M})$ in the nodes of the mesh into the associated components of a $\mathcal{S}_2^0(\mathcal{M})$-coefficient vector. To that end loop over the nodes of the mesh and retrieve the global numbers of the associated bases functions for both finite-element spaces.
2. Sample $u_h \in \mathcal{S}_1^0(\mathcal{M})$ at the midpoints of the edges, which can be done by averaging the nodal values in the endpoints, and then copy the obtained value into the edge-related component of the $\mathcal{S}_2^0(\mathcal{M})$-coefficient vector. This can be implemented as a loop over the edges of the mesh. Inside the loop you have to access the two degrees of freedom of $\mathcal{S}_1^0(\mathcal{M})$ *covering* the edge and the single degree of freedom of $\mathcal{S}_2^0(\mathcal{M})$ *associated* with the edge, *cf.* [Lecture → Section 2.5.3].

Access to the degrees of freedom is mediated through suitable **lf::assemble::DofHandler** objects and their methods `GlobalDofIndices()` and `InteriorGlobalDofIndices()`.

C++ code 3.18.7: Implementation of `trfLinToQuad()` ➙ GitLab

```
2  Eigen::VectorXd trfLinToQuad(
3      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_lin_p,
4      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_quad_p,
5      const Eigen::VectorXd &mu) {
6    using gdof_idx_t = lf::assemble::gdof_idx_t;
7    // Obtain local-to-global index mappings
8    const lf::assemble::DofHandler &dh_lin{fes_lin_p->LocGlobMap()};
9    const lf::assemble::DofHandler &dh_quad{fes_quad_p->LocGlobMap()};
10   LF_ASSERT_MSG(dh_lin.Mesh() == dh_quad.Mesh(),
11                 "DofHandlers must be based on the same mesh");
12   LF_ASSERT_MSG(dh_lin.NumDofs() == mu.size(), "Vector length mismath");
13   // Underlying mesh
14   std::shared_ptr<const lf::mesh::Mesh> mesh_p{dh_lin.Mesh()};
15   const lf::mesh::Mesh &mesh{*mesh_p};
16   LF_ASSERT_MSG(
17       (dh_lin.NumDofs() == mesh.NumEntities(2)) &&
18           (dh_quad.NumDofs() == mesh.NumEntities(2) + mesh.NumEntities(1)),
19       "#DOFs do not match #entities");
20   // The coefficients of a FE function in the quadratic Lagrangian FE
       space with
21   // respect to the nodal basis are simply its values at the nodes and
       the
22   // midpoints of the edges.
23   Eigen::VectorXd nu(dh_quad.NumDofs());
24   // Visit nodes (codimension =2) and copy values
25   for (const lf::mesh::Entity *node : mesh.Entities(2)) {
26     std::span<const gdof_idx_t> lf_dof_idx{dh_lin.GlobalDofIndices(*node)};
27     LF_ASSERT_MSG(lf_dof_idx.size() == 1,
28                   "Exactly one LFE basis functiona at a node");
```

```
29      LF_ASSERT_MSG(lf_dof_idx[0] < dh_lin.NumDofs(), "LFE dof number too large");
30      std::span<const gdof_idx_t> qf_dof_idx{dh_quad.GlobalDofIndices(*node)};
31      LF_ASSERT_MSG(lf_dof_idx.size() == 1,
32                    "Exactly one QFE basis functiona at a node");
33      LF_ASSERT_MSG(qf_dof_idx[0] < dh_quad.NumDofs(),
34                    "QFE dof number too large");
35      nu[qf_dof_idx[0]] = mu[lf_dof_idx[0]];
36    }
37    // Run through all edges
38    for (const lf::mesh::Entity *edge : mesh.Entities(1)) {
39      // Obtain global numbers of LFE DOFs associated with the endpoints
          of
40      // the edge, that is, of those DOFs covering the edge.
41      std::span<const gdof_idx_t> lf_dof_idx{dh_lin.GlobalDofIndices(*edge)};
42      LF_ASSERT_MSG(lf_dof_idx.size() == 2,
43                    "Exactly two basis functions must cover an edge");
44      LF_ASSERT_MSG((lf_dof_idx[0] < dh_lin.NumDofs()) &&
45                        (lf_dof_idx[1] < dh_lin.NumDofs()),
46                    "LFE dof numbers too large");
47      // Obtain global number of QFE basis function associated with the
          edge
48      std::span<const gdof_idx_t> qf_dof_idx{
49          dh_quad.InteriorGlobalDofIndices(*edge)};
50      LF_ASSERT_MSG(qf_dof_idx.size() == 1,
51                    "A single QFE basis function is associated to an edge!");
52      LF_ASSERT_MSG(qf_dof_idx[0] < dh_quad.NumDofs(),
53                    "QFE dof number too large");
54
55      // Value of p.w.  linear continuous FE function at the midpoint of
          the edge
56      const double mp_val = (mu[lf_dof_idx[0]] + mu[lf_dof_idx[1]]) / 2.0;
57      // Store value as appropriate coefficient
58      nu[qf_dof_idx[0]] = mp_val;
59    }
60    return nu;
61  }
```

SOLUTION of (3-18.f):

We obtain a basis $\widehat{\mathfrak{B}} := \left\{\widetilde{b}_h^1, \ldots, \widetilde{b}_h^M\right\}$ of the space $\widehat{\mathcal{S}}_2^0(\mathcal{M})$ by removing those standard nodal basis functions of $\mathcal{S}_2^0(\mathcal{M})$ that are associated with nodes of the mesh. The remaining nodal basis functions are the so-called **edge bubbles** and they form a basis of $\widehat{\mathcal{S}}_2^0(\mathcal{M})$. Hence writing $\mathfrak{B}_2$ for the standard nodal basis of $\mathcal{S}_2^0(\mathcal{M})$, we can write

$$\widehat{\mathfrak{B}} = \{b_h \in \mathfrak{B}_2 : \ b_h \text{is associated with an edge of } \mathcal{M}\} \ .$$

Alternatively, one can define the functions $\widetilde{b}_h^i$ through their values in the set of interpolation nodes for $\mathcal{S}_2^0(\mathcal{M})$:

$$b_h^i \in \mathcal{S}_2^0(\mathcal{M}) \quad \text{and} \quad b_h^i(\boldsymbol{p}) = 0 \quad \forall \boldsymbol{p} \in \mathcal{V} \ , \quad b_h^i(\boldsymbol{m}_j) = \delta_{ij} \ , \quad i,j \in \{1,\ldots,N_E\} \ ,$$

where $\boldsymbol{m}_j$ is the midpoint of the j-th edge of $\mathcal{M}$. Both constructions show $M = N_E$.

HINT 1 for (3-18.g): The function `trfLinToQuad()` from Sub-problem (3-18.e) can be used. ⌟

HINT 2 for (3-18.g): An auxiliary function

```
template <typename SCALAR, typename SELECTOR>
void dropMatrixRowsColumns(
SELECTOR &&selectidx,
lf::assemble::COOMatrix<SCALAR> &A);
```

is provided in the file `hierarchicalerrorestimator.h`. Its `selectidx` argument is a predicate of type `std::function<bool(int)>`. Its other argument `A` holds a sparse matrix $\mathbf{A} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$ (for `SCALAR = double`), in LEHRFEM++'s internal triplet format. That matrix is transformed into another matrix $\mathbf{A}' \in \mathbb{R}^{N,N}$ according to

$$\left(\mathbf{A}'\right)_{i,j} = \begin{cases} \left(\mathbf{A}\right)_{i,j} & \text{, if } \texttt{selectidx(i)==}\textbf{false} \text{ and } \texttt{selectidx(j)==}\textbf{false}, \\ 0 & \text{, if } i \neq j, \text{ and } \texttt{selectidx(i)==}\textbf{true} \text{ or } \texttt{selectidx(j)==}\textbf{true}, \\ 1 & \text{, if } i = j, \text{ and } \texttt{selectidx(i)==}\textbf{true} \text{ or } \texttt{selectidx(j)==}\textbf{true}. \end{cases}$$

HINT 3 for (3-18.g): LEHRFEM++ **lf::assemble::COOMatrix** objects offer the method `lf::assemble` which may be convenient. ⌟

SOLUTION of (3-18.g):

Let the standard nodal basis $\mathfrak{B}_2$ of $\mathcal{S}_2^0(\mathcal{M})$ be partition into basis functions associated with nodes of the mesh and others associated with its edges: $\mathfrak{B}_2 = \mathfrak{B}_n \cup \mathfrak{B}_e$. Assume – and this is really the case in LEHRFEM++– that the elements of $\mathfrak{B}_n$ are numbered before those of $\mathfrak{B}_e$. This induces a partitioning of the $\mathcal{S}_2^0(\mathcal{M})$-Galerkin matrix $\mathbf{A}$ for the bilinear form a and of the load vector $\vec{\varphi}$ spawned by ℓ:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{nn} & \mathbf{A}_{en} \\ \mathbf{A}_{ne} & \mathbf{A}_{ee} \end{bmatrix} \quad , \quad \vec{\varphi} = \begin{bmatrix} \vec{\varphi}_n \\ \vec{\varphi}_e \end{bmatrix} .$$

Let $\vec{\mu}'$ be the $\mathcal{S}_2^0(\mathcal{M})$-coefficient vector of $u_h \in \mathcal{S}_1^0(\mathcal{M})$ represented by the coefficient vector μ in the nodal basis of $\mathcal{S}_1^0(\mathcal{M})$:

$$\vec{\mu} \xmapsto{\texttt{trfLinToQuad()}} \vec{\mu}' .$$

The effect of `dropMatrixRowsColumns()` can be expressed similarly:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{nn} & \mathbf{A}_{en} \\ \mathbf{A}_{ne} & \mathbf{A}_{ee} \end{bmatrix} \xmapsto{\texttt{dropMatrixRowsColumns()}} \mathbf{A}' = \begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{O} & \mathbf{A}_{ee} \end{bmatrix}$$

Since the trial and test space in (3.18.9) is spanned by the functions in $\mathfrak{B}_e$ the coefficient vector representation of $\widehat{w}_h$ with respect to the basis $\mathfrak{B}_e$ is

$$\vec{\omega}_e = \mathbf{A}_{ee}^{-1}\left(\vec{\varphi}_e - \begin{bmatrix} \mathbf{O} & \mathbf{I} \end{bmatrix} \mathbf{A}\vec{\mu}'\right) .$$

Hence, in the full basis, the function $\widehat{w}_h$ has the representation

$$\vec{\omega} = (\mathbf{A}')^{-1} \begin{bmatrix} \mathbf{O} & \mathbf{O} \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \left(\vec{\varphi} - \mathbf{A}\vec{\mu}'\right) .$$

This vector is returned by `compHierSurplusSolution()`.

C++ code 3.18.10: Implementation of `compHierSurplusSolution()` ↪ GitLab

```
2  template <typename MESHFUNCTION_ALPHA, typename MESHFUNCTION_F>
3  Eigen::VectorXd compHierSurplusSolution(
4      const MESHFUNCTION_ALPHA &mf_alpha, const MESHFUNCTION_F &mf_f,
5      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_lin_p,
6      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_quad_p,
7      const Eigen::VectorXd &mu) {
8    // References to FE space
9    const lf::uscalfe::FeSpaceLagrangeO2<double> &quad_space{*fes_quad_p};
10   const lf::uscalfe::FeSpaceLagrangeO1<double> &lfe_space{*fes_lin_p};
11   // Get references to DofHandlers
12   const lf::assemble::DofHandler &dh_quad{quad_space.LocGlobMap()};
13   const lf::assemble::DofHandler &dh_lfe{lfe_space.LocGlobMap()};
14   LF_ASSERT_MSG(dh_lfe.Mesh() == dh_quad.Mesh(),
15                 "DofHandlers must be based on the same mesh");
16   LF_ASSERT_MSG(dh_lfe.NumDofs() == mu.size(), "Vector length mismath");
17   // Underlying mesh
18   std::shared_ptr<const lf::mesh::Mesh> mesh_p{dh_lfe.Mesh()};
19   const lf::mesh::Mesh &mesh{*mesh_p};
20   LF_ASSERT_MSG(
21       (dh_lfe.NumDofs() == mesh.NumEntities(2)) &&
```

```
22            (dh_quad.NumDofs() == mesh.NumEntities(2) + mesh.NumEntities(1)),
23        "#dof mismatch");
24    // Dimension of the quadratic finite element space
25    const lf::base::size_type N_qdofs(dh_quad.NumDofs());
26    // Vector for returning the result
27    Eigen::VectorXd quad_surplus(N_qdofs);
28    // Step I: Assemble full Galerkin matrix for quadratic FE
29    // Object for sparse matrix to be filled by cell-oriented assembly
30    lf::assemble::COOMatrix<double> A(N_qdofs, N_qdofs);
31    // Provider object for element matrices for scalar linear second-order
          pure
32    // diffusion operator with variable diffusion coefficient.  Uses
          numerical
33    // quadrature of order 3 and, thus, computes exact element matrices
          for
34    // locally constant diffusion coefficient.
35    lf::fe::DiffusionElementMatrixProvider<double, MESHFUNCTION_ALPHA>
          elmat_builder(fes_quad_p, mf_alpha);
36    // Invoke cell-oriented assembly of the finite-element Galerkin matrix
37    lf::assemble::AssembleMatrixLocally(0, dh_quad, dh_quad, elmat_builder, A);
38
39    // II: Assembly of right-hand-side vector in quadratic FE space
40    Eigen::Matrix<double, Eigen::Dynamic, 1> phi(N_qdofs);
41    phi.setZero();
42    // Assemble right-hand side vector depending on the source function f.
43    // Initialize object taking care of local computations on all cells.
44    lf::uscalfe::ScalarLoadElementVectorProvider<double, MESHFUNCTION_F>
          elvec_builder(fes_quad_p, mf_f);
45    // Invoke assembly on cells (codim == 0)
46    AssembleVectorLocally(0, dh_quad, elvec_builder, phi);
47
48    // III.:  Compute residual vector in quadratic FE space
49    // We take for granted that the components of mu corresponding to
          nodes on the
50    // boundary vanish
51    Eigen::VectorXd residual_quad =
          phi + A.MatVecMult(-1.0, trfLinToQuad(fes_lin_p, fes_quad_p, mu));
52
53    // predicate for selecting dofs associated with interior edges
54    // Create a predicate selecting edges on the boundary
55    lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
          lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
56    // Flag vector indicating inactive edges
57    std::vector<bool> dropped_ed(N_qdofs, false);
58    for (const lf::mesh::Entity *edge : mesh.Entities(1)) {
59      std::span<const lf::assemble::gdof_idx_t> qf_dof_idx{
            dh_quad.GlobalDofIndices(*edge)};
60      LF_ASSERT_MSG(qf_dof_idx.size() == 3,
61                    "Three QFE basis functions covering an edge!");
62      // The two first indices correspond to the basis functions
            associated with
63      // the endpoints of the edge, the third index is the number of the
            basis
64      // function associated with the edge itself.
65      dropped_ed[qf_dof_idx[0]] = true;
66      dropped_ed[qf_dof_idx[1]] = true;
67      if (bd_flags(*edge)) {
68        dropped_ed[qf_dof_idx[2]] = true;
69      }
70    }
```

```
76
77    // Annihilate inactive components of residual.  We have to do this,
          because we
78    // solve a linear system with the full-size Galerkin matrix.
79    for (int j = 0; j < N_qdofs; ++j) {
80      if (dropped_ed[j]) {
81        residual_quad[j] = 0.0;
82      }
83    }
84    // Set imactive rows and columns of A to zero
85    dropMatrixRowsColumns<double>(
86        [&dropped_ed](lf::assemble::gdof_idx_t idx) -> bool {
87          return dropped_ed[idx];
88        },
89        A);
90    // Solve linear system with surplus reduced Galerkin matrix
91    // Convert COO matrix A into CRS format using Eigen's internal
          conversion
92    // routines.
93    Eigen::SparseMatrix<double> A_crs = A.makeSparse();
94    // Solve linear system using Eigen's sparse direct elimination
95    // Examine return status of solver in case the matrix is singular
96    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
97    solver.compute(A_crs);
98    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
99    quad_surplus = solver.solve(residual_quad);
100   LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
101   return quad_surplus;
102 }
```

SOLUTION of (3-19.a):

All plots are doubly logarithmic and the neat alignment of the points corresponding to the norms of the FE discretization error on different meshes reveals empiric algebraic convergence for $h \to 0$.

We note that

- Experiments (I) and (II) have an analytic ("arbitrarily smooth") solution and
- that they involve a domain Ω on which the Dirichlet problem for $-\Delta$ is 2-regular.

Hence

- the convergemce of the $H^1(\Omega)$-seminorm (the energy norm) of the FE discretization error for both experiments will be solely determined by the polynomial degree of the Lagrangian finite element space. The rate of algebraic convergence will agree with that degree, recall [Lecture → Thm. 3.3.5.6].
- the $L^2(\Omega)$-norm of the FE discretization error will converge algebraically with a rate $+1$ the rate observed for the $H^1(\Omega)$-seminorm, see [Lecture → § 3.6.3.7].

On the L-shaped domain the manufactured solution has a singularity at $\mathbf{0}$ and does not even belong to $H^2(\Omega)$. This will limit algebraic convergence to a rate < 1 for the $H^1(\Omega)$-seminorms, regardless of the polynomial degree used for finite-element approximation. However, the energy norm of the discretization error will be smaller for $\mathcal{S}^0_{2,0}(\mathcal{M})$ than for $\mathcal{S}^0_{1,0}(\mathcal{M})$.

Plot A: We observe algebraic convergence with rate < 1. So we face the case of an experiment on the L-shaped domain and the recorded norm is the energy norm/$H^1(\Omega)$-seminorm. The error norms are consistently smaller than those in Plot D.

Exp. (V)

Plot B: Algebraic convergence with rate 1, which can only be expected for the energy norm of the discretization error in the case of $\mathcal{S}^0_{1,0}(\mathcal{M})$.

Exp. (III)

Plot C: Algebraic convergence with rate 2, which can be observed for the energy norm of the discretization error for $\mathcal{S}^0_{2,0}(\mathcal{M})$ **or** the $L^2(\Omega)$-norm of the discretization error for $\mathcal{S}^0_{1,0}(\mathcal{M})$.

Exp. (IV), (I)

Plot D: see discussion for Plot A

Exp. (VI)

Plot E: Algebraic convergence with rate 2, see discussion for Plot C

Exp. (IV), (I)

Plot F: Algebraic convergence with rate 3, which can only be achived by the $L^2(\Omega)$-norm of the discretization error for $\mathcal{S}^0_{2,0}(\mathcal{M})$.

Exp. (II)

SOLUTION of (3-19.b):

We have to choose $f := -\Delta u$ and $g := u|_{\partial\Omega}$.

- Since $-\Delta = -\frac{\partial^2}{\partial x_1^2} - \frac{\partial^2}{\partial x_1^2}$ we find

$$f(\boldsymbol{x}) = 2x_2(1-x_2) + 2x_1(1-x_1) \quad , \quad \boldsymbol{x} \in \Omega \, .$$

- Obviously, for $\Omega =]0,1[^2$ we have $u|_{\partial\Omega} = 0$, which means $g \equiv 0$.

SOLUTION of (3-19.c):

The function $\boldsymbol{x} \mapsto u(\boldsymbol{x})$ from (3.19.2) is a *polynomial* of degree 4: $u \in \mathcal{P}_4(\mathbb{R}^2)$, and vanishes on $\partial\Omega$

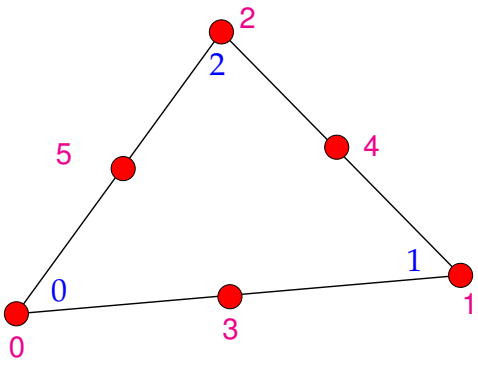
The smallest Lagrangian finite-element space that contains $\mathcal{P}_4(\mathbb{R}^2) \cap C_0^0(\overline{\Omega})$ for any mesh $\mathcal{M}$ is $V_h = \mathcal{S}_{4,}^0$

.

HINT 1 for (3-2.a): The **lf::assemble::DofHandler** permits you to loop over the global shape functions and retrieve the entities they are associated with. ⌟

HINT 2 for (3-2.a): The global shape functions of $\mathcal{S}_2^0(\mathcal{M})$ are a cardinal basis with respect to the set of interpolation nodes introduced in [Lecture → Ex. 2.6.1.2], see [Lecture → Fig. 103]. Therefore, in order to realize the interpolation operator you only need to *copy* the f-values in interpolations nodes in the corresponding components of the coefficient vector. ⌟

SOLUTION of (3-2.a):



The interpolation nodes for $\mathcal{S}_2^0(\mathcal{M})$ are both the nodes of $\mathcal{M}$ and the midpoints of edges. Consequently, the six local interpolation nodes are the three vertices of a triangle and the three midpoints of its edges. The local numbering convention of LEHRFEM++ is given in [Lecture → § 2.7.2.14] and [Lecture → Ex. 2.7.4.12].

The coordinates of the local interpolation nodes on the reference "unit" triangle are

$$\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}\frac{1}{2}\\0\end{bmatrix}, \begin{bmatrix}\frac{1}{2}\\\frac{1}{2}\end{bmatrix}, \begin{bmatrix}0\\\frac{1}{2}\end{bmatrix},$$

These reference coordinates have to be passed to the `GLobal()` method of the **lf::geometry::Geometry** object belonging to a cell in order to retrieve the global "world" coordinates of the interpolation nodes. At these points the functor has to be evaluated.

Note that interpolation is not an "assembly-style" operation; we do **not add** up cell contributions, but just **set** vector entries.

A possible implementation can be found in **??**.

C++ code 3.2.2: Sub-problem (3-2.a): function `interpolateOntoQuadFE()` → GitLab

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd interpolateOntoQuadFE(const lf::assemble::DofHandler &dofh,
4                                        FUNCTOR &&f) {
5    // Obtain a pointer to the mesh object
6    auto mesh = dofh.Mesh();
7    const size_type N_dofs(dofh.NumDofs());
8    // variable for returning result vector
9    Eigen::VectorXd result = Eigen::VectorXd::Zero(N_dofs);
10   // Reference coordinates of the interpolation nodes of the triangle
11   Eigen::Matrix<double, 2, 6> refnodes(2, 6);
12   refnodes << 0.0, 1.0, 0.0, 0.5, 0.5, 0.0, 0.0, 0.0, 1.0, 0.0, 0.5, 0.5;
13   // Loop over the cells of the mesh (codim-0 entities)
14   for (const lf::mesh::Entity *cell : mesh->Entities(0)) {
15     LF_ASSERT_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
16                   "Implemented for triangles only");
17     // Fetch pointer to asscoiated geometry object
18     const lf::geometry::Geometry *geom = cell->Geometry();
19     // Obtain actual coordinates of interpolation nodes
20     Eigen::MatrixXd nodes{geom->Global(refnodes)};
21     // get local to global index map for the current cell
22     auto glob_ind = dofh.GlobalDofIndices(*cell);
23     // Loop over local interpolation nodes
24     for (int i = 0; i < 6; i++) {
25       // update the result vector
26       result(glob_ind[i]) = f(nodes.col(i));
27     }
28   }
```

```
29    return result;
30  }
```

Remark. LEHRFEM++ also supplies the function `lf::fe::NodalProjection`, which, when supplied with an **lf::uscalfe::FeSpaceLagrangeO2** object evaluates the finite-element interpolant of a **MeshFu**

HINT 1 for (3-2.c): Recall from [Lecture → Section 3.1] the arguments that suggested that the energy norm provides a relevant norm for measuring the discretization error. ⌟

SOLUTION of (3-2.c):

Since u was said to be "smooth", that is, $u \in C^\infty(\overline{\Omega})$, we can take for granted $u \in H^3(\Omega)$. In this case Thm. 3.2.4 gives

$$\|u - \mathsf{I}_2 u\|_{H^1(\Omega)} \leq C h_{\mathcal{M}}^2 |u|_{H^3(\Omega)} \quad \forall u \in H^3(\Omega) \,, \tag{3.2.5}$$

with $C > 0$ depending only on the shape regularity measure of $\mathcal{M}$.

The next step is the "$a^2 - b^2 = (a+b)(a-b)$-formula" for symmetric bilinear forms:

$$\begin{aligned}
|\mathsf{a}(u,u) - \mathsf{a}(\mathsf{I}_2 u, \mathsf{I}_2 u)| &= |\mathsf{a}(u + \mathsf{I}_2 u, u - \mathsf{I}_2 u)| \\
&\leq \|u + \mathsf{I}_2 u\|_{\mathsf{a}} \|u - \mathsf{I}_2 u\|_{\mathsf{a}} && (\text{Cauchy-Schwarz}) \\
&\leq (\|u\|_{\mathsf{a}} + \|\mathsf{I}_2 u\|_{\mathsf{a}}) \|u - \mathsf{I}_2 u\|_{\mathsf{a}} && (\Delta\text{-ineq.}) \\
&= (\|u\|_{\mathsf{a}} + \|\mathsf{I}_2 u - u + u\|_{\mathsf{a}}) \|u - \mathsf{I}_2 u\|_{\mathsf{a}} && (\text{Add zero trick}) \\
&\leq (2\|u\|_{\mathsf{a}} + \|u - \mathsf{I}_2 u\|_{\mathsf{a}}) \|u - \mathsf{I}_2 u\|_{\mathsf{a}} && (\Delta\text{-ineq. and } \|-a\| = \|a\|) \\
&\leq C_1 |u|_{H^1(\Omega)} |u|_{H^3(\Omega)} h_{\mathcal{M}}^2 + C_2 |u|_{H^3(\Omega)}^2 h_{\mathcal{M}}^4 && \text{Use theorem Thm. 3.2.4} \\
&\leq C_3 |u|_{H^1(\Omega)} |u|_{H^3(\Omega)} h_{\mathcal{M}}^2 \,. &&
\end{aligned}$$

Hence, we can choose $T(h) := h^2$.

HINT 1 for (3-2.d): It is advisable to assemble the Galerkin matrix for $\mathsf{a}(\cdot,\cdot)$ based on $\mathcal{S}_2^0(\mathcal{M})$ and store it in a data member of type **Eigen::SparseMatrix<double>**. ⌟

HINT 2 for (3-2.d): Use the function `interpolateOntoQuadFE()` class developed in Sub-problem (3

SOLUTION of (3-2.d):

The main function of the constructor is to assembly the Galerkin matrix for the bilinear form a using different local assembly objects, whose type is given as a template argument. [Lecture → Ex. 2.7.4.39] gives details about the implementation in LEHRFEM++.

C++ code 3.2.6: Constructor of class QFEProvideTester ↬ GitLab

```
2  template <lf::assemble::EntityMatrixProvider ENTITY_MATRIX_PROVIDER>
3  QFEProviderTester<ENTITY_MATRIX_PROVIDER>::QFEProviderTester(
4      const lf::assemble::DofHandler &dofh,
5      ENTITY_MATRIX_PROVIDER &element_matrix_provider)
6      : dofh_(dofh), element_matrix_provider_(element_matrix_provider) {
7    // Assemble the Galerkin matrix and store it into A_
8    const lf::base::size_type N_dofs(dofh.NumDofs());
9    lf::assemble::COOMatrix<double> mat(N_dofs, N_dofs);
10   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, element_matrix_provider,
11                                       mat);
12   A_ = Eigen::SparseMatrix<double>(mat.makeSparse());
13 }
```

We use the fact

$$a(v_h, v_h) = \vec{\eta}^\top \mathbf{A}\vec{\eta} \quad \forall v_h \in \mathcal{S}_2^0(\mathcal{M}) ,$$

if $\mathbf{A}$ is the Galerkin matrix for a and $\vec{\eta}$ stands for the basis expansion coefficient vector with respect to the set of global shape functions used for assembly.

C++ code 3.2.7: Implementation of member function `energyOfInterpolant()` of class QFEProvideTester ↬ GitLab

```
2  template <lf::assemble::EntityMatrixProvider ENTITY_MATRIX_PROVIDER>
3  template <typename FUNCTOR>
4  double QFEProviderTester<ENTITY_MATRIX_PROVIDER>::energyOfInterpolant(
5      FUNCTOR &&u) const {
6    double energy = 0.0;
7    Eigen::VectorXd eta = DebuggingFEM::interpolateOntoQuadFE(dofh_, u);
8    // Compute ηᵀ A η
9    energy = eta.dot(A_ * eta);
10   return energy;
11 }
```

SOLUTION of (3-2.e):

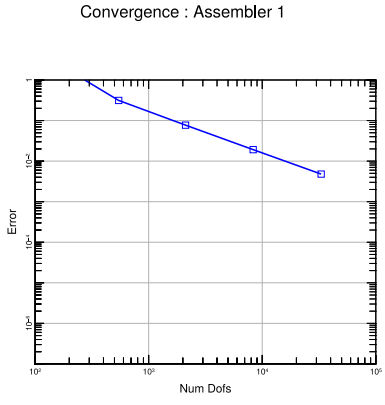
This should be the output of the executable:

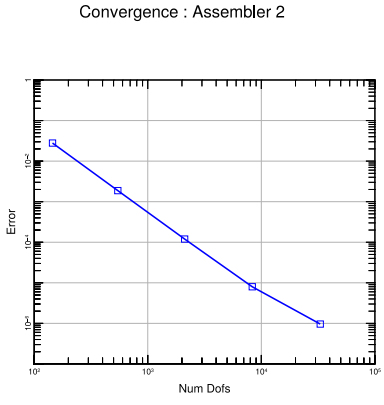
N	Assembler 1	Assembler 2	Assembler 3
145	1.32498	0.0278714	0.0289051
545	0.313257	0.00187436	0.00579826
2113	0.077044	0.000119856	0.0155225
8321	0.0191794	7.98826e−06	0.0612379
33025	0.00479006	9.61216e−07	0.244454

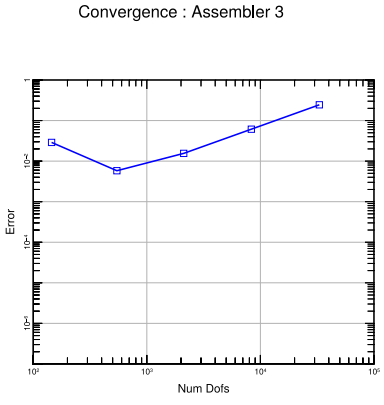
Note that the deviations err_i are tabulated versus the dimension N of the finite element space. Since we deal with 2D triangular meshes generated by regular refinement, we can take for granted $h \sim N^{-1/2}$. This means that the mesh width is about halved when moving from one row of the table to the next.

- `Assembler 1`: the deviation drops by about a factor of 4 in every row from top to bottom: $\mathrm{err}_i = O($
- `Assembler 2`: Same: $\mathrm{err}_i = O(h^2)$, but the error is significantly smaller than for `Assembler 1`.
- `Assembler 3`: the deviation fails to decrease as $h \to 0$.

Convergence plots for Sub-problem (3-2.e):







SOLUTION of (3-2.f):

We see that both **LocalLaplaceQFE1** and **LocalLaplaceQFE2** ensures convergence with rate 1 with respect to the number of degrees of freedom, which is rate 2 for the meshwidth as predicted in Sub-problem (3-2.c).

Moreover, **LocalLaplaceQFE2** achieves even convergence *with rate* 2, which is way more than predicted by the crude estimate of Sub-problem (3-2.c).

Obviously, **LocalLaplaceQFE3** fails to make the FEM converge so it must be flawed.

Unfortunately, we cannot decide whether **LocalLaplaceQFE1** is flawed. We can be rather certain that **LocalLaplaceQFE2** provides the correct implementation.

HINT 1 for (3-20.a): Which equations will u and $\mathbf{j}$ satisfy, if $J\big((u,\mathbf{j})\big) = 0$? ⌟

SOLUTION of (3-20.a):

Since $J((v,\mathbf{q})) \geq 0\ \forall (v,\mathbf{q}) \in V$ we conclude that , u and $\mathbf{j}$ satisfying $J(v,\mathbf{q}) = 0$ are necessarily minimizers of J:

$$J(u,\mathbf{j}) = 0 \quad \Rightarrow \quad (u,\mathbf{j}) \in \operatorname{argmin}_{(v,\mathbf{q})\in V} J\big((v,\mathbf{q})\big)\ . \tag{3.20.5}$$

It is clear from the definition of J:

$$J(v,\mathbf{q}) = 0 \quad \Rightarrow \quad \begin{cases} \sqrt{\alpha}\,\mathbf{grad}\,v = \frac{1}{\sqrt{\alpha}}\mathbf{q} \\ \operatorname{div}\mathbf{q} = f \end{cases} \quad \text{in} \quad \Omega\ . \tag{3.20.6}$$

By applying the divergence operator to the first equation of (**??**), since $v|_{\partial\Omega} = 0$ we have the following second-order scalar linear elliptic Dirichlet boundary value problem on Ω for the u-component solution to (**??**):

$$\begin{cases} \operatorname{div}(\alpha\,\mathbf{grad}\,u) = f & \text{on } \Omega, \\ u = 0 & \text{on } \partial\Omega. \end{cases} \tag{3.20.7}$$

The boundary condition for u is built into the space V, see (3.20.4).

The Dirichlet problem (**??**) always has a solution, from which we obtain $(u,\mathbf{j})$ with $J(u,\mathbf{q}) = 0$. This confirms that the minimal value 0 is attained by J.

SOLUTION of (3-20.b):

This problem can be tackled in two ways:

(I) We can rely on the calculus of variations approach:

To state the variational formulation of our problem, we write:

$$\lim_{t\to 0} \frac{J((v+tw,\mathbf{q}+t\mathbf{p}))-J((v,\mathbf{q}))}{t} = 0, \quad \forall (w,\mathbf{p}) \in V\,.$$

We obtain

$$\lim_{t\to 0} \frac{J((v+tw,\mathbf{q}+t\mathbf{p}))-J((v,\mathbf{q}))}{t} =$$
$$= \int_\Omega \alpha^{-1}\mathbf{q}\cdot\mathbf{p} + \alpha\,\mathbf{grad}\,v\cdot\mathbf{grad}\,w - \mathbf{grad}\,w\cdot\mathbf{q} - \mathbf{grad}\,v\cdot\mathbf{p} + \operatorname{div}\mathbf{q}\operatorname{div}\mathbf{p} - \operatorname{div}\mathbf{p}\, f\,\mathrm{d}\boldsymbol{x} = 0,$$

which leads to the following variational equation

$$\mathbf{a}_F((v,\mathbf{q}),(w,\mathbf{p})) - \ell((w,\mathbf{p})) = 0 \quad \forall (w,\mathbf{p}) \in V, \tag{3.20.8}$$

where

$$\ell((w,\mathbf{p})) = \int_\Omega \operatorname{div}\mathbf{p}\, f\,\mathrm{d}\boldsymbol{x}\,,$$

and $\mathbf{a}_F$ is a symmetric bilinear form defined by

$$\mathbf{a}_F((v,\mathbf{q}),(w,\mathbf{p})) = \int_\Omega \alpha^{-1}\mathbf{q}\cdot\mathbf{p} + \alpha\,\mathbf{grad}\,v\,\mathbf{grad}\,w - \mathbf{grad}\,w\cdot\mathbf{q} - \mathbf{grad}\,v\cdot\mathbf{p} + \operatorname{div}\mathbf{q}\operatorname{div}\mathbf{p}\,\mathrm{d}\boldsymbol{x}\,. \tag{3.20.9}$$

(II) We can start from the observation that (3.20.2) is a quadratic minimization problem:

$$J((v,\mathbf{q})) = \tfrac{1}{2}\mathbf{a}_F((v,\mathbf{q}),(v,\mathbf{q})) - \ell((v,\mathbf{q})) + \|f\|^2_{L^2(\Omega)}\,. \tag{3.20.10}$$

where $\mathbf{a}_F$ and ℓ are as in (**??**) and (**??**). The identity (**??**) emerges from straightforward elementary computations.

HINT 1 for (3-20.c): Use the two versions of the Cauchy-Schwarz inequality

$$\int_\Omega v\,w\,\mathrm{d}\boldsymbol{x} \le \|v\|_{L^2(\Omega)}\,\|w\|_{L^2(\Omega)} \quad \forall v,w \in L^2(\Omega)\,, \tag{3.20.12}$$

$$\left|\sum_{i=1}^n \alpha_i\beta_i\right| \le \left(\sum_{i=1}^n \alpha_i^2\right)^{\frac{1}{2}} \cdot \left(\sum_{i=1}^n \beta_i^2\right)^{\frac{1}{2}} \quad \forall \alpha_i,\beta_i \in \mathbb{R}\,. \tag{3.20.13}$$

⌟

SOLUTION of (3-20.c):

We want to prove the continuity of $\mathbf{a}_F$ w.r.t. the energy norm $\|\cdot\|_A$, that is

$$|\mathbf{a}_F((v,\mathbf{q}),(w,\mathbf{p}))|^2 \le C\|(v,\mathbf{q})\|_A^2\|(w,\mathbf{p})\|_A^2 \quad \forall (v,\mathbf{q}),(w,\mathbf{p}) \in V\ . \tag{3.20.14}$$

By applying the triangle inequality and the Cauchy-Schwarz inequality to $\mathbf{a}_F$ in (**??**), we obtain

$$\begin{aligned}|\mathbf{a}_F((v,\mathbf{q}),(w,\mathbf{p}))| \le C_1\Big(&\|\mathbf{q}\|_{L^2(\Omega)}\|\mathbf{p}\|_{L^2(\Omega)} + \|\operatorname{\mathbf{grad}} v\|_{L^2(\Omega)}\|\operatorname{\mathbf{grad}} w\|_{L^2(\Omega)} + \\ &\|\mathbf{q}\|_{L^2(\Omega)}\|\operatorname{\mathbf{grad}} w\|_{L^2(\Omega)} + \|\operatorname{\mathbf{grad}} v\|_{L^2(\Omega)}\|\mathbf{p}\|_{L^2(\Omega)} + \\ &\|\operatorname{div}\mathbf{q}\|_{L^2(\Omega)}\|\operatorname{div}\mathbf{p}\|_{L^2(\Omega)}\Big)\ ,\end{aligned}$$

where

$$C_1 = \max\{1, \|\alpha^{1/2}\|_{L^\infty(\Omega)}, \|\alpha^{-1/2}\|_{L^\infty(\Omega)}\}\ .$$

Since, by (**??**), $(A+B+C+D)^2 \le 4(A^2+B^2+C^2+D^2)$ for any real numbers A,B,C, and D, the following inequality holds:

$$\begin{aligned}|\mathbf{a}_F((v,\mathbf{q}),(w,\mathbf{p}))|^2 \le C\Big(&\|\mathbf{q}\|^2_{L^2(\Omega)}\|\mathbf{p}\|^2_{L^2(\Omega)} + \|\operatorname{\mathbf{grad}} v\|^2_{L^2(\Omega)}\|\operatorname{\mathbf{grad}} w\|^2_{L^2(\Omega)} + \\ &\|\mathbf{q}\|^2_{L^2(\Omega)}\|\operatorname{\mathbf{grad}} w\|^2_{L^2(\Omega)} + \|\operatorname{\mathbf{grad}} v\|^2_{L^2(\Omega)}\|\mathbf{p}\|^2_{L^2(\Omega)} + \\ &\|\operatorname{div}\mathbf{q}\|^2_{L^2(\Omega)}\|\operatorname{div}\mathbf{p}\|^2_{L^2(\Omega)}\Big)\end{aligned}$$

for $C = 4C_1^2$, which proves (**??**).

SOLUTION of (3-20.d):

We follow the considerations of [Lecture → Section 1.3.1]: We can use any finite element space for the Galerkin discretization of the minimization problem (3.20.2)/the FOSLS variational problem (**??**) for which the energy norm (3.20.11) is well defined, that is, finite.

In particular, by virtue of the trivial estimate $\|\mathrm{div}\,\mathbf{q}\|_{L^2(\Omega)} \leq \|\mathbf{q}\|_{H^1(\Omega)}$, this is the case for $(v, \mathbf{q}) \in H^1_0(\Omega)$ of which the proposed finite element space is a subspace.

Remark. It turns out that the continuity of v and the continuity of normal components of $\mathbf{q}$ are the key properties of valid arguments of a_F.

HINT 1 for (3-20.e): Base your arguments on Thm. 3.20.15 and the assertion proved in Sub-problem (3-2

SOLUTION of (3-20.e):

To begin with, we conclude from Thm. 3.20.15 that the bilinear form a_F is s.p.d. and, hence, induces a norm $\|\cdot\|_a$, which is *equivalent* to the energy norm $\|\cdot\|_A$.

Therefore we can apply Cea's lemma [Lecture → Thm. 3.1.3.7] and we obtain

$$\|(u-u_h, \mathbf{j}-\mathbf{j}_h)\|_a^2 = \inf_{(v_h,\mathbf{q}_h)\in V_{0,h}} \|(u-v_h, \mathbf{j}-\mathbf{q}_h)\|_a^2 \,.$$

We continue based on the result of Sub-problem (3-20.c):

$$\begin{aligned}\|(u-u_h, \mathbf{j}-\mathbf{j}_h)\|_A^2 &\leq C \inf_{(v_h,\mathbf{q}_h)\in V_{0,h}} \left(\|\mathbf{grad}(u-v_h)\|_{L^2(\Omega)}^2 + \|\operatorname{div}(\mathbf{j}-\mathbf{q}_h)\|_{L^2(\Omega)}^2 + \|\mathbf{j}-\mathbf{q}_h\|_{L^2(\Omega)}^2 \right) \\ &\leq C \inf_{v_h\in\mathcal{S}_2^0(\mathcal{M})} \|\mathbf{grad}(u-v_h)\|_{L^2(\Omega)}^2 + \inf_{\mathbf{q}_h\in(\mathcal{S}_2^0(\mathcal{M}))^2} \|\mathbf{j}-\mathbf{q}_h\|_{H^1(\Omega)}^2 \,.\end{aligned}$$

Then, to describe qualitatively and quantitatively the asympotic convergence of $\|u-u_h\|_{H^1(\Omega)}$ and $\|\operatorname{div}(\mathbf{j}$ for $N\to\infty$, we rely on [Lecture → Thm. 3.3.5.6].

Since the exact solutions u and $\mathbf{j}$ are smooth for this problem, by [Lecture → Thm. 3.3.5.6] we can rely on the following best approximation estimates for quadratic Lagrangian finite elements ($p=2$):

$$\inf_{v_h\in\mathcal{S}_2^0(\mathcal{M})} \|u-v_h\|_{H^1(\Omega)} \leq C\frac{h_{\mathcal{M}}^2}{2}\|u\|_{H^3(\Omega)} \,,$$

$$\inf_{\mathbf{q}_h\in\mathcal{S}_2^0(\mathcal{M})} \|\operatorname{div}(\mathbf{j}-\mathbf{q}_h)\|_{L^2(\Omega)} \leq \inf_{\mathbf{q}_h\in\mathcal{S}_2^0(\mathcal{M})} \|\mathbf{j}-\mathbf{q}_h\|_{H^1(\Omega)} \leq C\frac{h_{\mathcal{M}}^2}{2}\|\mathbf{j}\|_{H^3(\Omega)} \,,$$

with constants depending only on the shape-regularity measure of the meshes.

As $N\approx h_{\mathcal{M}}^{-2}$, this means that we have the following asymptotic convergence for $N\to\infty$:

$$\|u-u_h\|_{H^1(\Omega)} = O(N^{-1}) \,,$$

$$\|\operatorname{div}(\mathbf{j}-\mathbf{j}_h)\|_{L^2(\Omega)} = O(N^{-1}) \,.$$

In each case we expect algebraic convergence with rate $(-)1$.

SOLUTION of (3-21.a):

I. We follow the standard approach explained in detail in [Lecture → Section 1.5.3]. We first test with $v \in C_0^\infty(\Omega)$, perform integration by parts using [Lecture → Thm. 1.5.2.7] noticing that no boundary terms will emerge. This yields the PDE

$$-\Delta u = f \quad \text{in} \quad \Omega \ . \tag{3.21.2}$$

Next, we test with general $v \in C^\infty(\overline{\Omega})$ and use (**??**). Thus integration by parts makes it possible to isolate boundary terms and we arrive at the boundary conditions

$$\mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) + \kappa(\boldsymbol{x}) u(\boldsymbol{x}) = h(\boldsymbol{x}) \quad \text{on} \quad \partial\Omega \ . \tag{3.21.3}$$

II. An alternative approach just relies on "backward matching". We notice that (3.21.1) agrees with [Lecture → Eq. (1.8.0.8)] with $\Psi(u) = \kappa(\boldsymbol{x})u$. Thus the strong/PDE form of (3.21.1) is [Lecture → Eq. (1.8.0.7)] with that choice of Ψ. This is the same as (**??**) and (**??**).

SOLUTION of (3-21.j):

The impleentation of `error_I()` makes use of **MeshFunction**s, see the documentation and [Lecture → Suppl. 2.8.3.32] for explanations. The LEHRFEM++ helper function `lf::fe::integrateMeshFunct` can be used to integrate a mesh function over a collection of mesh entities, see its documentation for details.

Understanding **MeshFunction** and `lf::fe::integrateMeshFunction()` we infer from the code that

$$\text{Return value of } \texttt{error_I()} = \left| \int_{\partial\Omega} u(\boldsymbol{x}) - u_{h,\ell}(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) \right| .$$

HINT 1 for (3-21.k): It is useful to know the following special elliptic regularity result:

Lemma 3.21.16. Regularity of solutions of (3.21.1)

If $f \in H^1(\Omega)$, $h \in H^2(\partial\Omega)$, and Ω is the unit triangle, then the solution $u \in H^1(\Omega)$ of the linear variational problem will actually belong to $H^3(\Omega)$.

⌟

SOLUTION of (3-21.k):

As we saw in Sub-problem (3-21.j) the function `error_I()` returns the error of the linear output functional $F(u) := \int_{\partial\Omega} u(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})$. By virtue of the Cauchy-Schwarz inequality and the multiplicative trace estimate [Lecture → Thm. 1.9.0.10] this functional is clearly continuous on $H^1(\Omega)$. Hence, we can apply the duality estimates of [Lecture → Section 3.6.1].

Theorem [Lecture → Thm. 3.6.1.7]. Duality estimate for linear functional output

Define the ***dual solution*** $g_F \in V_0$ *belonging to the linear functional* $F : V_0 \to \mathbb{R}$ *as solution of the* ***dual variational problem***

$$g_F \in V_0: \quad \mathsf{a}(v, g_F) = F(v) \quad \forall v \in V_0 \,.$$

Then

$$|F(u) - F(u_h)| \le \|u - u_h\|_{\mathsf{a}} \inf_{v_h \in V_{0,h}} \|g_F - v_h\|_{\mathsf{a}} \,. \qquad \text{[Lecture → Eq. (3.6.1.14)]}$$

The dual variational problem agrees with (3.21.1) with right-hand side functional $v \mapsto \int_{\partial\Omega} v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})$, which is (3.21.1) with data $f \equiv 0$ and $h \equiv 1$. Obviously, the assumptions of **??** are satisfied and $g_F \in H^3(\Omega$

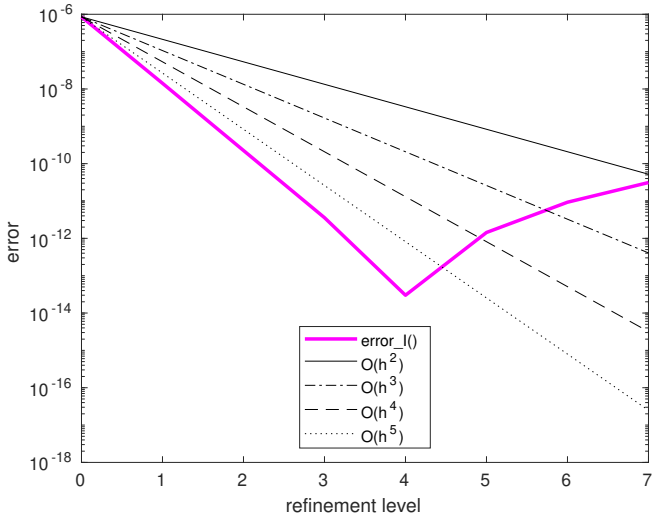
Of course, in the test setting we also have $u \in H^3(\Omega)$. Then we appeal to [Lecture → Thm. 3.3.5.6] and conclude

$$|F(u) - F(u_h)| \le \underbrace{\|u - u_h\|_{H^1(\Omega)}}_{=O(h^2)} \underbrace{\inf_{v_h \in \mathcal{S}_2^0(\mathcal{M})} \|g_F - v_h\|_{H^1(\Omega)}}_{=O(h^2)} = O(h^4) \quad \text{for} \quad h \to 0 \,.$$

Remark. This output functional has also been investigated in the homework problem 3-1 (Computing averages over the boundary).

The actual errors returned by `error_I()` in the setting of the numerical experiment are the following

level	error	$\log_2(\text{ratio})$
0	8.534e-07	
1	1.419e-08	5.9095
2	2.253e-10	5.9772
3	3.577e-12	5.9775
4	2.995e-14	6.8999
5	1.434e-12	5.5816
6	9.272e-12	2.6924
7	3.120e-11	1.7509



The third column of the error table records

$$\log_2 \frac{\text{error}(\ell - 1)}{\text{error}(\ell)} \,, \quad \ell = 1, \ldots, 7 \,.$$

Since in the course of regular refinement of a triangular mesh, the meshwidth is halved in every refinement step, $h_\ell \sim 2^{-\ell}$, this quantity can be regarded as an estimate of the algebraic rate of convergence. That estimate yields an empiric rate of 6 !

That rate (before the error is swamped by roundoff errors on fine meshes) is way better than predicted by theory. We witness a case of **superconvergence** for which an explanation is still lacking.

SOLUTION of (3-21.l):

Note that the results of the evaluation of two integrals are subtracted.

Return value of `error_IV()` $= \left| \|u_{h,\ell}\|^2_{L^2(\Omega)} - \|u\|^2_{L^2(\Omega)} \right| = \left| \int_\Omega u_{h,\ell}(\boldsymbol{x})^2 \, \mathrm{d}\boldsymbol{x} - \int_\Omega u(\boldsymbol{x})^2 \, \mathrm{d}\boldsymbol{x} \right|$.

SOLUTION of (3-21.m):

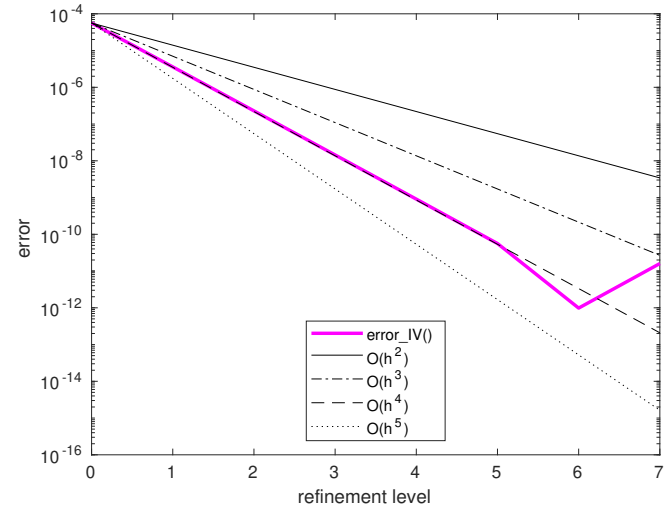
By a "binomial formula $a^2 - b^2 = (a+b)(a-b)$" for inner products we obtain

$$\|u_h\|^2_{L^2(\Omega)} - \|u\|^2_{L^2(\Omega)} = (u_h - u, u_h + u)_{L^2(\Omega)} \leq \|u_h - u\|_{L^2(\Omega)} \cdot \|u_h + u\|_{L^2(\Omega)} \,. \tag{3.21.18}$$

Here we used the Cauchy-Schwarz inequality. The first term has been estimated in Sub-problem (3-21.i) and behaves like $O(h^3)$ for $h \to 0$, whereas the second is merely bounded and may not tend to 0 for $h \to 0$. Summing up we obtain algebraic convergence like $O(h^3)$ for $h \to 0$.

The actual errors returned by `error_IV()` in the setting of the numerical experiment are the following

level	error	$\log_2(\text{ratio})$
0	5.608e-05	
1	3.592e-06	3.9648
2	2.283e-07	3.9754
3	1.441e-08	3.9855
4	9.059e-10	3.9922
5	5.615e-11	4.0119
6	9.827e-13	5.8364
7	1.600e-11	-4.0253



The measured data hint at an asymptotic behavior $|\|u_h\|^2_{L^2(\Omega)} - \|u\|^2_{L^2(\Omega)}| = O(h^4)$ for $h \to 0$. It seems that the estimates in (**??**) are too crude.

SOLUTION of (3-21.b):

Recall from [Lecture → § 2.7.5.27] that a local quadrature rule that is exact for piecewise polynomials of degree $\leq p$ has order $p+1$.

If $f \in \mathcal{S}_1^0(\mathcal{M})$ the integrand of $\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ is a polynomial of degree ≤ 3 on every cell of the mesh. In order to evaluate the local integrals exactly, the quadrature rule has to be of order 4 .

For $\kappa \in \mathcal{S}_1^0(\mathcal{M})\big|_{\partial\Omega}$ for $\int_{\partial\Omega} \kappa(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})$ we have to integrate piecewise polynomials of degree 5, which, again, requires a local quadrature rule of order 6 on the edges of the mesh that belong to $\partial\Omega$.

SOLUTION of (3-21.c):

According to (**??**) the formula for $h(\boldsymbol{x})$ is

$$h(\boldsymbol{x}) = \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) + u(\boldsymbol{x}) \,.$$

We have to compute the different ingredients:

- By the chain rule $\mathbf{grad}\, u(\boldsymbol{x}) = 2\boldsymbol{x}\exp(\|\boldsymbol{x}\|^2)$,
- The exterior unit normal vector field is piecewise constant

$$\boldsymbol{n}(\boldsymbol{x}) = \begin{cases} \begin{bmatrix} 0 \\ -1 \end{bmatrix} & \text{on} \quad \Gamma_1 \,, \\ \frac{1}{2}\sqrt{2}\begin{bmatrix} 1 \\ 1 \end{bmatrix} & \text{on} \quad \Gamma_2 \,, \\ \begin{bmatrix} -1 \\ 0 \end{bmatrix} & \text{on} \quad \Gamma_3 \,. \end{cases}$$

- On the different parts of $\partial\Omega$:

$$\text{on } \Gamma_1\text{:} \quad x_2 = 0 \quad , \quad \text{on } \Gamma_2\text{:} \quad x_1 + x_2 = 1 \quad , \quad \text{on } \Gamma_3\text{:} \quad x_1 = 0 \,.$$

Taking into account this information we compute

$$h(\boldsymbol{x}) = \begin{cases} \exp(\|\boldsymbol{x}\|^2) = \exp \\ (\sqrt{2}+1)\exp(x_1^2 \\ \exp(\|\boldsymbol{x}\|^2) = \exp \end{cases}$$

SOLUTION of (3-21.d):

Note that the **MeshFunction** `mf_fesolg` represents the gradient of the finite element solution $u_{h,\ell}$. Otherwise Code 3.21.4 is self-explanatory:

Return value of `error_II()`

$$= \|\mathbf{grad}\, u - \mathbf{grad}\, u_{h,\ell}\|_{L^2(\Omega)} = |u - u_{h,l}|_{H^1(\Omega)} = \left(\int_\Omega \|\mathbf{grad}(u - u_{h,\ell})\|^2 \, \mathrm{d}\boldsymbol{x} \right)^{\frac{1}{2}} .$$

SOLUTION of (3-21.e):

Let v_h^* be the best approximant

$$v_h^* = \operatorname*{argmin}_{v_h \in \mathcal{S}_2^0(\mathcal{M})} \|u - v_h\|_{H^1(\Omega)} . \tag{3.21.5}$$

Since $\boldsymbol{x} \mapsto u(\boldsymbol{x})$ is smooth (certainly $u \in H^2(\Omega)$), from [Lecture → Thm. 3.3.5.6] we know that on the given sequence of meshes

$$\|u - v_h^*\|_{H^1(\Omega)} = O(h^2) \quad \text{for} \quad h \to 0 . \tag{3.21.6}$$

The bilinear form of (3.21.1) is s.p.d. and induces the energy norm ($\kappa \equiv 1$)

$$\|v\|_a^2 := \int_\Omega \|\mathbf{grad}\, v\|^2 \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} v^2(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = |v|_{H^1(\Omega)}^2 + \|v\|_{L^2(\partial\Omega)}^2 , \quad v \in H^1(\Omega) ,$$

Firstly, by virtue of a Poincaré-Friedrichs-type estimate [Lecture → Thm. 1.8.0.26] the energy norm bounds the $H^1(\Omega)$-norm: with $C = C(\Omega) > 0$

$$\|v\|_{H^1(\Omega)} \le C\|v\|_a \quad \forall v \in H^1(\Omega) . \tag{3.21.7}$$

Secondly, by the multiplicative trace inequality [Lecture → Thm. 1.9.0.10] we can bound with "generic constants" $C > 0$ (depending only on Ω)

$$\|v\|_a^2 \le |v|_{H^1(\Omega)}^2 + C\|v\|_{L^2(\Omega)}\|v\|_{H^1(\Omega)} \le C\|v\|_{H^1(\Omega)} \quad \forall v \in H^1(\Omega) . \tag{3.21.8}$$

Next, Cea's lemma [Lecture → Thm. 3.1.3.7] tells us that that any finite-element Galerkin solution $u_h \in \mathcal{S}_2^0($ of (3.21.1) will satisfy

$$\|u - u_h\|_a = \inf_{v_h \in \mathcal{S}_2^0(\mathcal{M})} \|u - v_h\|_a \le \|u - v_h^*\|_a \le C\|u - v_h^*\|_{H^1(\Omega)} . \tag{3.21.9}$$

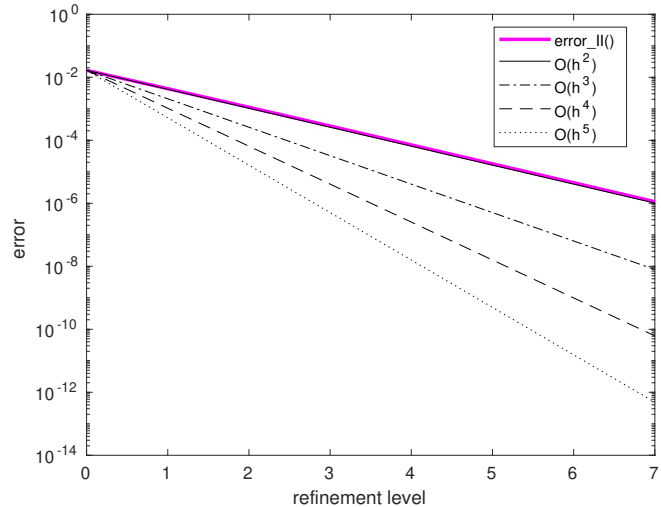
Combined with (**??**) and (**??**) we end up with

$$|u - u_h|_{H^1(\Omega)} \le C\|u - u_h\|_a \le C\|u - v_h^*\|_{H^1(\Omega)} = O(h^2) \quad \text{for} \quad h \to 0 ,$$

where the generic constants $C > 0$ may have different values at every occurrence.

The actual errors returned by `error_II()` in the setting of the numerical experiment are the following:

level	error	$\log_2(\text{ratio})$
0	1.661e-02	
1	4.353e-03	1.9318
2	1.119e-03	1.9597
3	2.837e-04	1.9797
4	7.143e-05	1.9901
5	1.791e-05	1.9952
6	4.486e-06	1.9976
7	1.122e-06	1.9988



The data reveal an empiric algebraic rate of convergence of $O(h^2)$ which agrees with the theoretical prediction.

The following code was used to compute the solutions:

C++ code 3.21.10: Code for `solveLaplImpBVP()`

```
template <typename SOURCE_FUNCTOR, typename IMPDATA_FUNCTOR>
Eigen::VectorXd solveLaplImpBVP(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fes_p,
    SOURCE_FUNCTOR f, IMPDATA_FUNCTOR h) {
  // Set up mesh functions for data
  lf::mesh::utils::MeshFunctionGlobal mf_f{f};
  lf::mesh::utils::MeshFunctionGlobal mf_h{h};
  lf::mesh::utils::MeshFunctionGlobal mf_one{
      [](Eigen::VectorXd /*x*/) -> double { return 1.0; }};

  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler& dofh{fes_p->LocGlobMap()};
  // Dimension of finite element space
  const std::size_t N_dofs(dofh.NumDofs());

  // I. Assembly of Galerkin matrix
  // Set up element matrix providers
  // For Dirichlet form in the volume
  lf::fe::DiffusionElementMatrixProvider<double, decltype(mf_one)>
      vol_elmat_provider(fes_p, mf_one);
  // For boundary contribution to the bilinear form
  // Obtain an array of boolean flags for the edges of the mesh, 'true'
  // indicates that the edge lies on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 1)};
  // Selector for edges on the boundary
  auto imp_sel = [&bd_flags](const lf::mesh::Entity& edge) -> bool {
    return bd_flags(edge);
  };
  lf::fe::MassEdgeMatrixProvider<double, decltype(mf_one), decltype(imp_sel)>
      bd_elmat_provider(fes_p, mf_one, imp_sel);
  // Assemble Galerkin matrix
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Cell-oriented assembly
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, vol_elmat_provider, A);
  // Edge-oriented assembly of boundary term
  lf::assemble::AssembleMatrixLocally(1, dofh, dofh, bd_elmat_provider, A);

  // II. Assembly of right-hand side vector
  Eigen::Matrix<double, Eigen::Dynamic, 1> phi(N_dofs);
  phi.setZero();
  // Assemble volume part of right-hand side vector depending on the
    source
  // function f.
  lf::fe::ScalarLoadElementVectorProvider<double, decltype(mf_f)> elvec_builder(
      fes_p, mf_f);
  // Invoke assembly on cells (codim == 0)
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);
  // Supply boundary contributions due to impedance boundary conditions
  lf::fe::ScalarLoadEdgeVectorProvider<double, decltype(mf_h),
                                       decltype(imp_sel)>
```

```
      bd_elvec_provider(fes_p, mf_h, imp_sel);
  // Invoke assembly on edges (codim == 1), update vector
  lf::assemble::AssembleVectorLocally(1, dofh, bd_elvec_provider, phi);

  // III. Solve linear system
  // Assembly completed:  Convert COO matrix A into CRS format using Eigen's
  // internal conversion routines.
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();

  // Solve linear system using Eigen's sparse direct elimination
  // Examine return status of solver in case the matrix is singular
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  const Eigen::VectorXd sol_vec = solver.solve(phi);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
  return sol_vec;
}
```

The driver routine is the following:

C++ code 3.21.11: Code for `studyAsymptoticCvg()`

```
Eigen::MatrixXd studyAsymptoticCvg(unsigned int reflev) {
  // Exact solution u
  auto u = [](Eigen::Vector2d x) -> double {
    return std::exp(x.squaredNorm());
  };
  // Its gradient
  auto grad_u = [&u](Eigen::Vector2d x) -> Eigen::Vector2d {
    return 2.0 * x * u(x);
  };
  // Right-hand side source function = -Laplacian of u
  auto rhs_f = [&u](Eigen::Vector2d x) -> double {
    return -4.0 * u(x) * (1.0 + x.squaredNorm());
  };
  // Impedance source function for unit triangle domain
  auto imp_h = [&u, &grad_u](Eigen::Vector2d x) -> double {
    Eigen::Vector2d ext_normal;
    // Geometric selection of boundary part
    if ((x[0] > x[1]) && (x[1] < 0.5 * (1.0 - x[0]))) {
      // Bottom edge
      ext_normal << 0.0, -1.0;
    } else if ((x[1] >= x[0]) && (x[0] < 0.5 * (1.0 - x[1]))) {
      ext_normal << -1.0, 0.0;
    } else {
      const double ir2 = 1.0 / std::sqrt(2.0);
      ext_normal << ir2, ir2;
    }
    return grad_u(x).dot(ext_normal) + u(x);
  };

  // Read mesh of unit triangle from file triangle.msh in directory ../meshes
  std::string mesh_file = "meshes/triangle.msh";
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
```

```
  lf::io::GmshReader reader(std::move(mesh_factory), mesh_file);
  std::shared_ptr<lf::mesh::Mesh> cmesh_p = reader.mesh();
  const lf::mesh::Mesh& cmesh{*cmesh_p};
  std::cout << "Read mesh from meshes/triangle.msh: " << cmesh.NumEntities(0)
            << " cells, " << cmesh.NumEntities(1) << " edges, "
            << cmesh.NumEntities(2) << " nodes " << std::endl;
  // Make sure that the mesh is purel triangular
  LF_VERIFY_MSG(cmesh.NumEntities(lf::base::RefEl::kQuad()) == 0,
                "Only triangular meshes admitted");
  // Create hierarchy of meshes by regular refinement
  std::shared_ptr<lf::refinement::MeshHierarchy> multi_mesh_p =
      lf::refinement::GenerateMeshHierarchyByUniformRefinemnt(cmesh_p, reflev);
  lf::refinement::MeshHierarchy& multi_mesh{*multi_mesh_p};
  // Ouput information about hierarchy of nested meshes
  std::cout << "\t Sequence of nested meshes used in experiment\n";
  multi_mesh.PrintInfo(std::cout);
  // Number of levels
  const int L = multi_mesh.NumLevels();

  // Matrix for collecting error numbers
  Eigen::MatrixXd errs(L, 6);
  // LEVEL LOOP: Do computations on all levels
  for (int level = 0; level < L; ++level) {
    std::shared_ptr<lf::mesh::Mesh> mesh_p = multi_mesh.getMesh(level);
    // Set up global FE space; quadratic Lagrangian finite elements
    auto fe_space =
        std::make_shared<const lf::uscalfe::FeSpaceLagrangeO2<double>>(mesh_p);
    // Compute finite element solution of impedance problem for
       Laplacian
    const Eigen::VectorXd mu_h =
        asymptoticcvgfem::solveLapIImpBVP(fe_space, rhs_f, imp_h);
    // L2 error of solution
    std::cout << "Level " << (errs(level, 0) = level) << ": L2 error = "
              << (errs(level, 5) = error_V(fe_space, mu_h, u))
              << ", L2 error bd = "
              << (errs(level, 3) = error_III(fe_space, mu_h, u))
              << ", delta L2^2 = "
              << (errs(level, 4) = error_IV(fe_space, mu_h, u))
              << ", bd avg = " << (errs(level, 1) = error_I(fe_space, mu_h, u))
              << ", H1 error = "
              << (errs(level, 2) = error_II(fe_space, mu_h, grad_u))
              << std::endl;
  }
  return errs;
}
```

SOLUTION of (3-21.f):

Return value of `error_V()` $= \|u - u_{h,l}\|_{L^2(\Omega)} = \left(\int_\Omega |u(\boldsymbol{x}) - u_{h,l}(\boldsymbol{x})|^2 \, \mathrm{d}\boldsymbol{x} \right)^{\frac{1}{2}}$.

HINT 1 for (3-21.g): You may appeal to the following elliptic regularity result

Lemma 3.21.13. 2-Regularity of (3.21.1)

If $f \in L^2(\Omega)$, $h \in H^1(\partial\Omega)$, *and* Ω *is convex, then the solution* $u \in H^1(\Omega)$ *of the linear variational problem* (3.21.1) *with* $\kappa \equiv 1$ *will actually belong to* $H^2(\Omega)$ *and will satisfy*

$$\|u\|_{H^2(\Omega)} \leq C\Big(\|f\|_{L^2(\Omega)} + \|h\|_{H^1(\partial\Omega)}\Big)$$

with a constant $C > 0$ *depending only on* Ω.

SOLUTION of (3-21.g):

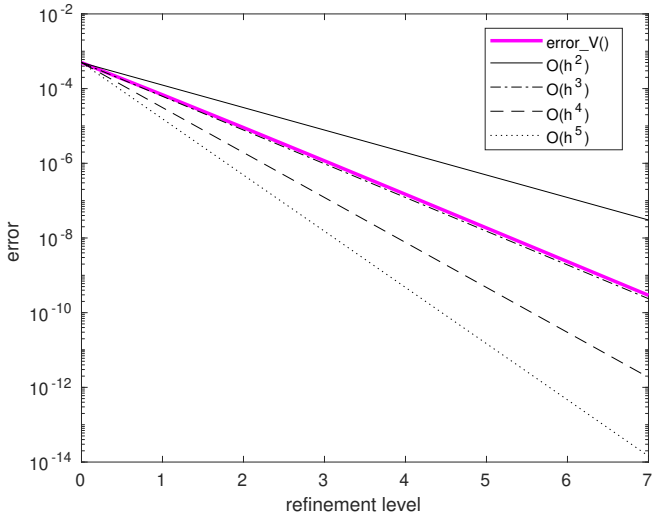
Following the considerations of [Lecture → Section 3.6.3] we see that the elliptic lifting result of **??** paves the way for duality estimates of the $L^2(\Omega)$-norm of the finite-element discretization error. As in [Lecture → § 3.6.3.7] we get

$$\|u - u_h\|_{L^2(\Omega)} \leq Ch\|u - u_h\|_a = O(h^3) \quad \text{for} \quad h \to 0\,,$$

where we used the estimate of Sub-problem (3-21.e).

The actual errors returned by `error_V()` in the setting of the numerical experiment are the following:

level	error	$\log_2(\text{ratio})$
0	4.969e-04	
1	6.832e-05	2.8625
2	9.041e-06	2.9179
3	1.163e-06	2.9582
4	1.475e-07	2.9793
5	1.857e-08	2.9897
6	2.329e-09	2.9949
7	2.919e-10	2.9967



In this case the observations match the theoretical predictions.

SOLUTION of (3-21.h):

Explanations are given in [Lecture → Ex. 3.2.3.4]:

Return value of `error_III()` $= \|u - u_{h,\ell}\|_{L^2(\partial\Omega)} = \left(\int_{\partial\Omega} |u(\boldsymbol{x}) - u_{h,\ell}(\boldsymbol{x})|^2 \, \mathrm{d}S(\boldsymbol{x}) \right)^{\frac{1}{2}}$.

SOLUTION of (3-21.i):

We can use the multiplicative trace inequality of [Lecture → Thm. 1.9.0.10]:

$$\|u - u_h\|^2_{L^2(\partial\Omega)} \leq C\|u - u_h\|_{H^1(\Omega)}\|u - u_h\|_{L^2(\Omega)}$$

and then appeal to the estimates of Sub-problem (3-21.e) and Sub-problem (3-21.g), which yields

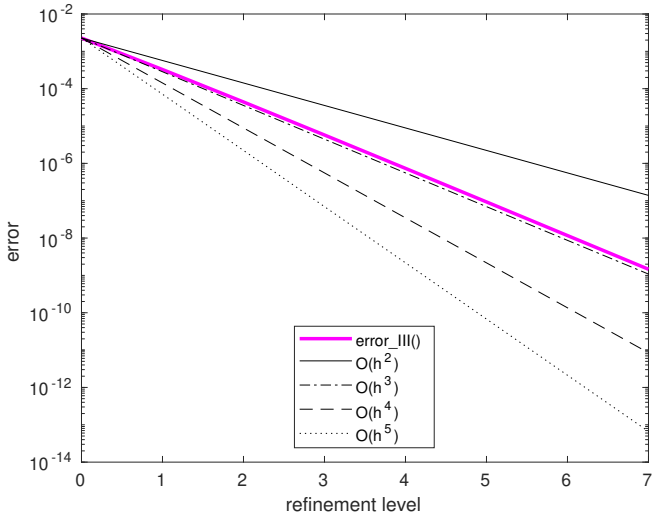
$$\|u - u_h\|^2_{L^2(\partial\Omega)} \leq C \underbrace{\|u - u_h\|_{H^1(\Omega)}}_{=O(h^2)} \underbrace{\|u - u_h\|_{L^2(\Omega)}}_{=O(h^3)} .$$

Finally, we get

$$\|u - u_h\|_{L^2(\partial\Omega)} = O(h^{5/2}) \quad \text{for} \quad h \to 0 .$$

The actual errors returned by `error_III()` in the setting of the numerical experiment are the following:

level	error	$\log_2(\text{ratio})$
0	1.661e-02	
1	1.419e-08	2.8036
2	2.253e-10	2.8861
3	3.577e-12	2.9388
4	2.995e-14	2.9687
5	1.434e-12	2.9843
6	9.272e-12	2.9922
7	3.120e-11	2.9960



We observe faster convergence than predicted by theory, namely an empiric asymptotic convergence like $O(h^3)$ for $h \to 0$.

SOLUTION of (3-22.a):

In [Lecture → Ex. 1.8.0.2] we learned the variational formulation of second-order elliptic Dirichlet boundary value problems. We conclude the variational formulations:

- For (3.22.1a): seek $w \in \{\boldsymbol{x} \mapsto 1\} + H_0^1(\Omega)$ such that

$$\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, \widetilde{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f^2(\boldsymbol{x})\widetilde{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall \widetilde{w} \in H_0^1(\Omega)\ . \tag{3.22.2}$$

- For (3.22.1b): seek $v \in H_0^1(\Omega)$ such that

$$\int_\Omega w(\boldsymbol{x})\, \mathbf{grad}\, v(\boldsymbol{x}) \cdot \mathbf{grad}\, \widetilde{v}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega w(\boldsymbol{x})\widetilde{v}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall \widetilde{v} \in H_0^1(\Omega)\ . \tag{3.22.3}$$

However, the sub-problem asks for *merging both variational equations* into a single one. This is achieved by switching to product spaces simply adding the equations: seek $(w, v) \in (\{\boldsymbol{x} \mapsto 1\} + H_0^1(\Omega)) \times H_0^1(\Omega)$ such that

$$\begin{aligned}\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, \widetilde{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega w(\boldsymbol{x})\, \mathbf{grad}\, v(\boldsymbol{x}) \cdot \mathbf{grad}\, \widetilde{v}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \\ \int_\Omega f^2(\boldsymbol{x})\widetilde{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega w(\boldsymbol{x})\widetilde{v}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall \begin{bmatrix}\widetilde{w}\\ \widetilde{v}\end{bmatrix} \in H_0^1(\Omega) \times H_0^1(\Omega)\ .\end{aligned} \tag{3.22.4}$$

Matching the components of (**??**) and [Lecture → Eq. (1.4.1.7)] we find

$$\begin{aligned}\widehat{V} &:= (\{\boldsymbol{x} \mapsto 1\} + H_0^1(\Omega)) \times H_0^1(\Omega)\ ,\\ V_0 &:= H_0^1(\Omega) \times H_0^1(\Omega)\ ,\\ \mathsf{a}\left(\left[\begin{smallmatrix} w\\ v\end{smallmatrix}\right], \left[\begin{smallmatrix} \widetilde{w}\\ \widetilde{v}\end{smallmatrix}\right]\right) &:= \int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, \widetilde{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega w(\boldsymbol{x})\, \mathbf{grad}\, v(\boldsymbol{x}) \cdot \mathbf{grad}\, \widetilde{v}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\ ,\\ \ell\left(\left[\begin{smallmatrix} \widetilde{w}\\ \widetilde{v}\end{smallmatrix}\right]\right) &:= \int_\Omega f^2(\boldsymbol{x})\widetilde{w}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega w(\boldsymbol{x})\widetilde{v}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall \begin{bmatrix}\widetilde{w}\\ \widetilde{v}\end{bmatrix} \in H_0^1(\Omega) \times H_0^1(\Omega)\ .\end{aligned}$$

Instead of $\{\boldsymbol{x} \mapsto 1\}$ one may simply write '1'.

SOLUTION of (3-22.b):

From [Lecture → Section 1.2.3.4] and [Lecture → § 1.3.4.13] we know that a solution of (**??**) is guaranteed, if $F := f^2 \in L^2(\Omega)$. The standard way of checking this is to verify $\|F\|_{L^2(\Omega)} < \infty$.

(i) The function $F(\boldsymbol{x}) := \log\|\boldsymbol{x}\|$ has a singularity in 0: "$f(0) = \infty$". Elsewhere it is smooth. Therefore we need only make sure that F^2 is integrable in a neighborhood of 0. Using polar coordinates (r, φ) [Lecture → Eq. (1.2.3.47)] we find, using integration by parts,

$$\int_0^{2\pi}\int_0^1 \underbrace{\log^2 r}_{\text{bounded}}\, r\, \mathrm{d}r = \Big[log^2 \tfrac{1}{2}r^2\Big]_0^1 - \Big[logr\tfrac{1}{2}r^2\Big]_0^1 + \int_0^1 \tfrac{1}{2}r\,\mathrm{d}r = \frac{1}{4} < \infty\,.$$

We conclude that $F \in L^2(\Omega)$

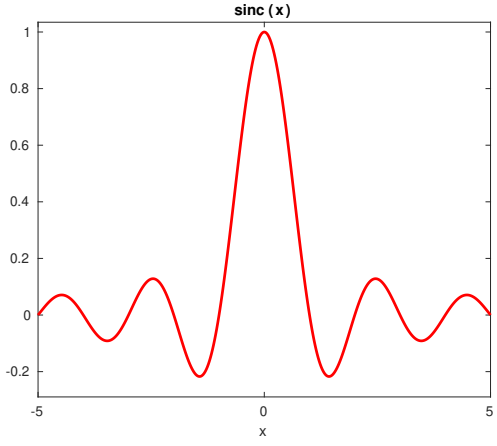
(ii) For $F(\boldsymbol{x}) := \frac{1}{\|\boldsymbol{x}\|^2}$ the attempt to compute its L^2-norm in a neighborhood of the singular point 0 fails:

$$\int_0^{2\pi}\int_0^1 \frac{1}{r^4}\, r\,\mathrm{d}r = 2\pi \int_0^1 \frac{1}{r^3}\,\mathrm{d}r = \text{“}\left[-\frac{1}{2r^2}\right]_0^1\text{”} = \infty\,.$$

As a consequence $F \notin L^2(\Omega)$.

(iii) The function $f(\boldsymbol{x}) := \sqrt[3]{\|\boldsymbol{x}\|}$ is continuous and bounded on Ω and so is $F := f^2$. Since $C^0(\overline{\Omega}) \subset L^2($ it belongs to $L^2(\Omega)$.

(iv) The function $\xi \mapsto \frac{\sin\xi}{\xi}$ ("sinc function") is is smooth on $\mathbb{R}$. Therefore $f(\boldsymbol{x}) := \frac{\sin(x_1)}{x_1}$ will also be smooth on $\overline{\Omega}$. It goes without saying that $\xi \mapsto \frac{\sin\xi}{\xi}$ lies in $L^2(\Omega)$.



So we have existence of a weak solution w for cases (i), (iii), and (iv). Only for **Case (ii)** ($f(\boldsymbol{x}) := \frac{1}{\|\boldsymbol{x}\|}$) we still have to investigate whether the right-hand-side functional

$$\widetilde{w} \mapsto \int_\Omega f^2(\boldsymbol{x})\widetilde{w}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega \frac{1}{\|\boldsymbol{x}\|^2}\widetilde{w}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$

is continuous on $H_0^1(\Omega)$. This is not the case, because if $\widetilde{w} \equiv 1$ in a neighborhood of zero, then $\int_\Omega f^2(\boldsymbol{x})\widetilde{w}($ So in Case (ii) a weak solution cannot exists, as we learned in [Lecture → Rem. 1.2.3.41].

HINT 1 for (3-22.c): Which properties of w guarantee the existence of a unique weak solution v of (3.22.1b)? ⌟

SOLUTION of (3-22.c):

- $f \in C^\infty(\overline{\Omega})$ implies $f^2 \in L^2(\Omega)$ which ensures the existence of a weak solution $w \in H^1(\Omega)$ of (3.22.1a).
- Since Ω is *convex*, we know from

Theorem [Lecture → Thm. 3.4.0.10]. Elliptic lifting theorem on convex domains

If $\Omega \subset \mathbb{R}^d$ *convex,* $u \in H_0^1(\Omega)$, $\Delta u \in L^2(\Omega)$ *then* $u \in H^2(\Omega)$.

that $w \in H^2(\Omega)$. Next,

Theorem [Lecture → Thm. 3.3.3.8]. Sobolev embedding theorem

$$m > \frac{d}{2} \quad \Rightarrow \quad H^m(\Omega) \subset C^0(\overline{\Omega}) \quad \wedge \quad \exists C = C(\Omega) > 0\colon \quad \|u\|_\infty \leq C\|u\|_{H^m(\Omega)} \quad \forall u \in H^m(\Omega)$$

ensures that w is *bounded*.

- From the **maximum principle**

Theorem [Lecture → Thm. 3.7.1.2]. Maximum principle for 2nd-order elliptic BVP

For $u \in C^0(\overline{\Omega}) \cap H^1(\Omega)$ *holds the* ***maximum principle***

$$\begin{aligned} -\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) \geq 0 \quad &\Longrightarrow \quad \min_{\boldsymbol{x}\in\partial\Omega} u(\boldsymbol{x}) = \min_{\boldsymbol{x}\in\Omega} u(\boldsymbol{x})\,, \\ -\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) \leq 0 \quad &\Longrightarrow \quad \max_{\boldsymbol{x}\in\partial\Omega} u(\boldsymbol{x}) = \max_{\boldsymbol{x}\in\Omega} u(\boldsymbol{x})\,. \end{aligned}$$

we conclude that $w(\boldsymbol{x}) \geq 1$ for all $\boldsymbol{x} \in \Omega$. As a consequence the diffusion coefficient w in (3.22.1b) is *uniformly positive* [Lecture → Def. 1.2.2.9].

- Since w is bounded and $w \geq 1$, the bilinear form of the variational formulation of (3.22.1b) defines an energy norm, which is equivalent to the $H^1(\Omega)$-norm on $H_0^1(\Omega)$ by [Lecture → Thm. 1.3.4.17]. This ensures existence and uniqueness of a solution $v \in H_0^1(\Omega)$, *cf.* [Lecture → Section 1.2.3.2].

HINT 1 for (3-22.d): When will the finite element solution $w_h \in \mathcal{S}_1^0(\mathcal{M})$ of (3.22.1a) have properties that ensure the existence of a unique finite element solution $v_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$ of (3.22.1b)? ⌟

SOLUTION of (3-22.d):

The arguments of Sub-problem (3-22.c) carry over: Since $f^2 \in L^2(\Omega)$ existence and uniqueness of Galerkin solutions of (**??**) is clear. So is existence and uniqueness of Galerkin solutions of (**??**), provided that the diffusion coefficient w_h is *uniformly positive* [Lecture → Def. 1.2.2.9].

We know that $w_h \geq 1$ pointwise, if the **discrete maximum principle** holds for the finite-element discretization of (3.22.1a). We invoke the theory of [Lecture → Section 3.7.2], in particular the reasoning of [Lecture → § 3.7.2.7], and [Lecture → Thm. 3.7.2.20].

Theorem [Lecture → Thm. 3.7.2.20]. Maximum principle for linear FE solution of Poisson equation

The linear finite element solution $u_h \in \mathcal{S}_1^0(\mathcal{M})$ of

$$-\Delta u = F \quad \text{in } \Omega \subset \mathbb{R}^2 \quad , \quad u = g \quad \text{on } \partial\Omega,$$

*on a triangular mesh $\mathcal{M}$ satisfies the **discrete maximum principle***

$$\begin{aligned} F \geq 0 \quad &\Longrightarrow \quad \min_{x \in \partial\Omega} u_h(\boldsymbol{x}) = \min_{x \in \Omega} u_h(\boldsymbol{x}) \, , \\ F \leq 0 \quad &\Longrightarrow \quad \max_{x \in \partial\Omega} u_h(\boldsymbol{x}) = \max_{x \in \Omega} u_h(\boldsymbol{x}) \, , \end{aligned}$$

*if $\mathcal{M}$ is a **Delaunay triangulation** [Lecture → Def. 4.2.2.5].*

So we conclude, that $\mathcal{M}$ being a Delaunay triangulation ensures existence and uniqueness of $\mathcal{S}_1^0(\mathcal{M})$ finite element Galerkin solutions of (3.22.1). Recall that a triangular mesh $\mathcal{M}$ is a Delaunay triangulation, if [Lecture → Thm. 4.2.2.4]

(i) sum of angles facing an interior edge $\leq \pi$,

(ii) angles facing boundary edges $\leq \pi/2$.

HINT 1 for (3-22.e): In the file `coupledbvps.h` you will find a potentially useful C++ function

```
template <typename MESHFN_ALPHA, typename MESHFN_F, typename G_FUNCTOR>
Eigen::VectorXd solveDirichletBVP(
  std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
  MESHFN_ALPHA alpha, MESHFN_F F, G_FUNCTOR &&g);
```

which computes the basis expansion coefficient vector of the $\mathcal{S}_1^0(\mathcal{M})$ finite-element solution of the Dirichlet boundary value problem

$$-\operatorname{div}(\alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) = F(\boldsymbol{x}) \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega\,.$$

The parameters `alpha` and `F` must be of type **MeshFunction** [Lecture → Suppl. 2.8.3.32] and provide the functions $\boldsymbol{x} \in \Omega \mapsto \alpha(\boldsymbol{x})$ for the diffusion coefficient, and $\boldsymbol{x} \in \Omega \mapsto F(\boldsymbol{x})$ for the right-hand side function F.

The argument `g`, which supplies the Dirichlet boundary values in the form of a continuous function $g : \partial \to \mathbb{R}$, must contain a functor object matching **std::function<double(Eigen::Vector2D)>**.

⌟

SOLUTION of (3-22.e):

C++ code 3.22.5: Implementation of `solveDirichletBVP()`

```
template <typename MESHFN_ALPHA, typename MESHFN_F, typename G_FUNCTOR>
Eigen::VectorXd solveDirichletBVP(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    MESHFN_ALPHA alpha, MESHFN_F F, G_FUNCTOR &&g) {
  // Obtain the mesh
  const lf::mesh::Mesh &mesh{*(fes_p->Mesh())};
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
  // Dimension of finite element space`
  const lf::base::size_type N_dofs(dofh.NumDofs());
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);

  // Define local computations for the LHS through a matrix provider
  lf::fe::DiffusionElementMatrixProvider elmat_builder(fes_p, alpha);
  // Construct the full matrix through local computations
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);

  // Define local computations for the RHS through a vector provider
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(F)>
      elvec_builder(fes_p, F);

  // Define the RHS vector
  Eigen::VectorXd phi(N_dofs);
  // We initialize to zero, because local assembly adds to the vector
  phi.setConstant(0);
  // Invoke assembly on cells (codim == 0)
  AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // Flag all nodes on the boundary (and only those)
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
  // Set up predicate:  Run through all global shape functions and check
     whether
  // they are associated with an entity on the boundary, store Dirichlet
     data.
  std::vector<std::pair<bool, double>> ess_dof_select{};
  for (lf::assemble::gdof_idx_t dofnum = 0; dofnum < N_dofs; ++dofnum) {
    const lf::mesh::Entity &dof_node{dofh.Entity(dofnum)};
    const Eigen::Vector2d node_pos{
        lf::geometry::Corners(*dof_node.Geometry()).col(0)};
    const double g_val = g(node_pos);
    if (bd_flags(dof_node)) {
      // Dof associated with a entity on the boundary:  "essential dof"
      // The value of the dof should be set to the value of the function
      // u at the location of the node.
      ess_dof_select.emplace_back(true, g_val);
    } else {
      // Interior node, also store value of solution for comparison
         purposes
      ess_dof_select.emplace_back(false, g_val);
    }
  }
  // modify linear system of equations
```

```
  lf::assemble::FixFlaggedSolutionCompAlt<double>(
      [&ess_dof_select](int dof_idx) -> std::pair<bool, double> {
        return ess_dof_select[dof_idx];
      },
      A, phi);

  // Assembly completed:  Convert COO matrix A into CRS format using Eigen's
  // internal conversion routines.
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();

  // Solve linear system using Eigen's sparse direct elimination
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  Eigen::VectorXd sol_vec = solver.solve(phi);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");

  // Return the solution
  return sol_vec;
}
```

Using `solveDirichletBVP()` twice the only trick is to wrap the finite-element solution for w into a **MeshFunction** object by means of **lf::fe::MeshFunctionFE**.

C++ code 3.22.6: Implementation of `solveModulatedHeatFlow()`

```
template <typename F_FUNCTOR>
std::pair<Eigen::VectorXd, Eigen::VectorXd> solveModulatedHeatFlow(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    F_FUNCTOR &&f) {
  // We need the following lambda functions to call solveDirichletBVP
  auto f2 = [&f](Eigen::Vector2d x) -> double { return f(x) * f(x); };
  auto g = [](Eigen::Vector2d /*x*/) -> double { return 1.; };
  auto zero = [](Eigen::Vector2d /*x*/) -> double { return 0.; };
  // We need the following MeshFunctions to call solveDirichletBVP
  lf::mesh::utils::MeshFunctionGlobal mf_f2{f2};
  lf::mesh::utils::MeshFunctionConstant mf_one(1.);

  Eigen::VectorXd w_gf;  // Basis expansion coefficients for w_h
  Eigen::VectorXd v_gf;  // Basis expansion coefficients for v_h
  // We first solve for w using the provided solver for diffusion problems
  // (solveDirichletBVP)
  w_gf = solveDirichletBVP(fes_p, mf_one, mf_f2, g);
  // We cast w into a meshfunction such that it can be used in solveDirichletBVP
  // as a coefficient and source
  lf::fe::MeshFunctionFE mf_w(fes_p, w_gf);
  // We solve for v using solveDirichletBVP using the previously computed source
  // and coefficient w
  v_gf = solveDirichletBVP(fes_p, mf_w, mf_w, zero);
  // We return the solution
  return std::make_pair(w_gf, v_gf);
}
```

SOLUTION of (3-23.a):

Following the considerations of [Lecture → Section 1.5.3] we obtain

$$\begin{aligned} -\operatorname{div}(\boldsymbol{\alpha}\,\mathbf{grad}\,u) &= f \quad \text{in } \Omega\,, \\ u &= 0 \quad \text{on } \partial\Omega\,. \end{aligned}$$

The homogeneous Dirichlet boundary conditions are built into the trial space, they occur as essential boundary conditions.

SOLUTION of (3-23.b):

We recall [Lecture → Def. 1.2.3.35] of the energy norm induced by an s.p.d. bilinear form:

$$\|v\|_a = (\mathsf{a}(v,v))^{1/2} = \left(\int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,v(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right)^{1/2}, \quad v \in H^1(\Omega)\,. \tag{3.23.3}$$

HINT 1 for (3-23.c): What you have at your disposal is a C++ function

```
Eigen::Matrix<double, 2, 4> exteriorUnitNormals(
const lf::geometry::Geometry& geo)
```

that computes the exterior unit normals for all edges of a triangle or quadrilateral, whose geometry information is passed via the **lf::geometry::Geometry** object `geo`. The normals are returned in the columns of a $2 \times p$-matrix, $p = 3, 4$, whose last column is simply $= \mathbf{0}$ in the case $p = 3$. ⌟

SOLUTION of (3-23.c):

The key idea is to rely on a loop over the mesh cells, testing their edges whether they are located on the boundary. Of course, this test is easy, when invoking the LEHRFEM++ function `flagEntitiesOnBound`

In any case, we have to compute $\mathbf{grad}\, u_h$. In code **??** this is done using the helper class **lf::fe::MeshFuncti** Alternatively, one can directly compute the (constant!) gradients of the local shape functions, here the barycentric coordinate functions, using the algorithm of [Lecture → Code 2.4.5.11]. In fact this function is provided in `neumanndatarecovery.cc` as

```
Eigen::Matrix<double, 2, 3> GradsBaryCoords(
    Eigen::Matrix<double, 2, 3> vertices);
```

Once we have $\mathbf{grad}\, u_h|_K$ for some $K \in \mathcal{M}$ with an edge $e \subset \partial\Omega_h$, then we can call `exteriorUnitNorr` to obtain $\boldsymbol{n}_e$ and compute $\boldsymbol{n}_e \cdot \mathbf{grad}\, u_h|_K$. This is the value that has to be stored in the e-element of the **lf::mesh::utils::CodimMeshDataSet** object that is returned.

C++ code 3.23.4: Function `getNeumannData()`

```
lf::mesh::utils::CodimMeshDataSet<double> getNeumannData(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
    const Eigen::VectorXd &mu) {
  lf::mesh::utils::CodimMeshDataSet<double> edge_vals(fe_space->Mesh(), 1, 0.0);
  // Obtain flags indicating edges on the boundary
  lf::mesh::utils::CodimMeshDataSet<bool> bded_flags{
      lf::mesh::utils::flagEntitiesOnBoundary(fe_space->Mesh(), 1)};
  // Compute the piecewise constant gradients for all cells of the mesh
  const lf::fe::MeshFunctionGradFE mf_grad_fefun(fe_space, mu);
  // Reference coordinates of barycenter of mesh
  Eigen::MatrixXd refc_center = Eigen::Vector2d(1.0 / 3.0, 1.0 / 3.0);
  // Loop over all cells of the mesh
  for (const lf::mesh::Entity *cell : fe_space->Mesh()->Entities(0)) {
    LF_VERIFY_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Only triangular cells admitted!");
    // Compute constant value of the gradient for the current cell
    const Eigen::Vector2d grad_fefun = mf_grad_fefun(*cell, refc_center)[0];
    // Visit edges of the cell and check whether they are located on the
    // boundary
    std::span<const lf::mesh::Entity *const> edges{cell->SubEntities(1)};
    int ed_cnt = 0;
    for (const lf::mesh::Entity *edge : edges) {
      if (bded_flags(*edge)) {
        // Edge is located on the boundary:
        // Obtain exterior unit normals for the triangle, pick that belonging to
        // the current edge and compute its Euclidean inner product with the
        // local gradient vector.
        const Eigen::Matrix<double, 2, 4> unit_normals =
            exteriorUnitNormals(*(cell->Geometry()));
        edge_vals(*edge) = unit_normals.col(ed_cnt).dot(grad_fefun);
      }
      ed_cnt++;
    }
  }
  return edge_vals;
}
```

SOLUTION of (3-23.d):

We appeal to Gauss theorem [Lecture → Thm. 1.5.2.4] or Green's first formula [Lecture → Thm. 1.5.2.7], the result of Sub-problem (3-23.a), and deduce for all $v \in H^1(\Omega)$

$$\begin{aligned}
\int_{\partial\Omega} \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) &= \int_\Omega \mathrm{div}(\boldsymbol{\alpha}\,\mathbf{grad}\,u\,v)(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\\
&= \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \underbrace{\mathrm{div}(\boldsymbol{\alpha}\,\mathbf{grad}\,u)(\boldsymbol{x})}_{=-f(\boldsymbol{x})}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\\
&= \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega -f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\\
&= \mathsf{a}(u,v)+\ell(-v)\ .
\end{aligned}$$

Note that the test function v does not satisfy any boundary conditions!

SOLUTION of (3-23.e):

Plugging $u_h = \sum_{j=1}^{N} (\vec{\boldsymbol{\mu}})_j b_h^j$, $\mathsf{N}_h u_h = \sum_{j=N+1}^{M} (\vec{\boldsymbol{\eta}})_j b_h^j$, and $v \leftarrow b_h^j$, $j = N+1, \ldots, M$ into (3.23.10) we arrive at the $(M-N) \times (M-N)$ linear system of equations

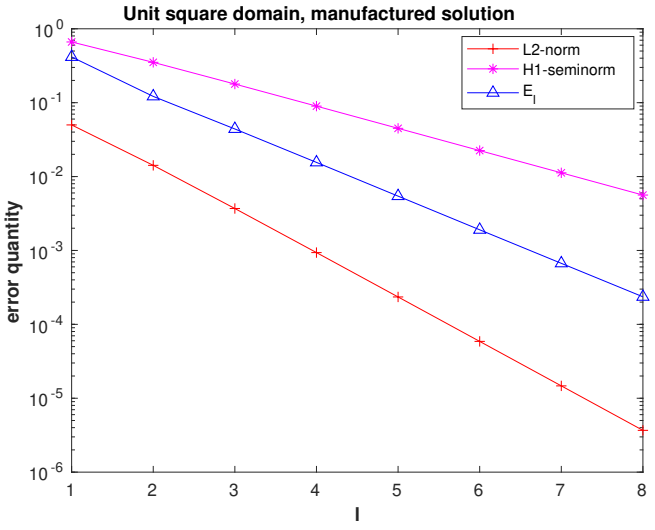
$$\mathbf{Q}_{\partial,\partial} \vec{\boldsymbol{\eta}} = \mathbf{A}_{0,\partial} \vec{\boldsymbol{\mu}} - \vec{\boldsymbol{\varphi}}_\partial \ , \quad (\vec{\boldsymbol{\varphi}}_\partial)_j := \int_\Omega f(\boldsymbol{x}) b_h^{j+N} \, \mathrm{d}\boldsymbol{x} \ , \quad j = 1, \ldots, M-N \ . \tag{3.23.11}$$

SOLUTION of (3-23.f):

C++ code 3.23.13: Function `getNeumannData()`

HINT 1 for (3-23.g): The error data are also available in CSV format in `errdata.txt`. ⌟

SOLUTION of (3-23.g):



◁ The errors are neatly aligned in a semilogarithmic plot as functions of ℓ

Since $h_l \sim 2^{-\ell}$, the errors points would also end up on lines in a doubly logarithmic plot as functions of h_ℓ.

Therefore we conclude algebraic convergence of all error quantities as $h_\ell \to 0$.

ℓ	$E_\ell^A : E_{\ell-1}^A$	$E_\ell^B : E_{\ell-1}^B$	$E_\ell^C : E_{\ell-1}^C$
2	0.2837	0.2924	0.5300
3	0.2606	0.3625	0.5084
4	0.2533	0.3527	0.5024
5	0.2510	0.3496	0.5006
6	0.2503	0.3503	0.5002
7	0.2501	0.3515	0.5000
8	0.2500	0.3524	0.5000

We write E_ℓ^X for the value tabulated above in row ℓ and column (X), $X \in \{A, B, C\}$.

The table beside shows the ratios of the error quantities in subsequent rows. Thus halving the meshwidth h_ℓ means

- a reduction by a factor ≈ 4 for E^A ► algebraic convergence $O(h_\ell^2)$,
- a reduction by a factor $\approx 0.35^{-1} = 2.8571$ for E^B ► algebraic convergence $O(h_\ell^p)$, $p = \frac{\log 2.8}{\log}$
- and a reduction by a factor ≈ 2 for E^C ► algebraic convergence $O(h_\ell)$.

Since the manufactured solution u is smooth in this case, we certainly have $u \in H^2(\Omega)$. Thus, according to [Lecture → Thm. 3.3.5.6] (for $p = 1$) and thanks to the optimality of the finite-element Galerkin solution, we conclude that $|u - u_\ell|_{H^1(\Omega)} = O(h_\ell)$. Thus, the values in Column (C) of the table correspond to $|u - u_\ell|_{H^1(\Omega)}$.

Since Ω is convex by [Lecture → Thm. 3.4.0.10] the homogeneous Dirichlet problem for $-\Delta$ is 2-regular. Thus, the results of [Lecture → Section 3.6.3] predict the asymptotic convergence $\|u - u_\ell\|_{L^2(\Omega)} = O(h_\ell^2)$

Thus, we conclude that column (A) of the table lists $\|u - u_\ell\|_{L^2(\Omega)}$.

The remaining column (B) must show the error quantity $\mathcal{E}_\ell$.

HINT 1 for (3-3.a): Remember a particular model discussed in [Lecture → Chapter 1].

SOLUTION of (3-3.a):

This is similar to the 2D membrane model [Lecture → Section 1.2.1] with a "point loading" at $\boldsymbol{x} = 0$. More details in [Lecture → Ex. 1.2.3.45] where this situation was described as a needle piercing the skin of a balloon.

HINT 1 for (3-3.j): A solid understanding of the LEHRFEM++ framework for assembly as discussed in the lecture in [Lecture → Section 2.7.5] is necessary. ⌟

HINT 2 for (3-3.j):

Be aware that in a floating-point environment you cannot test whether a point is exactly located inside a cell. You have to allow for some margin!

However, if you employ a fuzzy test for “is located inside”, then a point could be identified as lying inside several cells. By means of some internal counter/flag your **ENTITY_VECTOR_PROVIDER** object must deal with this situation.

⌟

SOLUTION of (3-3.j):

To evaluate $\ell(v_h) = v_h(\mathbf{0})$, we have to

1. find the unique cell $K \in \mathcal{M}$, which contains $\mathbf{0}$,
2. evaluate the local shape functions b_K^ℓ, $\ell = 1, \ldots, Q$, of that cell in $\mathbf{0}$,

For the second step, thanks to $\mathcal{S}_1^0(\mathcal{M})$ being a parametric finite element space, we determine the preimage $\widehat{\boldsymbol{x}}_0 := \boldsymbol{\Phi}_K^{-1}(\mathbf{0})$ and compute the values $\widehat{b}^\ell(\widehat{\boldsymbol{x}}_0)$, which correspond to the entries of the element vector for K.

C++11 code 3.3.12: Sub-problem (3-3.j): Implementation of `DeltaLocalVectorAssembler::Eval()` ➙ GitLab

```
Eigen::VectorXd DeltaLocalVectorAssembler::Eval(const lf::mesh::Entity &cell) {
  Eigen::VectorXd result;
  // get the coordinates of the corners of this cell
  const lf::geometry::Geometry *geo_ptr = cell.Geometry();
  auto vertices = lf::geometry::Corners(*geo_ptr);
  Eigen::Vector2d x_hat;
  // the margin we allow when we determine wether a point is inside an
  // element
  const double margin = 1e-10;
  if (lf::base::RefEl::kTria() == cell.RefEl()) {
    x_hat = PointEvaluationRhs::GlobalInverseTria(vertices, x_0);
    result.resize(3);
    result.setZero();
    if (x_hat(0) <= 1 + margin && x_hat(0) >= 0 - margin &&
        x_hat(1) <= 1 + margin && x_hat(1) >= 0 - margin &&
        x_hat(1) + x_hat(0) <= 1 + margin) {
      already_found = true;
      // Barycentric coordinates on reference triangle
      result[0] = 1.0 - x_hat(0) - x_hat(1);
      result[1] = x_hat(0);
      result[2] = x_hat(1);
    }
  } else if (lf::base::RefEl::kQuad() == cell.RefEl()) {
    x_hat = PointEvaluationRhs::GlobalInverseQuad(vertices, x_0);
    result.resize(4);
    result.setZero();
    if (x_hat(0) <= 1 + margin && x_hat(0) >= 0 - margin &&
        x_hat(1) <= 1 + margin && x_hat(1) >= 0 - margin) {
      already_found = true;
      // Local shape functions on unit square
      result[0] = (1.0 - x_hat(0)) * (1.0 - x_hat(1));
      result[1] = x_hat(0) * (1.0 - x_hat(1));
      result[2] = x_hat(0) * x_hat(1);
      result[3] = (1.0 - x_hat(0)) * x_hat(1);
    }
  } else {
    LF_ASSERT_MSG(
        false, "Function only defined for triangular or quadrilateral cells");
  }
  return result;
}
```

HINT 1 for (3-3.k): Implementation can be done in at least three different ways:

1. Directly by looping over the cells of the mesh. For local computations on quadrilaterals use the quadrature rule supplied by `lf::quad::make_QuadQR_P4O4()`. For triangular local evaluations can be done analytically based on [Lecture → Lemma 2.7.5.5].
2. An alternative is to use LEHRFEM++'s **MeshFunction** facilities like **lf::fe::MeshFunctionFE**, `lf::u` and `lf::mesh::utils::squaredNorm()` see [Lecture → Ex. 3.2.3.4] and [Lecture → Code 3
3. Another alternative can rely on the $\mathcal{S}_1^0(\mathcal{M})$ mass matrix, that is the finite element Galerkin matrix for the $L^2(\Omega)$-inner product $(u,v) \mapsto \int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$.

All these options are implemented in a demonstration code, some only in the case of triangular meshes
➥ GITHUB ⌟

SOLUTION of (3-3.k):

The following code implements the third option.

C++11 code 3.3.13: Sub-problem (3-3.k): Implementation of `computeL2normLinearFE()`
➥ GitLab

```
2  double computeL2normLinearFE(const lf::assemble::DofHandler &dofh,
3                               const Eigen::VectorXd &mu) {
4    double result = 0.0;
5    int N_dofs = dofh.NumDofs();
6    lf::assemble::COOMatrix<double> mass_matrix(N_dofs, N_dofs);
7    MassLocalMatrixAssembler my_mat_provider{};
8    lf::assemble::AssembleMatrixLocally(0, dofh, dofh, my_mat_provider,
9                                        mass_matrix);
10   const Eigen::SparseMatrix<double> mass_mat = mass_matrix.makeSparse();
11   result = std::sqrt(mu.dot(mass_mat * mu));
12   return result;
13 }
```

HINT 1 for (3-3.l):

Again, this function can be implemented in several different ways:

1. Direct computation in a loop over the cells using local numerical quadrature. For local computations on quadrilaterals use the quadrature rule supplied by `lf::quad::make_QuadQR_P4O4()`. The gradients of functions $\in \mathcal{S}_1^0(\mathcal{M})$ on are constant on triangles.
2. An alternative is to use `lf::uscalfe::IntegrateMeshFunction()` supplied with a **lf::fe::M** object that is built from the basis expansion coefficient vector for the finite element solution.
3. Another way relies on the $\mathcal{S}_1^0(\mathcal{M})$-finite-element Galerkin matrix (stiffness matrix) $\mathbf{A}$ arising from the bilinear form $(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$. Multiplying it from the right and from the left with the basis expansion coefficient vector of the finite element function $v_h \in \mathcal{S}_1^0(\mathcal{M})$ gives $|v_h|_{H^1(\Omega)}$.

⌟

SOLUTION of (3-3.l):

C++11 code 3.3.14: Sub-problem (3-3.l): Implementation of `computeH1seminormLinearFE()` ↪ GitLab

```
2  double computeH1seminormLinearFE(const lf::assemble::DofHandler &dofh,
3                                   const Eigen::VectorXd &mu) {
4    // calculate stiffness matrix by using the already existing local assembler
5    // LinearFELaplaceElementMatrix
6    double result = 0.0;
7    int N_dofs = dofh.NumDofs();
8    lf::assemble::COOMatrix<double> stiffness_matrix(N_dofs, N_dofs);
9    lf::uscalfe::LinearFELaplaceElementMatrix my_mat_provider{};
10   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, my_mat_provider,
11                                       stiffness_matrix);
12   const Eigen::SparseMatrix<double> stiffness_mat =
13       stiffness_matrix.makeSparse();
14   result = std::sqrt(mu.dot(stiffness_mat * mu));
15   return result;
16 }
```

SOLUTION of (3-3.m):

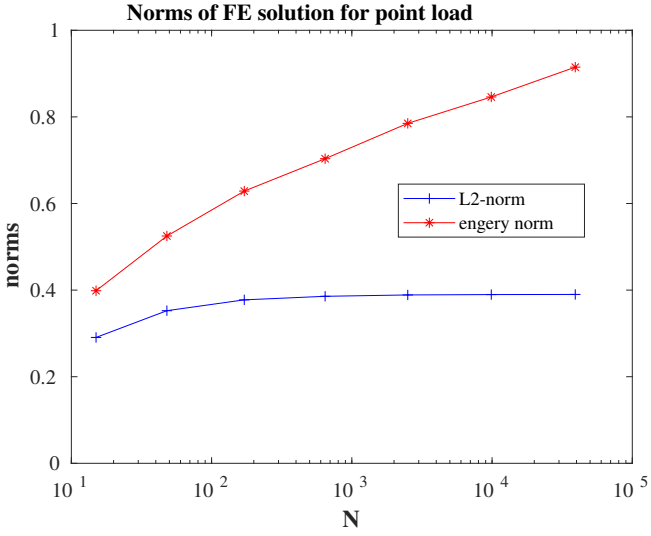
C++11 code 3.3.15: Sub-problem (3-3.m): Implementation of `normsSolutionPointLoadDirichletBVP()` ↬ GitLab

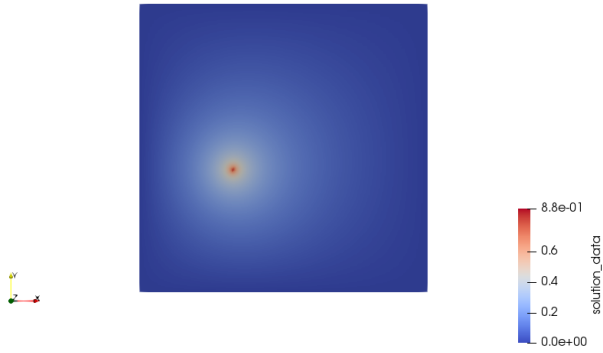
```
  // Assemble matrix A
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  lf::uscalfe::LinearFELaplaceElementMatrix loc_mat_laplace{};
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, loc_mat_laplace, A);
  // Build rhs vector using dedicated ENTITY_VECTOR_PROVIDER
  Eigen::VectorXd rhs(N_dofs);
  rhs.setZero();
  PointEvaluationRhs::DeltaLocalVectorAssembler myvec_pro(source_point);
  lf::assemble::AssembleVectorLocally(0, dofh, myvec_pro, rhs);

  // Enforce Dirichlet boundary conditions
  const double boundary_val = 0;  // zero Dirichlet boundary conditions
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(dofh.Mesh(), 2)};
  auto my_selector = [&dofh, &bd_flags, &boundary_val](unsigned int dof_idx) {
    if (bd_flags(dofh.Entity(dof_idx))) {
      return (std::pair<bool, double>(true, boundary_val));
    }  // interior node: the value we return here does not matter
    return (std::pair<bool, double>(false, 42.0));
  };
  lf::assemble::FixFlaggedSolutionComponents<double>(my_selector, A, rhs);

  // Solve linear system of equations A*x = rhs
  const Eigen::SparseMatrix<double> A_crs(A.makeSparse());
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  if (solver.info() == Eigen::Success) {
    sol_vec = solver.solve(rhs);
  } else {
    LF_ASSERT_MSG(false, "Eigen Factorization failed");
  }
```

SOLUTION of (3-3.n):

N	$\|u_h\|_{L^2(\Omega)}$	$\|u_h\|_{H^1(\Omega)}$
15	0.29066	0.398559
48	0.352459	0.524942
171	0.377485	0.628679
645	0.385684	0.703485
2505	0.388823	0.785053
9873	0.389681	0.846032
39201	0.389981	0.914899

We observe slow, but steady increase of the $H^1(\Omega)$-seminorm of the solution as the resolution of the finite element space is increased.
Conversely, the $L^2(\Omega)$-norm of the finite element Galerkin solution seems to approach a limit value.





SOLUTION of (3-3.o):

From the variational problem with $u = v$ we have $\ell(u) = \mathsf{a}(u,u)$. Thus

$$J(u) = \frac{1}{2}\mathsf{a}(u,u) - \ell(u) = -\frac{1}{2}\|u\|_{\mathsf{a}}^2 \,. \tag{3.3.17}$$

SOLUTION of (3-3.p):

We can see from Fig. 40 and **??** that the H^1-seminorm of the solution appears to diverge (albeit not very quickly), while the L^2-norm converges. This is because the solution of this variational problem is not in H^1 (but it is in L^2), *cf.* Sub-problem (3-3.h).

HINT 1 for (3-3.b): Refresh yourself on the contents of [Lecture → Section 1.2.3]. Also take another look at [Lecture → Section 1.3] and, in particular, [Lecture → Ex. 1.2.3.45]. ⌟

SOLUTION of (3-3.b):

As shown in [Lecture → Ex. 1.2.3.45], the linear form $\ell(v) = v(\mathbf{0})$ is not continuous with respect to the energy norm

$$\|v\|_{\mathsf{a}} := |v|_{H^1(\Omega)} = \left(\int_\Omega \|\mathbf{grad}\, v\|^2 \,\mathrm{d}\boldsymbol{x} \right)^{\frac{1}{2}}, \quad v \in H^1(\Omega) .$$

The associated quadratic functional will not be bounded from below and has no minimizer according to [Lecture → Lemma 1.2.3.39].

SOLUTION of (3-3.c):

1. On the restricted space $\mathcal{S}^0_{1,0}(\mathcal{M})$, ℓ *is* in fact continuous with respect to $\|\cdot\|_{\mathsf{a}}$. Remember from your analysis class that all linear functionals are continuous in finite dimensional spaces, because they produce finite values when acting on the finitely many elements of a basis:

$$\ell(\vec{\eta}) = \sum_{j=1}^{N} \eta_j \ell(\vec{\beta}_j) \leq \underbrace{\left(\sum_{j=1}^{N} \ell(\vec{\beta})^2\right)^{\frac{1}{2}}}_{=:C} \underbrace{\left(\sum_{j=1}^{N} \eta_j^2\right)^{\frac{1}{2}}}_{=\|\vec{\eta}\|},$$

   where $\{\vec{\beta}_1, \ldots, \vec{\beta}_N\}$ is a basis. Then invoke [Lecture $\rightarrow$ Thm. 1.2.3.44].

2. Alternatively, one can argue that the s.p.d. bilinear of (3.3.1) will spawn an s.p.d. Galerkin matrix, which is always invertible.

Remark. How is it possible that the discrete quadratic minimization problem admits a minimum, whereas the infinite-dimensional one does not? Well, that minimal value will be an "approximation of $-\infty$". Now you can imagine, what will happen as you refine the resolution of the discrete model.

HINT 1 for (3-3.d): See [Lecture → § 2.7.5.12] for details about affine mappings connecting triangles. The inverse of an affine mapping is again affine. ⌟

SOLUTION of (3-3.d):

Refer to [Lecture → § 2.7.5.12]. The unique affine mapping

$$\boldsymbol{\Phi}_K : \widehat{K} \to K \quad , \quad \boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) := \mathbf{A}\widehat{\boldsymbol{x}} + \boldsymbol{t}$$

taking the unit triangle $\widehat{K}$ to the general triangle K has the inverse

$$\boldsymbol{\Phi}_K^{-1}(\boldsymbol{x}) = \mathbf{A}^{-1}(\boldsymbol{x} - \boldsymbol{t}) \ .$$

The EIGEN-based implementation of this formula is given in **??**.

C++11 code 3.3.2: Sub-problem (3-3.d): Implementation of `GlobalInverseTria()`
→ GitLab

```
2  Eigen::Vector2d GlobalInverseTria(Eigen::Matrix<double, 2, 3> mycorners,
3                                    Eigen::Vector2d x) {
4    Eigen::Vector2d x_hat;
5
6    // The unique affine mapping of the unit triangle to a general
       triangle
7    // is given by the formula x = A_K * x_hat + s,
8    // Use the vertex coordinates to calculate matrix A and translation
       vector s
9    Eigen::Matrix2d A(2, 2);
10   A.col(0) = mycorners.col(1) - mycorners.col(0);
11   A.col(1) = mycorners.col(2) - mycorners.col(0);
12   // The inverse mapping:  x_hat = A_K^{-1} * (x - a_0)
13   x_hat = A.partialPivLu().solve(x - mycorners.col(0));
14   return x_hat;
15 }
```

SOLUTION of (3-3.e):

Rescale the equations to the form $\xi^2 + p\xi + q = 0$. Then check the sign of the discriminant $D = p^2 - 4q$; if $D < 0$ no real solutions exist.

If $D \geq 0$ compute the first root *avoiding cancellation*:

$$\xi_1 = \begin{cases} \frac{1}{2}(-b - \sqrt{D}) & \text{, if } b > 0\,, \\ \frac{1}{2}(-b + \sqrt{D}) & \text{, if } b \leq 0\,. \end{cases}$$

Compute the other root based on Vieta's rule $\xi_1\xi_2 = p$: $\xi_2 = p/\xi_1$.

C++11 code 3.3.3: Sub-problem (3-3.e): Implementation of `solveQuadraticEquation()`
➥ GitLab

```
2  std::pair<double, double> solveQuadraticEquation(double a, double b, double c) {
3    // Implement the cases which are solvable and return their solutions
4    if (a != 0.) {
5      b /= a;
6      c /= a;
7      const double D = b * b - 4 * c;  // discriminant
8      if (D >= 0) {                    // real solutions
9        const double rtD = std::sqrt(D);
10       // Real solutions, cancellation-free formulas !
11       if (b < 0) {
12         const double root = 0.5 * (-b + rtD);
13         return {root, c / root};
14       }  // b >= 0
15       const double root = 0.5 * (-b - rtD);
16       return {root, c / root};
17     }
18   } else {
19     if (b != 0.0) {
20       return {-c / b, -c / b};
21     }
22   }
23   // Return NAN if there are no (real) roots
24   return {NAN, NAN};
25 }
```

HINT 1 for (3-3.f): *After a suitable permutation* of the vertices of K you can write the bilinear mapping $\boldsymbol{\Phi}_K$ in the form

$$\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) = \boldsymbol{t} + \mathbf{A}\widehat{\boldsymbol{x}} + \boldsymbol{d}\widehat{x}_1\widehat{x}_2 \,, \quad \widehat{\boldsymbol{x}} = \begin{bmatrix} \widehat{x}_1 \\ \widehat{x}_2 \end{bmatrix} \,, \quad \boldsymbol{t} \in \mathbb{R}^2 \,, \quad \mathbf{A} \in \mathbb{R}^{2,2} \text{ regular} \,. \tag{3.3.4}$$

⌟

HINT 2 for (3-3.f): We have to solve $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) = \boldsymbol{x}$ for $\widehat{\boldsymbol{x}}$, where $\boldsymbol{\Phi}_K$ is given in the form (**??**). Note that this equation is equivalent to

$$\widehat{\boldsymbol{x}} + \boldsymbol{q}\widehat{x}_1\widehat{x}_2 = \boldsymbol{y}\,, \tag{3.3.5}$$

with suitable vectors $\boldsymbol{q} \in \mathbb{R}^2$ and $\boldsymbol{y} \in \mathbb{R}^2$, which arises from multiplying $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) = \boldsymbol{x}$ with $\mathbf{A}^{-1}$.

If $\boldsymbol{q} \neq \mathbf{0}$ (be careful and test for relative smallness!), multiply (**??**) with $\boldsymbol{q}^\perp := \begin{bmatrix} q_2 \\ -q_1 \end{bmatrix}$ from the left and obtain a formula for one component of $\widehat{\boldsymbol{x}}$, which you can plug into one equation of (**??**) in order to arrive at a quadratic equation for the other component. ⌟

SOLUTION of (3-3.f):

For the sake of stability, in a first step we have to cyclically permute the vertices of the QUAD K in a way so that the triangle spanned by $\boldsymbol{a}_K^1, \boldsymbol{a}_K^2, \boldsymbol{a}_K^4$, has the largest area of all triangles formed by three vertices of K. Then $\boldsymbol{\Phi}_K : \widehat{K} \to K$ has the form (**??**) with

$$\boldsymbol{t} = \boldsymbol{a}_K^1 \,, \quad \mathbf{A} = \left[\boldsymbol{a}_K^2 - \boldsymbol{a}_K^1 \; \boldsymbol{a}_K^4 - \boldsymbol{a}_K^1\right] \,, \quad \boldsymbol{d} = \boldsymbol{a}_K^3 - \boldsymbol{a}_K^4 - \boldsymbol{a}_K^2 + \boldsymbol{a}_K^1 \,.$$

It is easy to verify $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{a}}^\ell) = \boldsymbol{a}_K^\ell$, $\ell = 1,2,3,4$, where $\widehat{\boldsymbol{a}}^\ell$ are the vertices of the unit square numbered counter-clockwise. The initial permutation ensures that $\mathbf{A}$ is regular.

Given $\boldsymbol{x} \in \mathbb{R}^2$, its pre-image $\widehat{x} = [\widehat{x}_1 \, \widehat{x}_2]^\top$ under $\boldsymbol{\Phi}_K$ must satisfy (**??**) with

$$\boldsymbol{q} := [q_1 \, q_2]^\top := \mathbf{A}^{-1}\boldsymbol{d} \in \mathbb{R}^2 \quad , \quad \boldsymbol{y} := [y_1 \, y_2]^\top := \mathbf{A}^{-1}(\boldsymbol{x} - \boldsymbol{t}) \in \mathbb{R}^2 \,,$$

which amounts to the *non-linear system of equations*:

$$\begin{aligned} \widehat{x}_1 + q_1\widehat{x}_1\widehat{x}_2 &= y_1 \,, \\ \widehat{x}_2 + q_2\widehat{x}_1\widehat{x}_2 &= y_2 \,, \end{aligned} \tag{3.3.6}$$

We multiply the first equation by q_2, the second by $-q_1$, and add both:

$$\blacktriangleright \qquad q_2x_1 - q_1x_2 = q_2y_1 - q_1y_2 \,.$$

Next, we have to distinguish two cases:

(I) If $\|\boldsymbol{q}\| \ll \|\boldsymbol{y}\|$ (test relative smallness !), then set $\widehat{\boldsymbol{x}} := \boldsymbol{y}$,

(II) else if $|q_2| \ll \|\boldsymbol{y}\|$ (test relative smallness !) then set $\widehat{x}_2 = y_2$ and $\widehat{x}_1 = \frac{1}{1+q_1y_2}y_1$

(III) else if $|q_1| \ll \|\boldsymbol{y}\|$ (test relative smallness !) then set $\widehat{x}_1 = y_1$ and $\widehat{x}_2 = \frac{1}{1+q_2y_1}y_2$

(IV) else

$$\begin{aligned} & x_2 = -\frac{1}{q_1}(q_2y_1 - q_1y_2 - q_2\widehat{x}_1) \,, \\ \Rightarrow \quad & q_2\widehat{x}_1^2 + (1 - q_2y_1 + q_1y_2)\widehat{x}_1 - y_1 = 0 \,. \end{aligned} \tag{3.3.7}$$

We arrive at a quadratic equation for the component$\widehat{x}_1$. If it fails to have a real solution, $\widehat{\boldsymbol{x}}$ does not exist and the function should return a vector of `NaN`s (“Not a number”, see **std::nan()**).

Otherwise there will usually be two solutions, of which we pick one in $[0,1]$, if it exists. If not, we choose the one with the smaller modulus. The other component can be recovered by the expression in the first lines of (**??**) and (**??**), respectively.

Finally, we have to reverse the initial cyclic permutation by rotating $\widehat{\boldsymbol{x}}$ by an angle $\rho \in \{0, \pi/2, \pi, 3\pi/2\}$ around $\begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix}$:

$$\widehat{\boldsymbol{x}} \leftarrow \begin{bmatrix} \cos(\rho) & -\sin\rho \\ \sin(\rho) & \cos(\rho) \end{bmatrix} \left(\widehat{\boldsymbol{x}} - \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix} \right) + \begin{bmatrix} 1/2 \\ 1/2 \end{bmatrix} \,.$$

The angle ρ depends on which vertex of K was permuted into $\boldsymbol{a}_K^1$ in the beginning.

C++11 code 3.3.8: Sub-problem (3-3.f), `GlobalInverseQuad()`: seeking largest sub-triangle ⇝ GitLab

```
2   // I: Find sub-triangle with largest area and use its
3   // middle vertex as new "vertex 0".
4   unsigned int vt_zero_idx{0};
5   double max_area{0.0};
6   for (int l = 0; l < 4; ++l) {
7     const double sub_tria_area =
8         triaArea(vert.col(l), vert.col((l + 1) % 4), vert.col((l + 2) % 4));
9     if (sub_tria_area > max_area) {
10      vt_zero_idx = (l + 1) % 4;
11      max_area = sub_tria_area;
12    }
13  }
14  // vt_zero_idx stores the index of the vertex that will now
15  // be regarded as "pivotal vertex 0"
```

C++11 code 3.3.9: Sub-problem (3-3.f), `GlobalInverseQuad()`: inverting parameterization ⇝ GitLab

```
2   // Use corners to calculate matrix A, the translation t
3   // and the coefficient vector d
4   Eigen::Matrix2d A(2, 2);
5   A.col(0) = corner1 - corner0;
6   A.col(1) = corner3 - corner0;
7   Eigen::Vector2d t = corner0;
8   Eigen::Vector2d d = corner2 + corner0 - corner1 - corner3;
9
10  // Calculate the coefficients in the non-linear system of equations
11  Eigen::Vector2d y = A.partialPivLu().solve(x - t);
12  Eigen::Vector2d q = A.partialPivLu().solve(d);
13  Eigen::Vector2d q_orth(q(1), -q(0));
14
15  // Treat exceptional cases
16  // If q vanishes
17  const double ref_size = y.norm();
18  if (q.norm() < kEPS * ref_size) {
19    x_hat = y;
20  } else if (std::abs(q(0)) < kEPS * ref_size) {
21    x_hat(0) = y(0);
22    x_hat(1) = y(1) / (1 + q(1) * x_hat(0));
23  } else if (std::abs(q(1)) < kEPS * ref_size) {
24    x_hat(1) = y(1);
25    x_hat(0) = y(0) / (1 + q(0) * x_hat(1));
26  } else {
27    // Generic case; solve quadratic equation
28    double a = q(0);
29    double b = 1 + y.dot(q_orth);
30    double c = -y(1);
31
32    auto [x_op1, x_op2] = solveQuadraticEquation(a, b, c);
33    // No solution
34    if (std::isnan(x_op1) || std::isnan(x_op2)) {
35      x_hat(0) = NAN;
36      x_hat(1) = NAN;
```

```
    } else {
      // There is a solution
      // Find root in the unit interval -> x_op1
      if ((x_op1 < 0.) || (x_op1 > 1.)) {
        if ((x_op2 >= 0.) && (x_op2 <= 1.)) {
          x_op1 = x_op2;
        }
      }
      x_hat(1) = x_op1;
      x_hat(0) = (y.dot(q_orth) + q(0) * x_hat(1)) / q(1);
    }
  }
```

C++11 code 3.3.10: Sub-problem (3-3.f), `GlobalInverseQuad()`: reversing initial permutation of vertices ↪ GitLab

```
  // Reverse initial permutation
  switch (vt_zero_idx) {
    case 1: {
      x_hat = ((Eigen::Matrix2d() << 0., -1., 1., 0.)).finished() * x_hat +
              Eigen::Vector2d(1.0, 0.0);
      break;
    }
    case 2: {
      x_hat = -x_hat + Eigen::Vector2d(1.0, 1.0);
      break;
    }
    case 3: {
      x_hat = ((Eigen::Matrix2d() << 0., 1., -1., 0.)).finished() * x_hat +
              Eigen::Vector2d(0.0, 1.0);
      break;
    }
  }
```

SOLUTION of (3-3.g):

The test relies on the fact that `GlobalInverseQuad()` is the inverse of the `Global()` method of an **lf::mesh::Entity** object of type QUAD.

C++11 code 3.3.11: Sub-problem (3-3.g): Implementation of `testGlobalInverseQuad()`
➥ GitLab

```
2  void testGlobalInverseQuad(const lf::mesh::Entity &quad, Eigen::Vector2d xh) {
3    LF_ASSERT_MSG(quad.RefEl() == lf::base::RefEl::kQuad(),
4                  "Cell must be a quadrilateral");
5    // get the coordinates of the corners of this cell
6    const lf::geometry::Geometry *geo_ptr = quad.Geometry();
7    auto vertices = lf::geometry::Corners(*geo_ptr);
8    // Image of point in unit square under parametric mapping
9    Eigen::Vector2d x = geo_ptr->Global(xh);
10   Eigen::Vector2d xh_comp = PointEvaluationRhs::GlobalInverseQuad(vertices, x);
11   EXPECT_NEAR((xh - xh_comp).norm(), 0.0, 1.0E-8)
12       << "quadl " << quad << ": Mismatch xh = " << xh << ", x = " << x
13       << ", xh_comp = " << xh_comp << std::endl;
14 }
```

SOLUTION of (3-3.h):

Since there is no minimizer, we expect the value of the quadratic functional on $\mathcal{S}_1^0(\mathcal{M}_k)$ to diverge towards negative infinity for $k \to \infty$.

SOLUTION of (3-4.a):

From the DOXYGEN documentation: Objects of type **lf::uscalfe::LinearFELaplaceElementMatrix** compute the $\mathcal{S}_1^0(\mathcal{M})$-element matrices for the bilinear form $\mathsf{a}(u,v) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$, which is associated with the strong-form PDE operator $-\Delta$.

SOLUTION of (3-4.b):

According to the DOXYGEN documentation the class **lf::uscalfe::FeSpaceLagrangeO1** encodes a lowest-order Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$ on a hybrid mesh $\mathcal{M}$. Objects of this type fit the **lf::uscalfe::** interface ($\rightarrow$ [Lecture $\rightarrow$ § 2.8.3.28]),

- give access to the underlying mesh,
- and contain a description of local shape functions for all types of mesh entities on the respective reference elements.

SOLUTION of (3-4.c):

See ➙ (documentation), which offers comprehensive explanations.

HINT 1 for (3-4.d): To understand how Dirichlet boundary conditions are imposed in LEHRFEM++, please re-read [Lecture → § 2.7.6.13]. ⌟

SOLUTION of (3-4.d):

From the LEHRFEM++ documentation you learn that the code relies on the element matrix provider class for $-\Delta$. The right-hand-side vector is not even intialized. Thus the code attempts to solve the BVP

$$\begin{aligned} -\Delta u(\mathbf{x}) &= 0 \quad \text{in } \Omega \\ u(\mathbf{x}) &= g(\mathbf{x}) \quad \text{on } \partial\Omega, \end{aligned} \tag{3.4.1}$$

where

$$g(\mathbf{x}) = \begin{cases} 0, & x_2 > 0 \\ 1 - x_2, & x_2 \leq 0 \end{cases}, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}. \tag{3.4.2}$$

The code relies on lowest-order (piecewise linear) Lagrangian finite elements.

SOLUTION of (3-4.e):

To begin with be aware that every affine linear function $u \in Cp_1(\mathbb{R}^2)$ satisfies $\Delta u = 0$. Thus a Dirichlet BVP of the form (**??**) will have a solution $u \in Cp_1(\mathbb{R}^2)$, if $g \in \mathcal{P}_1(\mathbb{R}^2)\big|_{\partial\Omega}$.

The Dirichlet data from (**??**) are linear and continuous for the left and the right domain. In particular, the solution $u(x) = 0$ for the left domain, and it will be a linear function for the right domain.

HINT 1 for (3-4.f): Curves of the same color and style in Fig. 39 and Fig. 40 refer to the same situation. ⌟

SOLUTION of (3-4.f):

Observe that for Domain 1 $g(x) = 0$, so the solution will be $u(x) = 0$ which belongs to the finite element space. Moreover the number 0 is a machine number. Therefore, for this case the solution $u = 0$ can be represented exactly in the discrete finite element model. This situation corresponds to curve #1 (blue), as its seminorm is zero in Fig. 40.

For Domain 3, $g(x) = 1 - x_2$, which leads to a solution function $u(x)$ which is also contained in the finite element space, However, this time the solution is not zero, so we observe a flat curve with error on the scalar of machine precision, but a non-zero constant for the solution's seminorm in Fig. 40. This is shown in curve #3 (black).

Finally, curve #2 (red) shows the behavior of the norms for Domain #2.

HINT 1 for (3-4.g): Study again [Lecture → § 1.9.0.6]. For the finite element code the discontinuous Dirichlet data will look like continuous with ever increasing slope as the mesh is refined. ⌟

SOLUTION of (3-4.g):

In [Lecture $\rightarrow$ § 1.9.0.6] We learned that piecewise smooth Dirichlet data have to be *continuous*! However, for the situation in Fig. 37 the Dirichlet data g jump at $x_2 = 0$. As a consequence, the boundary value problem cannot have a weak solution in $H^1(\Omega)$.

The solution's seminorm blows up due to the discontinuity in the Dirichlet data $g(x)$ on the middle domain. Indeed we also see that the error in Fig. 39 does not converge to zero.

HINT 1 for (3-5.a): Do not forget the appropriate function spaces. ⌟

SOLUTION of (3-5.a):

Recall the material of [Lecture → Section 1.8]. As in [Lecture → Ex. 1.8.0.6], we multiply with a test function v and integrate, then applying Green's formula

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad \text{[Lecture → Eq. (13.2.3.2)]}$$

and the impedance boundary condition:

$$\begin{aligned} -\int_\Omega v\Delta u \,\mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} (\boldsymbol{n} \cdot \mathbf{grad}\, u) v \,\mathrm{dS}(\boldsymbol{x}) \\ &= \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} uv \,\mathrm{dS}(\boldsymbol{x})\,. \end{aligned}$$

$$\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} uv \,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega fv \,\mathrm{d}\boldsymbol{x} \tag{3.5.2}$$

for all $v \in H^1(\Omega)$.

SOLUTION of (3-5.j):

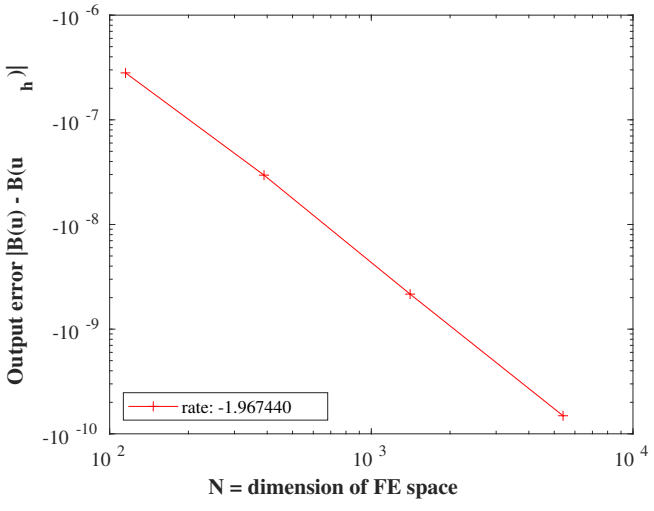
C++11 code 3.5.16: Sub-problem (3-5.i): function `bdFunctionalEval()` ↪ GITHUB

```
    // Load mesh into a Lehrfem++ object
    std::string mesh_file = "meshes/hex" + std::to_string(i) + ".msh";
    auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
    const lf::io::GmshReader reader(std::move(mesh_factory), mesh_file);
    auto mesh_p = reader.mesh();  // type shared_ptr<const lf::mesh::Mesh>

    // Finite element space
    auto fe_space = std::make_shared<linear_lagrange>(mesh_p);
    // Obtain local->global index mapping for current finite element space
    const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
    // Dimension of finite element space
    const lf::base::size_type N_dofs(dofh.NumDofs());
    results(i - 1, 0) = N_dofs;

    // Solve the boundary value problem with Robin boundary conditions
    Eigen::VectorXd sol_vec = solveBVP(fe_space);

    // Integrate the solution sol_vec over the flagged edges
    double bd_functional_val = bdFunctionalEval(fe_space, sol_vec);

    double error = std::abs(bd_functional_val - 2.081541059732923);
    results(i - 1, 1) = error;
```

See **??** for the implementation. This code produces the following output

```
(Ndofs = 115):  value: 2.081540778886353  error: -2.808465699999374e-07
(Ndofs = 389):  value: 2.081541030122551  error: -2.961037237625419e-08
(Ndofs = 1407):  value: 2.081541057570378  error: -2.162545253980852e-09
(Ndofs = 5405):  value: 2.081541059584104  error: -1.488196232912742e-10
```

The error versus the dimension of the finite element space is plotted in **??**.



◁ Convergence plot for Sub-problem (3-5.j).

The error points a nice aligned on a line with slope -2. This indicates *algebraic convergence* with rate 2.

HINT 1 for (3-5.k): Use the variational formulation of the boundary value problem and the special test function $v \equiv 1$ ⌟

SOLUTION of (3-5.k):

Using the variational formulation (**??**) with test function $v \equiv 1$, we end up with

$$\int_\Omega \mathbf{grad}\, u \cdot \underbrace{\mathbf{grad}\, 1}_{=\mathbf{0}} \,\mathrm{d}\boldsymbol{x} + B(u) = \int_\Omega f \,\mathrm{d}\boldsymbol{x}\,.$$

The result follows since $\mathbf{grad}\, 1 = \mathbf{0}$.

Exactly the same argument remains valid for the discrete variational problem posed on $\mathcal{S}_1^0(\mathcal{M})$, since $\{\boldsymbol{x} \mapsto 1\} \in \mathcal{S}_1^0(\mathcal{M})$ on *any* mesh. We combine this insight with the fact that $\{\boldsymbol{x} \mapsto 1\} \in \mathcal{S}_1^0(\mathcal{M})$ has the basis expansion coefficient vector $\mathbf{1}$ (due to the cardinal basis property of the tent functions) and that

$$\ell_h(\{\boldsymbol{x} \mapsto 1\}) = \mathbf{1}^\top \vec{\boldsymbol{\varphi}}\,,$$

where ℓ_h is the approximate right-hand-side functional realized by means of numerical quadrature.

SOLUTION of (3-5.l):

This is an immediate consequence of the (3.5.17), because in the absence of quadrature errors we have

$$B(u_h) = \mathbf{1}^\top \vec{\boldsymbol{\varphi}} = \int_\Omega f(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = B(u) \ .$$

HINT 1 for (3-5.b): Refine the argument of [Lecture → § 1.2.3.36]. ⌟

SOLUTION of (3-5.b):

The bilinear form

$$\mathrm{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} uv \,\mathrm{d}S(\boldsymbol{x}) \tag{3.5.3}$$

is symmetric and positive definite. The positivity of the bilinear form can be seen as follows: if it was zero for $u = v$, then u would have vanishing gradient, would necessarily have to be constant, and would have to vanish on $\partial\Omega$, which is possible only if $u = 0$.

HINT 1 for (3-5.c): To deal with the non-standard energy norm related to (3.5.1) you may rely on [Lecture → Thm. 1.8.0.20], [Lecture → Thm. 1.9.0.10], and consider the $H^1(\Omega)/L^2(\Omega)$-orthogonal splitting

$$H^1(\Omega) = H^1_*(\Omega) \oplus \{\boldsymbol{x} \mapsto c \in \mathbb{R}\} \,, \tag{3.5.4}$$

in combination with the Cauchy-Schwarz inequality in $L^2(\Omega)$. ⌟

SOLUTION of (3-5.c):

This problem is surprisingly difficult, because the energy does not comprise the $L^2(\Omega)$-norm. We explain two possible approaches, the first elementary, the second appealing to results from [Lecture → Suppl. 1.8.0.24].

❶ To establish the continuity of the right-hand side functional use (**??**):

$$\int_\Omega fv\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\overline{v}+v_*)\,\mathrm{d}\boldsymbol{x}\,,\quad \overline{v}(\boldsymbol{x}) = \frac{1}{|\Omega|}\int_\Omega v\,\mathrm{d}\boldsymbol{x}\,,\quad v_* := v-\overline{v}\,,$$

and note that thanks to elementary considerations and the Poincaré-Friedrichs-type inequality of [Lecture → Thm. 1.8.0.20] there is a constant $C = C(\Omega) > 0$ such that

$$\|v_*\|_{L^2(\Omega)} \le C\|\mathbf{grad}\, v_*\|_{L^2(\Omega)}\,, \tag{3.5.5}$$

$$\|\overline{v}\|_{L^2(\Omega)} \le C\|\overline{v}\|_{L^2(\partial\Omega)}\,. \tag{3.5.6}$$

Next, we appeal to the multiplicative trace estimate of [Lecture → Thm. 1.9.0.10] and get

$$\begin{aligned}\|v_*\|_{\mathsf{a}}^2 &= |v_*|_{H^1(\Omega)}^2 + \|v_*\|_{L^2(\partial\Omega)}^2\\ &\overset{\text{[Lecture → Thm. 1.9.0.10]}}{\le} |v_*|_{H^1(\Omega)}^2 + C\|v_*\|_{L^2(\Omega)}\|v_*\|_{H^1(\Omega)}\\ &\overset{\text{[Lecture → Thm. 1.8.0.20]}}{\le} C'|v_*|_{H^1(\Omega)}^2 \le C'\|v\|_{\mathsf{a}}^2\,.\end{aligned} \tag{3.5.7}$$

An application of the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)] finishes the argument: with constant depending only on Ω we have for all $v \in H^1(\Omega)$

$$\begin{aligned}\int_\Omega f(\overline{v}+v_*)\,\mathrm{d}\boldsymbol{x} &\le \|f\|_{L^2(\Omega)}\Big(\|\overline{v}\|_{L^2(\Omega)} + \|v_*\|_{L^2(\Omega)}\Big)\\ &\overset{(??)}{\le} C\|f\|_{L^2(\Omega)}(\|\overline{v}\|_{\mathsf{a}} + \|v_*\|_{\mathsf{a}})\\ &\overset{\triangle\text{-inequ.}}{\le} C'C'\|f\|_{L^2(\Omega)}(\|v\|_{\mathsf{a}} + \|v_*\|_{\mathsf{a}}) \overset{(??)}{\le} C''\|f\|_{L^2(\Omega)}\|v\|_{\mathsf{a}}\,.\end{aligned}$$

❷ The following result immediately shows that the energy norm induced by a from (**??**) is equivalent to the $H^1(\Omega)$-norm.

Theorem [Lecture → Thm. 1.8.0.26]. Poincaré-Friedrichs-type estimate

Let $\Omega \subset \mathbb{R}^d$ a bounded and connected Lipschitz domain and $F : H^1(\Omega) \to W$ a bounded/continuous linear operator, where W is some normed vector space.
If $F(\{x \mapsto 1\}) \neq 0$, then

$$\exists C > 0:\quad \|u\|_{L^2(\Omega)} \le C\big(\|\mathbf{grad}\, u\|_{L^2(\Omega)} + \|F(u)\|_W\big)\quad \forall u \in H^1(\Omega)\,.$$

[Lecture → Eq. (1.8.0.27)]

We apply this theorem with

- $W := L^2(\partial\Omega)$,
- $F(u) := u|_{\partial\Omega}$, which is a continuous mapping $F : H^1(\Omega) \to L^2(\partial\Omega)$ thanks to the multiplicative trace inequality:

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

$$\exists C = C(\Omega) > 0: \quad \|u\|^2_{L^2(\partial\Omega)} \leq C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega) \, .$$

Hence, from [Lecture → Thm. 1.8.0.26] we conclude

$$\|u\|_{L^2(\Omega)} \leq C\Big(\|\mathbf{grad}\, u\|_{L^2(\Omega)} + \|u\|_{L^2(\partial\Omega)}\Big) \leq C\sqrt{2}\|u\|_a \quad \forall u \in H^1(\Omega) \, ,$$

which yields the equivalence of the energy norm and the $H^1(\Omega)$-norm on $H^1(\Omega)$.

Finally, the continuity of $v \mapsto \int_\Omega fv \, \mathrm{d}\boldsymbol{x}$ in $H^1(\Omega)$ can easily be inferred from the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)], see [Lecture → Eq. (1.3.4.16)].

SOLUTION of (3-5.d):

The solution of the BVP (3.5.1) satisfies all assumptions the theorem requires for u; in particular, since $f \in L^2(\Omega)$ the theorem guarantees $u \in H^2(\Omega)$ and that, with some constant depending only on Ω

$$\|u\|_{H^2(\Omega)} \leq C\|f\|_{L^2(\Omega)} .$$

HINT 1 for (3-5.e):

- [Lecture → Thm. 3.1.3.7] applies and gives us quasi-optimality of Galerkin solutions so that *the energy norm of any interpolation error provides an upper bound for the energy norm of the finite element discretization error*.
- In order to deal with $L^2(\partial\Omega)$-norms of interpolation errors you can appeal to [Lecture → Thm. 1.9.0.1

⌟

SOLUTION of (3-5.e):

From (**??**) we learn that the energy norm induced by the bilinear form $\mathsf{a}(\cdot,\cdot)$ is

$$\|v\|_{\mathsf{a}}^2 := \|\mathbf{grad}\, v\|_{L^2(\Omega)}^2 + \|v\|_{L^2(\partial\Omega)}^2 \,. \tag{3.5.9}$$

Thus, by the optimality of finite element Galerkin solutions with respect to the energy norm according to [Lecture → Thm. 3.1.3.7] we know

$$\|\mathbf{grad}(u-u_i)\|_{L^2(\Omega)}^2 + \|u-u_i\|_{L^2(\partial\Omega)}^2 \le \|\mathbf{grad}(u-\mathsf{I}_1 u)\|_{L^2(\Omega)}^2 + \|u-\mathsf{I}_1 u\|_{L^2(\partial\Omega)}^2 \,,$$

where I_1 is the nodal interpolation operator $\mathsf{I}_1 : C^0(\overline{\Omega}) \to \mathcal{S}_1^0(\mathcal{M}_i)$ as introduced in [Lecture → Eq. (3.3.2.2
By the multiplicative trace inequality [Lecture → Thm. 1.9.0.10] we can further estimate

$$\|u-\mathsf{I}_1 u\|_{L^2(\partial\Omega)}^2 \le C\|u-\mathsf{I}_1 u\|_{L^2(\Omega)}\|u-\mathsf{I}_1 u\|_{H^1(\Omega)} \quad \text{with} \quad C = C(\Omega) > 0 \,.$$

Combining this with the estimates of

Theorem [Lecture → Thm. 3.3.2.21]. Error estimate for piecewise linear interpolation

For any $u \in C^2(\bar{\Omega})$ and 2D piecewise linear interpolation $\mathsf{I}_1 : C^0(\overline{\Omega}) \to \mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh, holds

$$\|u-\mathsf{I}_1 u\|_{L^2(\Omega)} \le \sqrt{\tfrac{3}{8}}\, h_{\mathcal{M}}^2 \left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)} \,,$$
$$\|\mathbf{grad}(u-\mathsf{I}_1 u)\|_{L^2(\Omega)} \le \sqrt{\tfrac{3}{24}}\, \rho_{\mathcal{M}} h_{\mathcal{M}} \left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)} \,.$$

where $h_{\mathcal{M}}$ denotes the mesh width (→ [Lecture → Def. 3.2.1.4]) and $\rho_{\mathcal{M}}$ the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$.

we obtain

$$\|\mathbf{grad}(u-u_i)\|_{L^2(\Omega)}^2 + \underbrace{\|u-u_i\|_{L^2(\partial\Omega)}^2}_{\ge 0} \le C(h_{\mathcal{M}}^2 + h_{\mathcal{M}}^3)|u|_{H^2(\Omega)}^2 \,, \tag{3.5.10}$$

where the constant $C > 0$ depends only on Ω and the shape-regularity measure of the triangulation.

Finally, note that $N_i \sim h_i^{-2}$, where h_i is the meshwidth of $\mathcal{M}_i$. According to [Lecture → Thm. 3.3.2.21] (see also [Lecture → Eq. (3.3.5.21)]), we find a constant $C > 0$, so that, for $N_i \to \infty$,

$$\|u-u_i\|_{H^1(\Omega)} \le C_2 N_i^{-1/2}|u|_{H^2(\Omega)} = O(N_i^{-1/2}) \,.$$

HINT 1 for (3-5.f):

- $L^2(\Omega)$-norm error estimates rely on dual BVP with the Galerkin discretization error as source function.
- Thm. 3.5.8 gives 2-regularity for the dual BVP and paves the way for applying the duality technique from [Lecture $\rightarrow$ Section 3.6.3].

⌟

SOLUTION of (3-5.f):

The duality techniques underlying $L^2(\Omega)$-estimates rely on the dual boundary value problem

$$-\Delta g = u - u_i \quad \text{in } \Omega \quad , \quad \boldsymbol{n} \cdot \mathbf{grad}\, g + g = 0 \quad \text{on } \partial\Omega \, , \tag{3.5.11}$$

whose variational formulation is: seek $g \in H^1(\Omega)$ such that

$$\int_\Omega \mathbf{grad}\, g \cdot \mathbf{grad}\, v \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} g\, v \, \mathrm{d}S = \int_\Omega (u - u_i) v \, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \, . \tag{3.5.12}$$

Now we use the result of Sub-problem (3-5.d) that ensures $g \in H^2(\Omega)$. By Thm. 3.5.8 we conclude

$$g \in H^2(\Omega) \quad , \quad \|g\|_{H^2(\Omega)} \leq C \|u - u_i\|_{L^2(\Omega)} \quad \text{with} \quad C = C(\Omega) > 0 \, . \tag{3.5.13}$$

Then apply the reasoning of [Lecture → § 3.6.3.3] with a slight change in the energy norm, which will also involve a boundary term now. We plug $v := u - u_i$ into the variational formulation (**??**) and get, using Galerkin orthogonality, for every $v_i \in \mathcal{S}_1^0(\mathcal{M}_i)$,

$$\begin{aligned} \|u - u_i\|^2_{L^2(\Omega)} &= \int_\Omega \mathbf{grad}(g - v_i) \cdot \mathbf{grad}(u - u_i) \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} (g - v_i)\,(u - u_i) \, \mathrm{d}S \\ &\leq \Big(\|\mathbf{grad}(g - v_i)\|^2_{L^2(\Omega)} + \|g - v_i\|^2_{L^2(\partial\Omega)} \Big)^{\frac{1}{2}} \cdot \\ &\quad \Big(\|\mathbf{grad}(u - u_i)\|^2_{L^2(\Omega)} + \|u - u_i\|^2_{L^2(\partial\Omega)} \Big)^{\frac{1}{2}} \end{aligned}$$

Then we can rely on the estimate (**??**) and those of [Lecture → Thm. 3.3.2.21] (refer to the solution of Sub-problem (3-5.e)) and conclude an asymptotic behavior $\|u - u_i\|_{L^2(\Omega)} = O(h^2_{\mathcal{M}})$.

Recall that $N_i \sim h_i^{-2}$, where h_i is the meshwidth of $\mathcal{M}_i$. Thus, there is a constant $C > 0$ such that, for $N_i \to \infty$,

$$\|u - u_i\|_{L^2(\Omega)} \leq C_1 N_i^{-1} |u|_{H^2(\Omega)} = O(N_i^{-1}) \, .$$

This means algebraic convergence with rate 1.

SOLUTION of (3-5.g):

$$\|u - u_i\|^2_{L^2(\partial\Omega)} \leq C_3 \|u - u_i\|_{L^2(\Omega)} \|u - u_i\|_{H^1(\Omega)} \leq C_1 C_2 C_3 N_i^{-3/2} |u|^2_{H^2(\Omega)}.$$

$$\|u - u_i\|_{L^2(\partial\Omega)} \leq C_4 h_i^{3/2} |u|_{H^2(\Omega)} = O(h_i^{3/2}) \ .$$

SOLUTION of (3-5.i):

The integral of a linear function v_h on a straight edge e with endpoint coordinates $\boldsymbol{p}_1, \boldsymbol{p}_2 \in \mathbb{R}^2$ evaluates to

$$\int_e v_h(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) = \tfrac{1}{2}|e|(v_h(\boldsymbol{p}_1) + v_h(\boldsymbol{p}_2)) \, .$$

This formula can be derived from the parameterization $\tau \mapsto \tau \boldsymbol{p}_2 + (1-\tau)\boldsymbol{p}_1, 0 \leq \tau \leq 1$.

C++ code 3.5.15: Sub-problem (3-5.i): function `bdFunctionalEval()` ➙ GITHUB

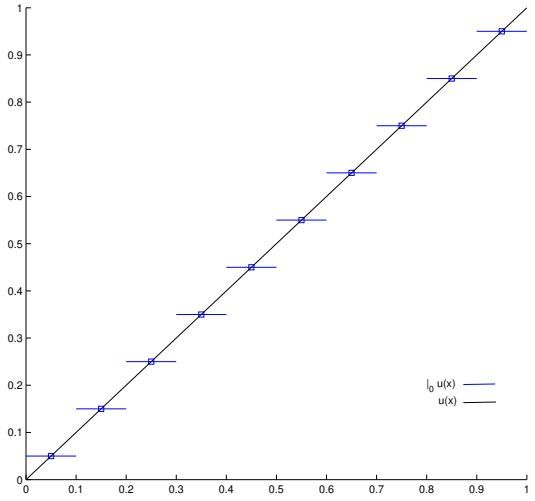
```
double bdFunctionalEval(
    std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space,
    Eigen::VectorXd &coeff_vec) {
  double bd_functional_val = 0;

  // Reference to mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p{fe_space->Mesh()};
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};

  // Obtain an array of boolean flags for the edges of the mesh: 'true'
  // indicates that the edge lies on the boundary. This predicate will guarantee
  // that the computations are carried only on the edges of the mesh.
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};

  // Computing the integral of function_vec on the flagged edges
  double edge_length;
  for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
    if (bd_flags(*edge)) {
      // Obtain endpoints of the edge
      auto endpoints = lf::geometry::Corners(*(edge->Geometry()));
      LF_ASSERT_MSG(endpoints.cols() == 2, "Wrong no endpoints in " << edge);
      // Compute length of the edge
      edge_length = (endpoints.col(1) - endpoints.col(0)).norm();
      // Find the endpoints global indices
      auto dof_idx = dofh.GlobalDofIndices(*edge);
      assert(dofh.NumLocalDofs(*edge) == 2);
      bd_functional_val +=
          (coeff_vec.coeff(dof_idx[0]) + coeff_vec.coeff(dof_idx[1])) *
          edge_length / 2.0;
    }
  }
  return bd_functional_val;
}
```

SOLUTION of (3-6.a):

We apply the definition of I_0 as from (3.6.2), and, since the interpolant is a staircase function, we obtain, by direct computation:

$$I_0 u = \sum_{j=1}^{M} \frac{1}{x_j - x_{j-1}} \int_{x_{j-1}}^{x_j} u(\xi)\, \mathrm{d}\xi \cdot \chi_{]x_{j-1},x_j[} = \sum_{j=1}^{M} \frac{1}{2} \frac{x_j^2 - x_{j-1}^2}{x_j - x_{j-1}} \cdot \chi_{]x_{j-1},x_j[}$$
$$= \sum_{j=1}^{M} \frac{1}{2} \frac{(x_j + x_{j-1})(x_j - x_{j-1})}{x_j - x_{j-1}} \cdot \chi_{]x_{j-1},x_j[} = \sum_{j=1}^{M} \frac{(x_j + x_{j-1})}{2} \cdot \chi_{]x_{j-1},x_j[}$$

Hence, $I_0 u = \frac{(x_j + x_{j-1})}{2}$ in $]x_{j-1}, x_j[$. The plot is given in **??**.



HINT 1 for (3-6.b): Use the fundamental theorem of calculus [Lecture → Eq. (1.5.1.6)]. Similar, even more complicated, developments are covered in [Lecture → § 3.3.1.6]. ⌟

SOLUTION of (3-6.b):

For $x_{j-1} \leq x \leq x_j$

$$(u - \mathsf{I}_0 u)(x) = u(x) - \mathsf{I}_0 u(x) = u(x) - \frac{1}{x_j - x_{j-1}} \int_{x_{j-1}}^{x_j} u(y)\,\mathrm{d}y$$

We tackle the right-hand side of (3.6.4). Note that,

$$\int_{x_{j-1}}^{x_j} \int_{y}^{x} u'(\xi) d\xi\,\mathrm{d}y = \int_{x_{j-1}}^{x_j} (u(x) - u(y))\,\mathrm{d}y = (x_j - x_{j-1})u(x) - \int_{x_{j-1}}^{x_j} u(y)\,\mathrm{d}y$$

Then, using the definition of I_0 as from (3.6.2):

$$\begin{aligned}(u - \mathsf{I}_0 u)(x) &= u(x) - \frac{1}{x_j - x_{j-1}} \int_{x_{j-1}}^{x_j} u(y)\,\mathrm{d}y \\ &= \frac{x_j - x_{j-1}}{x_j - x_{j-1}} u(x) - \frac{1}{x_j - x_{j-1}} \int_{x_{j-1}}^{x_j} u(y)\,\mathrm{d}y \\ &= \frac{1}{|x_j - x_{j-1}|} \int_{x_{j-1}}^{x_j} \int_{y}^{x} u'(\xi)\ \mathrm{d}\xi\,\mathrm{d}y\end{aligned}$$

HINT 1 for (3-6.c): Take the cue from what is done in [Lecture → § 3.3.1.6]. ⌟

HINT 2 for (3-6.c): Apply the Cauchy-Schwarz inequality for integrals

$$\left|\int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right| \leq \left(\int_\Omega |u(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2}\left(\int_\Omega |v(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2} = \|u\|_{L^2(\Omega)}\|v\|_{L^2(\Omega)}$$

[Lecture → Eq. (1.3.4.15)]

twice and for $\Omega =]x_{j-1}, x_j[$. ⌟

SOLUTION of (3-6.c):

Of course, we rely on the error representation formula (3.6.4)

$$\begin{aligned}\|u - \mathrm{I}_0 u\|^2_{L^2(]x_{j-1},x_j[)} &\overset{(3.6.4)}{=} \int_{x_{j-1}}^{x_j} \left| \frac{1}{|x_j - x_{j-1}|} \int_{x_{j-1}}^{x_j} \int_y^x u'(\xi)\,\mathrm{d}\xi\,\mathrm{d}y \right|^2 \mathrm{d}x \\ &= \left(\frac{1}{|x_j - x_{j-1}|}\right)^2 \int_{x_{j-1}}^{x_j} \left| \int_{x_{j-1}}^{x_j} 1 \cdot \int_y^x u'(\xi)\,\mathrm{d}\xi\,\mathrm{d}y \right|^2 \mathrm{d}x\,.\end{aligned}$$

As already indicated above, next we use the Cauchy-Schwarz inequality on $L^2(]x_{j-1}, x_j[)$ and, subsequently, on $L^2(]x, y[)$:

$$\begin{aligned}\|u - \mathrm{I}_0 u\|^2_{L^2(]x_{j-1},x_j[)} &\le \left(\frac{1}{|x_j - x_{j-1}|}\right)^2 \int_{x_{j-1}}^{x_j} \Big(\Big(\int_{x_{j-1}}^{x_j} 1^2\,\mathrm{d}y \Big) \Big(\int_{x_{j-1}}^{x_j} \left| \int_y^x u'(\xi)\,\mathrm{d}\xi \right|^2 \mathrm{d}y \Big) \Big)\,\mathrm{d}x \\ &= \left(\frac{1}{|x_j - x_{j-1}|}\right)^2 \int_{x_{j-1}}^{x_j} (x_j - x_{j-1}) \Big(\int_{x_{j-1}}^{x_j} \left| \int_y^x u'(\xi)\,\mathrm{d}\xi \right|^2 \mathrm{d}y \Big)\,\mathrm{d}x \\ &= \frac{1}{|x_j - x_{j-1}|} \int_{x_{j-1}}^{x_j} \left(\int_{x_{j-1}}^{x_j} \left| \int_y^x 1 \cdot u'(\xi)\,\mathrm{d}\xi \right|^2 \mathrm{d}y \right) \mathrm{d}x \\ &\le \frac{1}{|x_j - x_{j-1}|} \int_{x_{j-1}}^{x_j} \Big(\int_{x_{j-1}}^{x_j} \Big(\Big(\int_{x_{j-1}}^{x_j} 1^2\,\mathrm{d}\xi \Big) \Big(\int_{x_{j-1}}^{x_j} |u'(\xi)|^2\,\mathrm{d}\xi \Big) \Big)\,\mathrm{d}y \Big)\,\mathrm{d}x \\ &= \int_{x_{j-1}}^{x_j} \int_{x_{j-1}}^{x_j} |u|^2_{H^1(]x_{j-1},x_j[)}\,\mathrm{d}y\,\mathrm{d}x = |x_j - x_{j-1}|^2 |u|^2_{H^1(]x_{j-1},x_j[)}\,,\end{aligned}$$

where for the last inequality, while applying Cauchy-Schwarz, we also used the positivity of the integrand and the monotonicity of the interval to estimate $\int_y^x |u'(\xi)|^2\,\mathrm{d}\xi \le \int_{x_{j-1}}^{x_j} |u'(\xi)|^2\,\mathrm{d}\xi = |u|^2_{H^1(]x_{j-1},x_j[)}$.

SOLUTION of (3-6.d):

Of course, we rely on the *localization idea* as expressed by [Lecture → Eq. (3.3.1.7)]:

$$\begin{aligned}\|u - \mathrm{I}_0 u\|^2_{L^2(]a,b[)} &= \sum_{\mathcal{M}} \|u - \mathrm{I}_0 u\|^2_{L^2(]x_{j-1},x_j[)} \leq \sum_{j=1}^{M} |x_j - x_{j-1}|^2 |u|^2_{H^1(]x_{j-1},x_j[)} \\ &\leq h^2_{\mathcal{M}} |u|^2_{H^1(]a,b[)}\end{aligned}$$

Hence

$$\|u - \mathrm{I}_0 u\|_{L^2(]a,b[)} \leq h_{\mathcal{M}} |u|_{H^1(]a,b[)}.$$

SOLUTION of (3-7.a):

We face a BVP with homogeneous Dirichlet boundary conditions and can employ the standard arguments from [Lecture → Section 1.8] and [Lecture → Ex. 1.8.0.2]. The reaction term is not affected by integration by parts.

$$u \in H_0^1(\Omega): \quad \underbrace{\int_\Omega (1-c)\,\mathbf{grad}\, u \cdot \mathbf{grad}\, v + cu\,v \,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = \underbrace{\int_\Omega f\,v\,\mathrm{d}\boldsymbol{x}}_{=:\ell(v)} \quad \forall v \in H_0^1(\Omega)\ . \tag{3.7.2}$$

SOLUTION of (3-7.j):

Only the contribution from the reaction term cu is affected. Since the local shape functions are a cardinal basis withg respect to the quadrature nodes of the local trapezoidal rule, now the 3×3 element mass matrix becomes diagonal:

$$\mathbf{M}_K^{\mathrm{TR}} = \frac{1}{6} h^2 \mathbf{I} \,. \tag{3.7.12}$$

This entails straightforward modifications of the code for `computeGalerkinMatrix()`.

C++11 code 3.7.13: Sub-problem (3-7.k): function `computeGalerkinMatrixTR()`, ↪ GitLab

```
2  Eigen::SparseMatrix<double> computeGalerkinMatrixTR(int M, double c) {
3    Eigen::Matrix3d B_K;
4    double h = 1.0 / (M + 1);
5    Eigen::Matrix3d A_K;
6    A_K << 1.0, -0.5, -0.5, -0.5, 0.5, 0.0, -0.5, 0.0, 0.5;
7    Eigen::Matrix3d M_K = h * h / 6.0 * Eigen::Matrix3d::Identity();
8    B_K = (1.0 - c) * A_K + c * M_K;
9    return assemble(M, B_K);
10 }
```

HINT 1 for (3-7.k): Follow the arguments elaborated in [Lecture → § 3.7.2.2] (“averaging argument”). ⌟

SOLUTION of (3-7.k):

The case $c = 0$ is discussed in [Lecture → § 3.7.2.2] so that we can confine ourselves to $0 < c < 1$.

As in [Lecture → § 3.7.2.2] we use double indices to identify global shape functions and the entries of the basis expansion coefficient vector. To begin with the Galerkin matrix (or the difference stencil, whichever you prefer) gives us the linear relationship between the components of the solution vector, which agree with values of the finite element solution in nodes of the mesh:

$$(1-c)(4\mu_{i,j} - \mu_{i-1,j} - \mu_{i+1,j} - \mu_{i,j-1} - \mu_{i,j+1}) + ch^2\mu_{i,j} = \phi_{i,j} \leq 0\,. \tag{3.7.14}$$

Here (i,j) are the mesh coordinates of an *interior nodes*. Note that thanks to the use of local quadrature based on the 2D trapezoidal rule the value $\Phi_{i,j}$ is a scaled f-value in a node of the mesh, which accounts for the sign. Thus (**??**) leads to

$$(4 + \frac{c}{1-c}h^2)\mu_{i,j} = \underbrace{\frac{1}{1-c}\phi_{i,j}}_{<0} + (\mu_{i-1,j} + \mu_{i+1,j} + \mu_{i,j-1} + \mu_{i,j+1})\,.$$

$$\mu_{i,j} \leq \frac{1}{4 + \frac{c}{1-c}h^2}(\mu_{i-1,j} + \mu_{i+1,j} + \mu_{i,j-1} + \mu_{i,j+1})$$

$$\blacktriangleright \quad \mu_{i,j} \leq \frac{4}{4 + \frac{c}{1-c}h^2}\max\{\mu_{i-1,j}, \mu_{i+1,j}, \mu_{i,j-1}, \mu_{i,j+1}\}\,.$$

Now assume that $\mu_{i,j}$ is the entry of the coefficient vector μ of the solution which is *maximal* among all nodal values. Since $c > 0$, this immediately leads to the contraction

$$\max\{\mu_{i-1,j}, \mu_{i+1,j}, \mu_{i,j-1}, \mu_{i,j+1}\} \geq \left(1 + \frac{c}{4(1-c)}h^2\right) \cdot \mu_{i,j} > \mu_{i,j}\,.$$

As a consequence the maximum of the nodal values must be attained in a node on the boundary, for which all nodal values are zero. The finite element solution cannot be larger than 0 in any node.

SOLUTION of (3-7.b):

We rely on [Lecture → Thm. 1.4.1.8]:

Theorem [Lecture → Thm. 1.4.1.8]. Equivalence of quadratic minimization problem and linear variational problem

For a (generalized) quadratic functional $J(v) = \frac{1}{2}\mathsf{a}(v,v) - \ell(v) + c$ on a vector space V and with a symmetric positive definite bilinear form $\mathsf{a} : V \times V \to \mathbb{R}$ are equivalent:

(i) The quadratic minimization problem for J has unique minimizer $u_* \in \widehat{V}$ over the affine subspace $\widehat{V} = g + V_0$, $g \in V$.

(ii) The linear variational problem

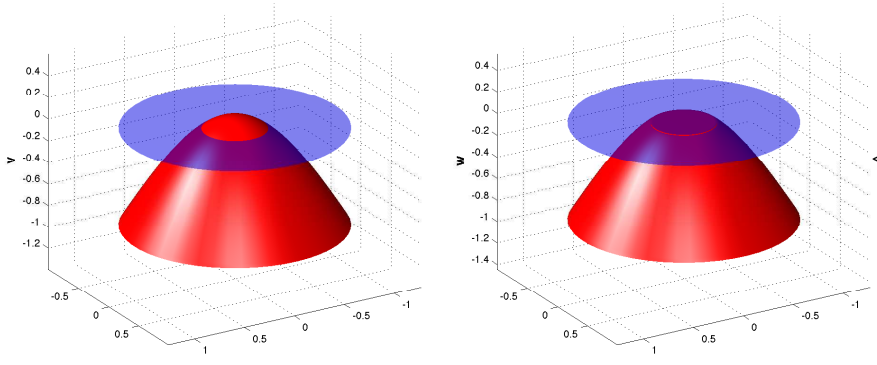
$$u \in \widehat{V}: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,,$$

has a unique solution $u_* \in \widehat{V}$.

The functional to be minimized is $J(u) = \frac{1}{2}\mathsf{a}(u,u) - \ell(u)$, where a and ℓ are the bilinear and linear forms associated with the weak form of (3.7.1), respectively, see (**??**). That is,

$$u = \operatorname*{argmin}_{u \in H^1_0(\Omega)} \frac{1}{2} \int_\Omega (1-c) \|\mathbf{grad}\, u(\boldsymbol{x})\|^2 + c u(\boldsymbol{x})^2 \,\mathrm{d}\boldsymbol{x} - \int_\Omega f(\boldsymbol{x}) u(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,.$$

HINT 1 for (3-7.c): Employ a reasoning similar to that in the “visual proof” of the maximum principle in the lecture, see **??** and **??**.



◁ A visual cue for the proof requested in Sub-problem (3-7.c).

⌞

SOLUTION of (3-7.c):

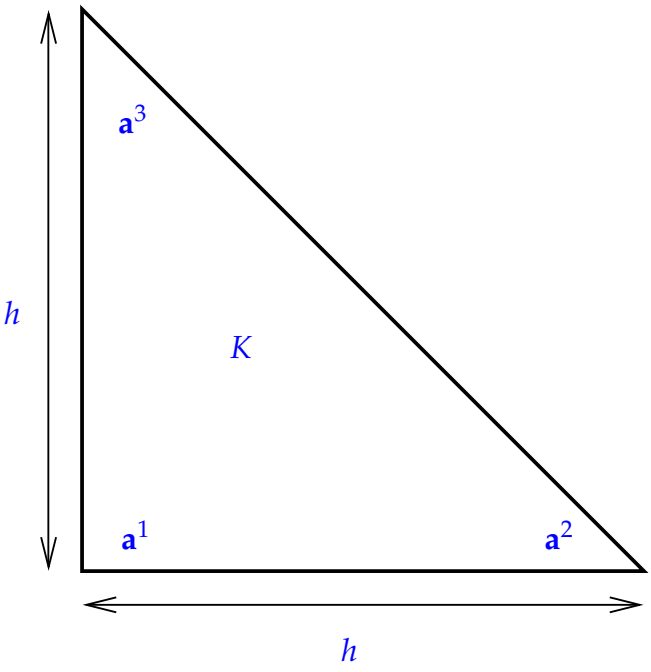
For the solution u it holds that

$$u = \underset{v=0 \text{ on } \partial\Omega}{\operatorname{argmin}} \frac{1}{2} \int_\Omega (1-c)\|\mathbf{grad}\, v\|^2 + c|v|^2 + (-f)v \, \mathrm{d}\boldsymbol{x} \tag{3.7.4}$$

Assume that there the maximum of u is attained a point $\boldsymbol{x}^\star$ with $u(\boldsymbol{x}^\star) > 0$. Then if we cut off the tip of the "mountain" centered at $\boldsymbol{x}^\star$, where after the cut-off the values are still positive, then we only decreased the quantity on the r.h.s. of (**??**), which is a contradiction to u being the argmin, thus there can be no points with $u(\boldsymbol{x}^\star) > 0$.

The condition $f \leq 0$ is important, because otherwise we might increase the value of the functional by this procedure. As it is, all three terms become smaller.

SOLUTION of (3-7.d):



The local shape functions are the barycentric coordinate functions λ_ℓ, $\ell = 1,2,3$. Their gradients on the triangle sketched beside with cathetis aligned with coordinate axes are [Lecture → Eq. (2.4.5.4)]

$$\mathbf{grad}\,\lambda_1 = h^{-1}\begin{bmatrix}-1\\-1\end{bmatrix},$$
$$\mathbf{grad}\,\lambda_2 = h^{-1}\begin{bmatrix}1\\0\end{bmatrix},$$
$$\mathbf{grad}\,\lambda_3 = h^{-1}\begin{bmatrix}0\\1\end{bmatrix},$$

Alternatively, you may just use the formula [Lecture → Eq. (2.4.5.8)] with $\omega_1 = \frac{\pi}{2}, \omega_2 = \frac{\pi}{4}, \omega_3 = \frac{\pi}{4}$.

This leads to the element matrix

$$\mathbf{A}_K = \frac{1}{2}\begin{bmatrix}2 & -1 & -1\\-1 & 1 & 0\\-1 & 0 & 1\end{bmatrix}. \tag{3.7.5}$$

SOLUTION of (3-7.e):

Again, the element mass matrix $\mathbf{M}_K$ is the same for all $K \in \mathcal{M}$. We can invoke [Lecture → Lemma 2.7.5.5] for $d = 2$,

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1! \alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, .$$

[Lecture → Eq. (2.7.5.6)]

and get

$$\mathbf{M}_K = \frac{1}{12}|K| \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} \, . \tag{3.7.6}$$

SOLUTION of (3-7.f):

Since the tent basis function are associated with *interior* nodes of the mesh, the dimension $\dim \mathcal{S}^0_{1,0}(\mathcal{M})$ is equal to the number of interior nodes of $\mathcal{M}$, of which there are M^2.

Each interior node is connected with six others via edges of the mesh. Edges give rise to non-zero entries of the Galerkin matrix $\mathbf{A}$, see [Lecture $\rightarrow$ Section 2.4.4]. Hence, every row/column of $\mathbf{A}$ can contain at most seven non-zero entries.

HINT 1 for (3-7.g): Provided the maximal number of non-zero entries per row/column is known, the initialization of a sparse matrix in EIGEN can be done efficiently by means of the member functions `reserve()` and `coeffRef()`, see ➙ documentation. ⌟

SOLUTION of (3-7.g):

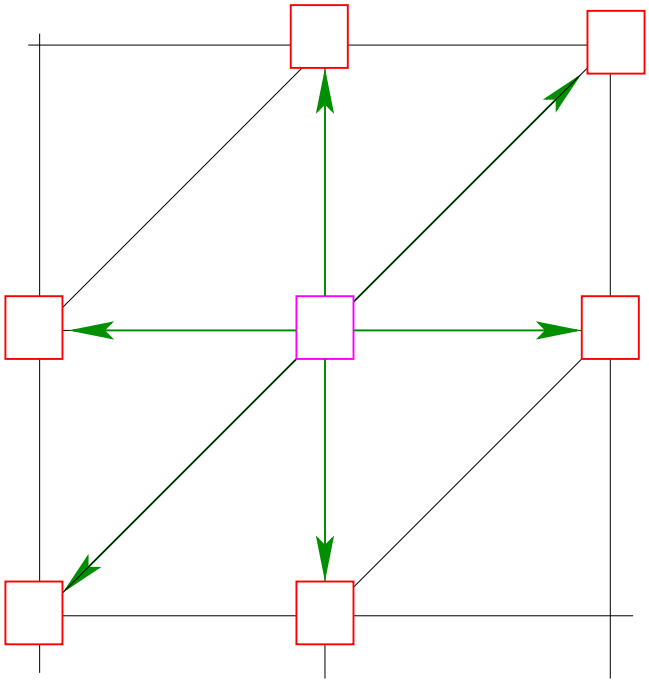
We first set up the element matrix and then do cell-oriented assembly exploiting the special very regular structure of the matrix in the auxiliary function `assemble()`.

C++11 code 3.7.7: Sub-problem (3-7.g): function `computeGalerkinMatrix()`, ↪ GitLab

```
2  Eigen::SparseMatrix<double> computeGalerkinMatrix(int M, double c) {
3    Eigen::Matrix3d B_K;
4    double h = 1.0 / (M + 1);
5    Eigen::Matrix3d A_K;
6    A_K << 1.0, -0.5, -0.5, -0.5, 0.5, 0.0, -0.5, 0.0, 0.5;
7    Eigen::Matrix3d M_K;
8    M_K << 2.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 2.0;
9    M_K *= h * h / 24.0;
10   B_K = (1.0 - c) * A_K + c * M_K;
11   return assemble(M, B_K);
12 }
```

The rows/columns of the Galerkin matrix correspond to interior nodes of the mesh, which can be indexed by their position $(i,j) \in \{1,\ldots,M\}$ in a 2D tensor-product lattice. The tent function with position index (i,j) located at node $\begin{bmatrix} ih \\ jh \end{bmatrix}$ has the global index (mathematical indexing!)

$$\mathrm{idx}(i,j) = (j-1) * M + i\,, \quad i,j \in \{1,\ldots,M\}\,. \tag{3.7.8}$$



Edges of the mesh connect a node with position index (i,j) to six other nodes with position indices

$$(i+1,j),(i,j+1),(i+1,j+1)\,,$$
$$(i-1,j),(i,j-1),(i-1,j-1)\,,$$

where indices $= 0$ mean nodes on $\partial\Omega$.

Since edges of $\mathcal{M}$ correspond to potentially non-zero entries of the $\mathcal{S}^0_{1,0}(\mathcal{M})$-Galerkin matrix, its structure can be described by a 7-point stencil.

Thus, we can traverse them in double loop with range $1,\ldots,M$.

In the loop body set the non-zero entries for the current row. Careful bookkeeping for the indices is required. In particular, nodes next to the boundary require a special treatment.

C++11 code 3.7.9: Sub-problem (3-7.g): Auxiliary function for specialized assembly on a tensor-product triangular mesh. ↪ GitLab

```cpp
Eigen::SparseMatrix<double> assemble(int M, const Eigen::Matrix3d &B_K) {
  int M2 = M * M;
  Eigen::SparseMatrix<double> A(M2, M2);
  double near_neighbour_contribution = 2.0 * B_K(0, 1);
  double far_neighbour_contribution = 2.0 * B_K(1, 2);
  double self_contribution = 2.0 * (B_K(0, 0) + B_K(1, 1) + B_K(2, 2));

  std::vector<double> contribution = {
      far_neighbour_contribution,  near_neighbour_contribution,
      near_neighbour_contribution, self_contribution,
      near_neighbour_contribution, near_neighbour_contribution,
      far_neighbour_contribution};

  std::vector<Eigen::Vector2i> shift = {{-1, -1}, {0, -1}, {-1, 0}, {0, 0},
                                        {1, 0},   {0, 1},  {1, 1}};

  std::vector<Eigen::Triplet<double>> tripletList;
  for (int i = 0; i < M; ++i) {
    for (int j = 0; j < M; ++j) {
      Eigen::Vector2i self = Eigen::Vector2i(i, j);
      for (int k = 0; k < 7; ++k) {
        Eigen::Vector2i other = self + shift[k];
        if (0 <= other(0) && other(0) < M && 0 <= other(1) && other(1) < M) {
          tripletList.push_back(Eigen::Triplet<double>(
              self(0) + M * self(1), other(0) + M * other(1), contribution[k]));
        }
      }
    }
  }
  A.setFromTriplets(tripletList.begin(), tripletList.end());
  return A;
}
```

Actually, since the number of non-zero entries per row/column of the Galerkin matrix is known in advance, one can directly set matrix entries after reserving enough space via EIGEN's `reserve()` method of **Eigen::SparseMatrix** ↪ documentation.

SOLUTION of (3-7.h):

Refer to [Lecture → § 2.4.6.9] for details. We get $(\vec{\varphi})_\ell = h^2 f(\boldsymbol{x}^\ell)$, $\boldsymbol{x}^\ell \in \mathcal{V}(\mathcal{M})$ a node of the mesh.

C++11 code 3.7.10: Sub-problem (3-7.h): function `computeLoadVector()`, ⇝ GitLab

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd computeLoadVector(int M, FUNCTOR &&f) {
4    Eigen::VectorXd phi(M * M);
5    double h = 1.0 / (M + 1);
6    for (int j = 0; j < M; ++j) {
7      for (int i = 0; i < M; ++i) {
8        phi(i + M * j) = h * h * f(h * (i + 1), h * (j + 1));
9      }
10   }
11   return phi;
12 }
```

HINT 1 for (3-7.i): Try $c \approx 1$, compute the inverse $\mathbf{A}^{-1}$ of the finite element Galerkin matrix $\mathbf{A}$ and watch out for negative entries of $\mathbf{A}^{-1}$. All this can be done computer-assisted, of course! ⌟

SOLUTION of (3-7.i):

You can achieve this by inspecting $\mathbf{A}^{-1}$ for negative entries, and manipulating f so that the load vector is close to zero everywhere except for an element corresponding to a column of $\mathbf{A}^{-1}$ with a negative element. These pop up more frequently for values of c close to 1; $c = 0.99$.

C++ code 3.7.11: Sub-problem (3-7.i): Implementation of counterexample, ↪ GitLab

```
2   unsigned int M = 4;
3   double c = 0.99;
4   std::function<double(double, double)> f = [](double x, double y) {
5     double ret = 0.0;
6     double dx = x - 0.25;
7     double dy = y - 0.25;
8     ret = std::exp(-100.0 * (dx * dx + dy * dy));
9     return ret;
10  };
11
12  Eigen::SimplicialLLT<Eigen::SparseMatrix<double>> solver;
13  Eigen::VectorXd phi = computeLoadVector(M, f);
14
15  Eigen::SparseMatrix<double> A = computeGalerkinMatrix(M, c);
16  Eigen::VectorXd mu = solver.compute(A).solve(phi);
17  std::cout << "mu = " << std::endl << mu << std::endl;
```

SOLUTION of (3-8.a):

Note that we face *mixed boundary conditions*, Dirichlet boundary conditions on one part (Γ_g) of $\partial\Omega$ and impedance boundary conditions [Lecture → Ex. 1.7.0.5] on another (Γ_0). Thus we have to follow the considerations of both [Lecture → Ex. 1.8.0.2] and [Lecture → Ex. 1.8.0.6] and the steps laid out in the beginning of [Lecture → Section 1.8]:

1. Multiply the PDE with a smooth test function v vanishing on the Dirichlet boundary part Γ_g,

   $$\blacktriangleright \quad -\int_\Omega \Delta u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^\infty(\Omega),\ v|_{\Gamma_g} = 0\,.$$

2. Integrate by parts using Green's first formula [Lecture → Thm. 1.5.2.7], which shifts the divergence in $-\Delta = -\operatorname{div}\mathbf{grad}$ as a $\mathbf{grad}$ onto the test function and introduces a boundary term

   $$\blacktriangleright \quad \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v\,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \mathbf{grad}\, u \cdot \boldsymbol{n}\, v\,\mathrm{d}S = 0 \quad \forall v \in C^\infty(\Omega),\ v|_{\Gamma_g} = 0\,,$$

   where $\boldsymbol{n}$ is the exterior unit normal with respect to Ω, $\boldsymbol{n} = -\boldsymbol{n}_0$!

3. Use the impedance boundary conditions (3.8.1b) to replace $\mathbf{grad}\, u \cdot \boldsymbol{n}_0$ with u in the boundary term,

4. pick the largest Sobolev function spaces as trial and test space that still guarantee continuity of the bilinear form resulting from the previous steps.

This yields the linear variational problem

$$u \in V\!: \quad \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v\,\mathrm{d}\boldsymbol{x} + \int_{\Gamma_0} uv\,\mathrm{d}S = 0 \quad \forall v \in V_0 \quad \forall v \in V_0\,, \tag{3.8.3}$$

where the appropriate Sobolev spaces are

$$V_0 := \{v \in H^1(\Omega) :\ v|_{\Gamma_g} = 0\} \qquad \text{(test space, a vector space)}\,, \tag{3.8.4a}$$

$$V := \{v \in H^1(\Omega) :\ v|_{\Gamma_g} = g\} \qquad \text{(trial space, an affine space)}\,. \tag{3.8.4b}$$

Note that test functions should vanish on Γ_0 according to the guideline "do not test where the solution is known". Therefore boundary terms for Γ_g do not occur in the variational formulation.

SOLUTION of (3-8.b):

We have to show that for $g \equiv 0$ the variational problem (3.8.3) has only the trivial solution $u \equiv 0$. This amounts to showing that the bilinear form on the left-hand-side of (3.8.3) is positive definite. We follow the considerations of [Lecture → § 1.2.3.36].

If $g \equiv 0$, we have $V = V_0$ and $u \in V_0$ is a solution of the variational problem, then

$$\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\Gamma_0} uv \,\mathrm{d}S = 0 \quad \forall v \in V_0 \,.$$

We can set $v := u$, which implies $\mathbf{grad}\, u = 0$ so that u has to be constant. Since it vanishes on Γ_g, we conclude $u \equiv 0$.

Alternatively, we can use the following result with $F(u) = u|_{\Gamma_0}$ and $W := L^2(\Gamma_0)$.

Theorem [Lecture → Thm. 1.8.0.26]. Poincaré-Friedrichs-type estimate

Let $\Omega \subset \mathbb{R}^d$ a bounded and connected Lipschitz domain and $F : H^1(\Omega) \to W$ a bounded/continuous linear operator, where W is some normed vector space.
If $F(\{\boldsymbol{x} \mapsto 1\}) \neq \mathbf{0}$, then

$$\exists C > 0: \quad \|u\|_{L^2(\Omega)} \leq C\big(\|\mathbf{grad}\, u\|_{L^2(\Omega)} + \|F(u)\|_W\big) \quad \forall u \in H^1(\Omega) \,.$$

[Lecture → Eq. (1.8.0.27)]

The continuity of F is a consequence of the multiplicative trace inequality [Lecture → Thm. 1.9.0.10]. Thus, [Lecture → Thm. 1.8.0.26] implies

$$\|\mathbf{grad}\, u\|_{L^2(\Omega)} + \|u\|_{L^2(\Gamma_0)} = 0 \quad \Rightarrow \quad u \equiv 0 \,.$$

We do not even need that $u|_{\Gamma_g} = 0$!

HINT 1 for (3-8.c): See [Lecture → Ex. 2.7.4.41].

SOLUTION of (3-8.c):

[Lecture → Ex. 2.7.4.41] discusses the assembly of the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrix for the bilinear form on the left-hand-side of (3.8.3). Of course, for the function `solveImpedanceBVP()` only the edges on Γ_0 should be visited by the `Eval()` method of **lf::fe::MassEdgeMatrixProvider**.

Of course, we also have to take care of the non-zero Dirichlet boundary conditions on Γ_g, as demonstrated in [Lecture → Code 2.7.6.17]; use the function `lf::assemble::FixFlaggedSolutionCompAlt(` with a **SELECTOR** argument that singles out edges on Γ_g. To that end query the position of the edge and rely on the information $\Omega_0 \subset [0.1, 0.9]^2$.

C++ code 3.8.5: Function `solveImpedanceBVP`: Computation of volume contribution to Galerkin matrix ➙ GITHUB

```
Eigen::VectorXd solveImpedanceBVP(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
    Eigen::Vector2d g) {
  // Related implementations:
  // Homework problem ErrorEstimatesForTraces:
  //
     https://gitlab.math.ethz.ch/ralfh/npdecodes/tree/master/homeworks/ErrorEstimatesFor

  // Pointer to current mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());

  Eigen::VectorXd discrete_solution(N_dofs);

  // I : ASSEMBLY
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Right hand side vector, must be initialized with 0!
  Eigen::Matrix<double, Eigen::Dynamic, 1> phi(N_dofs);
  phi.setZero();

  // I.i : Computing volume matrix for negative Laplace operator
  // Initialize object taking care of local mass (volume) computations.
  lf::uscalfe::LinearFELaplaceElementMatrix elmat_builder{};
  // Invoke assembly on cells (co-dimension = 0 as first argument)
  // Information about the mesh and the local-to-global map is passed through
  // a Dofhandler object, argument 'dofh'. This function call adds triplets to
  // the internal COO-format representation of the sparse matrix A.
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
```

C++ code 3.8.6: Function `solveImpedanceBVP`: Assembly of boundary contribution to Galerkin matrix, *cf.* boundary term in (3.8.3). ➙ GITHUB

```
  // I.ii : Computing mass edge matrix resulting from Robin B.C.
  // Obtain an array of boolean flags for the edges of the mesh, 'true'
  // indicates that the edge lies on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
```

```
  // Creating a predicate that will guarantee that the computations are carried
  // only on the interior boundary edges of the mesh using the boundary flags
  auto edges_predicate_RobinBC =
      [&bd_flags](const lf::mesh::Entity &edge) -> bool {
    if (bd_flags(edge)) {
      auto endpoints = lf::geometry::Corners(*(edge.Geometry()));
      return endpoints(0, 0) > 0.05 && 0.95 > endpoints(0, 0) &&
             endpoints(1, 0) > 0.05 && 0.95 > endpoints(1, 0);
    }
    return false;
  };
  // Coefficients used in the class template
  // MassEdgeMatrixProvider< SCALAR, COEFF, EDGESELECTOR >
  auto eta = lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> double { return 1.0; });
  lf::uscalfe::MassEdgeMatrixProvider<double, decltype(eta),
                                      decltype(edges_predicate_RobinBC)>
      edgemat_builder(fe_space_p, eta, edges_predicate_RobinBC);
  // Invoke assembly on edges by specifying co-dimension = 1
  lf::assemble::AssembleMatrixLocally(1, dofh, dofh, edgemat_builder, A);
```

C++ code 3.8.7: Function `solveImpedanceBVP`: Assembly of right-hand-side vector and solution of linear system of equations. ⇝ GITHUB

```
  // I.iii : Computing right-hand side vector
  // Right-hand side source function f
  auto mf_f = lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> double { return 0.0; });
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      elvec_builder(fe_space_p, mf_f);
  // Invoke assembly on cells (codim == 0)
  AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // I.iv : Imposing essential boundary conditions
  // Dirichlet data
  auto mf_g = lf::mesh::utils::MeshFunctionGlobal(
      [&g](Eigen::Vector2d x) -> double { return g.dot(x); });
  // Inspired by the example in the documentation of
  // InitEssentialConditionFromFunction()
  //
     https://craffael.github.io/lehrfempp/namespacelf_1_1uscalfe.html#a5afbd94919f0382cf
  // Creating a predicate that will guarantee that the computations are carried
  // only on the exterior boundary edges of the mesh using the boundary flags
  auto edges_predicate_Dirichlet =
      [&bd_flags](const lf::mesh::Entity &edge) -> bool {
    if (bd_flags(edge)) {
      auto endpoints = lf::geometry::Corners(*(edge.Geometry()));
      if (endpoints(0, 0) <= 0.05 || 0.95 <= endpoints(0, 0) ||
          endpoints(1, 0) <= 0.05 || 0.95 <= endpoints(1, 0)) {
        return true;
      }
    }
    return false;
  };
  // Determine the fixed dofs on the boundary and their values
  // Alternative: See lecturedemoDirichlet() in
```

```
33    //
          https://github.com/craffael/lehrfempp/blob/master/examples/lecturedemos/lecturedemo
34    auto edges_flag_values_Dirichlet{lf::fe::InitEssentialConditionFromFunction(
35        *fe_space_p, edges_predicate_Dirichlet, mf_g)};
36    // Eliminate Dirichlet dofs from the linear system
37    lf::assemble::FixFlaggedSolutionCompAlt<double>(
38        [&edges_flag_values_Dirichlet](lf::assemble::glb_idx_t gdof_idx) {
39          return edges_flag_values_Dirichlet[gdof_idx];
40        },
41        A, phi);
42
43    // Assembly completed! Convert COO matrix A into CRS format using Eigen's
44    // internal conversion routines.
45    Eigen::SparseMatrix<double> A_sparse = A.makeSparse();
46
47  // II : SOLVING  THE LINEAR SYSTEM
48    // II.i : Setting up Eigen's sparse direct elimination
49    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
50    solver.compute(A_sparse);
51    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
52    // II.ii : Solving
53    discrete_solution = solver.solve(phi);
54    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
55
56  // do nothing
57    return discrete_solution;
58  };
```

SOLUTION of (3-8.d):

The variational problem (3.8.3) features an s.p.d. bilinear form, which ensures optimality of the finite element Galerkin solution with respect to the energy norm, see [Lecture → Thm. 3.1.3.7]. Hence, we can estimate

$$\|\mathbf{grad}(u-u_\ell)\|^2_{L^2(\Omega)} + \|u-u_\ell\|^2_{L^2(\Gamma_0)} \le \|\mathbf{grad}(u-\mathsf{I}_\ell u)\|^2_{L^2(\Omega)} + \|u-\mathsf{I}_\ell u\|^2_{L^2(\Gamma_0)}\,, \tag{3.8.8}$$

where $\mathsf{I}_\ell : C^0(\overline{\Omega}) \to \mathcal{S}^0_1(\mathcal{M})$ is the nodal interpolation operator according to [Lecture → Def. 3.3.2.1].

Moreover, from the multiplicative trace inequality of

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

$$\exists C = C(\Omega) > 0\colon \quad \|u\|^2_{L^2(\partial\Omega)} \le C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega)\,.$$

we conclude

$$\|u-\mathsf{I}_\ell u\|^2_{L^2(\Gamma_0)} \le C\|u-\mathsf{I}_\ell u\|_{L^2(\Omega)}\,\|u-\mathsf{I}_\ell u\|_{H^1(\Omega)}\,, \tag{3.8.9}$$

with a constant $C > 0$ depending only on Ω_0. Combining this with (3.8.8), we infer

$$\|u-u_\ell\|^2_{\mathsf{a}} = \|\mathbf{grad}(u-\mathsf{I}_\ell u)\|^2_{L^2(\Omega)} + \|u-\mathsf{I}_\ell u\|^2_{L^2(\Gamma_0)} \le C\|u-\mathsf{I}_\ell u\|^2_{H^1(\Omega)}\,. \tag{3.8.10}$$

The additional information $u \in H^2(\Omega)$ paves the way for applying the interpolation estimates of

Theorem [Lecture → Thm. 3.3.2.21]. Error estimate for piecewise linear interpolation

For any $u \in C^2(\bar{\Omega})$ and 2D piecewise linear interpolation $\mathsf{I}_1 : C^0(\overline{\Omega}) \to \mathcal{S}^0_1(\mathcal{M})$, $\mathcal{M}$ a triangular mesh, holds

$$\|u-\mathsf{I}_1 u\|_{L^2(\Omega)} \le \sqrt{\tfrac{3}{8}}\, h^2_{\mathcal{M}} \left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)}\,,$$
$$\|\mathbf{grad}(u-\mathsf{I}_1 u)\|_{L^2(\Omega)} \le \sqrt{\tfrac{3}{24}}\, \rho_{\mathcal{M}} h_{\mathcal{M}} \left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)}\,.$$

where $h_{\mathcal{M}}$ denotes the mesh width (→ [Lecture → Def. 3.2.1.4]) and $\rho_{\mathcal{M}}$ the shape regularity measure (→ [Lecture → Def. 3.3.2.20]) of $\mathcal{M}$.

They immediately give us

$$\|u-\mathsf{I}_\ell u\|_{L^2(\Omega)} \le C h^2_{\mathcal{M}} |u|_{H^2(\Omega)} \quad , \quad |u-\mathsf{I}_\ell u|_{H^1(\Omega)} \le C h_{\mathcal{M}} |u|_{H^2(\Omega)}\,. \tag{3.8.11}$$

The bound on the shape regularity of the meshes $\mathcal{M}_\ell$ will enter the constants. Obviously, the asymptotic behavior of $\|u-u_\ell\|_{\mathsf{a}}$ will be determined by the slower-converging term $|u-\mathsf{I}_\ell u|_{H^1(\Omega)}$:

$$\|u-u_\ell\|_{\mathsf{a}} = O(h_\ell) \quad \text{for} \quad \ell \to \infty \ : \quad \text{algebraic convergence with rate 1}\,.$$

Remark. If Ω_0 is a disc, $\partial\Omega$ has *no re-entrant corners*. Thus, elliptic regularity theory as explained in [Lecture → Section 3.4] confirms that the exact solution of (3.8.1) will belong to $H^2(\Omega)$: $u \in H^2(\Omega)$.

HINT 1 for (3-8.e): You can use (3.8.1b). ⌟

SOLUTION of (3-8.e):

We can just use the impedance boundary condition (3.8.1b) on Γ_0 to substitute $\mathbf{grad}\, u \cdot \boldsymbol{n}_0$ and rewrite

$$\int_{\Gamma_0} (\boldsymbol{d} \cdot \boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}_0(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) = \boxed{F_\partial(\boldsymbol{d}; u) := \int_{\Gamma_0} (\boldsymbol{d} \cdot \boldsymbol{x}) u(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})} \ . \tag{3.8.13}$$

Note the opposite direction of the unit normals!

Remark. By the Cauchy-Schwarz inequality [Lecture → Eq. (1.3.4.15)] and the multiplicative trace inequality [Lecture → Thm. 1.9.0.10] this expression is continuous/bounded on $H^1(\Omega)$: for any $v \in H^1(\Omega)$

$$|F_\partial(\boldsymbol{d}; v)| \le \underbrace{\|\{\boldsymbol{x} \to (\boldsymbol{d} \cdot \boldsymbol{x})\}\|_{L^2(\Gamma_0)}}_{\text{enters constant!}} \|v\|_{L^2(\Gamma_0)} \le C \|v\|_{L^2(\Omega)}^{\frac{1}{2}} \cdot |v|_{H^1(\Omega)}^{\frac{1}{2}} \le C \|v\|_{H^1(\Omega)} \ ,$$

with $C > 0$ depending on Ω and $\boldsymbol{d}$.

SOLUTION of (3-8.f):

Functionals $F_\Omega(\boldsymbol{d};\cdot)$ can be derived in two ways. In any case we need a cut-off function

$$\psi \in H^1(\Omega) \quad , \quad \psi|_{\Gamma_0} \equiv 1 \quad , \quad \psi|_{\Gamma_g} \equiv 0 \,. \tag{3.8.14}$$

(I) We can follow [Lecture → Section 3.6.2] and rely on Green's formula [Lecture → Thm. 1.5.2.7].

$$\begin{aligned} F(\boldsymbol{d};u) &= \int_{\Gamma_0} \psi(\boldsymbol{x})(\boldsymbol{d}\cdot\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}_0(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &= -\int_\Omega \underbrace{\Delta u}_{=0}(\psi(\boldsymbol{x})(\boldsymbol{d}\cdot\boldsymbol{x})) + \mathbf{grad}\,u\cdot\mathbf{grad}\,\underbrace{(\psi(\boldsymbol{x})(\boldsymbol{d}\cdot\boldsymbol{x}))}_{\in H^1(\Omega)}\,\mathrm{d}\boldsymbol{x} \\ &= -\int_\Omega \mathbf{grad}\,u\cdot(\mathbf{grad}\,\psi(\boldsymbol{x})(\boldsymbol{d}\cdot\boldsymbol{x}) + \psi(\boldsymbol{x})\boldsymbol{d})\,\mathrm{d}\boldsymbol{x}\,. \end{aligned}$$

So, one way to write F_Ω is

$$F_\Omega(\boldsymbol{d};v) = \int_\Omega \mathbf{grad}\,v\cdot(\mathbf{grad}\,\psi(\boldsymbol{x})(\boldsymbol{d}\cdot\boldsymbol{x}) + \psi(\boldsymbol{x})\boldsymbol{d})\,\mathrm{d}\boldsymbol{x}\,.$$

Since $\mathbf{grad}\,\psi, \psi \in L^2(\Omega)$, we immediately have from the Cauchy-Schwarz inequality that

$$|F_\Omega(\boldsymbol{d};v)| \le C\|v\|_{H^1(\Omega)} \quad \forall v \in H^1(\Omega)\,,$$

where the constant depends on ψ and $\boldsymbol{d}$.

(II) We can simply use the variational formulation (3.8.3) with $v(\boldsymbol{x}) := \psi(\boldsymbol{x})(\boldsymbol{d}\cdot\boldsymbol{x}) \in V_0$ and get the same result as in Sub-problem (3-8.e), see (3.8.13).

SOLUTION of (3-8.g):

The abstract dual variational problem as defined in [Lecture → Thm. 3.6.1.7] is

$$w \in V_0: \quad \mathsf{a}(v,w) = F_\partial(\boldsymbol{d}; v) \quad \forall v \in V_0 \,. \tag{3.8.15}$$

More concretely, this means

$$w \in V_0: \quad \int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\Gamma_0} wv \,\mathrm{d}S = \int_{\Gamma_0} (\boldsymbol{d}\cdot\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \quad \forall v \in V_0 \,. \tag{3.8.16}$$

We appeal to [Lecture → Ex. 1.8.0.6] and find that (3.8.16) is the weak formulation of the second-order scalar elliptic boundary value problem

$$-\Delta w = 0 \quad \text{in } \Omega \quad , \quad \mathbf{grad}\, w(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) + w(\boldsymbol{x}) = \boldsymbol{d}\cdot\boldsymbol{x} \quad \text{on } \Gamma_0 \quad , \quad w = 0 \quad \text{on } \Gamma_g \,. \tag{3.8.17}$$

SOLUTION of (3-8.h):

Of course, we split the integral into edge contributions. Let e stand for an edge $\subset \Gamma_0$ with endpoints $\boldsymbol{p}^0$ and $\boldsymbol{p}^1$. Let $\nu_0, \nu_1 \in \mathbb{R}$ be the values of $v_h \in \mathcal{S}_1^0(\mathcal{M})$ in the endpoints of e. Note that these values are directly available as suitable entries of the basis expansion coefficient vector $\vec{\boldsymbol{\eta}}$ of v_h. Their indices can be obtained from the **lf::assemble::DofHandler** object.

Then we can compute directly, using the definition of a line integral along a segment:

$$\begin{aligned}\int_e (\boldsymbol{d}\cdot\boldsymbol{x})v_h(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) &= |e|\int_0^1 (\boldsymbol{d}\cdot(\boldsymbol{p}^0(1-\tau)+\boldsymbol{p}^1\tau))(\nu_0(1-\tau)+\nu_1\tau)\,\mathrm{d}\tau \\ &= |e|\tfrac{1}{6}\boldsymbol{d}\cdot\left(2\boldsymbol{p}^0\nu_0+\boldsymbol{p}^0\nu_1+\boldsymbol{p}^1\nu_0+2\boldsymbol{p}^1\nu_1\right), \quad |e| := \left\|\boldsymbol{p}^1-\boldsymbol{p}^0\right\|.\end{aligned}$$

C++ code 3.8.18: Computation of boundary version $F_\partial(\boldsymbol{d};v)$ of the output functional
➥ GITHUB

```
2  double computeBoundaryOutputFunctional(
3      const Eigen::VectorXd eta,
4      const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
5      Eigen::Vector2d d) {
6    double func_val = 0.0;
7    // Pointer to current mesh
8    std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
9    // Obtain local->global index mapping for current finite element space
10   const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
11
12   // Obtain an array of boolean flags for the edges of the mesh, 'true'
13   // indicates that the edge lies on the boundary
14   auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
15
16   // Creating a predicate that will guarantee that the computations are carried
17   // only on the interior boundary edges of the mesh using the boundary flags
18   auto edges_predicate_RobinBC =
19       [&bd_flags](const lf::mesh::Entity &edge) -> bool {
20     if (bd_flags(edge)) {
21       auto endpoints = lf::geometry::Corners(*(edge.Geometry()));
22       return endpoints(0, 0) > 0.05 && 0.95 > endpoints(0, 0) &&
23              endpoints(1, 0) > 0.05 && 0.95 > endpoints(1, 0);
24     }
25     return false;
26   };
27
28   // Computing value of the functional
29   for (const lf::mesh::Entity *edge : mesh_p->Entities(1)) {
30     if (edges_predicate_RobinBC(*edge)) {
31       // Find the endpoints global indices
32       auto dof_idx = dofh.GlobalDofIndices(*edge);
33       assert(dofh.NumLocalDofs(*edge) == 2);
34       // Value of linear function in he endpoints
35       const double nu0 = eta(dof_idx[0]);
36       const double nu1 = eta(dof_idx[1]);
37       // Find coordinates of endpoints
38       auto endpoints = lf::geometry::Corners(*(edge->Geometry()));
```

```
39        // Length of edge
40        const double elen = (endpoints.col(1) - endpoints.col(0)).norm();
41        LF_ASSERT_MSG(endpoints.cols() == 2, "Wrong no endpoints in " << *edge);
42        func_val += elen / 6.0 *
43                    d.dot(2 * endpoints.col(0) * nu0 + endpoints.col(0) * nu1 +
44                          endpoints.col(1) * nu0 + 2 * endpoints.col(1) * nu1);
45      }
46    }
47    return func_val;
48  };
```

HINT 1 for (3-8.i): Note that $\mathbf{g} \to u(\mathbf{g})$ is a *linear* mapping and so is $\boldsymbol{d} \mapsto F(\boldsymbol{d}; u)$. This paves the way for farming out all expensive computations to the constructor, because **operator** `()` realizes a bilinear form on $\mathbb{R}^2$. ⌟

SOLUTION of (3-8.i):

$(\mathbf{g}, \boldsymbol{d}) \mapsto F(\boldsymbol{d}, u(\mathbf{g}))$ is a *bilinear* mapping $\mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R}$ and, as such, can be represented by a 2×2-matrix:

$$F(\boldsymbol{d}, u(\mathbf{g})) = \boldsymbol{d}^\top \begin{bmatrix} F(\boldsymbol{e}_1, u_h(\boldsymbol{e}_1)) & F(\boldsymbol{e}_1, u_h(\boldsymbol{e}_2)) \\ F(\boldsymbol{e}_2, u_h(\boldsymbol{e}_1)) & F(\boldsymbol{e}_2, u_h(\boldsymbol{e}_2)) \end{bmatrix} \mathbf{g}\,,$$

where $\boldsymbol{e}_1, \boldsymbol{e}_2$ are the Cartesian unit vectors in $\mathbb{R}^2$. Hence, the efficient implementation of **EvalResponse** can be done as follows:

- Call `solveImpedanceBVP()` twice in the constructor to compute the finite element Galerkin solutions $u_h(\boldsymbol{e}_1) \in \mathcal{S}_1^0(\mathcal{M})$ and $u_h(\boldsymbol{e}_2) \in \mathcal{S}_1^0(\mathcal{M})$.
- Then invoke `computeBoundaryOutputFunctional()` with the computed finite element Galer solutions to obtain the matrix entries.
- Store only the 2×2-matrix as data member of the class and use it whenever the evaluation operator is called.

Remark. The policy outlined above still leaves room for improvement: for the computation of $u_h(\boldsymbol{e}_1)$ and $u_h(\boldsymbol{e}_2)$ much computational effort could be saved by assembling the Galerkin matrix only once.

C++ code 3.8.19: Definition of class EvalResponse. ➙ GITHUB

```
2  class EvalResponse {
3   public:
4    /* Constructor */
5    explicit EvalResponse(
6        const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>>
7            &fe_space_p);
8    /* Evaluation operator */
9    double operator()(Eigen::Vector2d g, Eigen::Vector2d d) const;
10
11  private:
12   // Matrix representation of bilinear form realized by the evaluation operator.
13   Eigen::Matrix<double, 2, 2> F_Mat_;
14 };  // class EvalResponse
```

C++ code 3.8.20: Implementation class EvalResponse. ➙ GITHUB

```
2  EvalResponse::EvalResponse(
3      const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p) {
4    // Basis vectors for 2D Euclidean space ("unit vectors")
5    Eigen::Vector2d e0{1.0, 0.0}, e1{0.0, 1.0};
6    // Compute approximate solutions
7    Eigen::VectorXd approx_sol_e0, approx_sol_e1;
8    approx_sol_e0 = solveImpedanceBVP(fe_space_p, e0);
9    approx_sol_e1 = solveImpedanceBVP(fe_space_p, e1);
10   // Initialize the
11   F_Mat_(0, 0) = computeBoundaryOutputFunctional(approx_sol_e0, fe_space_p, e0);
12   F_Mat_(0, 1) = computeBoundaryOutputFunctional(approx_sol_e1, fe_space_p, e0);
13   F_Mat_(1, 0) = computeBoundaryOutputFunctional(approx_sol_e0, fe_space_p, e1);
14   F_Mat_(1, 1) = computeBoundaryOutputFunctional(approx_sol_e1, fe_space_p, e1);
15 }
```

```
16
17 /* Operator overloading */
18 double EvalResponse::operator()(Eigen::Vector2d g, Eigen::Vector2d d) const {
19   double value;
20   value = d.dot(F_Mat_ * g);
21   return value;
22 }
```

SOLUTION of (3-9.a):

From the variational equation (3.9.2) we conclude the $L^2(\Omega)$-*orthogonality*

$$\int_\Omega (\mathbf{g}_h - \mathbf{grad}\, u_h) \cdot \mathbf{v}_h \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall \mathbf{v}_h \in \mathbf{V}_h \,. \tag{3.9.3}$$

As a consequence we have Pythagoras' theorem:

$$\begin{aligned} \|\mathbf{w}_h - \mathbf{grad}\, u_h\|^2_{L^2(\Omega)} &= \|\mathbf{w}_h - \mathbf{g}_h + \mathbf{g}_h - \mathbf{grad}\, u_h\|^2_{L^2(\Omega)} \\ &= \|\mathbf{w}_h - \mathbf{g}_h\|^2_{L^2(\Omega)} + \|\mathbf{g}_h - \mathbf{grad}\, u_h\|^2_{L^2(\Omega)} \quad \forall \mathbf{w}_h \in \mathbf{V}_h \,. \end{aligned}$$

This means that we can equivalently define $\mathbf{g}_h$ through

$$\mathbf{g}_h := \operatorname*{argmin}_{\mathbf{v}_h \in \mathbf{V}_h} \|\mathbf{v}_h - \mathbf{grad}\, u_h\|^2_{L^2(\Omega)} \,. \tag{3.9.4}$$

In fact this is the assertion of [Lecture → Thm. 3.1.3.7] when the energy norm is just the $(L^2(\Omega))^2$-norm. Citing this theorem and mentioning that it is applied for $V = \left(L^2(\Omega)\right)^2$ and $V_h := \mathbf{V}_h$ is enough.

SOLUTION of (3-9.j):

In each refinement step the meshwidth h_ℓ decreases by a factor of 2.

For δ_ℓ we observe that its value roughly decreases also by a factor of 2 when moving to a finer mesh. This is clear evidence of algebraic convergence with rate 1: $\delta_\ell = O(h_\ell)$ (in the meshwidth).

For ϵ_ℓ each step of refinement achieves a reduction by a factor of 4, which hints at algebraic convergence w
.

SOLUTION of (3-9.b):

The particular choice of basis functions (3.9.5) for $\mathbf{V}_h$ means that its (ordered) set of local shape functions is

$$\left\{ \begin{bmatrix} \lambda_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_1 \end{bmatrix}, \begin{bmatrix} \lambda_2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_2 \end{bmatrix}, \begin{bmatrix} \lambda_3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_3 \end{bmatrix} \right\} . \tag{3.9.7}$$

Any other ordering is also acceptable, but once LEHRFEM++ implementation is requested the above ordering is the only one compatible with LEHRFEM++'s internal conventions, see [Lecture → § 2.7.4.11].

SOLUTION of (3-9.c):

18 entries of $\mathbf{M}_K^{\text{vec}}$ will be zero. For instance, among the local shape functions of (3.9.7) there are three with vanishing second component and three with vanishing first component. All the entries of $\mathbf{M}_K^{\text{vec}}$ corresponding to pairs of local shape functions from these different sets will vanish. There are $3 \cdot 3 + 3 \cdot 3 = 18$ of them.

HINT 1 for (3-9.d): You may appeal to [Lecture → Lemma 2.7.5.5] for $d = 2$. ⌟

SOLUTION of (3-9.d):

For a triangle K with area $|K|$ the element matrix $\mathbf{M}_K \in \mathbb{R}^{3,3}$ for $\mathcal{S}_1^0(\mathcal{M})$ (with standard choice of barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ as local shape functions) corresponding to the bilinear form

$$(u, v) \mapsto \int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \qquad (L^2(\Omega) \text{ inner product})$$

is given by, *cf.* [Lecture → Lemma 2.7.5.5],

$$\mathbf{M}_K = \frac{|K|}{12} \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} . \tag{3.9.8}$$

The particular choice and arrangement of basis functions for $\mathbf{V}_h$ means that its *ordered* set of local shape functions is

$$\left\{ \begin{bmatrix} \lambda_1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_1 \end{bmatrix}, \begin{bmatrix} \lambda_2 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_2 \end{bmatrix}, \begin{bmatrix} \lambda_3 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ \lambda_3 \end{bmatrix} \right\} .$$

Thus the element matrix $\mathbf{M}_K^{\mathrm{vec}} \in \mathbb{R}^{6,6}$ for $\mathbf{V}_h$ and the bilinear form

$$(\mathbf{u}, \mathbf{v}) \mapsto \int_\Omega \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \qquad ((L^2(\Omega))^2 \text{ inner product}) ,$$

which is the bilinear form underlying the discrete variational problem (3.9.2), arises from interleaving two copies of $\mathbf{M}_K$:

$$\mathbf{M}_K^{\mathrm{vec}} = \begin{bmatrix} (\mathbf{M}_K)_{1,1} & 0 & (\mathbf{M}_K)_{1,2} & 0 & (\mathbf{M}_K)_{1,3} & 0 \\ 0 & (\mathbf{M}_K)_{1,1} & 0 & (\mathbf{M}_K)_{1,2} & 0 & (\mathbf{M}_K)_{1,3} \\ (\mathbf{M}_K)_{2,1} & 0 & (\mathbf{M}_K)_{2,2} & 0 & (\mathbf{M}_K)_{2,3} & 0 \\ 0 & (\mathbf{M}_K)_{2,1} & 0 & (\mathbf{M}_K)_{2,2} & 0 & (\mathbf{M}_K)_{2,3} \\ (\mathbf{M}_K)_{3,1} & 0 & (\mathbf{M}_K)_{3,2} & 0 & (\mathbf{M}_K)_{3,3} & 0 \\ 0 & (\mathbf{M}_K)_{3,1} & 0 & (\mathbf{M}_K)_{3,2} & 0 & (\mathbf{M}_K)_{3,3} \end{bmatrix} \tag{3.9.9}$$

$$\blacktriangleright \quad \mathbf{M}_K^{\mathrm{vec}} = \frac{|K|}{12} \begin{bmatrix} 2 & 0 & 1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 & 1 & 0 \\ 0 & 1 & 0 & 2 & 0 & 1 \\ 1 & 0 & 1 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 & 0 & 2 \end{bmatrix} . \tag{3.9.10}$$

C++ code 3.9.11: Implementation of `Eval()` method of VectorProjectionMatrixProvider
→ GITHUB

```
Eigen::MatrixXd VectorProjectionMatrixProvider::Eval(
    const lf::mesh::Entity &entity) {
  Eigen::MatrixXd elMat_vec;  // element matrix to be returned
  // Throw error in case cell is not Tria nor Quad
  LF_VERIFY_MSG(entity.RefEl() == lf::base::RefEl::kTria() ||
                    entity.RefEl() == lf::base::RefEl::kQuad(),
```

```
                "Unsupported cell type " << entity.RefEl());

  if (entity.RefEl() == lf::base::RefEl::kTria()) {
    elMat_vec = Eigen::MatrixXd::Zero(6, 6);
    // For TRIANGULAR CELLS
    // Compute the area of the triangle cell
    const double area = lf::geometry::Volume(*(entity.Geometry()));
    // Assemble the mass element matrix over the cell
    // clang-format off
      elMat_vec << 2.0, 0.0, 1.0, 0.0, 1.0, 0.0,
                   0.0, 2.0, 0.0, 1.0, 0.0, 1.0,
                1.0, 0.0, 2.0, 0.0, 1.0, 0.0,
                0.0, 1.0, 0.0, 2.0, 0.0, 1.0,
                1.0, 0.0, 1.0, 0.0, 2.0, 0.0,
                0.0, 1.0, 0.0, 1.0, 0.0, 2.0;
    // clang-format on
    elMat_vec *= area / 12.0;
  } else {
    // for QUADRILATERAL CELLS
    elMat_vec = Eigen::MatrixXd::Zero(8, 8);
    Eigen::MatrixXd elMat_scal =
        Eigen::MatrixXd::Zero(4, 4);  // element matrix for scalar FEM
    // Tensor product Gauss-Legendre quadrature rule of order 4
    const lf::quad::QuadRule qr{
        lf::quad::make_QuadRule(lf::base::RefEl::kQuad(), 3)};
    // Reference quadrature points
    const Eigen::MatrixXd zeta_ref{qr.Points()};
    // Quadrature weights
    const Eigen::VectorXd w_ref{qr.Weights()};
    // Number of quadrature points
    const lf::base::size_type P = qr.NumPoints();

    // Reference tensor product basis functions on quadrilateral ref entity
    std::vector<std::function<double(coord_t)>> ref_basis_vec;
    auto b0_ref = [](coord_t x) -> double { return (1 - x(0)) * (1 - x(1)); };
    auto b1_ref = [](coord_t x) -> double { return x(0) * (1 - x(1)); };
    auto b2_ref = [](coord_t x) -> double { return x(0) * x(1); };
    auto b3_ref = [](coord_t x) -> double { return (1 - x(0)) * x(1); };
    ref_basis_vec.push_back(b0_ref);
    ref_basis_vec.push_back(b1_ref);
    ref_basis_vec.push_back(b2_ref);
    ref_basis_vec.push_back(b3_ref);

    const lf::geometry::Geometry &geo{*(entity.Geometry())};
    const Eigen::VectorXd gram_dets{geo.IntegrationElement(zeta_ref)};
    for (int i = 0; i < 4; i++) {
      for (int j = 0; j < 4; j++) {
        for (int l = 0; l < P; l++) {
          elMat_scal(i, j) += w_ref[l] * ref_basis_vec[i](zeta_ref.col(l)) *
                              ref_basis_vec[j](zeta_ref.col(l)) * gram_dets[l];
        }
      }
    }
    // 8x8 element (mass) matrix for vectorial FEM
    // clang-format off
    elMat_vec << elMat_scal(0, 0), 0.0, elMat_scal(0, 1), 0.0, elMat_scal(0, 2),
        0.0, elMat_scal(0, 3), 0.0,
```

```
                0.0, elMat_scal(0, 0), 0.0, elMat_scal(0, 1), 0.0, elMat_scal(0,
                    2), 0.0, elMat_scal(0, 3),
                elMat_scal(1, 0), 0.0, elMat_scal(1, 1), 0.0, elMat_scal(1, 2),
                    0.0, elMat_scal(1, 3), 0.0,
            0.0, elMat_scal(1, 0), 0.0, elMat_scal(1, 1), 0.0, elMat_scal(1, 2),
                0.0, elMat_scal(1, 3),
            elMat_scal(2, 0), 0.0, elMat_scal(2, 1), 0.0, elMat_scal(2, 2), 0.0,
                elMat_scal(2, 3), 0.0,
            0.0, elMat_scal(2, 0), 0.0, elMat_scal(2, 1), 0.0, elMat_scal(2, 2),
                0.0, elMat_scal(2, 3),
                elMat_scal(3, 0), 0.0, elMat_scal(3, 1), 0.0, elMat_scal(3, 2),
                    0.0, elMat_scal(3, 3), 0.0,
                0.0, elMat_scal(3, 0), 0.0, elMat_scal(3, 1), 0.0, elMat_scal(3,
                    2), 0.0, elMat_scal(3, 3);
    // clang-format on

  }
  return elMat_vec;  // return the local mass element matrix
}  //
```

An alternative is to use numerical quadrature based on a quadrature rule that is exact for polynomials of degree ≤ 2.

HINT 1 for (3-9.e): Recall that the determinant of the Jacobian of any bilinear mapping $\mathbb{R}^2 \to \mathbb{R}^2$ is a linear function.

Thus a quadrature rule of sufficiently high order enables the exact evaluation of the entries of element matrices. ⌟

SOLUTION of (3-9.e):

Referring to [Lecture → Section 2.8.2] write $\boldsymbol{\Phi}_K : \widehat{K} \to K$ for the bilinear mapping taking the unit square $\widehat{K}$ to the general quadrilateral K, see [Lecture → Eq. (2.8.2.5)]. Then, by the transformation formula for integrals

$$\int_K \mathbf{u}(\boldsymbol{x}) \cdot \mathbf{v}(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} (\boldsymbol{\Phi}^*\mathbf{u})(\widehat{\boldsymbol{x}}) \cdot (\boldsymbol{\Phi}^*\mathbf{v})(\widehat{\boldsymbol{x}}) |\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})| \,\mathrm{d}\widehat{\boldsymbol{x}} \,,$$

where $\boldsymbol{\Phi}^*$ is the pullback according to [Lecture → Def. 2.8.1.2], e.g„

$$(\boldsymbol{\Phi}^* u)(\widehat{\boldsymbol{x}}) = u(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})) \,, \quad \widehat{\boldsymbol{x}} \in \widehat{K} \,.$$

Assuming that the $\widehat{\boldsymbol{x}} \mapsto \det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})$ does not change sign on $\widehat{K}$, we know from [Lecture → Eq. (2.8.3.18c)] that $\widehat{\boldsymbol{x}} \mapsto |\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})|$ is a *linear function*:

$$|\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})| = \alpha + \beta\widehat{x}_1 + \gamma\widehat{x}_2 \,. \tag{3.9.12}$$

First we compute the element matrix $\mathbf{M}_K$ for the bilinear form

$$(u, v) \mapsto \int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \qquad (L^2(\Omega) \text{ inner product})$$

and the finite element space $\mathcal{S}_1^0(\mathcal{M})$ using the local shape functions $\widehat{b}^1, \ldots, \widehat{b}^4$ on $\widehat{K}$ as given in [Lecture → Eq. (2.6.2.3)]. These are bilinear functions so that the integrand in

$$\left(\mathbf{M}_K\right)_{i,j} = \int_{\widehat{K}} \widehat{b}^j(\widehat{\boldsymbol{x}}) \widehat{b}^i(\widehat{\boldsymbol{x}}) |\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})| \,\mathrm{d}\widehat{\boldsymbol{x}}$$

is a tensor-product polynomial of degree 3, which is integrated exactly by the 4-point tensor-product Gauss-Legendre quadrature rule of order 4, whose nodes and weights are available through the LEHRFEM++ function call

```
const lf::quad::QuadRule qr{
    lf::quad::make_QuadRule(lf::base::RefEl::kQuad(), 3)};
```

Implementation will comprise the following steps:

(I) Obtain the nodes $\widehat{\zeta}_k \in \widehat{K}$ and the weights $\omega_k \in \mathbb{R}$ of the P-point quadrature rule, $P = 4$ for the above choice.

(II) Evaluate the basis functions $\widehat{b}^\ell$, $\ell = 1, \ldots, 4$, at the quadrature nodes $\widehat{\zeta}_k$, $k = 1, \ldots, P$: $\widehat{b}^\ell(\widehat{\zeta}_k)$.

(III) Obtain the values of $|\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\zeta}_k)|$, $k = 1, \ldots, P$. This can be done through the `IntegrationEle` method of the **lf::geometry::Geometry** object describing the shape of the quadrilateral.

(IV) Compute the entries of $\mathbf{M}_K$:

$$\mathbf{M}_K = \left[\sum_{k=1}^{P} \omega_k \widehat{b}^j(\widehat{\zeta}_k) \widehat{b}^i(\widehat{\zeta}_k) |\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\zeta}_k)| \right]_{i,j=1}^{4}$$

$$= \sum_{k=1}^{P} \omega_k |\det \mathrm{D}\boldsymbol{\Phi}(\widehat{\zeta}_k)| \begin{bmatrix} \widehat{b}^1(\widehat{\zeta}_k) \\ \vdots \\ \widehat{b}^4(\widehat{\zeta}_k) \end{bmatrix} \begin{bmatrix} \widehat{b}^1(\widehat{\zeta}_k) & \ldots & \widehat{b}^4(\widehat{\zeta}_k) \end{bmatrix} .$$

In fact, for the sake of efficiency, steps (I) and (II) should be done in the constructor, since they do not depend on the shape of the cell.

Once, $\mathbf{M}_K$ is available, the final 8×8 element matrix can be built by even-off interleaving parallel to (3.9.9).

HINT 1 for (3-9.f): An auxiliary function `gradbarycoordinates()` modeled after [Lecture → Code 2.4.5.11] is supplied in the file `zienkiewiczzhuestimator.cc`. It supplies the gradients of the barycentric coordinate function of a triangular cell as the columns of a matrix. ⌟

SOLUTION of (3-9.f):

Let $K \in \mathcal{M}$ be a triangle and denote by μ_ℓ, $\ell = 1,2,3$, the value of u_h at its ℓ-th vertex. This value can be retrieved directly from the basis expansion coefficient vector `_mu` for u_h via a suitable **lf::assemble::DofHar** object, which is furnished by the `_fe_space_p` class data member. Then we have, with λ_ℓ designating the barycentric coordinate functions,

$$\boldsymbol{d} := \mathbf{grad}\, u_h|_K = \mu_1 \mathbf{grad}\, \lambda_1 + \mu_2 \mathbf{grad}\, \lambda_2 + \mu_3 \mathbf{grad}\, \lambda_3 \in \mathbb{R}^2 .$$

The gradients of the barycentric coordinate functions can be obtained as in [Lecture → Code 2.7.4.37] or [Lecture → Code 2.4.5.11].

Then the entries of the element vector $\vec{\boldsymbol{\varphi}}_K \in \mathbb{R}^6$ are given by

$$\left(\vec{\boldsymbol{\varphi}}_K\right)_\ell = \begin{cases} \int_K (\boldsymbol{d})_1 \lambda_{(\ell+1)/2}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} & \text{for odd } \ell\,, \\ \int_K (\boldsymbol{d})_2 \lambda_{\ell/2}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} & \text{for even } \ell \end{cases} = \frac{1}{3}|K| \cdot \begin{cases} (\boldsymbol{d})_1 & \text{for odd } \ell\,, \\ (\boldsymbol{d})_2 & \text{for even } \ell\,. \end{cases} \tag{3.9.13}$$

Here we used that $\int_K \lambda_k(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{1}{3}$ for any k.

C++ code 3.9.14: Implementation of `Eval()` method of GradientProjectionVectorProvider
→ GITHUB

```
2  Eigen::VectorXd GradientProjectionVectorProvider::Eval(
3      const lf::mesh::Entity &entity) {
4    Eigen::VectorXd elVec(6);  // for returning the element vector
5    // Obtain local->global index mapping for current finite element space
6    const lf::assemble::DofHandler &dofh{_fe_space_p->LocGlobMap()};
7    // Obtain global indices of the vertices of the triangle entity
8    auto dof_idx_vec = dofh.GlobalDofIndices(entity);
9    LF_ASSERT_MSG(dofh.NumLocalDofs(entity) == 3,
10                 "Too many global indices were returned for a triangle entity!");
11
12   // Obtain the gradients of the barycentric coordinate functions
13   Eigen::Matrix<double, 2, 3> elgrad_Mat = gradbarycoordinates(entity);
14   // Compute the local constant gradient of the finite element solution
15   Eigen::Vector2d grad_vec(0.0, 0.0);
16   for (int i = 0; i < 3; i++) {
17     grad_vec = grad_vec + elgrad_Mat.col(i) * _mu(dof_idx_vec[i]);
18   }
19   // Assemble local element vector
20   // Compute the area of the triangle cell
21   const double area = lf::geometry::Volume(*(entity.Geometry()));
22   // clang-format off
23     elVec << grad_vec(0),
24              grad_vec(1),
25              grad_vec(0),
26              grad_vec(1),
27              grad_vec(0),
28              grad_vec(1);
29   // clang-format on
30   elVec *= area / 3.0;
31   return elVec;
32 }  // GradientProjectionVectorProvider::Eval
```

SOLUTION of (3-9.g):

The global shape functions of $\mathbf{V}_h$ enjoy a cardinal basis property with respect to the nodes of the mesh. Thus the element matrices will become diagonal: For a triangle $K \in \mathcal{M}$

$$\widetilde{\mathbf{M}}_K^{\rm vec} = \tfrac{1}{3}|K|\ \mathrm{diag}(1,\ldots,1) \in \mathbb{R}^{6,6}\ . \tag{3.9.16}$$

Also note that the use of the quadrature rule (3.9.15), which is exact for linear functions, does not affect the right-hand-side vector, so that the formula (3.9.13) for the element vectors remains unchanged.

Blending all these insights, and doing "assembly in our minds", we arrive at

$$\widetilde{\mathbf{g}}_h(\mathbf{p}) = \left(\sum_{\boldsymbol{p}\in\overline{K}} |K|\right)^{-1} \sum_{\boldsymbol{p}\in\overline{K}} |K|\,\boldsymbol{d}_K\ . \tag{3.9.17}$$

The factor $\frac{1}{3}$ occurs on both sides and can be canceled. Hence, the nodal value of $\widetilde{\mathbf{g}}_h$ is an area-weighted sum of $\mathbf{grad}\, u_h$ on the cells adjacent to a node.

SOLUTION of (3-9.h):

As one of many options, this task can be solved by the following steps:

(I) Initialize the $\mathbf{V}_h$-basis-expansion coefficient vector $\vec{\gamma} \in \mathbb{R}^{2N}$ with zero.

(II) Loop over the cells and for every cell $K \in \mathcal{M}$ add the value $|K| g_K$ to the right segment of the vector $\vec{\gamma}$. The location of that segment can be retrieved through `vec_dofh`. g_K is computed as in Sub-problem (3-9.f) based on `scal_dofh` and the vector `mu`.

(III) "Assemble" the quantities $\sum_{p \in \overline{K}} |K|$ for every node of mesh into an **lf::mesh::utils::CodimMeshDataS**
object. This can be done inside the same loop over the cells as (II).

(IV) Finally visit all nodes and scale the entries of $\vec{\gamma}$ as required by (3.9.17).

C++ code 3.9.18: Implementation of `Eval()` method of VectorProjectionMatrixProvider
➥ GITHUB

```
Eigen::VectorXd computeLumpedProjection(
    const lf::assemble::DofHandler &scal_dofh, const Eigen::VectorXd &mu,
    const lf::assemble::DofHandler &vec_dofh) {
  // Obtain shared_ptr to mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = scal_dofh.Mesh();
  // Dimension of vector-valued finite element space
  const lf::uscalfe::size_type N_vec_dofs(vec_dofh.NumDofs());
  // Initialize vector FE basis expansion coefficient vector with zeros
  Eigen::VectorXd proj_vec(N_vec_dofs);
  proj_vec.setZero();
  // Initialize temporary helper nodal DataSet (codim 2)
  auto nodal_sum_of_areas =
      lf::mesh::utils::CodimMeshDataSet<double>(mesh_p, 2, 0.0);

  // Loop over the triangular cells of the mesh in the spirit of
  // cell oriented assembly
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    LF_VERIFY_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Unsupported cell type " << cell->RefEl());
    // Obtain global scalar-FE indices of the vertices
    const auto scal_dof_idx_vec = scal_dofh.GlobalDofIndices(*cell);
    // Obtain the gradients of the barycentric coordinate functions
    const Eigen::Matrix<double, 2, 3> elgrad_Mat = gradbarycoordinates(*cell);
    // Obtain area of the triangular cell
    const double area = lf::geometry::Volume(*(cell->Geometry()));
// Compute the gradient of the passed coefficient vector
    const Eigen::Vector2d grad_mu =
        elgrad_Mat.col(0) * mu(scal_dof_idx_vec[0]) +
        elgrad_Mat.col(1) * mu(scal_dof_idx_vec[1]) +
        elgrad_Mat.col(2) * mu(scal_dof_idx_vec[2]);
    // Local contribution to the area of the cell patch surrounding a node
    for (const lf::mesh::Entity *node : cell->SubEntities(2)) {
      LF_VERIFY_MSG(node->RefEl() == lf::base::RefEl::kPoint(),
                    "Expected kPoint type!" << node->RefEl());
      auto vec_dofh_idx = vec_dofh.GlobalDofIndices(*node);
      proj_vec[vec_dofh_idx[0]] += area * grad_mu[0];
      proj_vec[vec_dofh_idx[1]] += area * grad_mu[1];
```

```
39        nodal_sum_of_areas(*node) = nodal_sum_of_areas(*node) + area;
40      }
41    }
42
43    // Scaling of components of vector of dofs
44    for (const lf::mesh::Entity *node : mesh_p->Entities(2)) {
45      LF_VERIFY_MSG(node->RefEl() == lf::base::RefEl::kPoint(),
46                    "Expected kPoint type!" << node->RefEl());
47      const double area_scal_fac = 1.0 / nodal_sum_of_areas(*node);
48      auto vec_dofh_idx = vec_dofh.GlobalDofIndices(*node);
49      proj_vec[vec_dofh_idx[0]] *= area_scal_fac;
50      proj_vec[vec_dofh_idx[1]] *= area_scal_fac;
51    }
52    return proj_vec;
53  };  // computeLumpedProjection
```

HINT 1 for (3-9.i): The deviation norm should be computed in a cell-oriented way by adding up cell contributions. ⌟

HINT 2 for (3-9.i): A possible way to compute the square of the deviation norm on a triangle K exactly is to use the edge-midpoint quadrature rule

$$\int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{1}{3}|K|\left(\psi(\tfrac{1}{2}(\boldsymbol{a}_K^1+\boldsymbol{a}_K^2)) + \psi(\tfrac{1}{2}(\boldsymbol{a}_K^2+\boldsymbol{a}_K^3)) + \psi(\tfrac{1}{2}(\boldsymbol{a}_K^3+\boldsymbol{a}_K^1))\right),$$

where $\boldsymbol{a}_K^1, \boldsymbol{a}_K^2, \boldsymbol{a}_K^3$ are the vertices of K. Moreover, we can use that $\mathbf{r}_h$ is linear on K:

$$\mathbf{r}_h(\boldsymbol{x}) = \mathbf{r}(\boldsymbol{a}_K^1)\lambda_1(\boldsymbol{x}) + \mathbf{r}(\boldsymbol{a}_K^2)\lambda_2(\boldsymbol{x}) + \mathbf{r}(\boldsymbol{a}_K^3)\lambda_3(\boldsymbol{x})\ .$$

SOLUTION of (3-9.i):

Of course, the norm is computed by looping over all cells and summing their contributions $\|\mathbf{r}_h - \mathbf{grad}\, v_h\|_{L^2}^2$
Note that on every triangle $K \in \mathcal{M}$ the function $\boldsymbol{x} \mapsto \|\mathbf{r}_h(\boldsymbol{x}) - \mathbf{grad}\, v_h(\boldsymbol{x})\|^2$ is a quadratic polynomial, which can be integrated exactly by the edge-midpoint quadrature rule

$$\int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{1}{3}|K|\Big(\psi(\tfrac{1}{2}(\boldsymbol{a}_K^1+\boldsymbol{a}_K^2)) + \psi(\tfrac{1}{2}(\boldsymbol{a}_K^2+\boldsymbol{a}_K^3)) + \psi(\tfrac{1}{2}(\boldsymbol{a}_K^3+\boldsymbol{a}_K^1))\Big)\,, \tag{3.9.19}$$

as recommended in the hint. The values $\mathbf{r}_h(\boldsymbol{a}_K^\ell)$ are snippets of `gamma` whose location can be learned from `vec_dofh`.

In addition, we know that $\mathbf{grad}\, v_h$ is constant on K; write $\boldsymbol{d}_K \in \mathbb{R}^2$ for its value. This gives us the final formula for the local contribution

$$\begin{aligned}\|\mathbf{r}_h - \mathbf{grad}\, v_h\|_{L^2(K)}^2 = \frac{|K|}{3}\Big(&\Big\|\tfrac{1}{2}\big(\mathbf{r}_h(\boldsymbol{a}_K^1)+\mathbf{r}_h(\boldsymbol{a}_K^2)\big) - \boldsymbol{d}_K\Big\|^2 + \\ &\Big\|\tfrac{1}{2}\big(\mathbf{r}_h(\boldsymbol{a}_K^2)+\mathbf{r}_h(\boldsymbol{a}_K^3)\big) - \boldsymbol{d}_K\Big\|^2 + \\ &\Big\|\tfrac{1}{2}\big(\mathbf{r}_h(\boldsymbol{a}_K^3)+\mathbf{r}_h(\boldsymbol{a}_K^1)\big) - \boldsymbol{d}_K\Big\|^2\Big)\end{aligned}$$

C++ code 3.9.20: Implementation of `computeL2Deviation()` ↪ GITHUB

```
double computeL2Deviation(const lf::assemble::DofHandler &scal_dofh,
                          const Eigen::VectorXd &eta,
                          const lf::assemble::DofHandler &vec_dofh,
                          const Eigen::VectorXd &gamma) {
  double deviation_norm_value = 0.0;  // For retrurning the result
  // Obtain shared_ptr to mesh
  auto mesh_p = scal_dofh.Mesh();
  // Cell-oriented computation of deviation norm (squared)
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    LF_VERIFY_MSG(cell->RefEl() == lf::base::RefEl::kTria(),
                  "Unsupported cell type " << cell->RefEl());
    // Obtain area of the triangular cell
    const double area = lf::geometry::Volume(*(cell->Geometry()));

    // Obtain global scalar-FE indices of the vertices
    auto scal_dof_idx_vec = scal_dofh.GlobalDofIndices(*cell);
    // Obtain the gradients of the barycentric coordinates functions
    Eigen::Matrix<double, 2, 3> elgrad_Mat = gradbarycoordinates(*cell);
// Compute the gradient of the passed coefficient vector eta
    const Eigen::Vector2d grad_eta =
        elgrad_Mat.col(0) * eta(scal_dof_idx_vec[0]) +
        elgrad_Mat.col(1) * eta(scal_dof_idx_vec[1]) +
        elgrad_Mat.col(2) * eta(scal_dof_idx_vec[2]);

    // Obtaining the values of the passed vector at each node
    std::vector<Eigen::Vector2d> r_vec_values;
    for (const lf::mesh::Entity *node : cell->SubEntities(2)) {
      LF_VERIFY_MSG(node->RefEl() == lf::base::RefEl::kPoint(),
                    "Expected kPoint type!" << node->RefEl());
      auto vec_dofh_idx = vec_dofh.GlobalDofIndices(*node);
      Eigen::Vector2d r_vec_at_node;
```

```
      r_vec_at_node[0] = gamma[vec_dofh_idx[0]];
      r_vec_at_node[1] = gamma[vec_dofh_idx[1]];
      r_vec_values.push_back(r_vec_at_node);
    }

    // Computing local contribution of the cell to the deviation norm
    double local_norm_value =
        (0.5 * (r_vec_values.at(0) + r_vec_values.at(1)) - grad_eta)
            .squaredNorm() +
        (0.5 * (r_vec_values.at(1) + r_vec_values.at(2)) - grad_eta)
            .squaredNorm() +
        (0.5 * (r_vec_values.at(2) + r_vec_values.at(0)) - grad_eta)
            .squaredNorm();
    local_norm_value *= area / 3.0;

    // Adding local contribution to the value of the deviation norm
    deviation_norm_value += local_norm_value;
  }
  return std::sqrt(deviation_norm_value);
};  // computeL2Deviation
```

There is an alternative way to realize the function `computeL2Deviation()` relying on LEHRFEM++'s **MeshFunction** facilities. However, this solution does not fit the code template. However, modification of code templates is acceptable.

SOLUTION of (4-1.a):

With M points in each direction, there are M^2 free nodes in total, so the matrix should be of size $M^2 \times M^2$, and the vector of size M^2.

SOLUTION of (4-1.j):

C++ code 4.1.11: Compute the error on a grid of a given size. ⇢ **GitLab**

```
double compgriderr(unsigned int M) {
  double err = 0;
  // Obtain the Mehrstellen solution to the PDE
  const auto f = [](double x, double y) {
    return std::sin(M_PI * x) * std::sin(M_PI * y);
  };
  const Eigen::VectorXd mu = solveMehrstellen(f, M);
  // The exact solution u
  const auto u = [](double x, double y) {
    return std::sin(M_PI * x) * std::sin(M_PI * y) / (2 * M_PI * M_PI);
  };
  // Find the maximum difference between the discretized and exact
     solution at
  // the mesh nodes
  const double h = 1. / (M + 1);
  for (int i = 0; i < M; ++i) {
    for (int j = 0; j < M; ++j) {
      const double x = (i + 1) * h;
      const double y = (j + 1) * h;
      // The index of the node at (i,j)
      const int k = i * M + j;
      // Update the error norm
      const double diff = std::fabs(u(x, y) - mu[k]);
      if (diff > err) {
        err = diff;
      }
    }
  }
  return err;
}
```

SOLUTION of (4-1.k):

C++ code 4.1.12: Print a table of the error norms. ⇢ GitLab

```
void tabulateMehrstellenError() {
  std::vector<double> errs;
  std::vector<unsigned int> Ms = {5, 10, 20, 40, 80, 160};
  // Fill the errs vector with the errors at the different mesh
     resolutions
  for (unsigned int M : Ms) {
    errs.push_back(compgriderr(M));
  }
  // Print the collected data out in a table
  std::cout << "| M   | h       | error        |\n";
  std::cout << "-------------------------------|\n";
  for (int i = 0; i < Ms.size(); ++i) {
    const unsigned int M = Ms[i];
    const double h = 1. / (M + 1);
    const double err = errs[i];
    std::cout << "| " << std::setw(3) << M << " | " << std::fixed
              << std::setprecision(5) << h << " | " << std::scientific << err
              << " |\n";
  }
}
```

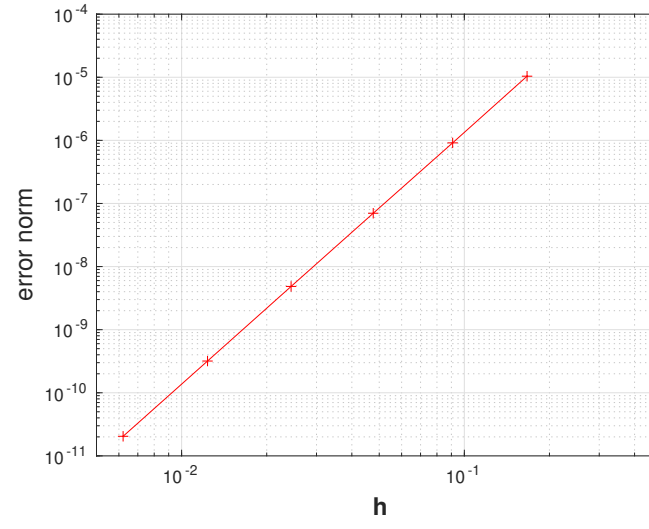
The code should produce the three left columns of the error table printed beside.

The right columns gives the ratio of two consecutive error norms

M	h	error norm	red.
5	0.16667	1.039435e-05	—
10	0.09091	9.127674e-07	11.4
20	0.04762	6.999649e-08	13.0
40	0.02439	4.842196e-09	14.5
80	0.01235	3.183002e-10	15.2
160	0.00621	2.042897e-11	15.6

As the meshwidth h is reduced by a factor of 2, the error norm drops by a factor $\approx 14 - 16$ asymptotically. We may conjecture an asymptotic behavior of the error norm like $O(h^4)$ as $h \to 0$. Note that this impressive rate is with respect to h and not the number of degrees of freedom (which is M^2, halving the rate).

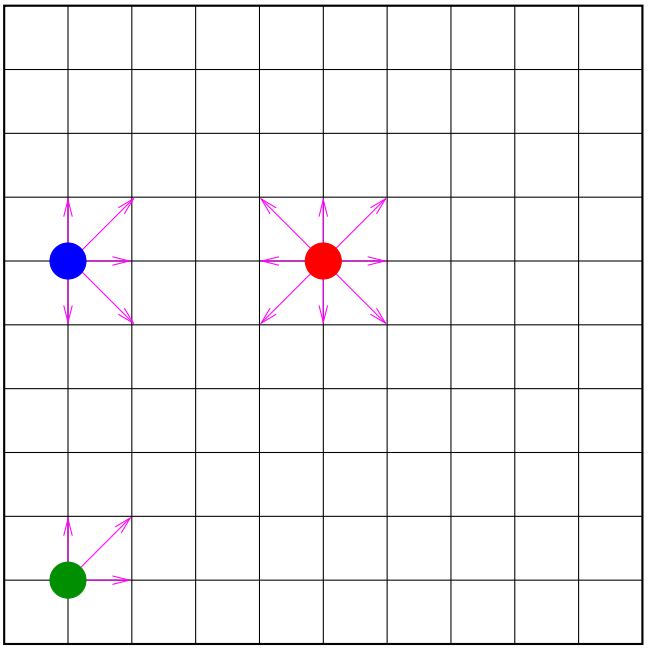
The error points nearly lie on a straight line in a doubly logarithmic plot of meshwidth h versus the error norm. According to [Lecture → § 3.2.2.5] this indicates **algebraic convergence**. The slope of the line tell us the rate of convergence. The convergence plot is shown beside.



SOLUTION of (4-1.b):

With the standard nodal basis, the entries of the solution vector correspond to the value of the finite element solution at the corresponding vertices of the mesh. Thus, the entries of the solution vector supply approximations for the values $u(\boldsymbol{x})$ for every grid points/mesh node $\boldsymbol{x}$.

SOLUTION of (4-1.c):



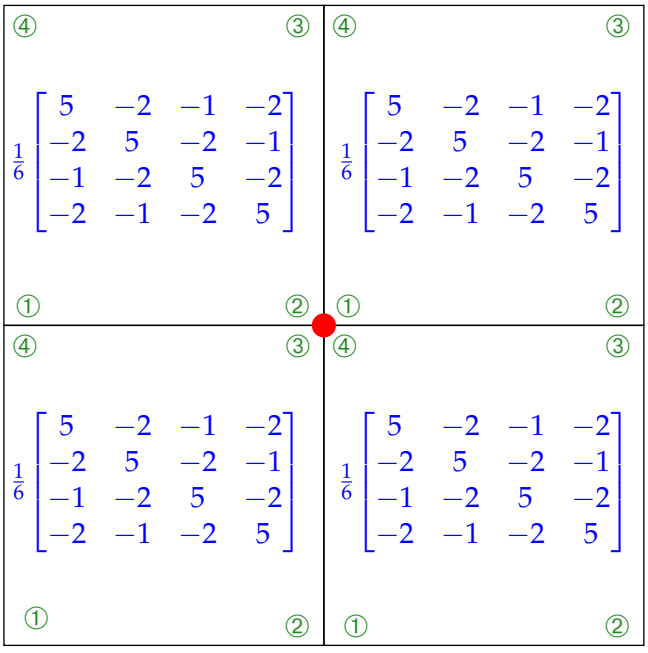
Since the element matrices (4.1.2) are fully populated, every node is “connected” (in the sense of a non-zero entry of the matrix $\mathbf{A}$) to all other nodes with which it shares a cell.

Hence, an interior node is connected to 9 other nodes, whereas a node next to a corner is connected to only three other nodes. ▷

Thus the number of non-zero entries of rows of $\mathbf{A}$ is between 3 and 9.

HINT 1 for (4-1.d):

Follow the “assembly on paper” approach of [Lecture → § 4.1.3.3].



SOLUTION of (4-1.d):

I. k-th row of the "Mehrstellenmatrix $\mathbf{A}$:

We may assume that the grid point with index k is completely interior, that is, not next to a grid point on $\partial\Omega$. This means, that that the k-th row of the matrix has 9 non-zero entries.

Notice that the element matrices $\mathbf{A}_K$ from (4.1.2) are symmetric and have constant diagonals, which greatly simplifies the computation of the entries of $(\mathbf{A})_{k,:}$. In the following formulas we denote by $\mathbf{A}_K^*$, $* = se, ne, nw, sw$ ("south-east, north-east, north-west, south-west), the element matrices of the four squares adjacent to grid point k •, see Fig. 91. We also use the local numbering of vertices given in that figure. With these notations and conventions combined with the lexikographic numbering of the grid points as given in Fig. 89 the following sums give the entries of $(\mathbf{A})_{k,:}$:

$$(\mathbf{A})_{k,j} = \begin{cases} (\mathbf{A}_K^{se})_{4,4} + (\mathbf{A}_K^{ne})_{1,1} + (\mathbf{A}_K^{nw})_{2,2} + (\mathbf{A}_K^{sw})_{3,3} & = \frac{20}{6} \quad \text{for } j = k\,, \\ (\mathbf{A}_K^{se})_{4,3} + (\mathbf{A}_K^{ne})_{1,2} & = -\frac{4}{6} \quad \text{for } j = k+1\,, \\ (\mathbf{A}_K^{sw})_{4,3} + (\mathbf{A}_K^{nw})_{1,2} & = -\frac{4}{6} \quad \text{for } j = k-1\,, \\ (\mathbf{A}_K^{sw})_{2,3} + (\mathbf{A}_K^{se})_{1,4} & = -\frac{4}{6} \quad \text{for } j = k-M\,, \\ (\mathbf{A}_K^{nw})_{2,3} + (\mathbf{A}_K^{ne})_{1,4} & = -\frac{4}{6} \quad \text{for } j = k+M\,, \\ (\mathbf{A}_K^{sw})_{1,3} & = -\frac{1}{6} \quad \text{for } j = k-M-1\,, \\ (\mathbf{A}_K^{se})_{2,4} & = -\frac{1}{6} \quad \text{for } j = k-M+1\,, \\ (\mathbf{A}_K^{ne})_{1,3} & = -\frac{1}{6} \quad \text{for } j = k+M+1\,, \\ (\mathbf{A}_K^{nw})_{2,4} & = -\frac{1}{6} \quad \text{for } j = k+M-1\,, \\ & = 0 \quad \text{for all other } j\,. \end{cases} \tag{4.1.4}$$

Since the row and column sums of $\mathbf{A}_K$ vanish, this must also hold for the k-th row sum of $\mathbf{A}$.

II. Entry k of the right-hand-side vector $\vec{\boldsymbol{\varphi}}$.

With the notational conventions introduced above and writing $\boldsymbol{p} \in \mathbb{R}^2$ for the position vector of the k-th

node, we deduce from (4.1.3) and the local numbering given in Fig. 88:

$$\vec{\boldsymbol{\varphi}}_K^{se} = \frac{h^2}{24}\begin{bmatrix} 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\-h\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right) \end{bmatrix},$$

$$\vec{\boldsymbol{\varphi}}_K^{ne} = \frac{h^2}{24}\begin{bmatrix} 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\h\end{smallmatrix}\right]\right) \end{bmatrix},$$

$$\vec{\boldsymbol{\varphi}}_K^{nw} = \frac{h^2}{24}\begin{bmatrix} 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\h\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right) \end{bmatrix},$$

$$\vec{\boldsymbol{\varphi}}_K^{sw} = \frac{h^2}{24}\begin{bmatrix} 4f\left(\boldsymbol{p}+\begin{bmatrix}-h\\-h\end{bmatrix}\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\begin{bmatrix}-h\\-h\end{bmatrix}\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right) \\ 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\begin{bmatrix}-h\\-h\end{bmatrix}\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right) \end{bmatrix}.$$

The entry of the right-hand-side vector can be computed by the following summation

$$\begin{aligned}(\vec{\boldsymbol{\varphi}})_k &= (\vec{\boldsymbol{\varphi}}_K^{se})_4 + (\vec{\boldsymbol{\varphi}}_K^{ne})_1 + (\vec{\boldsymbol{\varphi}}_K^{nw})_2 + (\vec{\boldsymbol{\varphi}}_K^{sw})_3 \\ &= \frac{h^2}{24}\Big(4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+ \\ &\qquad 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+ \\ &\qquad 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)+ \\ &\qquad 4f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)\Big) \\ &= \tfrac{h^2}{12}\Big(8f(\boldsymbol{p})+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\-h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}0\\h\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}h\\0\end{smallmatrix}\right]\right)+f\left(\boldsymbol{p}+\left[\begin{smallmatrix}-h\\0\end{smallmatrix}\right]\right)\Big)\,.\end{aligned} \tag{4.1.5}$$

SOLUTION of (4-1.e):

The stencil can be read off the matrix entries computed in (4.1.4). The “connection weights” of adjacent nodes on the same grid line are $-\frac{4}{6}$, whereas those of node at opposite corners of a square are $-\frac{1}{6}$. This can be visualized by the stencil

$$\frac{1}{6}\cdot\begin{bmatrix}-1 & -4 & -1\\ -4 & 20 & -4\\ -1 & -4 & -1\end{bmatrix}_h \quad . \tag{4.1.6}$$

HINT 1 for (4-1.f): The matrix $\mathbf{A}$ is a block tridiagonal matrix with tridiagonal blocks. ⌟

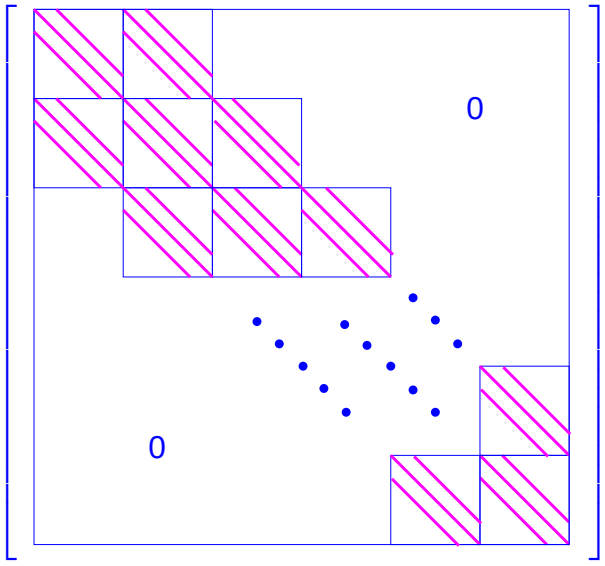
SOLUTION of (4-1.f):

All rows of $\mathbf{A}$ are given by (4.1.4) and for some we have to drop entries corresponding to grid points on the boundary. This yields the $M \times M$ block-tridiagonal matrix

$$\mathbf{A} := \begin{bmatrix} \mathbf{T} & \mathbf{S} & 0 & \cdots & \cdots & 0 \\ \mathbf{S} & \mathbf{T} & \mathbf{S} & & & \vdots \\ 0 & \mathbf{S} & \mathbf{T} & \mathbf{S} & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & \mathbf{S} & \mathbf{T} & \mathbf{S} \\ 0 & \cdots & \cdots & 0 & \mathbf{S} & \mathbf{T} \end{bmatrix},$$

$$\mathbf{T} := \frac{1}{6}\begin{bmatrix} 20 & -4 & 0 & & & 0 \\ -4 & 20 & -4 & \ddots & & \vdots \\ 0 & -4 & 20 & -4 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & -4 & 20 & -4 \\ 0 & \cdots & \cdots & 0 & -4 & 20 \end{bmatrix} \in \mathbb{R}^{M,M}, \quad \mathbf{S} := \frac{1}{6}\begin{bmatrix} -4 & -1 & 0 & & & 0 \\ -1 & -4 & -1 & \ddots & & \vdots \\ 0 & -1 & -4 & -1 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & & 0 & -1 & -4 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & -4 \end{bmatrix} \in \mathbb{R}^{M,M}.$$

$\mathbf{A} =$



◁ Structure of "Mehrstellenmatrix" $\mathbf{A}$.

The structure also reveals that

- rows of the matrix related to truly interior nodes, the red node • in Fig. 90, feature 9 non-zero entries
- rows of the matrix belonging to corner nodes, the green node • in Fig. 90, have 3 non-zero entries
- rows of the matrix for nodes at the boundary, the blue node • in Fig. 90, have 6 non-zero entries.

HINT 1 for (4-1.g): Since we already know the maximal number of non-zero entries of $\mathbf{A}$ per row, we can efficiently set the entries of an **Eigen::SparseMatrix** object, if we `reserve` sufficient space in advance, see for details or EIGEN's documentation on the use of sparse matrix data formats. ⌟

SOLUTION of (4-1.g):

The following implementation relies on direct initialization as in . In the outer loop the grid points are referenced by an index pair (i,j), $0 \leq i,j < M$, which is converted to the lexikographic index k through the formula $k = iM + j$.

The code directly implements (4.1.4) plus dropping of entries outside the grid. This is achieved by suitable index checks.

C++ code 4.1.7: Building of Mehrstellen system matrix. ⇝ GitLab

```
2  Eigen::SparseMatrix<double> compMehrstellenA(unsigned int M) {
3    // For the sake of efficiency the use of Eigen's sparse matrix data
       type is
4    // essential.  The matrix is stored in CCS format.
5    Eigen::SparseMatrix<double> A(M * M, M * M);
6    // We already know that the matrix has at most 9M non-zero entries
       per row
7    // and column.  This information is passed to Eigen via the reserve()
       member
8    // funtion.
9    A.reserve(Eigen::VectorXi::Constant(M * M, 9));
10   // Iterate over all interior nodes of the mesh and apply the stencil
       and
11   // initialize the matrix in column-wise order, from top to bottom in
       every
12   // column, which is most efficient for the CCS storage format.
13   for (int i = 0; i < M; ++i) {
14     for (int j = 0; j < M; ++j) {
15       // Index of the current node
16       const int k = i * M + j;
17       // Interaction term with the node below to the left
18       if (i > 0 && j > 0) {
19         A.insert(k - M - 1, k) = -1;
20       }
21       // Interaction term with the node below
22       if (i > 0) {
23         A.insert(k - M, k) = -4;
24       }
25       // Interaction term with the node below to the right
26       if (i > 0 && j < M - 1) {
27         A.insert(k - M + 1, k) = -1;
28       }
29       // Interaction term with the node to the left
30       if (j > 0) {
31         A.insert(k - 1, k) = -4;
32       }
33       // Interaction term with itself
34       A.insert(k, k) = 20;
35       // Interaction term with the node to the right
36       if (j < M - 1) {
37         A.insert(k + 1, k) = -4;
38       }
39       // Interaction term with the node above to the left
40       if (i < M - 1 && j > 0) {
41         A.insert(k + M - 1, k) = -1;
42       }
43       // Interaction term with the node above
```

```
44        if (i < M - 1) {
45          A.insert(k + M, k) = -4;
46        }
47        // Interaction term with the node above to th right
48        if (i < M - 1 && j < M - 1) {
49          A.insert(k + M + 1, k) = -1;
50        }
51      }
52    }
53    A.makeCompressed();
54    A /= 6;
55    return A;
56  }
```

Of course, an alternative is the initialization through a list of triplets, as discussed in and also in [Lecture → Rem. 2.4.5.25].

SOLUTION of (4-1.h):

The implementation is straightforward relying on (4.1.5). As in Code 4.1.7 in the loop the grid points are referenced by an index pair (i,j), $0 \leq i,j < M$, which is converted to the lexikographic index k through the formula $k = iM + j$.

C++ code 4.1.8: Building of the Mehrstellen right hand side vector. ⇝ GitLab

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd compMehrstellenf(FUNCTOR &&f, unsigned int M) {
4    Eigen::VectorXd phi = Eigen::VectorXd::Zero(M * M);
5    const double h = 1. / (M + 1);
6    // Iterate over all interior nodes of the mesh
7    for (int i = 0; i < M; ++i) {
8      for (int j = 0; j < M; ++j) {
9        // Translate i and j to x- and y-coordinates
10       const double x = (i + 1) * h;
11       const double y = (j + 1) * h;
12       // Compute the index of the node at (i,j)
13       const int k = i * M + j;
14       // Compute the k-th element of the load vector
15       phi[k] += f(x, y - h);
16       phi[k] += f(x - h, y);
17       phi[k] += 8 * f(x, y);
18       phi[k] += f(x + h, y);
19       phi[k] += f(x, y + h);
20     }
21   }
22   phi *= h * h / 12;
23   return phi;
24 }
```

HINT 1 for (4-1.i): Consult on how to solve a sparse linear system of equations in EIGEN. ⌟

SOLUTION of (4-1.i):

The following code gives a skeleton implementation. It should be augmented by checks of successful decomposition and backward elimination.

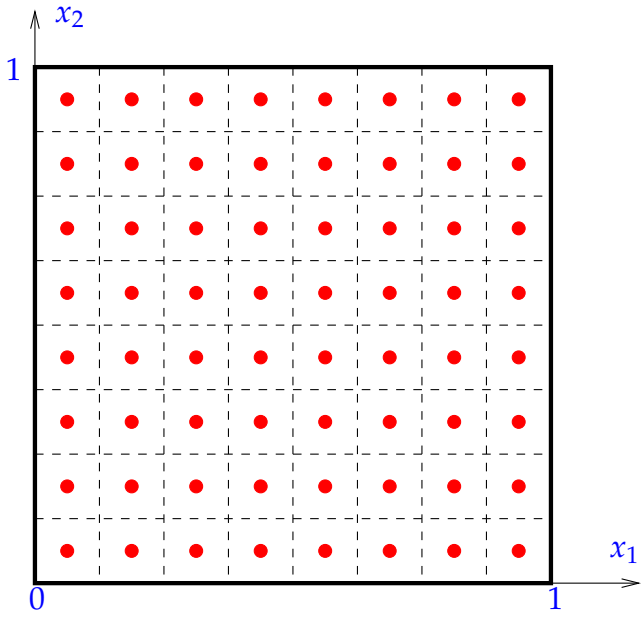
C++ code 4.1.9: Solve the PDE using the Mehrstellenverfahren. ⇝ GitLab

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd solveMehrstellen(FUNCTOR &&f, unsigned int M) {
4    Eigen::VectorXd mu = Eigen::VectorXd::Zero(M * M);
5    // Compute the stiffness matrix
6    Eigen::SparseMatrix<double> A = compMehrstellenA(M);
7    // Compute the load vector
8    Eigen::VectorXd phi = compMehrstellenf(f, M);
9    // Solve the LSE using sparse cholesky
10   Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> solver(A);
11   mu = solver.solve(phi);
12   return mu;
13 }
```

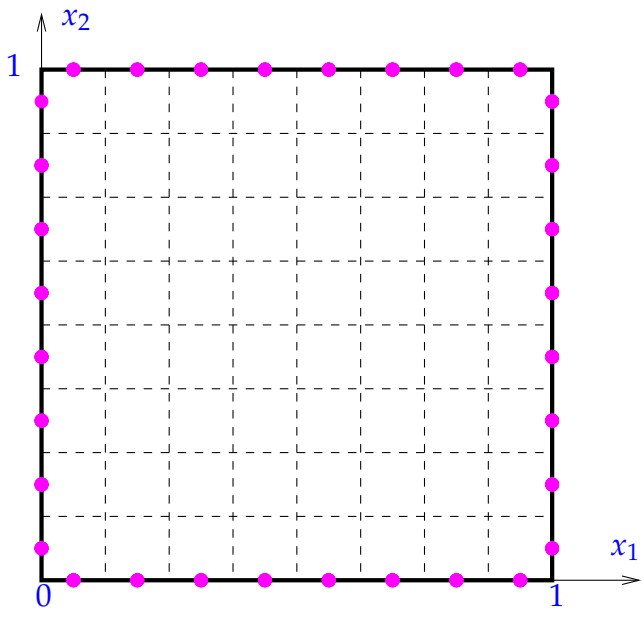
Any other direct elimination solver of EIGEN could also be used, in particular **SparseLU**, see EIGEN docume
The algorithm of **SimplicialLDLT** exploits the fact that $\mathbf{A}$ is *s.p.d.*.

SOLUTION of (4-2.a):

Note that in the following figure instead of ✖ the symbol • is used.



Points of Ω_h



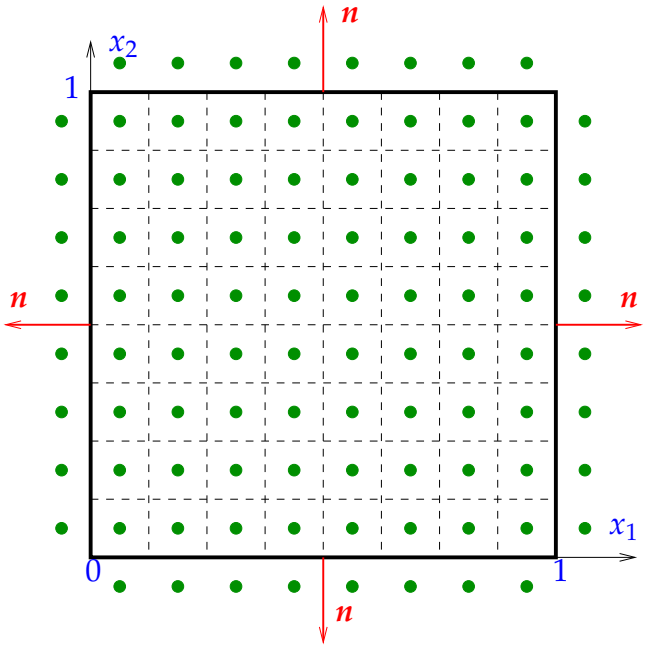
Points of Γ_h

SOLUTION of (4-2.b):

To every point in Ω_h and every point in Γ_h there corresponds an equation. We have

$$\sharp\Omega_h = n^2 \quad , \quad \sharp\Gamma_h = 4n \ .$$

This yields that the total number of equations is $n(n+4)$.



◁ Gridpoints corresponding to unknowns in (4.2.3)

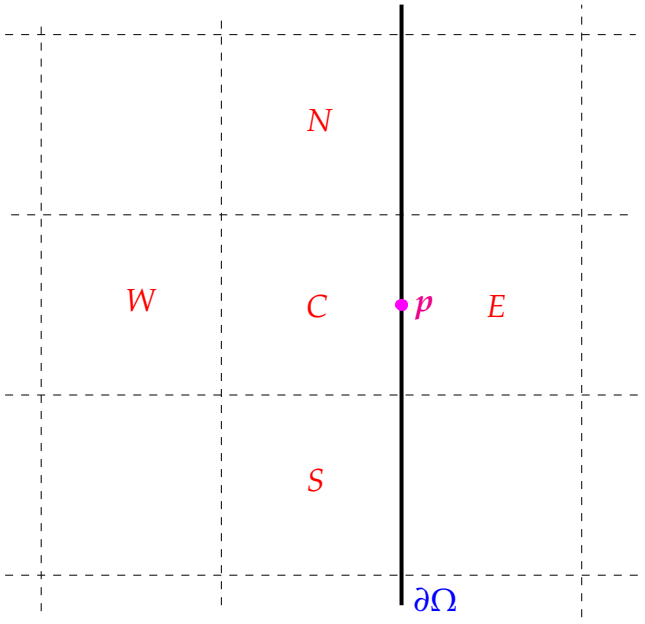
The total number of unknowns is n^2+4n .

Remark. This means that the number of equations and that of the unknowns in (4.2.3) are the same. This is a necessary condition for the existence and uniqueness of solutions of a *linear system of equations* (LSE). In other words, it must be an LSE with a square system matrix. (4.2.3).

HINT 1 for (4-2.c): Also the “corner points” of Ω_h require a special treatment! ⌟

SOLUTION of (4-2.c):

❶: For "interior" grid points $\boldsymbol{x} \in \widehat{\Omega}_h$ we can simply copy (4.2.3a). You may multiply the equations with h^2.



❷: grid point $\boldsymbol{x} \in \Omega_h$ at a side of Ω:

Let us consider $\boldsymbol{x} \in \Omega_h^E$ next to the "eastern" boundary. We label that point C ("center") and its direct neighbors in $\mathbb{L}_h$ E ("eastern neighbor"), N ("northern neighbor"), W ("western neighbor"), S ("southern neighbor"):

$$E \leftrightarrow \boldsymbol{x} + \begin{bmatrix} h \\ 0 \end{bmatrix} \quad , \quad N \leftrightarrow \boldsymbol{x} + \begin{bmatrix} 0 \\ h \end{bmatrix} ,$$
$$W \leftrightarrow \boldsymbol{x} - \begin{bmatrix} h \\ 0 \end{bmatrix} \quad , \quad S \leftrightarrow \boldsymbol{x} - \begin{bmatrix} 0 \\ h \end{bmatrix} .$$

The point on $\partial\Omega$ nearest to $\boldsymbol{x}$ is $\boldsymbol{p} \in \Gamma_h$: $\boldsymbol{p} = \boldsymbol{x} + \frac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix} s$, marked with • in Fig. 98 beside.

The equation of (4.2.3a) anchored at $\boldsymbol{x} \leftrightarrow C$ is

$$4u_C - u_E - u_N - u_W - u_S = h^2 f(\boldsymbol{x}) , \tag{4.2.5}$$

where we wrote

$$u_C := u_h(\boldsymbol{x}) , \quad u_E := u_h(\boldsymbol{x} + \begin{bmatrix} h \\ 0 \end{bmatrix}) , \quad u_N := u_h(\boldsymbol{x} + \begin{bmatrix} 0 \\ h \end{bmatrix}) , \quad u_W := u_h(\boldsymbol{x} - \begin{bmatrix} h \\ 0 \end{bmatrix}) ,$$
$$u_S := u_h(\boldsymbol{x} - \begin{bmatrix} 0 \\ h \end{bmatrix}) .$$

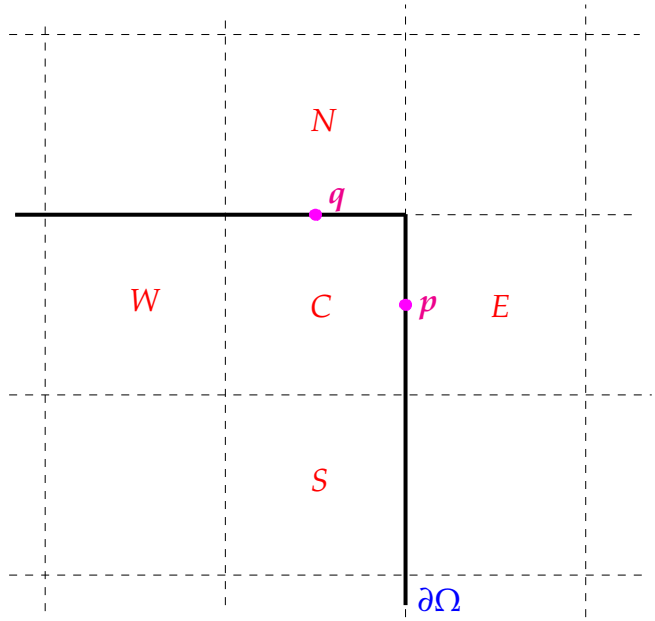
The equation of (4.2.3b) belonging to $\boldsymbol{p}$ is

$$u_E - u_C = hg(\boldsymbol{p}), \quad \text{because} \quad \boldsymbol{n}(\boldsymbol{p}) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} . \tag{4.2.6}$$

Using (4.2.6) we can eliminate u_E from (4.2.5) and obtain

$$3u_C - u_N - u_W - u_S = h^2 f(\boldsymbol{x}) + hg(\boldsymbol{p}) . \tag{4.2.7}$$

The same considerations can be pursued for all other boundaries and point sets Ω_h^N, Ω_h^W, and Ω_h^S, eliminating that grid point value accessed by the five-point stencil, but located outside Ω_h. In fact, this is consequence of the inherent axi-symmetry of the boundary value problem. It is very important to exploit that symmetry in order to carry out the computations efficiently.



❸: Point x of Ω_h near of corner of Ω

◁ Point $x \leftrightarrow C$ adjacent to upper right corner of Ω_h: $x = h\begin{bmatrix} \frac{2n-1}{2} \\ \frac{2n-1}{2} \end{bmatrix}$. Nearest boundary points

$$p = x + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}, \quad q = x + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}.$$

We use the same notations as above. The equation (4.2.3a) for x reads

$$4u_C - u_E - u_N - u_W - u_S = h^2 f(x). \tag{4.2.8}$$

The equations (4.2.3b) for p/q are

$$u_E - u_C = hg(p), \tag{4.2.9}$$
$$u_N - u_C = hg(q). \tag{4.2.10}$$

From them we eliminatie u_E and u_N in (4.2.8) and get

$$2u_C - u_W - u_S = h^2 f(x) + hg(p) + hg(q). \tag{4.2.11}$$

Analogous equations arise for the other three corners of Ω.

Finally we arrive at the $n^2 \times n^2$ linear system

$$4u_h(x) - u_h(x - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x - \begin{bmatrix} 0 \\ h \end{bmatrix}) - u_h(x + \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) \quad \forall x \in \widehat{\Omega}_h, \tag{4.2.12a}$$

$$3u_h(x) - u_h(x - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x - \begin{bmatrix} 0 \\ h \end{bmatrix}) - u_h(x + \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) + hg(x + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) \quad \forall x \in \widehat{\Omega}_h^E, \tag{4.2.12b}$$

$$3u_h(x) - u_h(x - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x - \begin{bmatrix} 0 \\ h \end{bmatrix}) - u_h(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) = h^2 f(x) + hg(x + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}) \quad \forall x \in \widehat{\Omega}_h^N, \tag{4.2.12c}$$

$$3u_h(x) - u_h(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x - \begin{bmatrix} 0 \\ h \end{bmatrix}) - u_h(x + \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) + hg(x - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) \quad \forall x \in \widehat{\Omega}_h^W, \tag{4.2.12d}$$

$$3u_h(x) - u_h(x - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x + \begin{bmatrix} 0 \\ h \end{bmatrix}) - u_h(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) = h^2 f(x) + hg(x - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}) \quad \forall x \in \widehat{\Omega}_h^S, \tag{4.2.12e}$$

$$2u_h(x) - u_h(x - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x - \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) + hg(x + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(x + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; x = h\begin{bmatrix} \frac{2n-1}{2} \\ \frac{2n-1}{2} \end{bmatrix}, \tag{4.2.12f}$$

$$2u_h(x) - u_h(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x - \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) + hg(x - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(x + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; x = h\begin{bmatrix} \frac{1}{2} \\ \frac{2n-1}{2} \end{bmatrix}, \tag{4.2.12g}$$

$$2u_h(x) - u_h(x + \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x + \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) + hg(x - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(x - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; x = h\begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}, \tag{4.2.12h}$$

$$2u_h(x) - u_h(x - \begin{bmatrix} h \\ 0 \end{bmatrix}) - u_h(x + \begin{bmatrix} 0 \\ h \end{bmatrix}) = h^2 f(x) + hg(x + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(x - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; x = h\begin{bmatrix} \frac{2n-1}{2} \\ \frac{1}{2} \end{bmatrix}. \tag{4.2.12i}$$

Of course, any equation can be scaled with some power of h without changing anything.

HINT 1 for (4-2.d): Take the cue from the way how the block-tridiagonal Poisson matrix is written in [Lecture → Eq. (4.1.2.8)]. ⌟

SOLUTION of (4-2.d):

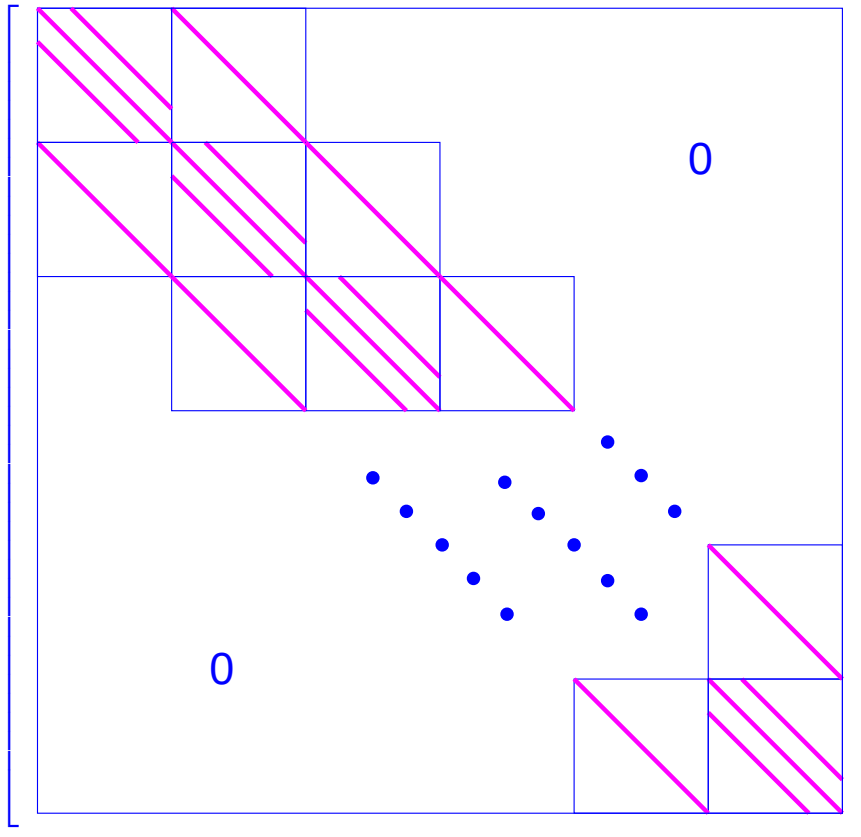
Let us write μ_k, $k = 1, \ldots, n^2$, for $u_h(\boldsymbol{x})$, where $\boldsymbol{x} \in \Omega_h$ is the interior grid point with index k in the lexikographic numbering. Then (4.2.12) can be rewritten as

$$
\begin{aligned}
4\mu_k - \mu_{k-1} - \mu_{k+1} - \mu_{k+n} - \mu_{k-n} &= h^2 f_k \quad \forall k \in \widehat{\Omega}_h \,, \\
3\mu_k - \mu_{k-1} - \mu_{k-n} - \mu_{k+n} &= h^2 f_k + hg(\boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) \quad \forall k \in \widehat{\Omega}_h^E \,, \\
3\mu_k - \mu_{k-1} - \mu_{k+1} - \mu_{k-n} &= h^2 f_k + hg(\boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}) \quad \forall k \in \widehat{\Omega}_h^N \,, \\
3\mu_k - \mu_{k+1} - \mu_{k+n} - \mu_{k-n} &= h^2 f_k + hg(\boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) \quad \forall k \in \widehat{\Omega}_h^W \,, \\
3\mu_k - \mu_{k-1} - \mu_{k+1} - \mu_{k+n} &= h^2 f_k + hg(\boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}) \quad \forall k \in \widehat{\Omega}_h^S \,, \\
2\mu_k - \mu_{k-1} - \mu_{k-n} &= h^2 f_k + hg(\boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(\boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; k = n^2 \,, \\
2\mu_k - \mu_{k+1} - \mu_{k-n} &= h^2 f_k + hg(\boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(\boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; k = (n-1)n \,, \\
2\mu_k - \mu_{k+1} - \mu_{k+n} &= h^2 f_k + hg(\boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(\boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; k = 1 \,, \\
2\mu_k - \mu_{k-1} - \mu_{k+n} &= h^2 f_k + hg(\boldsymbol{x} + \tfrac{1}{2}\begin{bmatrix} h \\ 0 \end{bmatrix}) + hg(\boldsymbol{x} - \tfrac{1}{2}\begin{bmatrix} 0 \\ h \end{bmatrix}), \; k = n \,.
\end{aligned}
$$

Here, for instance $k \in \widehat{\Omega}_h$ means that the equation has to hold for all indices k for which the associated grid point belongs to $\widehat{\Omega}_h$.

The matrix $\mathbf{A}$ has the following sparsity pattern and block structure, which is the same as for a $n^2 \times n^2$ Poisson matrix:

$\mathbf{A} =$



In detail $\mathbf{A}$ is and $n \times n$ block matrix wit blocks of size $n \times n$:

$$\mathbf{A} := \begin{bmatrix} \mathbf{S} & -\mathbf{I} & 0 & \cdots & \cdots & 0 \\ -\mathbf{I} & \mathbf{T} & -\mathbf{I} & & & \vdots \\ 0 & -\mathbf{I} & \mathbf{T} & -\mathbf{I} & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & -\mathbf{I} & \mathbf{T} & -\mathbf{I} \\ 0 & \cdots & \cdots & 0 & -\mathbf{I} & \mathbf{S} \end{bmatrix} \in \mathbb{R}^{n^2,n^2}, \tag{4.2.13}$$

$$\mathbf{T} := \begin{bmatrix} 3 & -1 & 0 & & & 0 \\ -1 & 4 & -1 & & & \vdots \\ 0 & -1 & 4 & -1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & -1 & 4 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 3 \end{bmatrix}, \quad \mathbf{S} := \begin{bmatrix} 2 & -1 & 0 & & & 0 \\ -1 & 3 & -1 & & & \vdots \\ 0 & -1 & 3 & -1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & -1 & 3 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 2 \end{bmatrix} \in \mathbb{R}^{n,n}. \tag{4.2.14}$$

SOLUTION of (4-2.e):

(I) The right-hand side of a square linear system has no influence whatsoever on uniqueness of solutions. Solutions will be unique if and only if the kernel of the system matrix is trivial. So let us examine the nullspace $\mathcal{N}(\mathbf{A})$.

All row sums of $\mathbf{A}$ vanish. This means that $\mathbf{A1} = \mathbf{0}$ for $\mathbf{1} = [1,\ldots,1]^\top \in \mathbb{R}^{n^2}$. Since $\mathbf{A}$ is diagonally dominant, this vector spans $\mathcal{N}(\mathbf{A})$. Hence, $\mathbf{A}$ has a non-trivial kernel and no conditions satisfied by f or g can restore uniqueness of solutions, even if they exist. Solutions of (4.2.12) will never be unique !

(II) We use the following result from linear algebra

Theorem 4.2.15. Existence of solutions of a square linear system

For a symmetric matrix $\mathbf{A} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$ and $\vec{\boldsymbol{\varphi}} \in \mathbb{R}^N$ the following is equivalent:

(i) The linear system of equations $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}$ has a solution.

(ii) $\vec{\boldsymbol{\varphi}}^\top \vec{\boldsymbol{\zeta}} = 0$ for all $\vec{\boldsymbol{\zeta}} \in \mathcal{N}(\mathbf{A})$.

Since we know $\mathcal{N}(\mathbf{A}) = \operatorname{Span}\{[1,\ldots,1]^\top\}$ the necessary and sufficient condition for the existence of a solution is that $[1,\ldots,1]^\top \vec{\boldsymbol{\varphi}} = 0$, where $\vec{\boldsymbol{\varphi}}$ denotes the right-hand side vector of the linear system of equations (4.2.12), which means that the sum of all components of $\vec{\boldsymbol{\varphi}}$ has to vanish. For the concrete right-hand-side vector is amounts to

$$h^2 \sum_{\boldsymbol{x}\in\Omega_h} f(\boldsymbol{x}) + h \sum_{\boldsymbol{x}\in\Gamma_h} g(\boldsymbol{x}) = 0 \quad . \tag{4.2.16}$$

Remark. In [Lecture → Ex. 1.8.0.10] we found that the data f, g for the Neumann boundary value problem have to satisfy the compatibility condition

$$\int_\Omega f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} g(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0\,. \tag{4.2.17}$$

The condition (4.2.16) is a discrete version of (4.2.17), the integrals being approximated by a composite midpoint quadrature formula.

HINT 1 for (5-1.a): The test space is the function space of admissible perturbations and those have to vanish where the solution is fixed. ⌟

SOLUTION of (5-1.a):

This subproblem has two solutions:

(I) We can work in the "classical" function space $(C^1_{\mathrm{pw}}([0,1]))^2$. Since the endpoints of the curves are fixed, perturbations/test functions have to vanish in $\xi = 0$ and $\xi = 1$, which yields the test space

$$V_0 = (C^1_{0,\mathrm{pw}}([0,1]))^2 := \left\{ \mathbf{v} \in (C^1_{\mathrm{pw}}([0,1]))^2 : \mathbf{v}(0) = \mathbf{v}(1) = \mathbf{0} \right\} . \tag{5.1.7}$$

(II) Applying the Cauchy-Schwarz inequality [Lecture $\rightarrow$ Eq. (1.3.4.15)] we can estimate

$$J_{\mathrm{tens}}(u) = \left(\int_0^1 1 \cdot \left\| \mathbf{u}'(\xi) \right\| \mathrm{d}\xi \right)^2 \leq \int_0^1 1 \, \mathrm{d}\xi \cdot \int_0^1 \left\| \mathbf{u}'(\xi) \right\|^2 \mathrm{d}\xi \leq \|\mathbf{u}\|^2_{H^1(]0,1[)} .$$

Thus, the functionals J_{tense} and J_{f} are both well-defined for $\mathbf{u} \in \left(H^1(]0,1[)\right)^2$, so that we could consider the variational problems also in that Sobolev space. In this case, the test space would be

$$V_0 := \left(H^1_0(]0,1[) \right)^2 . \tag{5.1.8}$$

HINT 1 for (5-1.b): Observe that J_f is a *linear* functional and that the derivative of a linear mapping is easy to compute, *cf.* [Lecture $\to$ Ex. 5.2.2.1]. ⌟

SOLUTION of (5-1.b):

A linear functional is "its own derivative": for all $\mathbf{v} \in V_0$

$$\lim_{t\to 0} \frac{J_f(\mathbf{u}+t\mathbf{v}) - J_f(\mathbf{u})}{t} = -\lim_{t\to 0} \frac{1}{t} \int_0^1 \mathbf{f}(\xi)\cdot t\mathbf{v}(\xi)\,\mathrm{d}\xi = -\int_0^1 \mathbf{f}(\xi)\cdot \mathbf{v}(\xi)\,\mathrm{d}\xi\ .$$

Note that the directional derivative is "constant", that is, it does not depend on $\mathbf{u}$, as we already observed with [Lecture $\rightarrow$ Eq. (5.2.2.2)].

HINT 1 for (5-1.c): You may use the following result: for $\mathbf{x} = \left[\begin{smallmatrix} x_1 \\ x_2 \end{smallmatrix}\right] \in \mathbb{R}^2 \setminus \{0\}$, $\mathbf{h} \in \mathbb{R}^2$, and $\mathbb{R} \ni t \to 0$ we find

$$\begin{aligned} \|\mathbf{x}+t\mathbf{h}\| &= \sqrt{(x_1+th_1)^2+(x_2+th_2)^2} = \sqrt{\|\mathbf{x}\|^2+2t\mathbf{x}\cdot\mathbf{h}+t^2\|\mathbf{h}\|^2} \\ &= \|\mathbf{x}\|\sqrt{1+2t\frac{\mathbf{x}\cdot\mathbf{h}}{\|\mathbf{x}\|^2}+t^2\frac{\|\mathbf{h}\|^2}{\|\mathbf{x}\|^2}} = \|\mathbf{x}\|+t\frac{\mathbf{x}\cdot\mathbf{h}}{\|\mathbf{x}\|}+O(t^2)\,. \end{aligned} \tag{5.1.9}$$

The identity $A^2-B^2=(A-B)(A+B)$ may also come in handy. ⌟

SOLUTION of (5-1.c):

The definition of the directional derivative of the tension energy J_{tens} in direction $\mathbf{v} \in V_0$ requires us to compute the limit

$$\langle \mathrm{D}J_{\text{tens}}(\mathbf{u}), \mathbf{v} \rangle = \lim_{t \to 0} \frac{J_{\text{tens}}(\mathbf{u} + t\mathbf{v}) - J_{\text{tens}}(\mathbf{u})}{t} .$$

For this, we make use of the identity

$$A^2 - B^2 = (A - B)(A + B) , \tag{5.1.10}$$

where

$$A = \sqrt{J_{\text{tens}}(\mathbf{u} + t\mathbf{v})} = \int_0^1 \left\| \mathbf{u}'(\xi) + t\mathbf{v}'(\xi) \right\| \mathrm{d}\xi \tag{5.1.11}$$

and

$$B = \sqrt{J_{\text{tens}}(\mathbf{u})} = \int_0^1 \left\| \mathbf{u}'(\xi) \right\| \mathrm{d}\xi .$$

Recall (5.1.9)

$$\|\mathbf{x} + t\mathbf{h}\| = \|\mathbf{x}\| + t\frac{\mathbf{x} \cdot \mathbf{h}}{\|\mathbf{x}\|} + O(t^2) \quad \text{for} \quad t \to 0 , \tag{5.1.12}$$

and use it with $\mathbf{x} := \mathbf{u}'(\xi)$ and $\mathbf{h} := \mathbf{v}'(\xi)$ to obtain

$$\left\| \mathbf{u}'(\xi) + t\mathbf{v}'(\xi) \right\| = \left\| \mathbf{u}'(\xi) \right\| + t\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2) \quad \text{for} \quad t \to 0 . \tag{5.1.13}$$

Plug this expansion into the expression (5.1.11) for A and compute for $t \to 0$

$$\begin{aligned} A - B &= \int_0^1 \left\| \mathbf{u}'(\xi) + t\mathbf{v}'(\xi) \right\| - \left\| \mathbf{u}'(\xi) \right\| \mathrm{d}\xi \\ &= \int_0^1 \left\| \mathbf{u}'(\xi) \right\| + t\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2) - \left\| \mathbf{u}'(\xi) \right\| \mathrm{d}\xi \\ &= \int_0^1 t\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2) \, \mathrm{d}\xi \quad \text{for} \quad t \to 0 \end{aligned}$$

and

$$A + B = \int_0^1 \left\| \mathbf{u}'(\xi) + t\mathbf{v}'(\xi) \right\| + \left\| \mathbf{u}'(\xi) \right\| \mathrm{d}\xi = \int_0^1 2\left\| \mathbf{u}'(\xi) \right\| + t\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2) \, \mathrm{d}\xi .$$

Combine these expansions according to (5.1.10) and compute the limit

$$\langle \mathrm{D}J_{\text{tens}}(\mathbf{u}), \mathbf{v} \rangle = \lim_{t \to 0} \frac{J_{\text{tens}}(\mathbf{u} + t\mathbf{v}) - J_{\text{tens}}(\mathbf{u})}{t} = \lim_{t \to 0} \frac{A^2 - B^2}{t} = \lim_{t \to 0} \frac{(A - B)(A + B)}{t}$$

$$
\begin{aligned}
&= \lim_{t\to 0} \frac{1}{t} \left(\int_0^1 t \frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2)\,\mathrm{d}\xi \right) \left(\int_0^1 2\|\mathbf{u}'(\xi)\| + t\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2)\,\mathrm{d}\xi \right) \\
&= \lim_{t\to 0} \left(\int_0^1 \frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t)\,\mathrm{d}\xi \right) \left(\int_0^1 2\|\mathbf{u}'(\xi)\| + t\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2)\,\mathrm{d}\xi \right) \\
&= 2\int_0^1 \frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}\,\mathrm{d}\xi \underbrace{\int_0^1 \|\mathbf{u}'(\xi)\|\,\mathrm{d}\xi}_{=:\ell(\mathbf{u})} = 2\ell(\mathbf{u}) \int_0^1 \frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}\,\mathrm{d}\xi
\end{aligned}
$$

Here, $\ell(\mathbf{u})$ is the **length of the curve** $\mathbf{u} : [0,1] \to \mathbb{R}^2$.

Note: There are alternative ways to compute the directional derivative $\langle \mathrm{D}J_{\mathrm{tens}}(\mathbf{u}), \mathbf{v}\rangle$. One may expand straightforwardly

$$
\begin{aligned}
J_{\mathrm{tens}}(\mathbf{u}+t\mathbf{v}) &= \left(\int_0^1 \|\mathbf{u}'(\xi) + t\mathbf{v}'(\xi)\|\,\mathrm{d}\xi \right)^2 \\
&\overset{(5.1.13)}{=} \left(\int_0^1 \|\mathbf{u}'(\xi)\| + t\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} + O(t^2)\,\mathrm{d}\xi \right)^2 \\
&= \left(\int_0^1 \|\mathbf{u}'(\xi)\|\,\mathrm{d}\xi + t\int_0^1 \frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}\,\mathrm{d}\xi + O(t^2) \right)^2 \\
&= \left(\int_0^1 \|\mathbf{u}'(\xi)\|\,\mathrm{d}\xi \right)^2 + 2t\int_0^1 \|\mathbf{u}'(\xi)\|\,\mathrm{d}\xi \cdot \int_0^1 \frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}\,\mathrm{d}\xi + O(t^2)\,.
\end{aligned}
$$

HINT 1 for (5-1.d): Do not forget to specify the trial and test spaces. ⌟

SOLUTION of (5-1.d):

According to [Lecture → Thm. 5.2.1.5] the variational problem is: Find $\mathbf{u} \in \left\{\mathbf{w} \in (C^1_{\mathrm{pw}}([0,1]))^2 : \mathbf{w}(0) =\right.$ such that

$$\langle \mathrm{D}J(\mathbf{u}), \mathbf{v} \rangle = \int_0^1 2\ell(\mathbf{u}) \frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} - \mathbf{f}(\xi) \cdot \mathbf{v}(\xi) \, \mathrm{d}\xi = 0 \quad \forall \mathbf{v} \in (C^1_{0,\mathrm{pw}}([0,1]))^2 \ .$$

Alternatively, the variational problem could also be set in the Sobolev spaces $\widehat{V} := \{\mathbf{w} \in \left(H^1(]0,1[)\right)^2 : \mathbf{w}$ and $V_0 = \left(H^1_0(]0,1[)\right)^2$.

SOLUTION of (5-1.e):

By using integration by parts, we have that

$$2\ell(\mathbf{u}) \int_0^1 \frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} \, \mathrm{d}\xi = -2\ell(\mathbf{u}) \int_0^1 \frac{\mathrm{d}}{\mathrm{d}\xi} \left(\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|} \right) \cdot \mathbf{v}(\xi) \, \mathrm{d}\xi$$

where we used the fact that $\mathbf{v}(0) = \mathbf{v}(1) = \mathbf{0}$. The variational equation can be rewritten as

$$\int_0^1 \left(-2\ell(\mathbf{u}) \frac{\mathrm{d}}{\mathrm{d}\xi} \left(\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|} \right) - \mathbf{f}(\xi) \right) \cdot \mathbf{v}(\xi) \, \mathrm{d}\xi = 0$$

which, under the necessary smoothness assumptions, results in the following two-point boundary value problem:

$$2\ell(\mathbf{u}) \frac{\mathrm{d}}{\mathrm{d}\xi} \left(\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|} \right) = -\mathbf{f}(\xi) \quad \text{in} \quad]0,1[\, ,$$

with the boundary conditions

$$\mathbf{u}(0) = \mathbf{u}_0, \qquad \mathbf{u}(1) = \mathbf{u}_1 \, .$$

SOLUTION of (5-2.a):

Writing $\mathbf{u}^i = \begin{bmatrix} u_1^i \\ u_2^i \end{bmatrix}$, we assume that $u_2^i \leq 0$ for all $i = 0, \ldots, N+1$ and we set $\mathbf{u}^{N+1} := \boldsymbol{b}$.

Each straight segment $[\mathbf{u}^i, \mathbf{u}^{i+1}]$, $i = 0, \ldots, N$, has length $\|\mathbf{u}^i - \mathbf{u}^{i+1}\|$, and according to Ass. 5.2.3 we approximate the speed of the ball on this segment by the constant speed

$$v^i := \sqrt{-\frac{1}{2}(u_2^i + u_2^{i+1})}\,, \quad i = 0, \ldots, N\,. \tag{5.2.4}$$

This means that the time t_i it takes the ball to cover segment $[\mathbf{u}^i, \mathbf{u}^{i+1}]$ is approximately

$$t_i := t(\mathbf{u}^i, \mathbf{u}^{i+1}) := \frac{\|\mathbf{u}^{i+1} - \mathbf{u}^i\|}{v^i} = \frac{\|\mathbf{u}^i - \mathbf{u}^{i+1}\|}{\sqrt{-\frac{1}{2}(u_2^i + u_2^{i+1})}}\,, \tag{5.2.5}$$

and, since friction is neglected, the total time required for rolling down the curve is

$$T_N(\mathbf{u}^1, \ldots, \mathbf{u}^N) = \sum_{i=0}^{N} t(\mathbf{u}^i, \mathbf{u}^{i+1}) = \sum_{i=0}^{N} \frac{\|\mathbf{u}^i - \mathbf{u}^{i+1}\|}{\sqrt{-\frac{1}{2}(u_2^i + u_2^{i+1})}}\,. \tag{5.2.6}$$

The knot positions $\mathbf{u}^i$ of the optimal polygon minimize $T_N(\mathbf{u}^1, \ldots, \mathbf{u}^N)$.

SOLUTION of (5-2.j):

Remember that teh coefficient function is given by

$$\sigma(\mathbf{v},\mathbf{g}) := \frac{1}{\sqrt{-v_2}\,\|\mathbf{g}\|} \quad \Longrightarrow \quad \sigma(\mathbf{u}(\xi),\mathbf{u}'(\xi)) = \frac{1}{\sqrt{-(\mathbf{u}(\xi))_2}\,\|\mathbf{u}'(\xi)\|}\,.$$

Since the components of $\xi \mapsto \mathbf{u}_h(\xi)$ are $\mathcal{M}$-piecewise linear, we have (C++ indexing !)

$$\begin{aligned}
\mathbf{u}_h(x^-_{j-1/2}) &= (\tfrac{1}{2} - \tfrac{1}{2\sqrt{3}})\mathbf{u}_h(x_j) + (\tfrac{1}{2} + \tfrac{1}{2\sqrt{3}})\mathbf{u}_h(x_{j-1}) && (5.2.32)\\
&= (0.5 - 0.5 * \texttt{sqrt}(3))(\texttt{knots.col}(j) + (0.5 + 0.5 * \texttt{sqrt}(3))\texttt{knots.col}(j-1))\,,\\
\mathbf{u}_h(x^+_{j-1/2}) &= (\tfrac{1}{2} + \tfrac{1}{2\sqrt{3}})\mathbf{u}_h(x_j) + (\tfrac{1}{2} - \tfrac{1}{2\sqrt{3}})\mathbf{u}_h(x_{j-1}) && (5.2.33)\\
&= (0.5 + 0.5 * \texttt{sqrt}(3))(\texttt{knots.col}(j) + (0.5 - 0.5 * \texttt{sqrt}(3))\texttt{knots.col}(j-1))\,,\\
\mathbf{u}'_h(x^\pm_{j-1/2}) &= \frac{1}{h}(\mathbf{u}_h(x_j) - \mathbf{u}_h(x_{j-1})) = \frac{1}{h}(\texttt{knots.col}(j) - \texttt{knots.col}(j-1))\,, \quad h := M^{-1}, && (5.2.34)
\end{aligned}$$

for $j = 1,\ldots,M$.

These values have to be plugged into the formula for σ as given in (5.2.25). The implementation is straightforward and can be based on simple C++ loops of compact linear-algebra operations supplied by the EIGEN library.

C++ code 5.2.35: Implementation of `coeff_sigma()` ⇢ GitLab

```
Eigen::VectorXd coeff_sigma(
    const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots) {
  int M = knots.cols() - 1;
  Eigen::VectorXd sigma(M);
  // Definition of locations quadrature points (2-point Gauss rule)
  const double rh = .5 + .5 / std::sqrt(3.);
  const double lh = .5 - .5 / std::sqrt(3.);
  const double h = 1.0 / static_cast<double>(M);
  // Loop over all cells of the mesh
  for (int i = 0; i < M; ++i) {
    // Evaluate the y-component of uh on both quadrature points
    const double uh2l = (1 - lh) * knots(1, i) + lh * knots(1, i + 1);
    const double uh2r = (1 - rh) * knots(1, i) + rh * knots(1, i + 1);
    // Compute the derivative
    const Eigen::Vector2d duh = (1. / h) * (knots.col(i + 1) - knots.col(i));
    // Norm of the derivative vector
    const double duh_norm = duh.norm();
    // Compute sigma integrated over the segment normalized to unit
       length
    sigma[i] = 0.5 * (1. / (std::sqrt(-uh2r) * duh_norm) +
                      1. / (std::sqrt(-uh2l) * duh_norm));
  }
  return sigma;
}
```

SOLUTION of (5-2.k):

From (5.2.25) recall that

$$\mathbf{f}(\mathbf{v},\mathbf{g}) := \frac{\|\mathbf{g}\|}{2\sqrt{-v_2}\,v_2}\begin{bmatrix}0\\1\end{bmatrix}$$

$$\blacktriangleright \quad \left(\mathbf{f}(\mathbf{u}_h(x_{j-1/2}^{\pm}),\mathbf{u}_h'(x_{j-1/2}^{\pm}))\right)_2 = \frac{\|\mathbf{g}\|}{2\sqrt{-v_2}\,v_2}\,, \quad \begin{aligned}\mathbf{g} &:= \mathbf{grad}\,\mathbf{u}_h|_{]x_{j-1},x_j[}\,,\\ v_2 &:= \left(\mathbf{u}_h(x_{j-1/2}^{\pm})\right)_2\,.\end{aligned}$$

Thus, based on the formulas (5.2.32), (5.2.33) and (5.2.34) a loop-based implementation is straightforward.

C++ code 5.2.36: Implementation of `sourcefn2()` ➔ GitLab

```
2  Eigen::Matrix<double, Eigen::Dynamic, 2> sourcefn2(
3      const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots) {
4    const std::size_t M = knots.cols() - 1;
5    Eigen::Matrix<double, Eigen::Dynamic, 2> f(M, 2);
6    // Definition quadrature points (2-point Gauss)
7    const double rh = .5 + .5 / std::sqrt(3.);
8    const double lh = .5 - .5 / std::sqrt(3.);
9    const double h = 1. / static_cast<double>(M);
10
11   // Loop over all segments of the curve
12   for (int i = 0; i < M; ++i) {
13     // Compute the derivative of uh in the current cell
14     const Eigen::VectorXd uhp = 1. / h * (knots.col(i + 1) - knots.col(i));
15     const double uhp_norm = uhp.norm();
16     // Evaluate the y-component of uh on both quadrature points
17     const double uh2l = (1 - lh) * knots(1, i) + lh * knots(1, i + 1);
18     const double uh2r = (1 - rh) * knots(1, i) + rh * knots(1, i + 1);
19     // Compute the second components of the function f
20     f(i, 0) = uhp_norm / (2.0 * std::sqrt(-uh2l) * uh2l);
21     f(i, 1) = uhp_norm / (2.0 * std::sqrt(-uh2r) * uh2r);
22   }
23   return f;
24 }
```

HINT 1 for (5-2.l): To find the right formulas for $\vec{\varphi}_i(\mathbf{u}_{l_h})$ you have to understand how to handle offset functions in the context of finite-element Galerkin discretization, see [Lecture → Rem. 2.3.3.15]. ⌟

SOLUTION of (5-2.l):

The linear system of equations (5.2.37) arises from the discrete variational problem

$$\mathbf{w}_h^{(k+1)} \in V_0: \quad \int_0^1 \sigma(\mathbf{u}_h^{(k)}(\xi), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(\xi)) \frac{d(\mathbf{u}_0 + \mathbf{w}_h^{(k+1)})}{d\xi}(\xi) \cdot \mathbf{v}_h'(\xi)\, d\xi = \int_0^1 \mathbf{f}(\mathbf{u}_h^{(k)}(\xi), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(\xi)) \cdot \mathbf{v}_h(\xi)\, d\xi \quad \forall \mathbf{v}_h \in V_{0,h}\,. \tag{5.2.38}$$

More precisely, since we are supposed to use the 2-point Gauss quadrature rule, the actual variational problem is: find $\mathbf{w}_h^{(k+1)} \in V_{0,h}$ such that

$$\begin{aligned} h \sum_{j=1}^{M} \tfrac{1}{2} \Big(& \sigma(\mathbf{u}_h^{(k)}(x_{j-1/2}^-), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(x_{j-1/2}^-)) \frac{d(\mathbf{u}_0 + \mathbf{w}_h^{(k+1)})}{d\xi}(x_j^-) \cdot \mathbf{v}_h'(x_{j-1/2}^-) + \\ & \sigma(\mathbf{u}_h^{(k)}(x_{j-1/2}^+), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(x_{j-1/2}^+)) \frac{d(\mathbf{u}_0 + \mathbf{w}_h^{(k+1)})}{d\xi}(x_{j-1/2}^+) \cdot \mathbf{v}_h'(x_{j-1/2}^+) \Big) \\ = h \sum_{j=1}^{M} \tfrac{1}{2} \Big(& \mathbf{f}(\mathbf{u}_h^{(k)}(x_{j-1/2}^-), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(x_{j-1/2}^-)) \cdot \mathbf{v}_h(x_{j-1/2}^-) + \\ & \mathbf{f}(\mathbf{u}_h^{(k)}(x_{j-1/2}^+), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(x_{j-1/2}^+)) \cdot \mathbf{v}_h(x_{j-1/2}^+) \Big) \quad \forall \mathbf{v}_h \in V_{0,h}\,, \end{aligned} \tag{5.2.39}$$

where the points $x_j^\pm$ are defined in (5.2.28).

❶ We realize that the block partitioning of the linear system (5.2.37) arises from the partitioning of the basis functions according to $\mathfrak{B}_h = \mathfrak{B}_h^1 \cup \mathfrak{B}_h^2$. The off-diagonal zero blocks are zero due to the fact that

$$\mathbf{v}_h \in \mathrm{Span}\{\mathfrak{B}_1\},\ \mathbf{w}_h \in \mathrm{Span}\{\mathfrak{B}_2\} \quad \Rightarrow \quad \mathbf{v}_h'(\xi) \cdot \mathbf{w}_h'(\xi) = 0 \quad \forall \xi \in [0.1]\,. \tag{5.2.40}$$

The diagonal blocks of $\mathbf{R}(\mathbf{u}_h)$ turn out to be the $\mathcal{S}_{1,0}^0(\mathcal{M})$-spawned finite-element Galerkin matrices for the bilinear form

$$(w_h, v_h) \mapsto h \sum_{j=1}^{M} \tfrac{1}{2} \Big(\sigma_{j-1/2}^- w_h'(x_{j-1/2}^-) v_h'(x_{j-1/2}^-) + \sigma_{j-1/2}^+ w_h'(x_{j-1/2}^+) v_h'(x_{j-1/2}^+) \Big)\,, \tag{5.2.41}$$

$$\sigma_{j-1/2}^\pm := \sigma(\mathbf{u}_h(x_{j-1/2}^\pm), \mathbf{u}_h'(x_{j-1/2}^\pm))\,, \quad j = 1, \ldots, M\,, \tag{5.2.42}$$

where $\mathbf{u}_h := \mathbf{u}_0 + \mathbf{w}_h$, $\mathbf{w}_h \in V_{0,h}$. We can exploit that w_h' and v_h' are $\mathcal{M}$-piecewise constant to simplify the expression for the bilinear form

$$(w_h, v_h) \mapsto h \sum_{j=1}^{M} \sigma_{j-1/2}\, w_h'|_{]x_{j-1}, x_j[} v_h'|_{]x_{j-1}, x_j[}\,, \quad \sigma_{j-1/2} := \tfrac{1}{2}(\sigma_{j-1/2}^- + \sigma_{j-1/2}^+)\,. \tag{5.2.43}$$

This means for the matrix entries

$$\begin{aligned} (\mathbf{R}(\mathbf{u}_h))_{i,k} &= h \sum_{j=1}^{M} \sigma_{j-1/2} \left. \frac{db_h^k}{d\xi} \right|_{]x_{j-1}, x_j[} \left. \frac{db_h^i}{d\xi} \right|_{]x_{j-1}, x_j[} \\ &= \frac{1}{h} \cdot \begin{cases} \sigma_{(i+k)/2} & \text{, if } |i-k| = 1\,, \\ \sigma_{i-1/2} + \sigma_{i+1/2} & \text{, if } i = k\,, \\ 0 & \text{, if } |i-k| > 1\,, \end{cases} \qquad i,k \in \{1, \ldots, M-1\}\,, \end{aligned} \tag{5.2.44}$$

with $\sigma_{j-1/2}$, $j = 1, \ldots, M$, as in (5.2.43). The formulas (5.2.44) are a direct consequence of the properties of the scalar tent basis functions b_h^i. Refer to [Lecture → § 2.3.3.1] and [Lecture → § 2.3.3.10] for a detailed explanation of (5.2.44).

This means that $\mathbf{R}(\vec{\boldsymbol{\mu}})$ is the $(M-1) \times (M-1)$ tridiagonal matrix

$$\mathbf{R}(\mathbf{u}_h) = \frac{1}{h} \begin{bmatrix} \sigma_{1/2}+\sigma_{3/2} & -\sigma_{3/2} & 0 & \ldots & & & \ldots & 0 \\ -\sigma_{3/2} & \sigma_{3/2}+\sigma_{5/2} & -\sigma_{5/2} & \ldots & & & & \\ 0 & \ddots & \ddots & \ddots & & & & \\ \vdots & & & & & & & \vdots \\ & & & \ddots & \ddots & \ddots & & 0 \\ \vdots & & & & -\sigma_{M-5/2} & \sigma_{M-5/2}+\sigma_{M-3/2} & -\sigma_{M-3/2} & \\ 0 & \ldots & & \ldots & 0 & -\sigma_{M-3/2} & \sigma_{M-3/2}+\sigma_{M-1/2} & \end{bmatrix} . \tag{5.2.45}$$

Note that, writing `sigma_vec` for the vector returned by `coeff_sigma()`, we have $\sigma_{j-1/2} =$ `sigma_ve` $j = 1, \ldots, M$.

❷ Now we tackle the computation of $\vec{\boldsymbol{\varphi}}_i(\vec{\boldsymbol{\mu}})$, $i = 1, 2$. To that end we move linear terms in (5.2.39) containing the offset function to the right-hand side and get

$$\begin{aligned} h \sum_{j=1}^{M} \sigma_{j-1/2} \left. \frac{d(\mathbf{w}_h^{(k+1)})}{d\xi} \right|_{]x_{j-1},x_j[} \cdot \mathbf{v}_h' \big|_{]x_{j-1},x_j[} & \\ = -h \sum_{j=1}^{M} \sigma_{j-1/2} \left. \frac{d\mathbf{u}_0}{d\xi} \right|_{]x_{j-1},x_j[} \cdot \mathbf{v}_h' \big|_{]x_{j-1},x_j[} + & \\ h \sum_{j=1}^{M} \frac{1}{2} \Big(\mathbf{f}(\mathbf{u}_h^{(k)}(x_{j-1/2}^-), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(x_{j-1/2}^-)) \cdot \mathbf{v}_h(x_{j-1/2}^-) + & \\ \mathbf{f}(\mathbf{u}_h^{(k)}(x_{j-1/2}^+), \frac{d\mathbf{u}_h^{(k)}}{d\xi}(x_{j-1/2}^+)) \cdot \mathbf{v}_h(x_{j-1/2}^+) \Big) \quad \forall \mathbf{v}_h \in V_{0,h} . & \end{aligned}$$

Here we assumed $\mathbf{u}_0 \in \mathcal{S}_1^0(\mathcal{M})$ so that its derivative is $\mathcal{M}$-piecewise constant, too.

Next, note that $(\mathbf{f})_1 = 0$, which means that $\mathbf{f}(v, \mathbf{g}) \cdot \mathbf{v}_h = 0$, if $\mathbf{v}_h \in \mathrm{Span}\{\mathfrak{B}_1\}$, so that the second sum on the right-hand side does not contribute to $\vec{\boldsymbol{\varphi}}_1$. This gives us for $\mathbf{u}_h \in \left(\mathcal{S}_1^0(\mathcal{M})\right)^2$ and $i = 1, \ldots, M-1$

$$(\vec{\boldsymbol{\varphi}}_1(\mathbf{u}_h))_i = -h \sum_{j=1}^{M} \sigma_{j-1/2} \left. \frac{d\mathbf{u}_0}{d\xi} \right|_{]x_{j-1},x_j[} \cdot \begin{bmatrix} (b_h^i)' \big|_{]x_{j-1},x_j[} \\ 0 \end{bmatrix}$$

$$\begin{aligned} (\vec{\boldsymbol{\varphi}}_2(\mathbf{u}_h))_i = & -h \sum_{j=1}^{M} \sigma_{j-1/2} \left. \frac{d\mathbf{u}_0}{d\xi} \right|_{]x_{j-1},x_j[} \cdot \begin{bmatrix} 0 \\ (b_h^i)' \big|_{]x_{j-1},x_j[} \end{bmatrix} + \\ & h \sum_{j=1}^{M} \frac{1}{2} \left(\mathbf{f}(\mathbf{u}_h(x_{j-1/2}^-), \frac{d\mathbf{u}_h}{d\xi}(x_{j-1/2}^-)) \cdot \begin{bmatrix} 0 \\ b_h^i(x_{j-1/2}) \end{bmatrix} + \mathbf{f}(\mathbf{u}_h(x_{j-1/2}^+), \frac{d\mathbf{u}_h}{d\xi}(x_{j-1/2}^+)) \cdot \begin{bmatrix} b_h^i(x \end{bmatrix} \right. \end{aligned}$$

Only the offset function contributes to $\vec{\boldsymbol{\varphi}}_1$ and due to the special choice (5.2.30) we end up with

$$(\vec{\boldsymbol{\varphi}}_1(\mathbf{u}_h))_i = \frac{1}{h} \cdot \begin{cases} \sigma_{1/2}\, a_1 & \text{for} \quad i = 1 , \\ 0 & \text{for} \quad i = 2, \ldots, M-2 , \\ \sigma_{M-1/2}\, b_1 & \text{for} \quad i = M-1 . \end{cases} \tag{5.2.46}$$

The tail part of the right-hand side vector will see the $\mathbf{u}$-dependent source function plus a contribution from the offset function. To derive expressions for its components we recall that for the tent functions we have

$$\begin{aligned} b_h^i(x_{i-1/2}^-) &= \tfrac{1}{2} - \tfrac{1}{2\sqrt{3}}\,, \\ b_h^i(x_{i-1/2}^+) &= \tfrac{1}{2} + \tfrac{1}{2\sqrt{3}} \end{aligned} \qquad \text{and} \qquad \begin{aligned} b_h^i(x_{i+1/2}^-) &= \tfrac{1}{2} + \tfrac{1}{2\sqrt{3}}\,, \\ b_h^i(x_{i+1/2}^+) &= \tfrac{1}{2} - \tfrac{1}{2\sqrt{3}}\,. \end{aligned}$$

This gives us the formulas

$$(\vec{\varphi}_2(\mathbf{u}_h))_i = \begin{cases} \dfrac{\sigma_{1/2}}{h}\, a_2 \quad + \quad \tfrac{1}{2}h\big(\omega_1 f_{1/2}^- + \omega_2 f_{1/2}^+ + \omega_2 f_{3/2}^- + \omega_1 f_{3/2}^+\big) & \text{for} \quad i = 1\,, \\ \qquad\qquad\quad \tfrac{1}{2}h\big(\omega_1 f_{i-1/2}^- + \omega_2 f_{i-1/2}^+ + \omega_2 f_{i+1/2}^- + \omega_1 f_{i+1/2}^+\big) & \text{for} \quad i = 2, \ldots, M-2\,, \\ \dfrac{\sigma_{M-1/2}}{h}\, b_2 \quad + \quad \tfrac{1}{2}h\big(\omega_1 f_{M-3/2}^- + \omega_2 f_{M-3/2}^+ + \omega_2 f_{M-1/2}^- + \omega_1 f_{M-1/2}^+\big) & \text{for} \quad i = M-1\,, \end{cases} \tag{5.2.47}$$

with the abbreviations

$$\begin{aligned} f_{i-1/2}^{\pm} &:= \Big(\mathbf{f}(\mathbf{u}_h(x_{i-1/2}^{\pm}), \mathbf{u}_h'(x_{i-1/2}^{\pm}))\Big)_2\,, \\ \omega_1 &:= \tfrac{1}{2} - \tfrac{1}{2\sqrt{3}} \quad , \quad \omega_2 := \tfrac{1}{2} + \tfrac{1}{2\sqrt{3}}\,. \end{aligned}$$

To understand this formula, note that for $\mathbf{v}_h = \begin{bmatrix} 0 \\ b_h^i \end{bmatrix}$ only the summands for $j = i$ and $j = i+1$ do not vanish in the sum

$$h \sum_{j=1}^{M} \tfrac{1}{2} \left(\mathbf{f}(\mathbf{u}_h(x_{j-1/2}^-), \frac{d\mathbf{u}_h}{d\xi}(x_{j-1/2}^-)) \cdot \mathbf{v}_h(x_{j-1/2}^-) + \mathbf{f}(\mathbf{u}_h(x_{j-1/2}^+), \frac{d\mathbf{u}_h}{d\xi}(x_{j-1/2}^+)) \cdot \mathbf{v}_h(x_{j-1/2}^+) \right)\,.$$

If `f_mat` denotes the return value of `sourcefn2()`, then

$$f_{i-1/2}^- = \texttt{f_mat(i,0)} \quad , \quad f_{i-1/2}^+ = \texttt{f_mat(i,1)}\,.$$

HINT 1 for (5-2.m): It is convenient to rely on the function `coeff_sigma()` from Sub-problem (5-2.j). ⌟

HINT 2 for (5-2.m): Take care to initialize the sparse matrix efficiently. Since we know that $\mathbf{R}(\mathbf{u}_{l_l})$ has at most three non-zero entries per row and column, we can reserve enough space in advance and then set entries with the `insert()` member function. The code demonstrates this. ⌟

SOLUTION of (5-2.m):

A call to `coeff_sigma()` gives the numbers $\sigma_{i-1/2}$, $i = 1, \ldots, M$. Then (5.2.44) tells us how to set the matrix entries directly. No cell-based assembly is required here.

Note that the number M of mesh cells can be inferred from the size of the `knots` argument.

C++ code 5.2.48: Implementation of `matR()` ↪ GitLab

```
2  Eigen::SparseMatrix<double> matR(
3      const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots) {
4    const Eigen::Index M = knots.cols() - 1;
5    // Reserve enough space for the tridiagonal matrix
6    Eigen::SparseMatrix<double> R(M - 1, M - 1);
7    R.reserve(Eigen::RowVectorXi::Constant(M - 1, 3));
8    // Compute the cell values for the coefficient
9    const Eigen::VectorXd sigma = coeff_sigma(knots);
10   // Fill in the matrix
11   double h = 1. / static_cast<double>(M);
12   R.insert(0, 0) = (sigma(0) + sigma(1)) / h;
13   R.insert(0, 1) = -sigma(1) / h;
14   for (int i = 1; i < M - 2; ++i) {
15     R.insert(i, i - 1) = -sigma(i) / h;
16     R.insert(i, i) = (sigma(i) + sigma(i + 1)) / h;
17     R.insert(i, i + 1) = -sigma(i + 1) / h;
18   }
19   R.insert(M - 2, M - 3) = -sigma(M - 2) / h;
20   R.insert(M - 2, M - 2) = (sigma(M - 2) + sigma(M - 1)) / h;
21   // Clean the CRS format
22   R.makeCompressed();
23   return R;
24 }
```

HINT 1 for (5-2.n): You may rely on the functions `coeff_sigma()` and `sourcefn2()` you wrote in Sub-problem (5-2.j) and Sub-problem (5-2.k). ⌟

SOLUTION of (5-2.n):

We simply implement the formulas (5.2.46) and (5.2.47) and get the numbers $f_{i-1/2}$, $i = 1, \ldots, M$ by calling `sourcefn2()` and `coeff_sigma()` gives you the numbers $\sigma_{i-1/2}$.

C++ code 5.2.49: Implementation of `compute_rhs()` ↪ GitLab

```
2  Eigen::VectorXd compute_rhs(
3      const Eigen::Matrix<double, 2, Eigen::Dynamic> &knots, Eigen::Vector2d a,
4      Eigen::Vector2d b) {
5    Eigen::Index M = knots.cols() - 1;
6    double h = 1. / static_cast<double>(M);
7    // Compute the "coefficients" for all cells
8    // A bit wasteful, because we only need sigma[0] and sigma[M-1]
9    const Eigen::VectorXd sigma = coeff_sigma(knots);
10   const Eigen::Matrix<double, Eigen::Dynamic, 2> f = sourcefn2(knots);
11   // Right-hand side vector
12   Eigen::VectorXd rhs(2 * (M - 1));
13   // Compute head part half of rhs:  φ⃗_1
14   rhs.setZero();
15   rhs[0] = sigma[0] * a[0] / h;
16   rhs[M - 2] = sigma[M - 1] * b[0] / h;
17   // Compute second half of rhs based on the 2-point Gauss rule
18   const double w1 = .5 - .5 / std::sqrt(3.);
19   const double w2 = .5 + .5 / std::sqrt(3.);
20   for (int i = 0; i < M - 1; ++i) {
21     rhs[M - 1 + i] =
22         0.5 * h *
23         (w1 * f(i, 0) + w2 * f(i, 1) + w2 * f(i + 1, 0) + w1 * f(i + 1, 1));
24   }
25   // Take into account offset function
26   rhs[M - 1] += sigma[0] * a[1] / h;
27   rhs[2 * M - 3] += sigma[M - 1] * b[1] / h;
28   return rhs;
29 }
```

SOLUTION of (5-2.o):

In step $k \in \mathbb{N}_0$ of the iteration loop

- assemble the matrix $\mathbf{R}(\mathbf{u}_h^{(k)}) \in \mathbb{R}^{M-1,M-1}$ by calling the function `matR()` implemented in Sub-problem (5-2.m),
- initialize the right-hand side vector $\left[\vec{\varphi}_1(\mathbf{u}_h^{(k)}), \vec{\varphi}_2(\mathbf{u}_h^{(k)})\right]^\top \in \mathbb{R}^{2M-2}$ by means of `compute_rhs()`
- solve the linear system of equations

$$\begin{bmatrix} \mathbf{R}(\mathbf{u}_h^{(k)}) & \mathbf{O} \\ \mathbf{O} & \mathbf{R}(\mathbf{u}_h^{(k)}) \end{bmatrix} \vec{\mu}^{(k+1)} = \begin{bmatrix} \vec{\varphi}_1(\mathbf{u}_h^{(k)}) \\ \vec{\varphi}_2(\mathbf{u}_h^{(k)}) \end{bmatrix}, \tag{5.2.37}$$

- build the next iterate

$$\mathbf{u}_h^{(k+1)} = \mathbf{u}_0 + \begin{bmatrix} \sum_{\ell=1}^{M-1} \left(\vec{\mu}^{(k+1)}\right)_\ell b_h^\ell \\ \sum_{\ell=1}^{M-1} \left(\vec{\mu}^{(k+1)}\right)_{\ell+M-1} b_h^\ell \end{bmatrix}.$$

- and check for convergence: stop, when

$$\left\| \mathbf{u}_h^{(k-1)} - \mathbf{u}_h^{(k)} \right\|_{0,h} \leq \min\left\{ \texttt{rtol} \cdot \left\| \mathbf{u}_h^{(k)} \right\|_{0,h}, \texttt{atol} \right\}.$$

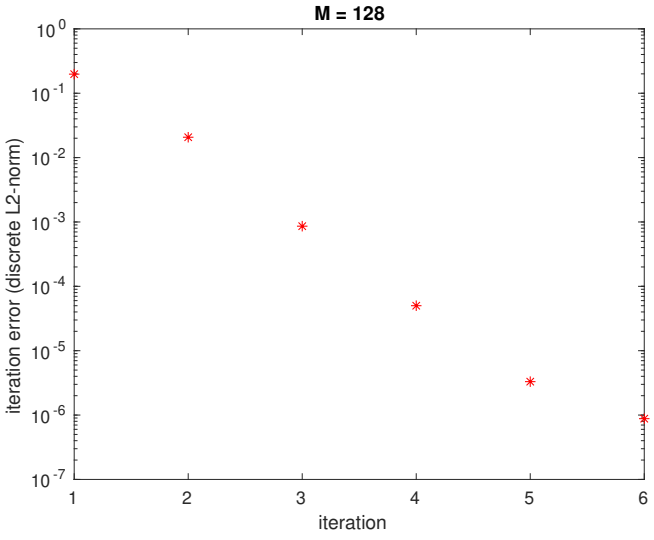
The iterates may be stored in the format (5.2.52), but storing the basis expansion coefficient vectors $\vec{\mu}^{(k)}$ is an option, too.

C++ code 5.2.53: Implementation of `brachistochrone()` ↪ GitLab

```
2  template <typename RECORDER = std::function<
3                void(const Eigen::Matrix<double, 2, Eigen::Dynamic> &)>>
4  Eigen::Matrix<double, 2, Eigen::Dynamic> brachistochrone(
5      unsigned int M, Eigen::Vector2d a, Eigen::Vector2d b, double atol,
6      double rtol, unsigned int itmax,
7      RECORDER &&rec = [](const Eigen::Matrix<double, 2, Eigen::Dynamic> &)
8          -> void { return; }) {
9    // Initialize knots
10   Eigen::Matrix<double, 2, Eigen::Dynamic> knots(2, M + 1);
11   // Linear interpolant as intial guess
12   for (int i = 0; i <= M; ++i) {
13     double s = static_cast<double>(i) / M;
14     knots.col(i) = (1.0 - s) * a + s * b;
15   }
16   Eigen::Matrix<double, 2, Eigen::Dynamic> knots_prev = knots;
17   // Record the initial conditions of the knots
18   rec(knots);
19   // Perform the fixed point-iteration
20   unsigned int q = 0;
21   do {
22     // Store the previous knots data for comparison later
23     knots_prev = knots;
```

```
24
25      // Compute R-matrix and RHS
26      Eigen::SparseMatrix<double> R = matR(knots);
27      Eigen::VectorXd rhs = compute_rhs(knots, a, b);
28      // Initialize the LU solver
29      Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
30      // Do LU decomposition once
31      solver.compute(R);
32      if (solver.info() != Eigen::Success) {
33        std::cerr << "Cannot factorize R!" << std::endl;
34        return knots;
35      }
36      // Solve the first system
37      Eigen::VectorXd uh = solver.solve(rhs.segment(0, M - 1));
38      for (int i = 0; i < M - 1; ++i) knots(0, i + 1) = uh(i);
39      // Solve the second system
40      Eigen::VectorXd uh2 = solver.solve(rhs.segment(M - 1, M - 1));
41      for (int i = 0; i < M - 1; ++i) knots(1, i + 1) = uh2(i);
42      // Update and record
43      rec(knots);
44      // Abort when max iterations is reached or error is small enough.
45    } while ((++q < itmax) &&
46             (Brachistochrone::L2norm(knots - knots_prev) >
47              std::min<double>(atol, rtol * Brachistochrone::L2norm(knots))));
48
49    // Output a warning if fixed-point iteration was aborted because max
50    // iterations was reached.
51    if (q == itmax)
52      std::cout << "M = " << M
53                << " : Max iterations reached. Truncated with L2 error: "
54                << L2norm(knots - knots_prev) << "\n";
55    return knots;
56  };
```

SOLUTION of (5-2.p):



◁ Semi-logarithmic plot of $k \mapsto \left\| \mathbf{u}_h^{(k)} - \mathbf{u}_h^{(100)} \right\|_{0,h}$.

The error points (except for the last one, which can be regarded as an outlier) are roughly located on a straight line, *cf.* .

We may conjecture linear convergence of the fixed-point iteration.

There is too much noise in the data to give an estimate of the rate of convergence.

C++ code 5.2.54: Implementation of `tabiterr()` ➙ GitLab

```
2  void tabiterr(unsigned int M, double rtol, double atol, unsigned int maxit) {
3    // Start of the (half) cyloid curve (xi = 0.5)
4    Eigen::Vector2d a({0.5 * M_PI - 1.0, -1.0});
5    // Initialize b to agree with endpoint of cycloid curve
6    Eigen::Vector2d b({M_PI, -2.});
7    // Carry out the fixed-point iteration
8    std::vector<Eigen::Matrix<double, 2, Eigen::Dynamic>> iterates;
9    Eigen::Matrix<double, 2, Eigen::Dynamic> mu = brachistochrone(
10       M, a, b, atol, rtol, maxit,
11       [&iterates](const Eigen::Matrix<double, 2, Eigen::Dynamic> &mu) -> void {
12         iterates.push_back(mu);
13       });
14   // Print table with results
15   std::cout << "| k   | ||u_h^(k)-u_h^(" << iterates.size() - 1
16             << ")||_{L2} |\n";
17   std::cout << "-----------------------------------|\n";
18   for (int i = 0; i < iterates.size() - 1; ++i) {
19     std::cout << "| " << std::setw(3) << i << " | " << std::setw(26)
20               << std::setprecision(5) << std::scientific
21               << L2norm(iterates.at(i) - iterates.at(iterates.size() - 1))
22               << " |\n";
23   }
24 }
```

HINT 1 for (5-2.q): [Lecture → § 3.2.2.5] explains how to predict type and speed of convergence from empirical data. ⌟

SOLUTION of (5-2.q):

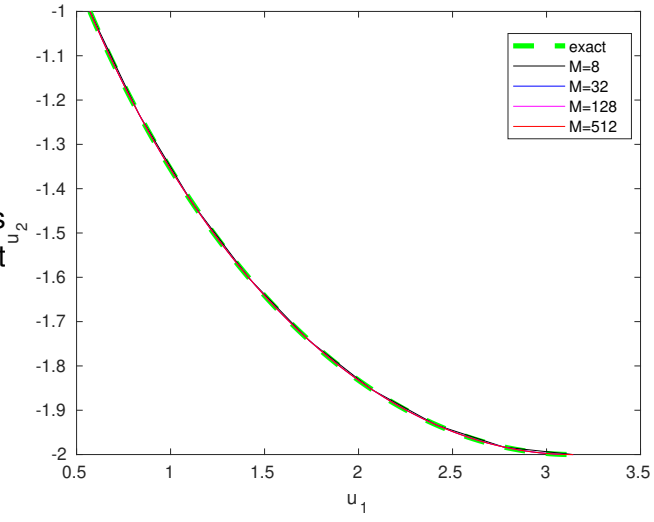
M	$\|\mathbf{u}_h - \mathsf{I}_M\mathbf{u}\|_{0,h}$
8	6.95044e-02
16	8.37491e-02
32	8.78687e-02
64	8.89394e-02
128	8.92902e-02
256	8.92976e-02
512	8.92994e-02

Halving the meshwidth has little impact on the discretization error!

Possible explanations:

- A bug in the code that disrupts convergence
- No proper convergence of the fixed-point
- Failure of the finite-element discretization to converge.

However, there is “visual convergence”. The curves obtained from the finite-element discretization almost lie on top of the exact solution in the plot.



C++ code 5.2.55: Code for `brachistochrone_cvg()` ↬ GitLab

```
void brachistochrone_cvg(std::string filename) {
  // Set minimal number of cells in the mesh
  int M = 8;
  // Start on the cyloid curve (xi = 0.5)
  Eigen::Vector2d a({0.5 * M_PI - 1.0, -1.0});
  // Set b to the endpoint of the cycloid curve
  Eigen::Vector2d b({M_PI, -2.});
  // Initialize parameters for brachistochrone function
  double atol = 1e-6;
  double rtol = 1e-4;
  double itmax = 100;
  std::vector<double> L2errors;
  std::ofstream outfile;
  if (!filename.empty()) {
    outfile.open(filename);
  }

  // Refine the mesh 8 times and compute the associated L2 errors
  for (int j = 0; j < 8; ++j) {
    // Compute the exact solution (i.e.  the cycloid)
    Eigen::Matrix<double, 2, Eigen::Dynamic> mu_cycloid(2, M + 1);
    for (int i = 0; i <= M; ++i) {
      const double xi = 0.5 * (static_cast<double>(i) / M + 1.0);
      mu_cycloid(0, i) = M_PI * xi - std::sin(M_PI * xi);
      mu_cycloid(1, i) = std::cos(M_PI * xi) - 1.;
```

```
    }
    // Solve the brachistochrone problem
    Eigen::Matrix<double, 2, Eigen::Dynamic> mu =
        brachistochrone(M, a, b, atol, rtol, itmax);
    // Compute and store the L2 error
    L2errors.push_back(L2norm(mu - mu_cycloid));
    // Half the mesh width/double the number of cells
    M *= 2;
    if (!filename.empty() && outfile.good()) {
      outfile << "mu{" << j + 1 << "} = [ ";
      for (int k = 0; k < mu.cols(); ++k) {
        outfile << mu(0, k) << " ";
      }
      outfile << ";" << std::endl;
      for (int k = 0; k < mu.cols(); ++k) {
        outfile << mu(1, k) << " ";
      }
      outfile << " ];" << std::endl;
    }
  }

  // Print a table with results.
  std::cout << "| k   | ||u_h-I_M u_h||_{L2} |\n";
  std::cout << "-------------------------|\n";
  for (int i = 0; i < L2errors.size() - 1; ++i) {
    std::cout << "| " << std::setw(3) << 8 * std::pow(2, i) << " | "
              << std::setw(20) << std::setprecision(5) << std::scientific
              << L2errors.at(i) << " |\n";
  }
}
```

SOLUTION of (5-2.r):

In Sub-problem (5-2.d) we found

$$T(\mathbf{u}) = \int_0^1 \sqrt{\frac{(u_1'(\xi))^2 + (u_2'(\xi))^2}{-u_2(\xi)}}\, \mathrm{d}\xi \,. \tag{5.2.57}$$

Now we perform the substitution

$$x = u_1(\xi) \quad \Rightarrow \quad \mathrm{d}x = u_1'(\xi)\mathrm{d}\xi \,, \tag{5.2.58}$$

noting that for $y(x) = u_2(\xi)$

$$\frac{dy}{dx}(x) = u_2'(\xi)\frac{d\xi}{dx} = \frac{u_2'(\xi)}{u_1'(\xi)} \,, \quad x = u_1(\xi) \,. \tag{5.2.59}$$

This transforms the integral (5.2.57) into

$$T(\mathbf{u}) = \int_0^1 \sqrt{\frac{(u_1'(\xi))^2 + (u_2'(\xi))^2}{-u_2(\xi)}}\, \mathrm{d}\xi = \int_{a_1}^{b_1} \sqrt{\frac{(u_1'(\xi))^2 + (u_2'(\xi))^2}{-y(x)}}\, \frac{1}{u_1'(\xi)}\, \mathrm{d}x \,,$$

(5.2.59) ► $T_g(y) = \int_0^1 \sqrt{\frac{1 + (dy/dx)^2}{-y(x)}}\, \mathrm{d}x$ for $y \in C^1_{\mathrm{pw}}([a_1, b_1])$. (5.2.60)

SOLUTION of (5-2.s):

There are two possible approaches:

❶ We directly obtain the variational equations from the minimization problem. As in Sub-problem (5-2.e) we employ the technique of nested Taylor expansion using

$$\sqrt{1+\delta} = 1 + \tfrac{1}{2}\delta + O(\delta^2) \quad , \quad \frac{1}{\sqrt{1+\delta}} = 1 - \tfrac{1}{2}\delta + O(\delta^2) \, ,$$

which yields for $w \in C^1_{\mathrm{pw}}([a_1, b_1])$ and $t \to 0$

$$\sqrt{1 + (y'(x) + tw'(x))^2} = \sqrt{1 + (y'(x))^2}\left(1 + t\,\frac{y'(x)w'(x)}{1 + (y'(x))^2} + O(t^2)\right) ,$$
$$\frac{1}{\sqrt{-y(x) - tw(x)}} = \frac{1}{\sqrt{-y(x)}}\left(1 - \tfrac{1}{2}t\,\frac{w(x)}{y(x)} + O(t^2)\right) .$$

We conclude

$$\frac{\sqrt{1 + (y'(x) + tw'(x))^2}}{\sqrt{-y(x) - tw(x)}} = \frac{\sqrt{1 + (y'(x))^2}}{\sqrt{-y(x)}} + t\left(\frac{y'(x)w'(x)}{\sqrt{1 + (y'(x))^2}} \cdot \frac{1}{\sqrt{-y(x)}} - \frac{1}{2}\,\frac{w(x)\sqrt{1 + (y'(x))^2}}{y(x)\sqrt{-y(x)}}\right) + O(t^2) \, ,$$

and get the **directional derivative**

$$\langle \mathrm{D}T_g(y), w\rangle = \int_{a_1}^{b_1} \frac{y'(x)w'(x)}{\sqrt{1 + (y'(x))^2}} \cdot \frac{1}{\sqrt{-y(x)}} - \frac{1}{2}\,\frac{w(x)\sqrt{1 + (y'(x))^2}}{y(x)\sqrt{-y(x)}}\,\mathrm{d}x \, . \tag{5.2.61}$$

This leads to the variational equation: Find $u \in C^1_{\mathrm{pw}}([a_1, b_1])$, $u(a_1) = 0$, $u(b_1) = b_2$, such that

$$\int_{a_1}^{b_1} \frac{y'(x)w'(x)}{\sqrt{1 + (y'(x))^2}} \cdot \frac{1}{\sqrt{-y(x)}} - \frac{1}{2}\,\frac{\sqrt{1 + (y'(x))^2}\,w(x)}{y(x)\sqrt{-y(x)}}\,\mathrm{d}x = 0 \quad \forall w \in C^1_{\mathrm{pw},0}([a_1, b_1]) \, , \tag{5.2.62}$$

where, abusing the notation, we wrote $y'(x) := \frac{dy}{dx}(x)$, $w'(x) := \frac{dw}{dx}(x)$.

❷ We can apply the transformations (5.2.58) and (5.2.59) to the variational equation obtained in Sub-problem (5-2.e). We start with choosing a test function $\mathbf{v}(\xi) = \begin{bmatrix} 0 \\ v(\xi) \end{bmatrix}$ in (5.2.19):

$$\int_0^1 \left(\frac{u_2'(\xi)v'(\xi)}{\sqrt{-u_2(\xi)}\sqrt{(u_1'(\xi))^2 + (u_2'(\xi))^2}} - \frac{1}{2}\,\frac{\sqrt{(u_1'(\xi))^2 + (u_2'(\xi))^2}\,v(\xi)}{u_2(\xi)\sqrt{-u_2(\xi)}}\right)\mathrm{d}\xi = 0 \tag{5.2.63}$$

for all $v \in C^1_{\mathrm{pw}],0}([0,1])$. Now transform the integral using (5.2.58) and (5.2.59):

$$\int_0^1 \left(\frac{u_2'(\xi) v'(\xi)}{\sqrt{-u_2(\xi)}\sqrt{(u_1'(\xi))^2 + (u_2'(\xi))^2}} - \frac{1}{2} \frac{\sqrt{(u_1'(\xi))^2 + (u_2'(\xi))^2}\, v(\xi)}{u_2(\xi)\sqrt{-u_2(\xi)}} \right) \mathrm{d}\xi$$

$$= \int_{a_1}^{b_1} \left(\frac{u_2'(\xi) v'(\xi)}{\sqrt{-y(x)}\sqrt{(u_1'(\xi))^2 + (u_2'(\xi))^2}} - \frac{1}{2} \frac{\sqrt{(u_1'(\xi))^2 + (u_2'(\xi))^2}\, v(\xi)}{y(x)\sqrt{-y(x)}} \right) \frac{1}{u_1'(\xi)} \mathrm{d}x$$

$$= \int_{a_1}^{b_1} \left(\frac{\frac{u_2'(\xi)}{u_1'(\xi)} \frac{v'(\xi)}{u_1'(\xi)}}{\sqrt{-y(x)}\sqrt{\left(1 + \left(\frac{u_2'(\xi)}{u_1'(\xi)}\right)^2\right)}} - \frac{1}{2} \frac{\sqrt{\left(1 + \left(\frac{u_2'(\xi)}{u_1'(\xi)}\right)^2\right)}\, v(\xi)}{y(x)\sqrt{-y(x)}} \right) \mathrm{d}x$$

$$= \int_{a_1}^{b_1} \left(\frac{y'(x)\, w'(x)}{\sqrt{-y(x)}\sqrt{(1 + (y'(x))^2}} - \frac{1}{2} \frac{\sqrt{(1 + (y'(x))^2}\, w(\xi)}{y(x)\sqrt{-y(x)}} \right) \mathrm{d}x \,.$$

Here we wrote $w(x) := v(\xi) = v(u_1^{-1}(x))$, which implies

$$w'(x) := \frac{dw}{dx}(x) = v'(\xi)\frac{d\xi}{dx} \overset{(5.2.58)}{=} \frac{v'(\xi)}{u_1'(\xi)} \,.$$

This gives the variational equation (5.2.62).

SOLUTION of (5-2.b):

The total travel time is a mapping $T_N : \mathbb{R}^{2N} \to \mathbb{R}$. From (5.2.6) we learn that it is continuously differentiable on

$$D := \left\{ \left[u_1^1, u_2^1, u_1^2, u_2^2, \ldots, u_1^N, u_2^N\right]^\top \in \mathbb{R}^{2N} : \; u_2^j < 0, \; j = 1, \ldots, N \right\}$$

From analysis we know that at a global minimum $\vec{\mathbf{u}}^* \in D$ we have

$$\mathbf{grad}\, T_N(\vec{\boldsymbol{u}}^*) = \mathbf{0} \,.$$

This tells us that $F = \mathbf{grad}\, T_N$ so that the task boils down to computing the gradient of T_N.

In light of the sum formula (5.2.6) we first compute the partial derivatives of the summands from (5.2.5),

$$t_i = t(\mathbf{u}^i, \mathbf{u}^{i+1}) \quad \text{with} \quad t(\mathbf{x}, \mathbf{y}) = \sqrt{\frac{\|\mathbf{x}-\mathbf{y}\|^2}{-\frac{1}{2}(x_2+y_2)}} \,, \quad i = 0, \ldots, N \,.$$

The chain rule combined with the quotient rule gives

$$\frac{\partial t}{\partial \mathbf{x}}(\mathbf{x}, \mathbf{y}) = -\frac{1}{\sqrt{2}} \sqrt{\frac{-(x_2+y_2)}{\|\mathbf{x}-\mathbf{y}\|^2}} \cdot \frac{2(x_2+y_2)(\mathbf{x}-\mathbf{y}) - \|\mathbf{x}-\mathbf{y}\|^2 \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]}{(x_2+y_2)^2} \,,$$

$$\frac{\partial t}{\partial \mathbf{y}}(\mathbf{x}, \mathbf{y}) = \frac{1}{\sqrt{2}} \sqrt{\frac{-(x_2+y_2)}{\|\mathbf{x}-\mathbf{y}\|^2}} \cdot \frac{2(x_2+y_2)(\mathbf{x}-\mathbf{y}) + \|\mathbf{x}-\mathbf{y}\|^2 \left[\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right]}{(x_2+y_2)^2} \,.$$

From (5.2.6) we infer

$$\mathbf{grad}\, T_N(\mathbf{u}^1, \ldots, \mathbf{u}^N) = \begin{bmatrix} \frac{\partial t}{\partial \mathbf{y}}(\boldsymbol{a}, \mathbf{u}^1) + \frac{\partial t}{\partial \mathbf{x}}(\mathbf{u}^1, \mathbf{u}^2) \\ \frac{\partial t}{\partial \mathbf{y}}(\boldsymbol{u}^1, \mathbf{u}^2) + \frac{\partial t}{\partial \mathbf{x}}(\mathbf{u}^2, \mathbf{u}^3) \\ \vdots \\ \frac{\partial t}{\partial \mathbf{y}}(\mathbf{u}^{N-1}, \mathbf{u}^N) + \frac{\partial t}{\partial \mathbf{x}}(\mathbf{u}^N, \boldsymbol{b}) \end{bmatrix} \in \mathbb{R}^{2N} \,. \tag{5.2.8}$$

The resulting non-linear system of $2N$ equations for the $2N$ unknowns $u_1^1, u_2^1, u_1^2, u_2^2, \ldots, u_1^N, u_2^N$ is

$$
\begin{aligned}
&\frac{1}{\sqrt{2}}\sqrt{\frac{-a_2-u_2^1}{\|\mathbf{u}^1\|^2}}\cdot\frac{2(a_2+u_2^1)(\boldsymbol{a}-\mathbf{u}^1)+\left\|\boldsymbol{a}-\mathbf{u}^1\right\|^2\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]}{\left(a_2+u_2^1\right)^2}-\\
&\frac{1}{\sqrt{2}}\sqrt{\frac{-(u_2^1+u_2^2)}{\|\mathbf{u}^1-\mathbf{u}^2\|^2}}\cdot\frac{2(u_2^1+u_2^2)(\mathbf{u}^1-\mathbf{u}^2)-\left\|\mathbf{u}^1-\mathbf{u}^2\right\|^2\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]}{\left(u_2^1+u_2^2\right)^2}=\mathbf{0}\,,\\
&\frac{1}{\sqrt{2}}\sqrt{\frac{-(u_2^{i-1}+u_2^i)}{\|\mathbf{u}^{i-1}-\mathbf{u}^i\|^2}}\cdot\frac{2(u_2^{i-1}+u_2^i)(\mathbf{u}^{i-1}-\boldsymbol{u}^i)+\left\|\mathbf{u}^{i-1}-\mathbf{u}^i\right\|^2\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]}{\left(u_2^{i-1}+u_2^i\right)^2}-\\
&\frac{1}{\sqrt{2}}\sqrt{\frac{-(u_2^i+u_2^{i+1})}{\|\mathbf{u}^i-\mathbf{u}^{i+1}\|^2}}\cdot\frac{2(u_2^i+u_2^{i+1})(\mathbf{u}^i-\mathbf{u}^{i+1})-\left\|\mathbf{u}^i-\mathbf{u}^{i+1}\right\|^2\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]}{\left(u_2^i+u_2^{i+1}\right)^2}=\mathbf{0}\,,\\
&\frac{1}{\sqrt{2}}\sqrt{\frac{-(u_2^{N-1}+u_2^N)}{\|\mathbf{u}^{N-1}-\mathbf{u}^N\|^2}}\cdot\frac{2(u_2^{N-1}+u_2^N)(\mathbf{u}^{N-1}-\boldsymbol{u}^N)+\left\|\mathbf{u}^{N-1}-\mathbf{u}^N\right\|^2\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]}{\left(u_2^{N-1}+u_2^N\right)^2}-\\
&\frac{1}{\sqrt{2}}\sqrt{\frac{-(u_2^N+b_2)}{\|\mathbf{u}^N-\boldsymbol{b}\|^2}}\cdot\frac{2(u_2^N+b_2)(\mathbf{u}^N-\boldsymbol{b})-\left\|\mathbf{u}^N-\boldsymbol{b}\right\|^2\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]}{\left(u_2^N+b_2\right)^2}=\mathbf{0}\,.
\end{aligned}
\tag{5.2.9}
$$

for $i=2,\ldots,N-1$ (N equations in $\mathbb{R}^2$).

HINT 1 for (5-2.d): As in [Lecture → Section 5.1.1.3] Taylor expansions are the key tool. Take the cue from [Lecture → Eq. (5.1.1.29)] concerning the Taylor expansion of the term $\left\|\mathbf{u}(\xi^{i+1}) - \mathbf{u}(\xi^{i})\right\|$. ⌟

SOLUTION of (5-2.d):

As in [Lecture → § 5.1.1.26] we rely on nested Taylor expansions to find the limit

$$T(\mathbf{u}) := \lim_{N\to\infty} \sqrt{2}\sum_{i=0}^{N} \frac{\|\mathbf{u}^{i+1}-\mathbf{u}^i\|}{\sqrt{u_2^{i+1}-u_2^i}} = \lim_{N\to\infty} \sqrt{2}\sum_{i=0}^{N} \frac{\|\mathbf{u}(\xi^{i+1})-\mathbf{u}(\xi^i)\|}{\sqrt{u_2(\xi^{i+1})-u_2(\xi^i)}} ,$$

where we have made use of (5.2.10) already.

For $\xi \mapsto \mathbf{u}(\xi) = \begin{bmatrix} u_1(\xi) \\ u_2(\xi) \end{bmatrix} \in C^2([0,1])$ with derivative $\xi \mapsto \mathbf{u}'(\xi)$, $1 \gg \eta \to 0$, Taylor expansion gives us

$$\begin{aligned} \|\mathbf{u}(\xi+\eta)-\mathbf{u}(\xi-\eta)\| &= \sqrt{(u_1(\xi+\eta)-u_1(\xi-\eta))^2+(u_2(\xi+\eta)-u_2(\xi-\eta))^2} \\ &= \sqrt{(2u_1'(\xi)\eta+O(\eta^2))^2+(2u_2'(\xi)\eta+O(\eta^2))^2} \\ &= 2\eta\|\mathbf{u}(\xi)'\|\sqrt{1+O(\eta)} = 2\eta\|\mathbf{u}'(\xi)\|+O(\eta^2) , \end{aligned} \tag{5.2.11}$$

because

$$\sqrt{1+\tau} = 1+\frac{1}{2}\tau+O(\tau^2) \quad \text{for} \quad \tau\to 0 . \tag{5.2.12}$$

To deal with the denominator we invoke Taylor expansion to see for a C^2-smooth function u

$$\frac{1}{\sqrt{u_2(\xi+\eta)+u_2(\xi-\eta)}} = \frac{1}{\sqrt{2u_2(\xi)+O(\eta^2)}} = \frac{1}{\sqrt{2u_2(\xi)}}(1+O(\eta^2)) , \tag{5.2.13}$$

because $(1+\tau)^{-1/2} = 1-\frac{1}{2}\tau+O(\tau^2)$ for $\tau\to 0$.

We use these formulas with $\eta = \frac{1}{2(N+1)}$ and $\xi = \xi^{i+1/2} := \frac{1}{2}(\xi^{i+1}+\xi^i)$ and get for $N\to\infty$

$$\begin{aligned} \frac{\|\mathbf{u}(\xi^{i+1})-\mathbf{u}(\xi^i)\|}{\sqrt{-u_2(\xi^{i+1})-u_2(\xi^i)}} &= \left(\frac{1}{N+1}\|\mathbf{u}'(\xi)\|+O(N^{-2})\right)\left(\frac{1}{\sqrt{-2u_2(\xi^{i+1/2})}}+O(N^{-2})\right) \\ &= \frac{1}{N+1}\frac{\|\mathbf{u}'(\xi)\|}{\sqrt{-2u_2(\xi^{i+1/2})}}+O(N^{-2}) . \end{aligned}$$

Thus,

$$T_N(\mathbf{u}) = \frac{1}{N+1}\sum_{i=0}^{N}\frac{\|\mathbf{u}'(\xi^{i+1/2})\|}{\sqrt{-u_2(\xi^{i+1/2})}}+O(N^{-2}) \quad \text{for} \quad N\to\infty$$

This is a Riemann-type sum and according to [Lecture → Eq. (5.1.1.31)] for $N\to\infty$ it will converge to

$$T(\mathbf{u}) := \lim_{N\to\infty} T_N(\mathbf{u}) = \int_0^1 \frac{\|\mathbf{u}'(\xi)\|}{\sqrt{-u_2(\xi)}}\,\mathrm{d}\xi . \tag{5.2.14}$$

Remark. If $(\mathbf{u}(0))_2 = 0$ the integrand will have a singularity in $\xi = 0$. However, since $\int_0^1 \xi^{-1/2}\,\mathrm{d}\xi < \infty$, (5.2.14) will exist as an improper integral, provided that $\xi \mapsto \mathbf{u}(\xi)$ is at least Lipschitz continuous.

However, a singularity of the integrand will compound difficulties as regards discretization. Therefore, we will later mainly consider $(\mathbf{u}(0))_2 < 0$.

HINT 1 for (5-2.e): Get inspired by [Lecture → Ex. 5.2.2.1] and use the Taylor expansion of $x \mapsto x^{-\frac{1}{2}}$ for $x < 0$:

$$\frac{1}{\sqrt{-x-th}} = \frac{1}{\sqrt{-x}}\left(1 - \tfrac{1}{2}t\,\frac{h}{x}\right) + O(t^2) \quad \text{for} \quad t \to 0\,. \tag{5.2.16}$$

⌟

SOLUTION of (5-2.e):

Remark. As pointed out above the integral of (5.2.14) will exist as an improper integral for $\mathbf{u} \in \widehat{V}$.

As we have learned in [Lecture → Section 5.2.2] the variational equation is obtained by setting the directional derivative of $u \in \widehat{V} \mapsto T(u)$

$$\langle DT(\mathbf{u}), \mathbf{v} \rangle = \lim_{t \to 0} \frac{T(\mathbf{u} + t\mathbf{v}) - T(\mathbf{u})}{t}, \quad \mathbf{v} \in V_0 := \left(C^1_{\mathrm{pw},0}([0,1]) \right)^2,$$

to zero for all perturbation directions $\mathbf{v} \in V_0$. Note that admissible perturbations $\mathbf{v}$ have to vanish in $\xi = 0, 1$, where $\mathbf{u}$ is known.

Along the lines of (5.2.11) we find for $u, v \in \mathbb{R}^2$

$$\begin{aligned} \|u + tv\| &= \sqrt{\|u\|^2 + 2tu \cdot v + t^2 \|v\|^2} = \|u\| \cdot \sqrt{1 + 2t \frac{u \cdot v}{\|u\|^2} + O(t^2)} \\ &\overset{(5.2.12)}{=} \|u\| \left(1 + t \frac{u \cdot v}{\|u\|^2} + O(t^2) \right) = \|u\| + t \frac{u \cdot v}{\|u\|} + O(t^2) \quad \text{for} \quad t \to 0 . \end{aligned} \tag{5.2.17}$$

Using (5.2.16) from the hint we can expand the integrand for $t \to 0$ (argument ξ suppressed)

$$\begin{aligned} \frac{\|\mathbf{u}' + t\mathbf{v}'\|}{\sqrt{-u_2 - tv_2}} &= \left(\|\mathbf{u}'\| + \frac{\mathbf{u}' \cdot \mathbf{v}'}{\|\mathbf{u}'\|} t + O(t^2) \right) \left(\frac{1}{\sqrt{-u_2}} \left(1 - \frac{tv_2}{2u_2} \right) + O(t^2) \right) \\ &= \frac{\|\mathbf{u}'\|}{\sqrt{-u_2}} + t \left(\frac{\mathbf{u}' \cdot \mathbf{v}'}{\sqrt{-u_2} \|\mathbf{u}'\|} - \frac{v_2 \|\mathbf{u}'\|}{2\sqrt{-u_2} u_2} \right) + O(t^2). \end{aligned}$$

Finally, integrating this expansion from $\xi = 0$ to $\xi = 1$ gives us

$$T(\mathbf{u} + t\mathbf{v}) = T(\mathbf{u}) + t \int_0^1 \left(\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\sqrt{-u_2(\xi)} \|\mathbf{u}'(\xi)\|} - \frac{v_2(\xi) \|\mathbf{u}'(\xi)\|}{2\sqrt{-u_2(\xi)} u_2(\xi)} \right) d\xi + O(t^2) .$$

So

$$\langle DT(\mathbf{u}), \mathbf{v} \rangle = \int_0^1 \left(\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\sqrt{-u_2(\xi)} \|\mathbf{u}'(\xi)\|} - \frac{v_2(\xi) \|\mathbf{u}'(\xi)\|}{2\sqrt{-u_2(\xi)}\, u_2(\xi)} \right) d\xi . \tag{5.2.18}$$

The resulting variational problem reads: Find $\mathbf{u} \in \widehat{V}$ so that

$$a(\mathbf{u}; \mathbf{v}) := \int_0^1 \left(\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\sqrt{-u_2(\xi)} \|\mathbf{u}'(\xi)\|} - \frac{v_2(\xi) \|\mathbf{u}'(\xi)\|}{2\sqrt{-u_2(\xi)}\, u_2(\xi)} \right) d\xi = 0 \quad \forall \mathbf{v} \in V_0 , \tag{5.2.19}$$

where the test space V_0 is

$$V_0 = \left(C^1_{\mathrm{pw},0}([0,1]) \right)^2 := \left\{ \mathbf{v} \in (C^1_{\mathrm{pw}}([0,1]))^2 : \ \mathbf{v}(0) = \mathbf{v}(1) = \mathbf{0} \right\} . \tag{5.2.20}$$

SOLUTION of (5-2.f):

First, note that for $\mathbf{u} \in \widehat{V}$ and $\mathbf{v} \in V_0$ we can rewrite the directional derivative from 5.2.18 in the form

$$\langle \mathrm{D}T(\mathbf{u}), \mathbf{v} \rangle = \int_0^1 \left(\frac{\mathbf{u}'(\xi) \cdot \mathbf{v}'(\xi)}{\sqrt{-u_2(\xi)}\|\mathbf{u}'(\xi)\|} - \frac{1}{2}\begin{bmatrix}0\\1\end{bmatrix} \frac{\|\mathbf{u}'(\xi)\|}{\sqrt{-u_2(\xi)}u_2(\xi)} \cdot \mathbf{v}(\xi) \right) \mathrm{d}\xi \, .$$

Using integration by parts [Lecture → Eq. (1.5.1.8)] to shift the derivative off the test function in the first term we end up with

$$\langle \mathrm{D}T(\mathbf{u}), \mathbf{v} \rangle = - \int_0^1 \left(\frac{d}{d\xi} \left(\frac{\mathbf{u}'}{\sqrt{-u_2}\|\mathbf{u}'\|} \right)(\xi) + \frac{1}{2}\begin{bmatrix}0\\1\end{bmatrix} \frac{\|\mathbf{u}'(\xi)\|}{\sqrt{-u_2(\xi)}u_2(\xi)} \right) \cdot \mathbf{v}(\xi) \; \mathrm{d}\xi.$$

Boundary terms do not come into play, because the test funtions vanishes in $\xi = 0$ and $xi = 1$, *cf.* (5.2.20)..

By [Lecture → Lemma 1.5.1.13], we can now turn this into the second-order ODE

$$\left(\frac{\mathbf{u}'}{\sqrt{-u_2}\|\mathbf{u}'\|} \right)' + \frac{1}{2}\begin{bmatrix}0\\1\end{bmatrix} \frac{\|\mathbf{u}'\|}{\sqrt{-u_2}u_2} = 0 \; . \tag{5.2.21}$$

Remark. Supplementing (5.2.21) with the essential boundary conditions $\mathbf{u}(0) = \boldsymbol{a}$ and $\mathbf{u}(1) = \boldsymbol{b}$ we get the 2-point boundary value problem solved by the brachistochrone curve connecting $\boldsymbol{a}$ and $\boldsymbol{b}$, $(\boldsymbol{b})_2 < 0$.

SOLUTION of (5-2.g):

The derivative of (5.2.22) with respect to the parameter xi is

$$\mathbf{u}'(\xi) = \begin{bmatrix} \pi - \pi\cos(\pi\xi) \\ -\pi\sin(\pi\xi) \end{bmatrix}, \quad 0 \le \xi \le 1\,.$$

Thus, for the cycloid curve $\xi \mapsto \mathbf{u}(\xi)$ from (5.2.22) we compute

$$\begin{aligned} \|\mathbf{u}'(\xi)\|^2 &= (\pi - \pi\cos(\pi\xi))^2 + (\pi\sin\pi\xi)^2 \\ &= \pi^2(1 - 2\cos(\pi\xi) + \cos(\pi\xi)^2 + \sin(\pi\xi)^2) \\ &= \pi^2(2 - 2\cos(\pi\xi)) = -2\pi^2 u_2(\xi)\,, \end{aligned}$$

which implies $\|\mathbf{u}'(\xi)\| = \sqrt{2}\pi\sqrt{-u_2}$. Inserting this converts the ODE (5.2.21) into

$$-\frac{1}{\pi\sqrt{2}}\frac{\mathrm{d}}{\mathrm{d}\xi}\left(\frac{\mathbf{u}'}{u_2}\right) + \frac{\pi}{\sqrt{2}}\begin{bmatrix}0\\1\end{bmatrix}\frac{1}{u_2} = 0. \tag{5.2.23}$$

We now examine the first term, component by component:

$$\frac{d}{d\xi}\left(\frac{u_1'}{u_2}\right) = -\frac{d}{d\xi}\left(\frac{\pi - \pi\cos(\pi\xi)}{1 - \cos(\pi\xi)}\right) = \frac{d}{d\xi}\pi = 0\,.$$

So we see that the ODE holds in the first component. What about the second?

$$\begin{aligned} \frac{d}{d\xi}\left(\frac{u_2'}{u_2}\right) &= \frac{d}{d\xi}\left(\frac{-\pi\sin(\pi\xi)}{\cos(\pi\xi) - 1}\right) = \frac{-\pi^2\cos(\pi\xi)(\cos(\pi\xi) - 1) + \pi\sin(\pi\xi)(-\pi\sin(\pi\xi))}{(\cos(\pi\xi) - 1)^2} \\ &= \frac{-\pi^2\cos(\pi\xi)^2 - \pi^2\sin(\pi\xi)^2 + \pi^2\cos(\pi\xi)}{(\cos(\pi\xi) - 1)^2)} = \pi^2\frac{\cos(\pi\xi) - 1}{(\cos(\pi\xi) - 1)^2} = \frac{\pi^2}{u_2}\,. \end{aligned}$$

So in conclusion we get

$$-\frac{1}{\sqrt{2}\pi}\frac{d}{d\xi}\left(\frac{\mathbf{u}'}{u_2}\right) = -\frac{1}{\sqrt{2}\pi}\begin{bmatrix}0\\1\end{bmatrix}\frac{\pi^2}{u_2} = -\frac{\pi}{\sqrt{2}}\begin{bmatrix}0\\1\end{bmatrix}\frac{1}{u_2}\,.$$

Matching this with (5.2.23) we see that the cycloid curve is a solution of the brachistochrone ODE (5.2.21).

SOLUTION of (5-2.h):

Comparing (5.2.19) and (5.2.24) it is immediate that

$$\sigma(\mathbf{v},\mathbf{g}) := \frac{1}{\sqrt{-v_2}\,\|\mathbf{g}\|} \quad , \quad \mathbf{f}(\mathbf{v},\mathbf{g}) := \frac{\|\mathbf{g}\|}{2\sqrt{-v_2}\,v_2}\begin{bmatrix}0\\1\end{bmatrix} . \tag{5.2.25}$$

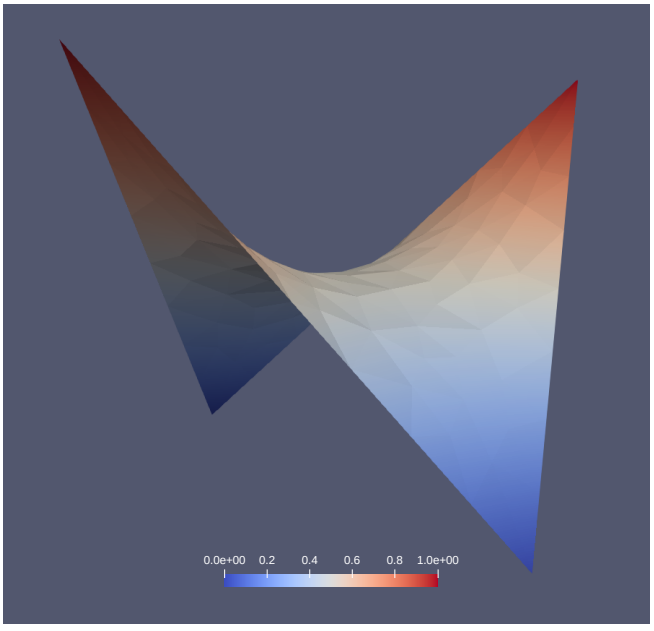
SOLUTION of (5-2.i):

Obviously, (5.2.24) is a *linear* variational equation, if we ignore the dependence of σ and $\mathbf{f}$ on $\mathbf{u}$. Therefore, letting the $\mathbf{u}$-arguments of those coefficient functions "lag behind" in the course of an iteration yields a valid fixed-point iteration in function space. It creates a sequence of approximate solutions $\mathbf{u}^{(k)} \in \widehat{V}$ of (5.2.19) by the recursion

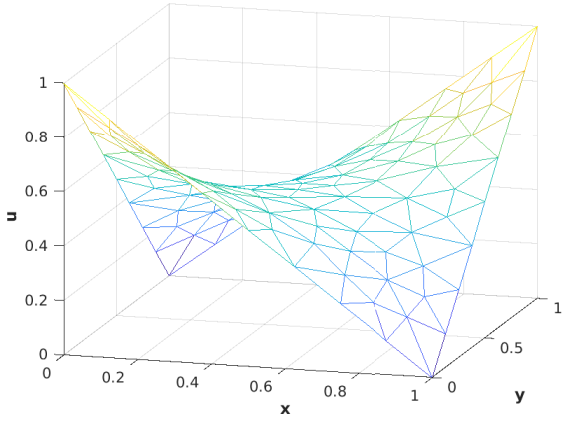
$$\mathbf{u}^{(k+1)} \in \widehat{V}: \quad \int_0^1 \sigma(\mathbf{u}^{(k)}(\xi), \frac{d\mathbf{u}^{(k)}}{d\xi}(\xi)) \frac{\mathbf{u}^{(k+1)}}{d\xi}(\xi) \cdot \mathbf{v}'(\xi)\, \mathrm{d}\xi = \int_0^1 \mathbf{f}(\mathbf{u}^{(k)}(\xi), \frac{d\mathbf{u}^{(k)}}{d\xi}(\xi)) \cdot \mathbf{v}(\xi)\, \mathrm{d}\xi \quad \forall \mathbf{v} \in V_0\,. \tag{5.2.26}$$

The test space V_0 is defined in (5.2.20).

SOLUTION of (5-3.j):



◁ Minimal surface attached to a frame whose shape is given by the boundary data (5.3.18).



C++ code 5.3.19: Implementation of `graphMinSurfVis()` ⇝ GitLab

```
void graphMinSurfVis(std::string meshfile, std::string vtkfile) {
  // Read mesh for unit square from file
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  const lf::io::GmshReader reader(std::move(mesh_factory), meshfile.c_str());
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = reader.mesh();
  // Finite element space
  auto fe_space_p =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
  // Solve non-linear BVP by means of Newton's method
  std::vector<Eigen::VectorXd> iterates{};
  int itcnt = 0;
  Eigen::VectorXd mu = graphMinimalSurface(
      fe_space_p,
      [](Eigen::Vector2d x) -> double { return std::abs(x[0] + x[1] - 1.0); },
      1.0E-4, 1.0E-6, 20,
      [&iterates, &itcnt](const Eigen::VectorXd& mu) -> void {
        iterates.push_back(mu);
        itcnt++;
      }
  );
  // Tabulate progress of iteration
  unsigned int N = mu.size();
  for (int i = 0; i < itcnt; ++i) {
    double mu_norm = iterates[i].norm() / std::sqrt(N);
    std::cout << "k = " << i << ": |mu(" << i << ")| = " << mu_norm;
    if (i > 0) {
      std::cout << ", |correction| = "
                << (iterates[i] - iterates[i - 1]).norm() / std::sqrt(N);
    }
    std::cout << std::endl;
  }
  // Output solution for visualization
```

```
34    lf::io::VtkWriter vtk_writer(mesh_p, vtkfile);
35    // Create a MeshFunction encoding the FE solution
36    lf::fe::MeshFunctionFE<double, double> mf_uh(fe_space_p, mu);
37    // Write nodal data to file
38    vtk_writer.WritePointData("mPoint", mf_uh);
39    std::cout << "\n The finite element solution was written to:" << std::endl;
40    std::cout << ">> " << vtkfile << std::endl;
41    // Supplement:  output for visualization with MATLAB
42    PrepTriMesh::prepTriMesh(fe_space_p, mu, "minsurf");
43    /* MATLAB code for visualization:
44       [x,y,TRI] = minsurf();
45       nodvals_minsurf;
46       trimesh(TRI(:,1:3),x,y,nodevals);
47       xlabel("{\bf x}");
48       ylabel("{\bf y}");
49       zlabel("{\bf u}");
50       print -depsc2 'minsurf.eps';
51     */
52  }
```

SOLUTION of (5-3.b):

Observe that [Lecture → Eq. (5.2.2.29)] fits the class of variational problems of the form

$$u \in \widehat{V} \subset C^1_{\mathrm{pw}}(\Omega: \quad \int_\Omega \mathbf{A}(\boldsymbol{x}, \mathbf{grad}\, u(\boldsymbol{x}))\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + c(u(\boldsymbol{x}))u(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in V_0 \subset C^1_{\mathrm{pw},0}(\Omega\,. \quad \text{[Lecture → Eq. (5.3.2.1)]}$$

In the case of [Lecture → Eq. (5.2.2.29)] we have

- $\mathbf{A}(\boldsymbol{x}, \mathbf{grad}\, u(\boldsymbol{x})) := \dfrac{1}{\sqrt{1 + \|\mathbf{grad}\, u(\boldsymbol{x})\|^2}} \cdot \mathbf{I}$, and
- $c \equiv 0$, $f \equiv 0$.

The sought recursion is the specialization of

$$u^{(k+1)} \in \widehat{V} \subset H^1(\Omega): \quad \int_\Omega \mathbf{A}(\boldsymbol{x}, \mathbf{grad}\, u^{(k)}(\boldsymbol{x}))\, \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + c(u^{(k)}(\boldsymbol{x}))u^{(k+1)}(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in V_0 \subset H^1(\Omega)\,, \quad \text{[Lecture → Eq. (5.3.2.3)]}$$

and reads: given $u^{(k)}$ find $u^{(k+1)} \in C^1_{\mathrm{pw}}(\Omega)$, $u^{(k+1)}\big|_{\partial\Omega} = d$, such that

$$\int_\Omega \frac{\mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1 + \|\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\|_2^2}}\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\,. \tag{5.3.1}$$

Note that (5.3.1) is a *linear* variational equation.

SOLUTION of (5-3.d):

In this case we face

$$\mathsf{a}(u;v) = \int_\Omega \frac{\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1+\|\mathbf{grad}\, u(\boldsymbol{x})\|_2^2}}\,\mathrm{d}\boldsymbol{x}\,, \quad u \in C^1_{\mathrm{pw}}(\overline{\Omega}),\ v \in V_0 := C^1_{\mathrm{pw},0}(\overline{\Omega})\,. \tag{5.3.2}$$

We have to compute the directional derivative

$$\mathsf{D}_u\mathsf{a}(u^{(k)};v)w = \lim_{t\to 0} \frac{\mathsf{a}(u+tw;v)-\mathsf{a}(u;v)}{t}\,, \quad u^{(k)} \in \widehat{V}\,, \quad v,w \in V_0\,. \tag{5.3.3}$$

As in [Lecture → Ex. 5.3.2.13] we rely on the tool of nested Taylor expansion (in t) applied to the integrand of

$$\mathsf{a}(u+tw;v) = \int_\Omega \frac{(\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, w(\boldsymbol{x}))\cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1+\|\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, w(\boldsymbol{x})\|_2^2}}\,\mathrm{d}\boldsymbol{x}\,. \tag{5.3.4}$$

We use $\sqrt{1+\delta} = 1+\frac{1}{2}\delta + O(\delta^2)$ for $\delta \to 0$ and start with the expansion

$$\begin{aligned}\frac{1}{\sqrt{\xi+t\eta}} &= \frac{1}{\sqrt{\xi}}\cdot\frac{1}{\sqrt{1+t\frac{\eta}{\xi}}}\\ &= \frac{1}{\sqrt{\xi}}\cdot\frac{1}{1+\frac{1}{2}t\frac{\eta}{\xi}+O(t^2)}\\ &= \frac{1}{\sqrt{\xi}}\left(1-\frac{1}{2}t\frac{\eta}{\xi}+O(t^2)\right) \quad \text{for} \quad t\to 0\,,\end{aligned}$$

and use it to expand

$$\frac{\alpha+t\beta}{\sqrt{\xi+t\eta}} = \frac{1}{\sqrt{\xi}}(\alpha+t\beta)\left(1-\frac{1}{2}t\frac{\eta}{\xi}+O(t^2)\right) = \frac{\alpha}{\sqrt{\xi}} + t\left(\frac{\beta}{\sqrt{\xi}} - \frac{\alpha\eta}{2\xi^{3/2}}\right)+O(t^2) \quad \text{for} \quad t\to 0\,.$$

Since $1+\|\mathbf{grad}\, u + t\,\mathbf{grad}\, w\|_2^2 = 1+\|\mathbf{grad}\, u\|^2 + 2t\,\mathbf{grad}\, u\cdot\mathbf{grad}\, w + O(t^2)$ for $t\to 0$, we can apply the previous expansion with

$$\begin{aligned}\alpha &:= \mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x})\,, \quad \beta := \mathbf{grad}\, w(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x})\,,\\ \xi &:= 1+\|\mathbf{grad}\, u(\boldsymbol{x})\|^2\,, \quad \eta := 2\,\mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, w(\boldsymbol{x}) + O(t)\,,\end{aligned}$$

and get

$$\begin{aligned}&\frac{(\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, w(\boldsymbol{x}))\cdot\mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1+\|\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, w(\boldsymbol{x})\|_2^2}}\\ &\quad = \frac{\mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1+\|\mathbf{grad}\, u(\boldsymbol{x})\|^2}} + t\left(\frac{\mathbf{grad}\, w(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1+\|\mathbf{grad}\, u(\boldsymbol{x})\|^2}} - \right.\\ &\qquad\qquad \left.\frac{(\mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x}))(\mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, w(\boldsymbol{x}))}{(1+\|\mathbf{grad}\, u(\boldsymbol{x})\|^2)^{3/2}}\right)\end{aligned}$$

$$+ O(t^2) \quad \text{for} \quad t \to 0$$

From this we conclude

$$\lim_{t\to 0} \frac{\mathsf{a}(u+tw;v) - \mathsf{a}(u;v)}{t} = \left(\frac{\mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1 + \|\mathbf{grad}\, u(\boldsymbol{x})\|^2}} - \frac{(\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}))(\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, w(\boldsymbol{x}))}{(1 + \|\mathbf{grad}\, u(\boldsymbol{x})\|^2)^{3/2}} \right) .$$

This leads to the following *linear* variational equation for the Newton correction in the $k+1$-th step of the Newton iteration: Seek $w \in V_0$ such that

$$\int_\Omega \frac{\mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1 + \|\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\|^2}} - \frac{(\mathbf{grad}\, u^{(k)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}))(\mathbf{grad}\, u^{(k)}(\boldsymbol{x}) \cdot \mathbf{grad}\, w(\boldsymbol{x}))}{(1 + \|\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\|^2)^{3/2}} \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \frac{\mathbf{grad}\, u^{(k)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})}{\sqrt{1 + \|\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\|_2^2}} \,\mathrm{d}\boldsymbol{x} \quad \forall v \in V_0 . \tag{5.3.5}$$

Then, from [Lecture → Eq. (5.3.2.5)] we learn that the next iterate is obtained from

$$u^{(k+1)} := u^{(k)} + w .$$

SOLUTION of (5-3.e):

The starting point is the variational equation (5.3.5). Let us first tackle the left-hand side bilinear form

$$(w,v) \mapsto \int_\Omega \frac{\mathbf{grad}\, w(x) \cdot \mathbf{grad}\, v(x)}{\sqrt{1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2}} - \frac{(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, v(x))(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, w(x))}{(1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2)^{3/2}}\,\mathrm{d}x\,.$$

Notice that owing to the associativity of matrix multiplication

$$\begin{aligned}(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, v(x))(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, w(x)) &\\ = \mathbf{grad}\, u^{(k)}(x)(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, w(x)) \cdot \mathbf{grad}\, v(x) &\\ = \underbrace{\left(\mathbf{grad}\, u^{(k)}(x)\left(\mathbf{grad}\, u^{(k)}\right)^\top\right)}_{\in \mathbb{R}^{2,2}} \mathbf{grad}\, w(x) \cdot \mathbf{grad}\, v(x)\,.&\end{aligned}$$

As a consequence

$$\begin{aligned}\int_\Omega \frac{\mathbf{grad}\, w(x) \cdot \mathbf{grad}\, v(x)}{\sqrt{1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2}} - \frac{(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, v(x))(\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, w(x))}{(1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2)^{3/2}}\,\mathrm{d}x &\\ = \int_\Omega \mathbf{A}(x)\,\mathbf{grad}\, w(x) \cdot \mathbf{grad}\, v(x)\,\mathrm{d}x &\end{aligned}$$

with

$$\mathbf{A}(x) = \frac{1}{\sqrt{1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2}}\left(\mathbf{I}_2 - \frac{\mathbf{grad}\, u^{(k)}(x)\left(\mathbf{grad}\, u^{(k)}\right)^\top}{1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2}\right), \quad x \in \Omega\,. \tag{5.3.7}$$

The right-hand side linear form of the linearized variational problem (5.3.5) is

$$v \mapsto -\int_\Omega \frac{\mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, v(x)}{\sqrt{1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|_2^2}}\,\mathrm{d}x\,,$$

from which it is obvious that it will match the right-hand side of (5.3.6), if we choose

$$c(x) := -\frac{1}{\sqrt{1+\left\|\mathbf{grad}\, u^{(k)}(x)\right\|^2}}, \quad x \in \Omega\,. \tag{5.3.8}$$

HINT 1 for (5-3.f): You may use LEHRFEM++ **MeshFunction** objects (→ [Lecture → Suppl. 2.8.3.32]), the auxiliary functions `lf::fe::IntegrateMeshFunction()`, `lf::mesh::utils::squaredN`

⌟

SOLUTION of (5-3.f):

The easiest implementation relies on high-level LEHRFEM++ facilities

- Wrap the coefficient vector `mu_vec` into a **lf::fe::MeshFunctionGradFE** object.
- Use `lf::mesh::utils::squaredNorm()` to create a mesh function object representing $\|\mathbf{gra}$
- Based on that object implement your own **MeshFunction** class that realizes $\sqrt{1+\|\mathbf{grad}\, u_h\|^2}$. Alternatively, a lambda function could serve the same purpose.
- Finally invoke `lf::fe::IntegrateMeshFunction()` to perform local numerical quadrature on $\mathcal{M}$.

C++ code 5.3.11: Implementation of `computeGraphArea()` ↪ GitLab

```
2  double computeGraphArea(
3      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
4      const Eigen::VectorXd& mu_vec) {
5    double area;
6    // Create a MeshFunction representing the gradient
7    lf::fe::MeshFunctionGradFE<double, double> graduh(fes_p, mu_vec);
8    // A lambda function realizing a MeshFunction sqrt(1+||grad u_h||^2)
9    auto integrand = [&graduh](
10                       const lf::mesh::Entity& e,
11                       const Eigen::MatrixXd& refc) -> std::vector<double> {
12     const std::vector<Eigen::VectorXd> gradvals{graduh(e, refc)};
13     std::vector<double> ret(gradvals.size());
14     for (int i = 0; i < gradvals.size(); ++i) {
15       ret[i] = std::sqrt(1.0 + gradvals[i].squaredNorm());
16     }
17     return ret;
18   };
19   const unsigned int quad_ord = 2;
20   area = lf::fe::IntegrateMeshFunction(*fes_p->Mesh(), integrand, quad_ord);
21   return area;
22 }
```

SOLUTION of (5-3.g):

Once you are familiar with the concept of a **MeshFunction** as used in LEHRFEM++, know how to use **lf::fe::MeshFunctionGradFE** objects, and given the concrete formulas (5.3.7) and (5.3.8), the implementation is straightforward.

C++ code 5.3.12: Implementation of constructor and evaluation operator of class CoeffTensorA. ↪ GitLab

```
CoeffTensorA::CoeffTensorA(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    const Eigen::VectorXd& mu)
    : graduh_(fes_p, mu) {}
```

```
std::vector<Eigen::Matrix2d> CoeffTensorA::operator()(
    const lf::mesh::Entity& e, const Eigen::MatrixXd& refc) const {
  // Number of points for which evaluation is requested
  const int nvals = refc.cols();
  // For returning values
  std::vector<Eigen::Matrix2d> Avals(nvals);
  // Gradients of FE function in those points
  const std::vector<Eigen::VectorXd> gradvals{graduh_(e, refc)};
  LF_ASSERT_MSG(gradvals.size() == nvals, "Wrong number of gradients");
  // Compute tensor A at all input locations
  for (int i = 0; i < nvals; ++i) {
    const Eigen::Vector2d g{gradvals[i]};
    const double norms_g = g.squaredNorm();
    Avals[i] =
        1.0 / std::sqrt(1.0 + norms_g) *
        (Eigen::Matrix2d::Identity() - g * g.transpose() / (1.0 + norms_g));
  }
  return Avals;
}
```

C++ code 5.3.13: Implementation of constructor and evaluation operator of class CoeffScalarc ↪ GitLab

```
CoeffScalarc::CoeffScalarc(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    const Eigen::VectorXd& mu)
    : graduh_(fes_p, mu) {}
```

```
std::vector<double> CoeffScalarc::operator()(
    const lf::mesh::Entity& e, const Eigen::MatrixXd& refc) const {
  // Number of points for which evaluation is requested
  const int nvals = refc.cols();
  // For returning values
  std::vector<double> cvals(nvals);
  // Gradients of FE function in those points
  const std::vector<Eigen::VectorXd> gradvals{graduh_(e, refc)};
  LF_ASSERT_MSG(gradvals.size() == nvals, "Wrong number of gradients");
  // Compute coefficient c for all input points
  for (int i = 0; i < nvals; ++i) {
    cvals[i] = -1.0 / std::sqrt(1.0 + gradvals[i].squaredNorm());
```

```
14      }
15      return cvals;
16  }
```

HINT 1 for (5-3.h): The LEHRFEM++ class **lf::uscalfe::ReactionDiffusionElementMatrixProvider** is an **ENTITY_MATRIX_PROVIDER** and can be used together with the classes **CoeffTensorA** and **CoeffScalarc** implemented in Sub-problem (5-3.g). ⌟

HINT 2 for (5-3.h): To impose zero Dirichlet boundary conditions use `lf::assemble::FixFlagged` see [Lecture → § 2.7.6.13]. ⌟

HINT 3 for (5-3.h): To compute the right-hand side vector $\vec{\boldsymbol{\varphi}}$ for the linear system of equations arising from (5.3.14)

1. first assemble the $\mathcal{S}_1^0(\mathcal{M})$ Galerkin matrix $\mathbf{T}$ for the bilinear form

$$(u, v) \mapsto \int_\Omega c(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\ ,$$

2. then compute $\vec{\boldsymbol{\varphi}} := \mathbf{T}\vec{\boldsymbol{\mu}}$, where $\vec{\boldsymbol{\mu}}$ is the basis expansion coefficient vector of $u_h^{(k)} \in \mathcal{S}_1^0(\mathcal{M})$.

Note that coefficient vectors invariably refer to the full finite element space $\mathcal{S}_1^0(\mathcal{M})$ and not $\mathcal{S}_{1,0}^0(\mathcal{M})$. So their lengths always agree with $\dim \mathcal{S}_1^0(\mathcal{M})$. ⌟

SOLUTION of (5-3.h):

C++ code 5.3.15: Implementation of `computeNewtonCorrection()` ⇝ GitLab

```
Eigen::VectorXd computeNewtonCorrection(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    const Eigen::VectorXd& mu_vec) {
  // Obtain reference to the underlying finite element mesh
  const lf::mesh::Mesh& mesh{*fes_p->Mesh()};
  // The local-to-global index mapping
  const lf::assemble::DofHandler& dofh{fes_p->LocGlobMap()};
  // Get the number of degrees of freedom = dimension of FE space
  const lf::base::size_type N_dofs(dofh.NumDofs());
  LF_ASSERT_MSG(mu_vec.size() == N_dofs, "Vector length mismatch!");
  // Solution vector = return value
  Eigen::VectorXd sol_vec(N_dofs);
  // Set up an empty sparse matrix to hold the Galerkin matrix
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Initialize ELEMENT_MATRIX_PROVIDER object
  // Tensor coefficient provided by auxiliary MeshFunction object
  CoeffTensorA mf_alpha(fes_p, mu_vec);
  // No zero-order term
  lf::mesh::utils::MeshFunctionConstant<double> mf_zero(0.0);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider elmat_builder(
      fes_p, std::move(mf_alpha), mf_zero);
  // Cell-oriented assembly over the whole computational domain
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
  // Assembly of right-hand-side vector
  Eigen::VectorXd phi(N_dofs);
  {
    // Auxliary sparse matrix
    lf::assemble::COOMatrix<double> T(N_dofs, N_dofs);
    // Scalar coefficient c(x) provided by auxiliary MeshFunction object
    CoeffScalarc mf_c(fes_p, mu_vec);
    lf::uscalfe::ReactionDiffusionElementMatrixProvider Tmat_builder(
        fes_p, std::move(mf_c), mf_zero);
    // Cell-oriented assembly over the whole computational domain
    lf::assemble::AssembleMatrixLocally(0, dofh, dofh, Tmat_builder, T);
    phi = T.MatVecMult(1.0, mu_vec);
  }
  // Enforce zero Dirichlet boundary conditions
  // Obtain an array of boolean flags for the edges of the mesh, 'true'
  // indicates that the edge lies on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
  // Elimination of degrees of freedom on the boundary
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&bd_flags,
       &dofh](lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
        const lf::mesh::Entity& node{dofh.Entity(gdof_idx)};
        return (bd_flags(node) ? std::make_pair(true, 0.0)
                               : std::make_pair(false, 0.0));
      },
      A, phi);
  // Assembly completed: Convert COO matrix A into CRS format using Eigen's
  // internal conversion routines.
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();
```

```
54
55    // Solve linear system using Eigen's sparse direct elimination
56    // Examine return status of solver in case the matrix is singular
57    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
58    solver.compute(A_crs);
59    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
60    sol_vec = solver.solve(phi);
61    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
62    return sol_vec;
63  }
```

SOLUTION of (5-3.i):

C++ code 5.3.17: Implementation of `graphMinimalSurface()` ⇝ GitLab

```
template <typename D_FUNCTOR,
          typename RECORDER = std::function<void(const Eigen::VectorXd&)>>
Eigen::VectorXd graphMinimalSurface(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    D_FUNCTOR&& boundary_data, double rtol, double atol, unsigned int itmax,
    RECORDER&& rec = [](const Eigen::VectorXd&) -> void {}) {
  // Obtain reference to the underlying finite element mesh
  const lf::mesh::Mesh& mesh{*fes_p->Mesh()};
  // The local-to-global index mapping
  const lf::assemble::DofHandler& dofh{fes_p->LocGlobMap()};
  // Get the number of degrees of freedom = dimension of FE space
  const lf::base::size_type N_dofs(dofh.NumDofs());

  // I. Solve Dirichlet problem for -Δu = 0 in order to obtain a good
  // initial guess
  Eigen::VectorXd uh(N_dofs);
  {
    // Set up an empty sparse matrix to hold the Galerkin matrix
    lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
    lf::mesh::utils::MeshFunctionConstant<double> mf_one(1.0);
    lf::mesh::utils::MeshFunctionConstant<double> mf_zero(0.0);
    lf::uscalfe::ReactionDiffusionElementMatrixProvider elmat_builder(
        fes_p, mf_one, mf_zero);
    lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
    Eigen::VectorXd phi(N_dofs);
    phi.setZero();
    // Impose Dirichlet boundary conditions
    auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
    // Set fixed values for d.o.f.  on the boundary
    const Eigen::Matrix<double, 0, 1> v_zero;
    lf::assemble::FixFlaggedSolutionComponents<double>(
        [&bd_flags, &v_zero, &boundary_data,
         &dofh](lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
          const lf::mesh::Entity& node{dofh.Entity(gdof_idx)};
          if (bd_flags(node)) {
            // Get location of the node on the boundary
            const lf::geometry::Geometry* geo = node.Geometry();
            // Query node coordinates
            Eigen::MatrixXd pos = geo->Global(v_zero);
            // Fecth boundary value in current node
            const double val = boundary_data(pos.col(0));
            return std::make_pair(true, val);
          }
          return std::make_pair(false, 0.0);
        },
        A, phi);
    // Solve Galerkin LSE
    Eigen::SparseMatrix<double> A_crs = A.makeSparse();
    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
    solver.compute(A_crs);
    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
    uh = solver.solve(phi);
```

```
54      LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
55    }
56    // II. Newton iteration with correction based termination
57    double update_norm;       // Norm of Newton correction
58    unsigned int it_cnt = 0;  // Iteration counter
59    rec(uh);
60    do {
61      it_cnt++;
62      // Compute Newton correction
63      const Eigen::VectorXd corr = computeNewtonCorrection(fes_p, uh);
64      update_norm = corr.norm() / std::sqrt(N_dofs);
65      uh += corr;  // Update approximate solution
66      rec(uh);     // Register updated solution
67      // Correction-based termination
68    } while ((update_norm > rtol * uh.norm() / std::sqrt(N_dofs)) &&
69             (update_norm > atol) && (it_cnt < itmax));
70    return uh;
71  }
```

SOLUTION of (5-4.a):

The correct solution is

$$J(v) = \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, v(\boldsymbol{x})\|^2 + \cosh(v(\boldsymbol{x})) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \;, \quad v \in C^1_{\mathrm{pw},0}(\overline{\Omega}) \;.$$

❶ We can directly verify this

$$\begin{aligned}\frac{J(u+tv)-J(u)}{t} &= \frac{1}{t}\Big(\int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, v(\boldsymbol{x})\|^2 + \cosh(u(\boldsymbol{x}) + tv(\boldsymbol{x})) - f(\boldsymbol{x})(u(\boldsymbol{x}) + tv(\boldsymbol{x}))\\ &\qquad \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, u(\boldsymbol{x})\|^2 - \cosh(u(\boldsymbol{x})) - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\Big)\\ &= \frac{1}{t}\int_\Omega t\ \mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x}) + t\ \sinh(u(\boldsymbol{x}))v(\boldsymbol{x}) - t\, f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + O(t)\;,\end{aligned}$$

where we use the Taylor expansions for $t \to 0$

$$\begin{aligned}\|\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, v(\boldsymbol{x})\|^2 &= \|\mathbf{grad}\, u(\boldsymbol{x})\|^2 + 2t\,\mathbf{grad}\, u(\boldsymbol{x})\cdot\mathbf{grad}\, v(\boldsymbol{x}) + O(t^2)\;,\\ \cosh(u(\boldsymbol{x}) + tv(\boldsymbol{x})) &= \cosh(u(\boldsymbol{x})) + t\sinh(u(\boldsymbol{x}))v(\boldsymbol{x}) + O(t^2)\;.\end{aligned}$$

❷ We can also appeal to the results of [Lecture → Ex. 5.2.2.6], because we have

$$J(u) = \int_\Omega F(\boldsymbol{x}, \mathbf{grad}\, u(\boldsymbol{x}), u(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} \quad \text{with} \quad F(\boldsymbol{x}, \mathbf{g}, v) = \tfrac{1}{2}\|\mathbf{g}\|^2 + \cosh(v) - f(\boldsymbol{x})v\;.$$

Then we can apply [Lecture → Eq. (5.2.2.10)].

SOLUTION of (5-4.b):

In principle a variational problem gives rise to a valid fixed-point iteration for (5.4.1), if, setting $u := u^{(k+1)} =$ recovers a solution u of (5.4.1). This property is called **consistency**. It holds for the following **two** linear variational problems in the above list:

$$\int_\Omega \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k)})v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\ ,$$

$$\int_\Omega \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \frac{\sinh(u^{(k)})}{u^{(k)}(\boldsymbol{x})}\, u^{(k+1)}(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\ .$$

These are the correct solutions.

SOLUTION of (5-4.c):

The non-linear variational problem (5.4.1) can be written in "canonical form" [Lecture → Def. 5.2.2.20]

$$\boxed{u \in \widehat{V}: \quad \mathsf{a}(u;v) = 0 \quad \forall v \in V_0} \,, \qquad \text{[Lecture → Eq. (5.2.2.21)]}$$

with $V_0 := C^1_{\mathrm{pw},0}(\overline{\Omega})$ and

$$\mathsf{a}(u;v) := \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,, \quad u, v \in V_0 \,. \tag{5.4.2}$$

The variational equation to be solved to obtain the Newton update (in function space) is

$$w \in V_0: \quad \mathsf{D}_u \mathsf{a}(u^{(k-1)};v)w = -\mathsf{a}(u^{(k-1)};v) \quad \forall v \in V_0 \,. \qquad \text{[Lecture → Eq. (5.3.2.9)]}$$

So we have to compute the directional derivative $w \mapsto \mathsf{D}_u \mathsf{a}(u^{(k)};v)w$ of a with respect to u. Concerning a we can split it into a linear and non-linear part

$$\mathsf{a}(u;v) := \underbrace{\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}}_{\text{affine linear in } u} + \underbrace{\int_\Omega \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}}_{\text{non-linear in } u} \,. \tag{5.4.3}$$

The derivative of the linear part "is itself". The directional derivative of the non-linear part can be computed as follows:

$$\lim_{t\to 0} \frac{1}{t} \left(\int_\Omega \sinh(u(\boldsymbol{x}) + t w(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} - \int_\Omega \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right) = \int_\Omega \cosh(u(\boldsymbol{x})) w(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,,$$

where we used the Taylor expansion

$$\sinh(u(\boldsymbol{x}) + t w(\boldsymbol{x})) = \sinh(u(\boldsymbol{x})) + t \sinh'(u(\boldsymbol{x})) w(\boldsymbol{x}) + O(t^2) \,.$$

Summing up we have

$$\mathsf{D}_u \mathsf{a}(u^{(k)};v)w = \int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_\Omega \cosh(u(\boldsymbol{x})) w(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$$

and the update equation [Lecture → Eq. (5.3.2.9)] becomes: seek $w \in C^1_{\mathrm{pw},0}(\overline{\Omega})$ such that

$$\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k-1)}(\boldsymbol{x}))\, w(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \mathbf{grad}\, u^{(k-1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k-1)}(\boldsymbol{x})) v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$$

for all $v \in C^1_{\mathrm{pw},0}(\overline{\Omega})$.

The third option **(C)** is the only correct one.

HINT 1 for (5-5.): The class **lf::uscalfe::ReactionDiffusionElementMatrixProvider** is compatible with the concept of **ENTITY_MATRIX_PROVIDER** [Lecture → Ex. 2.7.4.22]. It is supposed to compute the element matrices for Lagrangian finite elements on a hybrid 2D mesh and the bilinear form

$$(u,v) \mapsto \int_\Omega \alpha(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + \gamma(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1(\Omega) \tag{5.5.3}$$

The coefficient functions $\boldsymbol{x} \mapsto \alpha(\boldsymbol{x})$ and $\boldsymbol{x} \mapsto \gamma(\boldsymbol{x})$ have to be passed to the constructor of an **lf::uscalfe::R** object as **MeshFunction**s:

```
ReactionDiffusionElementMatrixProvider(
    std::shared_ptr<const UniformScalarFESpace<SCALAR>> fe_space,
    DIFF_COEFF alpha, REACTION_COEFF gamma);
```

HINT 2 for (5-5.): Recall that the C++ expression `(<expr>)?a:b` evaluates to `a`, if `<expr> == true`, otherwise to `b`.

SOLUTION of (5-5.a):

I. To begin with, we have to understand the purpose of **FunctionMFWrapper**. Writing

$$\operatorname{sinhc}(\xi) := \begin{cases} \frac{\sinh(\xi)}{\xi} & \text{for} \quad \xi \neq 0\,, \\ 1 & \text{for} \quad \xi = 0\,, \end{cases} \tag{5.5.5}$$

which is a *smooth* function $\mathbb{R} \to \mathbb{R}$, we see that the object constructed by

```
const FunctionMFWrapper mf_coeff(mf_uh_prev, [](double xi) -> double {
  return (xi == 0.0) ? 1.0 : std::sinh(xi) / xi;
});
```

from a **MeshFunction** object `mf` satisfies

$$\texttt{mf_coeff}(\texttt{K}, \widehat{\boldsymbol{x}}) = \operatorname{sinhc}(\texttt{mf}(\texttt{K}, \widehat{\boldsymbol{x}})) \quad \forall \widehat{\boldsymbol{x}} \in \widehat{K}\,,$$

where `K` is a **lf::mesh::Entity** object for a cell of the mesh and $\widehat{K}$ the corresponding reference element.

II. To analyze the function `fixedPointNextIt()` we write $u_h^{(k)} \in \mathcal{S}_1^0(\mathcal{M})$ for the function whose basis expansion coefficients are passed in `mu_vec`. Then `A` will be the $\mathcal{S}_1^0(\mathcal{M})$ finite-element Galerkin matrix for the bilinear form (5.5.3) with

- $\alpha \equiv 1$, and
- $\gamma(\boldsymbol{x}) = \operatorname{sinhc}(u_h^{(k)}(\boldsymbol{x})) = \frac{\sinh(u_h^{(k)}(\boldsymbol{x}))}{u_h^{(k)}(\boldsymbol{x})}$.

Within `fixedPointNextIt()` zero Dirichlet boundary conditions are imposed on `A` as explained in [Lecture → § 2.7.6.13].

III. Examining `solveSemilinearBVP()` reveals that the vector `rhs_vec` is the $\mathcal{S}_1^0(\mathcal{M})$ finite-element load vector induced by the linear functional $v \mapsto \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$. That load vector is built once in `solveSemilinearBVP()` and then passed to `fixedPointNextIt()`, where its components corresponding to d.o.f.s located on $\partial\Omega$ are set to zero.

From these insights we conclude that `fixedPointNextIt()` overwrites `mu_vec` with the basis expansion coefficient vector of $u_h^{(k+1)} \in \mathcal{S}_{1,0}^0(\mathcal{M})$, which solves

$$\int_\Omega \mathbf{grad}\, u_h^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x}) + \frac{\sinh(u_h^{(k)}(\boldsymbol{x}))}{u_h^{(k)}(\boldsymbol{x})} u_h^{(k+1)}(\boldsymbol{x}) v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \tag{5.5.6}$$

for all $v_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$.

Setting $u_h^{(k+1)} = u_h^{(k)} =: u_h$, that is, assuming that the fixed-point iteration has converged, and carrying out the obvious cancellation yields the sought discrete variational problem. Seek $u_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$ such that

$$\int_\Omega \mathbf{grad}\, u_h(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x}) + \sinh(u_h(\boldsymbol{x})) v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$

for all $v_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$.

HINT 1 for (5-5.b): Recall that for all $\xi \in \mathbb{R}$

$$\sinh(\xi) = \tfrac{1}{2}(e^{\xi} - e^{-\xi}) \quad , \quad \sinh'(\xi) = \cosh(\xi) = \tfrac{1}{2}(e^{\xi} + e^{-\xi}) \quad , \quad \cosh'(\xi) = \sinh(\xi)\,. \tag{5.5.7}$$

⌟

SOLUTION of (5-5.b):

From the solution of Sub-problem (5-5.a) we immediately see that $V_h = \mathcal{S}^0_{1,0}(\mathcal{M})$. In order to determine $J : V_h \to \mathbb{R}$ we have to pursue the considerations of [Lecture → Section 5.2.1] in reverse. Writing

$$\mathsf{a}(u_h; v) := \int_\Omega \mathbf{grad}\, u_h(\boldsymbol{x}) \cdot \mathbf{grad}\, v_h(\boldsymbol{x}) + \sinh(u_h(\boldsymbol{x}))v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} - \int_\Omega f(\boldsymbol{x})v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u_h, v_h \in \mathcal{S}^0_{1,0}(\mathcal{M})$$

we look for $J : \mathcal{S}^0_{1,0}(\mathcal{M}) \to \mathbb{R}$ such that

$$\langle \mathrm{D}J(u_h), v_h\rangle = \mathsf{a}(u_h; v_h) \quad \forall u_h, v_h \in \mathcal{S}^0_{1,0}(\mathcal{M})\,.$$

Remark. It does not matter that we deal with finite-dimensional spaces now. The considerations of [Lecture → Section 5.2.1] apply to all kinds of function spaces.

From the hint we learn that cosh is the principal of sinh. Moreover, if the sinh-part was missing, we know from [Lecture → Section 1.4.2] that in this case we would have $J(v_h) = \frac{1}{2}\int_\Omega \|\mathbf{grad}\, v_h(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x}$. So we guess

$$J(v_h) = \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, v_h(\boldsymbol{x})\|^2 + \cosh(v_h(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x}\,, \quad v_h \in \mathcal{S}^0_{1,0}(\mathcal{M})\,. \tag{5.5.8}$$

This result can be validated in two ways (for the sake of brevity we drop the subscript h):

❶ We can directly verify (5.5.8)

$$\begin{aligned}\frac{J(u+tv)-J(u)}{t} = \\ \frac{1}{t}\Big(\int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, v(\boldsymbol{x})\|^2 + \cosh(u(\boldsymbol{x}) + tv(\boldsymbol{x})) - f(\boldsymbol{x})(u(\boldsymbol{x}) + tv(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} - \\ \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, u(\boldsymbol{x})\|^2 - \cosh(u(\boldsymbol{x})) - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\Big) \\ = \frac{1}{t}\int_\Omega t\ \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + t\ \sinh(u(\boldsymbol{x}))v(\boldsymbol{x}) - t\, f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + O(t)\,,\end{aligned}$$

where we use the Taylor expansions for $t \to 0$

$$\begin{aligned}\|\mathbf{grad}\, u(\boldsymbol{x}) + t\,\mathbf{grad}\, v(\boldsymbol{x})\|^2 &= \|\mathbf{grad}\, u(\boldsymbol{x})\|^2 + 2t\,\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + O(t^2)\,, \\ \cosh(u(\boldsymbol{x}) + tv(\boldsymbol{x})) &= \cosh(u(\boldsymbol{x})) + t\sinh(u(\boldsymbol{x}))v(\boldsymbol{x}) + O(t^2)\,.\end{aligned}$$

❷ We can also appeal to the results of [Lecture → Ex. 5.2.2.6], because we have

$$J(u) = \int_\Omega F(\boldsymbol{x}, \mathbf{grad}\, u(\boldsymbol{x}), u(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} \quad \text{with} \quad F(\boldsymbol{x}, \mathbf{g}, v) = \tfrac{1}{2}\|\mathbf{g}\|^2 + \cosh(v) - f(\boldsymbol{x})v\,.$$

Then we can apply [Lecture → Eq. (5.2.2.10)].

SOLUTION of (5-5.c):

We impose zero Dirichlet boundary conditions on the finite-element functions and these boundary conditions are essential boundary conditions and they will directly induce boundary conditions for the BVP in strong form:

$$u = 0 \quad \text{on} \quad \partial\Omega \ .$$

Stated in a suitable infinite-dimensional function space the non-linear variational problem found in Sub-problem (5-5.a) reads: seek $u \in C^1_{\mathrm{pw},0}(\overline{\Omega})$:

$$\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega}) \ . \tag{5.5.11}$$

Assuming sufficient smoothness of u we can move the gradient off v by integration by parts based on [Lecture → Thm. 1.5.2.7].

$$\int_\Omega -\operatorname{div}(\mathbf{grad}\, u(\boldsymbol{x}))\, v(\boldsymbol{x}) + \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega}) \ . \tag{5.5.12}$$

Boundary terms do not appear, because the test function v vanishes on $\partial\Omega$. Appealing to the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4] we infer the PDE

$$-\Delta u + \sinh(u) = f \quad \text{in} \quad \Omega \ .$$

SOLUTION of (5-5.d):

Again, for the sake of brevity we omit subscripts h. Anyway, the arguments are alike in a finite element space or an infinite-dimensional function space.

The non-linear variational problem (5.5.11) can be written in "canonical form" [Lecture → Def. 5.2.2.20]

$$\boxed{u \in \widehat{V}: \quad \mathsf{a}(u;v) = 0 \quad \forall v \in V_0} \,, \qquad \text{[Lecture → Eq. (5.2.2.21)]}$$

with $V_0 := C^1_{\mathrm{pw},0}(\overline{\Omega})$ and

$$\mathsf{a}(u;v) := \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,, \quad u, v \in V_0 \,. \tag{5.5.15}$$

The variational equation to be solved to obtain the Newton update (in function space) is

$$w \in V_0: \quad \mathsf{D}_u \mathsf{a}(u^{(k-1)};v) w = -\mathsf{a}(u^{(k-1)};v) \quad \forall v \in V_0 \,. \qquad \text{[Lecture → Eq. (5.3.2.9)]}$$

So we have to compute the directional derivative $w \mapsto \mathsf{D}_u \mathsf{a}(u^{(k)};v)w$ of a with respect to u. Concerning a we can split it into a linear and non-linear part

$$\mathsf{a}(u;v) := \underbrace{\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}}_{\text{affine linear in } u} + \underbrace{\int_\Omega \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}}_{\text{non-linear in } u} \,. \tag{5.5.16}$$

The derivative of the linear part "is itself". The directional derivative of the non-linear part can be computed as follows:

$$\lim_{t\to 0} \frac{1}{t} \left(\int_\Omega \sinh(u(\boldsymbol{x}) + t w(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} - \int_\Omega \sinh(u(\boldsymbol{x}))\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right) = \int_\Omega \cosh(u(\boldsymbol{x})) w(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,,$$

where we used the Taylor expansion

$$\sinh(u(\boldsymbol{x}) + t w(\boldsymbol{x})) = \sinh(u(\boldsymbol{x})) + t \sinh'(u(\boldsymbol{x})) w(\boldsymbol{x}) + O(t^2) \,,$$

and $\sinh' = \cosh$ from (5.5.7).

Summing up we have

$$\mathsf{D}_u \mathsf{a}(u^{(k)};v) w = \int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_\Omega \cosh(u(\boldsymbol{x})) w(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$$

and the **update equation** [Lecture → Eq. (5.3.2.9)] becomes: seek $w \in C^1_{\mathrm{pw},0}(\overline{\Omega})$ such that

$$\int_\Omega \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \cosh(u^{(k-1)}(\boldsymbol{x}))\, w(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \mathbf{grad}\, u^{(k-1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + \sinh(u^{(k-1)}(\boldsymbol{x})) v(\boldsymbol{x}) - f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \tag{5.5.17}$$

for all $v \in C^1_{\mathrm{pw},0}(\overline{\Omega})$.

HINT 1 for (5-6.a):

- For $p > -1$ the derivative of $\xi \mapsto \xi^p$ is $\xi \mapsto p\xi^{p-1}$.
- For a smooth function $f : \mathbb{R}_0^+ \to \mathbb{R}$ we have

$$\mathbf{grad}\{\boldsymbol{x} \in \mathbb{R}^2 \mapsto f(\|\boldsymbol{x}\|)\}(\boldsymbol{x}_0) = f'(\|\boldsymbol{x}_0\|)\frac{\boldsymbol{x}_0}{\|\boldsymbol{x}_0\|}\,, \quad \boldsymbol{x}_0 \in \mathbb{R}^2 \setminus \{\mathbf{0}\}\,. \tag{5.6.2}$$

⌟

SOLUTION of (5-6.a):

To begin with, for $p = 2$ we already know the answer [Lecture → Section 1.4.1], [Lecture → § 1.2.3.7]:

$$J_2(v) := \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, v(\boldsymbol{x})\|^2 - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

❶ Intuitive reasoning: Since "2" occurs in this formula, let us try

$$J_p(v) = \int_\Omega \frac{1}{p}\|\mathbf{grad}\, v(\boldsymbol{x})\|^p - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,. \tag{5.6.3}$$

To verify that this is the right choice we have to compute the directional derivative of J_p.

$$\begin{aligned}\frac{J_p(u+tv)-J_p(u)}{t} &= \int_\Omega \frac{\frac{1}{p}\left(\|\mathbf{grad}(u+tv)\|^p - \|\mathbf{grad}\, u\|^p\right)}{t} - fv\,\mathrm{d}\boldsymbol{x}\\ &= \int_\Omega \|\mathbf{grad}\, u\|^{p-1}\frac{\mathbf{grad}\, u}{\|\mathbf{grad}\, u\|}\cdot\mathbf{grad}\, v - fv + O(t)\,\mathrm{d}\boldsymbol{x}\\ &= \int_\Omega \|\mathbf{grad}\, u\|^{p-2}\,\mathbf{grad}\, u\cdot\mathbf{grad}\, v - fv\,\mathrm{d}\boldsymbol{x} + O(t) \quad \text{for} \quad t\to 0\,.\end{aligned}$$

Here we used (5.6.2) in the form

$$\|\boldsymbol{x}+\boldsymbol{h}\|^p = \|\boldsymbol{x}\| + p\|\boldsymbol{x}\|^{p-1}\frac{\boldsymbol{x}\cdot\boldsymbol{h}}{\|\boldsymbol{x}\|} + O(\|\boldsymbol{h}\|^2)\,, \quad \boldsymbol{x},\boldsymbol{h}\in\mathbb{R}^2\,, \quad \boldsymbol{x}\neq\mathbf{0}\,. \tag{5.6.4}$$

This means that

$$\langle \mathsf{D}J_p(u), v\rangle = \int_\Omega \|\mathbf{grad}\, u\|^{p-2}\,\mathbf{grad}\, u\cdot\mathbf{grad}\, v - fv\,\mathrm{d}\boldsymbol{x}\,,$$

which is just the expression in (5.6.1). Why the equation $\frac{\partial \mathsf{D}J_p(u)}{\partial v} = 0$ is satisfied for all $v \in V_0$ is explained in [Lecture → Section 5.2.1].

❷ More structured reasoning: We start with

$$J_p(v) = \int_\Omega \Phi(\mathbf{grad}\, v(\boldsymbol{x})) - f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \text{for some } C^2\text{-function} \quad \Phi:\mathbb{R}^2\to\mathbb{R}\,. \tag{5.6.5}$$

- The occurrence of the term $f(\boldsymbol{x})v(\boldsymbol{x})$ is strongly suggested by the right-hand side functional of (5.6.1).
- The other term in the integrand is suggested by the left-hand side of (5.6.1), which depends on u and v only through their gradients.

At this point the function Φ is unknown and has to be determined. To that we compute the directional derivative using the Taylor expansion of Φ at $\mathbf{grad}\, u$:

$$\begin{aligned}\frac{J_p(u+tv)-J_p(u)}{t} &= \int_\Omega \frac{\Phi(\mathbf{grad}\, u + t\,\mathbf{grad}\, v) - \Phi(\mathbf{grad}\, u)}{t} - fv\,\mathrm{d}\boldsymbol{x}\\ &= \int_\Omega \mathbf{grad}\,\Phi(\mathbf{grad}\, u)\cdot\mathbf{grad}\, v + O(t) - fv\,\mathrm{d}\boldsymbol{x} \quad \text{for} \quad t\to 0\,.\end{aligned}$$

In order to match (5.6.1), Φ has to satisfy

$$\mathbf{grad}\,\Phi(\boldsymbol{w}) = \|\boldsymbol{w}\|^{p-2}\boldsymbol{w} = \|\boldsymbol{w}\|^{p-1}\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|} \quad \forall \boldsymbol{w} \in \mathbb{R}^2\,. \tag{5.6.6}$$

By the formula (5.6.2), this suggests the choice $\Phi(\boldsymbol{w}) = \frac{1}{p}\|\boldsymbol{w}\|^p$, which, when plugged into (5.6.5), yields (5.6.3).

Remark. $\mathbf{grad}\, u = \mathbf{0}$ locally in Ω is not a problem, because the integrand will always be bounded since $p > 1$.

SOLUTION of (5-6.b):

Of course, there are usually many fixed-point iterations that are consistent with a given non-linear variational problem. In this case we face a non-linear variational problem of the form

$$u \in \widehat{V}: \quad \int_\Omega \mathbf{A}(\boldsymbol{x}, \mathbf{grad}\, u(\boldsymbol{x}))\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) + c(u(\boldsymbol{x}))u(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in V_0\,. \quad [\text{Lecture} \to \text{Eq. (5.3.2.1)}]$$

A suitable fixed-point iteration is defined in [Lecture → Eq. (5.3.2.3)] and for the concrete case of (5.6.1) it reads: seek $u^{(k+1)} \in V_0$ such that

$$\int_\Omega \left\| \mathbf{grad}\, u^{(k)} \right\|^{p-2} \mathbf{grad}\, u^{(k+1)}(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in V_0\,. \tag{5.6.7}$$

It is hard to imagine another reasonable consistent fixed-point iteration.

HINT 1 for (5-6.c): The following class is available in the file `plaplacian.h`.

C++ code 5.6.8: MeshFunction wrapper class FunctionMFWrapper ⇢ GitLab

```
template <typename MESHFUNCTION, typename FUNCTION>
class FunctionMFWrapper {
 public:
  static_assert(lf::mesh::utils::MeshFunction<MESHFUNCTION>);
  using mf_vector_t = decltype(std::declval<MESHFUNCTION>()(
      std::declval<const lf::mesh::Entity>(),
      std::declval<const Eigen::MatrixXd>()));
  using mf_result_t = typename mf_vector_t::value_type;
  using F_result_t =
      decltype(std::declval<FUNCTION>()(std::declval<mf_result_t>()));

  explicit FunctionMFWrapper(MESHFUNCTION mf, FUNCTION F)
      : mf_(std::move(mf)), F_(std::move(F)) {}
  FunctionMFWrapper(const FunctionMFWrapper &) = default;
  FunctionMFWrapper(FunctionMFWrapper &&) noexcept = default;
  FunctionMFWrapper &operator=(const FunctionMFWrapper &) = delete;
  FunctionMFWrapper &operator=(FunctionMFWrapper &&) = delete;
  ~FunctionMFWrapper() = default;

  std::vector<F_result_t> operator()(const lf::mesh::Entity &e,
                                     const Eigen::MatrixXd &local) const {
    LF_ASSERT_MSG(e.RefEl().Dimension() == local.rows(),
                  "mismatch between entity dimension and local.rows()");
    const std::vector<mf_result_t> mf_result = mf_(e, local);
    std::vector<F_result_t> result(local.cols());
    for (long i = 0; i < local.cols(); ++i) {
      result[i] = F_(mf_result[i]);
    }
    return result;
  }

 private:
  MESHFUNCTION mf_;
  FUNCTION F_;
};
```

It is meant to “wrap a function around a **MeshFunction**” and might be useful for the the implementation of `fixedPointNextIt()`. ⌟

HINT 2 for (5-6.c): Your implementation should rely on LEHRFEM++'s facilities for local computations and assembly. In particular the classes **lf::fe::DiffusionElementMatrixProvider** and **lf::fe::ScalarLoadEle** will be useful. ⌟

SOLUTION of (5-6.c):

C++ code 5.6.9: Function realizing fixed-point update ⇢ GitLab

```
template <typename F_FUNCTOR>
void fixedPointNextIt(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    F_FUNCTOR f, double p, Eigen::VectorXd &mu_vec) {
  // Wrap right hand side source function into a Mesh Function
  lf::mesh::utils::MeshFunctionGlobal mf_f(f);
  // Reference to current mesh
  const lf::mesh::Mesh &mesh{*(fes_p->Mesh())};
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
  // Dimension of finite element space'
  const std::size_t N_dofs(dofh.NumDofs());
  // We will need a vanishing MeshFunction
  lf::mesh::utils::MeshFunctionConstant mf_zero(0.);
  lf::mesh::utils::MeshFunctionConstant mf_one(1.);
  // We need the gradient of the current iteration in MeshFunction
     format
  const lf::fe::MeshFunctionGradFE mf_grad_uh_prev(fes_p, mu_vec);

  // We need the diffusion coefficient as MeshFunction
  const FunctionMFWrapper mf_abs_uh_pmin2(mf_grad_uh_prev,
                                          [p](Eigen::Vector2d xi) -> double {
                                            return std::pow(xi.norm(), p - 2);
                                          });

  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  Eigen::VectorXd phi(N_dofs);
  phi.setZero();
  // Assemble matrix A, representing (||grad uh_prev||^(p-2) grad uh,
     grad vh)
  // in COO format
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<
      double, decltype(mf_abs_uh_pmin2), decltype(mf_zero)>
      elmat_provider(fes_p, mf_abs_uh_pmin2, mf_zero);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider, A);

  // Assemble right-hand side vector
  // Assemble volume part of right-hand side vector depending on the
     source
  // function f.
  // Initialize object taking care of local computations on all cells.
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      elvec_builder(fes_p, mf_f);
  // Invoke assembly on cells (codim == 0)
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // Enforce homogeneous Dirichlet boundary conditions
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
  // Elimination of degrees of freedom on the boundary.  Also sets the
  // corresponding entries of phi to zero.
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&bd_flags,
       &dofh](lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
```

```
        const lf::mesh::Entity &node{dofh.Entity(gdof_idx)};

        const Eigen::Vector2d node_pos{
            lf::geometry::Corners(*node.Geometry()).col(0)};
        return (bd_flags(node) ? std::make_pair(true, 0.0)
                               : std::make_pair(false, 0.0));
      },
      A, phi);

  // Convert matrix A in COO format to A_crs in Eigen::SparseMatrix
     format
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();

  // Solve the system and store the solution in mu_vec
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  mu_vec = solver.solve(phi);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");

  return;
}
```

Remark. If u_h^k is constant on a mesh cell, then the discretized fixed-point iteration (5.6.7) will not be well-defined and terminate with a "division by zero" error. In order to avoid this, a cut-off can be employed:

```
return std::pow(xi.norm() < tol?1.0/tol : xi.norm(), p - 2);
```

SOLUTION of (5-6.d):

We can rewrite the variational problem (5.6.1) as

$$u \in V_0: \quad \mathsf{a}(u;v) = 0 \quad \forall v \in V_0 \tag{5.6.10}$$

$$\text{with} \qquad \mathsf{a}(u;v) := \int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\|^{p-2}\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) - f(\boldsymbol{x})v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,. \tag{5.6.11}$$

Be alerted to the fact that $v \mapsto \mathsf{a}(u;v)$ is *linear*.

We know from [Lecture → Eq. (5.3.2.5)] that the linear variational problem for the Newton update $w := u^{(k+}$ is

$$w \in V_0: \quad \mathsf{D}_u\mathsf{a}(u^{(k)};v)w = -\mathsf{a}(u^{(k)};v) \quad \forall v \in V_0\,. \tag{5.6.12}$$

Hence, the key task is to compute the directional derivative $\mathsf{D}_u\mathsf{a}(u;v)w$ for $u, v, w \in V_0$. We do this directly based on the definition

$$\mathsf{D}_u\mathsf{a}(u^{(k)};v)w = \lim_{t\to 0} \frac{\mathsf{a}(u+tw;v) - \mathsf{a}(u;v)}{t}\,, \quad u^{(k)} \in \widehat{V}\,, \quad v, w \in V_0 \tag{5.3.2.7}$$

. [Lecture → Eq. (5.3.2.7)]

Matters become slightly simpler by noticing that the derivative can be pulled inside the integral. Hence, it suffices to compute the derivative of $u \mapsto \|\mathbf{grad}\, u\|^{p-2}\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v$

$$\begin{aligned}
&\|\mathbf{grad}(u+tw)\|^{p-2}\, \mathbf{grad}(u+tw) \cdot \mathbf{grad}\, v - \|\mathbf{grad}\, u\|^{p-2}\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \\
=&\left(\|\mathbf{grad}\, u\|^{p-2} + t(p-2)\|\mathbf{grad}\, u\|^{p-3} \frac{\mathbf{grad}\, u}{\|\mathbf{grad}\, u\|} \cdot \mathbf{grad}\, w + O(t^2) \right) (\mathbf{grad}\, u + t\, \mathbf{grad}\, w) \cdot \mathbf{grad}\, v - \\
&\|\mathbf{grad}\, u\|^{p-2}\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \\
=&t(p-2)\|\mathbf{grad}\, u\|^{p-3} \left(\frac{\mathbf{grad}\, u}{\|\mathbf{grad}\, u\|} \cdot \mathbf{grad}\, w \right) (\mathbf{grad}\, u \cdot \mathbf{grad}\, v) + \\
&t\|\mathbf{grad}\, u\|^{p-2}\, \mathbf{grad}\, w \cdot \mathbf{grad}\, v + O(t^2) \\
=&t\|\mathbf{grad}\, u\|^{p-2} \left((p-2) \frac{\mathbf{grad}\, u\ \mathbf{grad}\, u^\top}{\|\mathbf{grad}\, u\|^2} + \mathbf{I}_2 \right) \mathbf{grad}\, w \cdot \mathbf{grad}\, v + O(t^2) \quad \text{for} \quad t \to 0\,.
\end{aligned}$$

We infer that for any $u, v, w \in V_0$

$$\mathsf{D}_u\mathsf{a}(u;v)w = \int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\|^{p-2} \left((p-2) \frac{\mathbf{grad}\, u(\boldsymbol{x})\ \mathbf{grad}\, u(\boldsymbol{x})^\top}{\|\mathbf{grad}\, u(\boldsymbol{x})\|^2} + \mathbf{I}_2 \right) \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,.$$

Of course, $\mathbf{grad}\, u = \mathbf{0}$ in parts of Ω with positive volume is not admitted.

Now we are in a position to write down (5.6.12) explicitly for (5.6.1): seek $w \in V_0$

$$\int_\Omega \left\|\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\right\|^{p-2} \left((p-2) \frac{\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\ \mathbf{grad}\, u^{(k)}(\boldsymbol{x})^\top}{\left\|\mathbf{grad}\, u^{(k)}(\boldsymbol{x})\right\|^2} + \mathbf{I}_2 \right) \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$$

$$= -\int_\Omega \left\| \mathbf{grad}\, u^{(k)}(x) \right\|^{p-2} \mathbf{grad}\, u^{(k)}(x) \cdot \mathbf{grad}\, v(x) - f(x)v(x)\,\mathrm{d}x \quad \forall v \in V_0 \ . \tag{5.6.13}$$

Highlighting the structure of the bilinear form of (5.6.13) we can express it as

$$\mathsf{D}_u \mathsf{a}(u^{(k)}; v)w = \int_\Omega \mathbf{M}(u^{(k)})\, \mathbf{grad}\, w \cdot \mathbf{grad}\, v\,\mathrm{d}x \tag{5.6.14}$$

with
$$\mathbf{M}(u) := \|\mathbf{grad}\, u(x)\|^{p-2} \left((p-2) \frac{\mathbf{grad}\, u(x)\ \mathbf{grad}\, u(x)^\top}{\|\mathbf{grad}\, u(x)\|^2} + \mathbf{I}_2 \right) . \tag{5.6.15}$$

HINT 1 for (5-6.e): The hints for Sub-problem (5-6.c) also apply here. ⌟

SOLUTION of (5-6.e):

C++ code 5.6.16: Function realizing Newton update ⇢ GitLab

```
template <typename F_FUNCTOR>
Eigen::VectorXd newtonUpdate(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fes_p,
    F_FUNCTOR f, double p, const Eigen::VectorXd &mu_vec) {
  // Initialize the return value
  Eigen::VectorXd upd(mu_vec.size());
  // Wrap right hand side source function into a Mesh Function
  lf::mesh::utils::MeshFunctionGlobal mf_f(f);
  // Reference to current mesh
  const lf::mesh::Mesh &mesh{*(fes_p->Mesh())};
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
  // Dimension of finite element space'
  const std::size_t N_dofs(dofh.NumDofs());
  // We will need a vanishing MeshFunction
  lf::mesh::utils::MeshFunctionConstant mf_zero(0.);
  // We need the gradient of the current iteration in MeshFunction
     format
  const lf::fe::MeshFunctionGradFE mf_grad_uh_prev(fes_p, mu_vec);

  // We need ||grad uh_prev||^(p-2)*[(p-2)/||grad uh_prev||^2*grad
  // uh_prev*grad uh_prevT + I] as MeshFunction
  const FunctionMFWrapper mf_coeff_LHS(
      mf_grad_uh_prev, [p](Eigen::Vector2d xi) -> Eigen::MatrixXd {
        return std::pow(xi.norm(), p - 2) *
               ((p - 2) / xi.squaredNorm() * xi * xi.transpose() +
                Eigen::Matrix2d::Identity());
      });
  // We need ||grad uh_prev||^(p-2) as MeshFunction
  const FunctionMFWrapper mf_coeff_RHS(
      mf_grad_uh_prev,
      [p](Eigen::Vector2d xi) -> double { return std::pow(xi.norm(), p - 2); });

  // Assemble matrix A, representing (||grad
     uh_prev||^(p-2)*[(p-2)/||grad
  // uh_prev||^2*grad uh_prev*grad uh_prevT + I] grad uh, grad vh) in COO
  // format
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<
      double, decltype(mf_coeff_LHS), decltype(mf_zero)>
      elmat_provider_A(fes_p, mf_coeff_LHS, mf_zero);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider_A, A);

  // Assemble matrix M, representing (||grad uh_prev||^(p-2) grad uh,
     grad vh)
  // in COO format
  lf::assemble::COOMatrix<double> M(N_dofs, N_dofs);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<
      double, decltype(mf_coeff_RHS), decltype(mf_zero)>
      elmat_provider_M(fes_p, mf_coeff_RHS, mf_zero);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_provider_M, M);

  // Convert M to Eigen::SparseMatrix format
```

```
  Eigen::SparseMatrix<double> M_crs = M.makeSparse();

  // Assemble right-hand side vector
  Eigen::VectorXd phi(N_dofs);
  // phi = 0
  phi.setZero();
  // Assemble volume part of right-hand side vector depending on the source
  // function f.
  // Initialize object taking care of local computations on all cells.
  lf::uscalfe::ScalarLoadElementVectorProvider<double, decltype(mf_f)>
      elvec_builder(fes_p, mf_f);
  // Invoke assembly on cells (codim == 0)
  // phi = (f,vh)
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);

  // phi = -(||grad uh_prev||^(p-2) grad uh_prev, grad vh)
  phi -= M_crs * mu_vec;

  // Enforce homogeneous Dirichlet boundary conditions
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fes_p->Mesh(), 2)};
  // Elimination of degrees of freedom on the boundary.  Also sets the
  // corresponding entries of phi to zero.
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&bd_flags,
       &dofh](lf::assemble::glb_idx_t gdof_idx) -> std::pair<bool, double> {
        const lf::mesh::Entity &node{dofh.Entity(gdof_idx)};

        const Eigen::Vector2d node_pos{
            lf::geometry::Corners(*node.Geometry()).col(0)};
        return (bd_flags(node) ? std::make_pair(true, 0.0)
                               : std::make_pair(false, 0.0));
      },
      A, phi);

  // Convert matrix A in COO format to A_crs in Eigen::SparseMatrix format
  Eigen::SparseMatrix<double> A_crs = A.makeSparse();

  // Solve the system and add the solution to mu_vec
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(A_crs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
  upd = solver.solve(phi);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");

  return upd;
}
```

HINT 1 for (6-1.a): Remember from what property distinguishes an orthogonal matrix. Thus you see that the assertion we want to verify boils down to showing that the bilinear expression $t \mapsto \mathbf{Y}(t)^\top \mathbf{Y}(t)$ does not vary along trajectories, that is, its time derivative must vanish. ⌟

HINT 2 for (6-1.a): Since the matrix product is bilinear, the fact that $\frac{d}{dt}\mathbf{Y}(t)^\top\mathbf{Y}(t) = \mathbf{O}$ can be established by means of the product rule and using the differential equation. Sloppily speaking, the derivative of a product of time-dependent matrices can be computed in almost the same way as the derivative of a product of functions, except that you must change the order of the factors. ⌟

SOLUTION of (6-1.a):

Let us consider the time derivative of $t \mapsto \mathbf{Y}(t)^\top \mathbf{Y}(t)$. Applying the product rule we find ($\dot{\mathbf{Y}}(t) := \frac{d}{dt}\mathbf{Y}(t)$)

$$\begin{aligned}\frac{d}{dt}(\mathbf{Y}^\top \mathbf{Y}) = \dot{\mathbf{Y}}^\top \mathbf{Y} + \mathbf{Y}^\top \dot{\mathbf{Y}} &= (\mathbf{A}\mathbf{Y})^\top \mathbf{Y} + \mathbf{Y}^\top \mathbf{A}\mathbf{Y} \\ &= -\mathbf{Y}^\top \mathbf{A}\mathbf{Y} + \mathbf{Y}^\top \mathbf{A}\mathbf{Y} = 0\,.\end{aligned}$$

This implies $\mathbf{Y}(t)^\top \mathbf{Y}(t)$ is constant. From this, it follows that the orthogonality is preserved, as claimed. In fact, $\mathbf{I} = \mathbf{Y}(0)^\top \mathbf{Y}(0) = \mathbf{Y}(t)^\top \mathbf{Y}(t)$ for all t.

HINT 1 for (6-1.b): Since (6.1.1) is a linear ordinary differential equations, implicit timestepping schemes give rise to linear systems of equations, which can be solved using EIGEN's built-in elimination solvers. ⌟

SOLUTION of (6-1.b):

The explicit Euler method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}(t_k, \mathbf{Y}_k).$$

For the given differential equation we therefore obtain

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{A}\mathbf{Y}_k.$$

C++11-code 6.1.2: Explicit Euler method. ↪ GITHUB

```
2 Eigen::MatrixXd eeulstep(const Eigen::MatrixXd& A, const Eigen::MatrixXd& Y0,
3                          double h) {
4   Eigen::MatrixXd res = Eigen::MatrixXd::Zero(Y0.rows(), Y0.cols());
5   res = Y0 + h * A * Y0;
6   return res;
7 }
```

The implicit Euler method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}(t_{k+1}, \mathbf{Y}_{k+1}).$$

For the given differential equation this yields

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{A}\mathbf{Y}_{k+1} \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = (\mathbf{I} - h\mathbf{A})^{-1}\mathbf{Y}_k$$

Hence, every step of the single-step method entails solving a linear system of equations, which can be done using EIGEN's direct elimination solvers introduced in .

C++11-code 6.1.3: Implicit Euler method.↪ GITHUB

```
2 Eigen::MatrixXd ieulstep(const Eigen::MatrixXd& A, const Eigen::MatrixXd& Y0,
3                          double h) {
4   Eigen::MatrixXd res = Eigen::MatrixXd::Zero(Y0.rows(), Y0.cols());
5   int n = A.rows();
6   res = (Eigen::MatrixXd::Identity(n, n) - h * A).partialPivLu().solve(Y0);
7   return res;
8 }
```

Finally, the implicit mid-point method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}\left(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{Y}_k + \mathbf{Y}_{k+1})\right),$$

hence

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}(\mathbf{Y}_k + \mathbf{Y}_{k+1}) \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = \left(\mathbf{I} - \tfrac{h}{2}\mathbf{A}\right)^{-1}\left(\mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}\mathbf{Y}_k\right).$$

C++11-code 6.1.4: Implicit midpoint method.↪ GITHUB

```
2 Eigen::MatrixXd impstep(const Eigen::MatrixXd& A, const Eigen::MatrixXd& Y0,
```

```
                        double h) {
  Eigen::MatrixXd res = Eigen::MatrixXd::Zero(Y0.rows(), Y0.cols());
  int n = A.rows();
  res = (Eigen::MatrixXd::Identity(n, n) - h * 0.5 * A)
            .partialPivLu()
            .solve(Y0 + h * 0.5 * A * Y0);
  return res;
}
```

!

The method `PartialPivLu()` can also be replaced with the generic call `lu()`. The last code well showcases “expression programming” in EIGEN.

SOLUTION of (6-1.c):

C++11-code 6.1.5: Orthogonality testing code.↪ GITHUB

```
  double h = 0.01;  // stepsize
  Eigen::Vector3d norms;
  // Build M
  Eigen::Matrix3d M;
  M << 8, 1, 6, 3, 5, 7, 9, 9, 2;
  // Build A
  Eigen::Matrix3d A;
  A << 0, 1, 1, -1, 0, 1, -1, -1, 0;
  Eigen::MatrixXd I = Eigen::Matrix3d::Identity();
  // Build Q
  Eigen::HouseholderQR<Eigen::Matrix3d> qr(3, 3);
  qr.compute(M);
  Eigen::MatrixXd Q = qr.householderQ();
  // Set initial conditions
  Eigen::Matrix3d Meeul = Q, Mieul = Q, Mimp = Q;

  std::vector<int> sep = {5, 15};
  std::cout << std::setw(sep[0]) << "step" << std::setw(sep[1]) << "exp. Eul"
            << std::setw(sep[1]) << "imp. Eul" << std::setw(sep[1]) << "Mid-Pt"
            << std::endl;
  // Norm of Y'Y-I for initial value
  std::cout << std::setw(sep[0]) << "0" << std::setw(sep[1])
            << (Meeul.transpose() * Meeul - I).norm() << std::setw(sep[1])
            << (Mieul.transpose() * Mieul - I).norm() << std::setw(sep[1])
            << (Mimp.transpose() * Mimp - I).norm() << std::endl;
  // Norm of Y'Y-I for 20 steps
  for (unsigned int j = 0; j < 20; ++j) {
    Meeul = MatODE::eeulstep(A, Meeul, h);
    Mieul = MatODE::ieulstep(A, Mieul, h);
    Mimp = MatODE::impstep(A, Mimp, h);

    norms[0] = (Meeul.transpose() * Meeul - I).norm();
    norms[1] = (Mieul.transpose() * Mieul - I).norm();
    norms[2] = (Mimp.transpose() * Mimp - I).norm();

    std::cout << std::setw(sep[0]) << j + 1 << std::setw(sep[1]) << norms[0]
              << std::setw(sep[1]) << norms[1] << std::setw(sep[1]) << norms[2]
              << std::endl;
  }
```

The following table records the Frobenius norm of $\mathbf{Y}_k^\top \mathbf{Y}_k - \mathbf{I}$ for the three different single-step method investigated in this problem:

step	exp. Eul	imp. Eul	IMP
0	7.28022e-16	7.28022e-16	7.28022e-16
1	0.000424264	0.000424137	5.32444e-16
2	0.000848655	0.000848146	9.35491e-16
3	0.00127317	0.00127203	1.47288e-15
4	0.00169782	0.00169578	1.97201e-15
5	0.00212259	0.00211941	2.18124e-15
6	0.00254749	0.00254291	2.71154e-15
7	0.00297252	0.00296629	3.28877e-15
8	0.00339768	0.00338954	3.69389e-15
9	0.00382296	0.00381266	4.15556e-15
10	0.00424837	0.00423565	4.6497e-15
11	0.00467391	0.00465852	5.27444e-15
12	0.00509958	0.00508125	5.9441e-15
13	0.00552537	0.00550387	6.20137e-15
14	0.00595129	0.00592635	6.59532e-15
15	0.00637734	0.00634871	7.06934e-15
16	0.00680352	0.00677095	7.37985e-15
17	0.00722983	0.00719305	8.12551e-15
18	0.00765626	0.00761503	8.60154e-15
19	0.00808282	0.00803688	8.79041e-15
20	0.00850951	0.00845861	9.21619e-15

Orthogonality is preserved only by the implicit midpoint rule. Explicit and implicit Euler methods do not preserve orthogonality.

SOLUTION of (6-10.a):

In this case we have

$$\mathbf{f}(\mathbf{q},t) = q \quad , \quad \mathbf{g}(\mathbf{p}) = -p \quad \Rightarrow \quad f(q) := q \quad , \quad g(p) := -p \,.$$

Also note that the timestepping scheme (6.10.2) is *explicit* and, hence, the implementation is straightforward.

C++ code 6.10.5: Sub-problem (6-10.a): A single timestep of the symplectic single-step method ↪ GITHUB

```
2  void sympTimestep(double tau, Eigen::Vector2d &pq_j) {
3    // Coefficients of the method
4    const Eigen::Vector3d a{2. / 3., -2. / 3., 1.};
5    const Eigen::Vector3d b{7. / 24., 3. / 4., -1. / 24.};
6    // Single step
7    for (int i = 0; i < 3; ++i) {
8      pq_j(0) += tau * b(i) * pq_j(1);
9      pq_j(1) -= tau * a(i) * pq_j(0);
10   }
11 }
```

HINT 1 for (6-10.b): The exact solution of the initial value problem for (6.10.4) is $p(t) = \sin(t)$ and $q(t) = \cos(t)$, hence $p(2\pi) = 0$ and $q(2\pi) = 1$. ⌟

SOLUTION of (6-10.b):

C++ code 6.10.6: Sub-problem (6-10.a): Monitoring convergence of symplectic timestepping method for oscillator equation ↬ GITHUB

```
void sympTimesteppingODETest() {
  int nIter = 7;   // total number of iterations
  unsigned int m;  // number of equidistant steps
  Eigen::Vector2d approx_sol;

  // Initial conditions
  Eigen::Vector2d init_cond;
  init_cond << 0.0, 1.0;  // initial conditions

  // Evaluating the error at the final step between the approx solutions as
  // given by the symplectic method and the exact solution computed from
  // the anlytic formula.
  double errors[nIter];  // errors vector for all approx. sols
  double tau;
  for (int k = 0; k < nIter; k++) {
    m = 10 * std::pow(2, k);
    // Computing approximate solution
    approx_sol = init_cond;
    tau = 2.0 * PI / m;
    for (int i = 0; i < m; i++) {
      sympTimestep(tau, approx_sol);
    }
    // Computing the error in the maximum norm
    errors[k] = std::abs(std::sin(2.0 * PI) - approx_sol[0]) +
                std::abs(std::cos(2.0 * PI) - approx_sol[1]);
  }
  // Printing results
  std::cout << "Convergence of Symplectic Time Stepping Method:\n";
  std::cout << "\tM\t\terr(M)\n";
  for (int k = 0; k < nIter; k++) {
    std::cout << "\t" << 10 * std::pow(2, k) << "\t\t" << errors[k]
              << std::endl;
  }
}
```

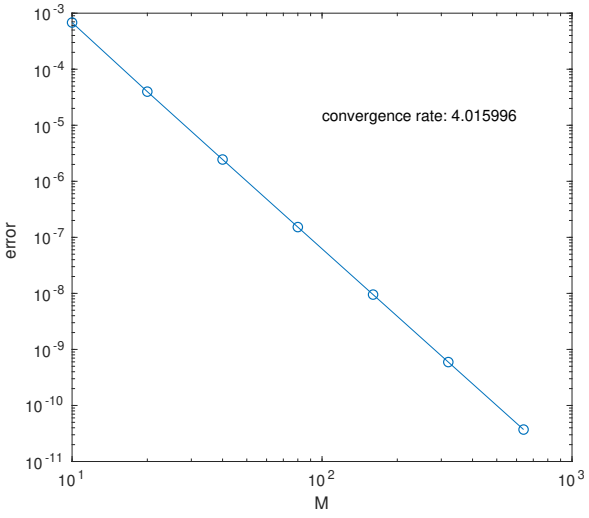


Table of errors:

M	$\mathrm{err}(M)$
10	0.000680607
$10 \cdot 2^1$	3.9793e-05
$10 \cdot 2^2$	2.4477e-06
$10 \cdot 2^3$	1.52386e-07
$10 \cdot 2^4$	9.51496e-09
$10 \cdot 2^5$	5.94542e-10
$10 \cdot 2^6$	3.71564e-11

◁ Log-log plot of `err` vs. M. The error curve is almost a line and from its slope we can read off a convergence rate of ~ 4.

A doubling of M leads to a decrease of the error by roughly a factor of 16: $\mathrm{err}(M) = O(M^{-4})$, algebraic convergence with rate 4.

SOLUTION of (6-10.c):

In order to show that $t \mapsto E(t)$ is constant we prove that its derivative vanishes: Using the *chain rule* in the form

$$\frac{d}{dt}\{t \mapsto G(\mathbf{F}(t))\}(t_0) = (\mathbf{grad}\, G)(\mathbf{F}(t_0))^\top \frac{d\mathbf{F}}{dt}(t_0)\,, \quad t_0 \in I\,, \qquad \begin{array}{l} G : \mathbb{R}^m \to \mathbb{R}\,, \\ \mathbf{F} : I \subset \mathbb{R} \to \mathbb{R}^m\,, \end{array}$$

we get

$$\begin{aligned} \dot{E}(t) &= \left(\mathbf{grad}\left\{\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix} \mapsto H(\mathbf{p},\mathbf{q})\right\}(\mathbf{p}(t),\mathbf{q}(t))\right)^\top \frac{d}{dt}\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix}(t) \\ &= \frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}(t),\mathbf{q}(t))^\top \dot{\mathbf{p}}(t) + \frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}(t),\mathbf{q}(t))^\top \dot{\mathbf{q}}(t) \\ &\overset{(6.10.8)}{=} -\frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}(t),\mathbf{q}(t))^\top \frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}(t),\mathbf{q}(t)) + \frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}(t),\mathbf{q}(t))^\top \frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}(t),\mathbf{q}(t)) \\ &= 0\,. \end{aligned}$$

SOLUTION of (6-10.d):

We calculate the partial derivatives using the chain rule for $h(\mathbf{x}) = (f \circ g)(\mathbf{x})$, $f(\xi) := \xi^2$, $g(\mathbf{x}) := \|\mathbf{x}\|^2$, in the case of the first one:

$$\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p},\mathbf{q}) = \mathbf{grad}\left\{\mathbf{x} \mapsto \|\mathbf{x}\|_2^4\right\}(\mathbf{q}) = 4\|\mathbf{q}\|_2^2\mathbf{q}\,,$$
$$\frac{\partial H}{\partial \mathbf{p}}(\mathbf{p},\mathbf{q}) = \mathbf{grad}\left\{\mathbf{x} \mapsto \tfrac{1}{2}\|\mathbf{x}\|_2^2\right\}(\mathbf{p}) = \mathbf{p}\,.$$

These derivatives can also be computed “in the pedestrian way” by componentwise differentiation.

Hence, the Hamiltonian ODE for the special Hamiltonian from (6.10.10) is of the form (6.10.1) with

$$\mathbf{f}(\mathbf{q},t) := -4\|\mathbf{q}\|^2\mathbf{q} \quad , \quad \mathbf{g}(\mathbf{p}) := \mathbf{p}\,. \tag{6.10.11}$$

SOLUTION of (6-10.e):

The implementation of (6.10.2) is straightforward. The inner loop can be replaced with a sequence of assignments.

C++ code 6.10.12: Sub-problem (6-10.e): Symplectic timestepping for (6.10.11) ⇝ **GITHUB**

```
Eigen::MatrixXd simulateHamiltonianDynamics(const Eigen::VectorXd &p0,
                                            const Eigen::VectorXd &q0, double T,
                                            unsigned int M) {
  int n = p0.size();
  Eigen::MatrixXd PQ(2 * n, M + 1);

  // Coefficients of the method
  Eigen::VectorXd a(3);
  a << 2. / 3., -2. / 3., 1.;
  Eigen::VectorXd b(3);
  b << 7. / 24., 3. / 4., -1. / 24.;

  double tau = T / M;
  Eigen::VectorXd pj(p0), qj(q0);
  PQ.col(0) << pj, qj;
  for (int j = 1; j <= M; j++) {   // integrate
    for (int k = 0; k < 3; k++) {  // one step
      // f(q) = - 4 * |q|^2 * q
      pj -= tau * b(k) * 4. * qj.squaredNorm() * qj;
      // g(p) = p
      qj += tau * a(k) * pj;
      PQ.col(j) << pj, qj;
    }
  }
  return PQ;
}
```

SOLUTION of (6-10.f):

Owing to the result of Sub-problem (6-10.c), for the exact solution $t \mapsto (\mathbf{p}(t), \mathbf{q}(t))$, $t \in \mathbb{R}$, we have

$$E(t) = H(\mathbf{p}_0, \mathbf{q}_0) = 1 \quad \forall t \in \mathbb{R} \, .$$

Thus, it is natural to study the difference $\mathcal{E}(M) - 1$. We observe that this difference approximately shrinks by a factor of 8 whenever the number of timesteps is doubled. This is a clear indication of *algebraic convergence* $\mathcal{E}(M) \to 1$ for $M \to \infty$ with *rate* 3.

SOLUTION of (6-2.a):

There are many ways to implement the formulas of [Lecture → Def. 6.4.0.9]. Thanks to the restriction to explicit RK-SSM, all merely involve $\mathbf{f}$-evaluations.

C++11-code 6.2.1: Solution of (6-2.a): Constructor and private data ↪ GITHUB

```
class RKIntegrator {
 public:
  // Constructor for the RK method.
  RKIntegrator(const Eigen::MatrixXd &A, const Eigen::VectorXd &b)
      : A_(A), b_(b), s_(b.size()) {
    assert(A.cols() == A.rows() && "Matrix must be square.");
    assert(A.cols() == b.size() && "Incompatible matrix/vector size.");
  }

  // Explicit Runge-Kutta numerical integrator
  template <class Function>
  std::vector<Eigen::VectorXd> solve(Function &&f, double T,
                                     const Eigen::VectorXd &y0, int M) const;

 private:
  // Butcher data
  const Eigen::MatrixXd A_;
  const Eigen::VectorXd b_;
  int s_;  // size of Butcher tableau
};
```

C++11-code 6.2.2: Solution of (6-2.a) Solve method ↪ GITHUB

```
template <typename Function>
std::vector<Eigen::VectorXd> RKIntegrator::solve(Function &&f, double T,
                                                 const Eigen::VectorXd &y0,
                                                 int M) const {
  int dim = y0.size();  // dimension
  double h = T / M;     // step size
  std::vector<Eigen::VectorXd> sol;
  sol.reserve(M + 1);

  // Initial data
  sol.push_back(y0);

  // RK looping tools
  Eigen::VectorXd incr(dim);
  std::vector<Eigen::VectorXd> k;
  k.reserve(s_);  // Runge-Kutta Increments

  // Stepping
  Eigen::VectorXd step(dim);
  for (int iter = 0; iter < M; ++iter) {
    // clear looping variables
    step.setZero();
    k.clear();
    // explicit RK
    k.push_back(f(sol.at(iter)));
```

```
27      step = step + b_[0] * k[0];
28      for (int i = 1; i < s_; ++i) {
29        incr.setZero();
30        for (int j = 0; j < i; ++j) {
31          incr += A_(i, j) * k[j];
32        }
33        k.push_back(f(sol.at(iter) + h * incr));
34        step += b_[i] * k.back();
35      }
36
37      // step forward
38      sol.push_back(sol[iter] + h * step);
39    }
40    return sol;
41  }
```

SOLUTION of (6-2.b):

The main task is to prepare the inputs for the application of **RKIntegrator**. The right-hand-side function **f** can be passed as lambda function, see .

C++11-code 6.2.5: Solution of (6-2.b) ↬ GITHUB

```
2  // Data for the Butcher scheme describing the explicit 3-stage Runge-Kutta
3  // method
4  Eigen::Matrix3d A;
5  Eigen::Vector3d b;
6  A << 0, 0, 0, 1.0 / 3.0, 0, 0, 0, 2.0 / 3.0, 0;
7  b << 0.25, 0, 0.75;
8
9  // Right-hand-side vectorfield for predator-prey model
10 auto f = [](Eigen::VectorXd y) {
11   double alpha1 = 3.0, alpha2 = 2.0, beta1 = 0.1, beta2 = 0.1;
12   Eigen::Vector2d eval = y;
13   eval(0) *= alpha1 - beta1 * y(1);
14   eval(1) *= -alpha2 + beta2 * y(0);
15   return eval;
16 };
17
18 // Final time for model
19 double T = 10.;
20
21 // Initial value for model
22 Eigen::Vector2d y0;
23 y0 << 100, 5;
24
25 // Array of number of steps (for convergence study)
26 int M[8] = {128, 256, 512, 1024, 2048, 4096, 8192, 16384};
27
28 // Reference "exact" value y(10) at final time T = 10 (approximated)
29 Eigen::Vector2d y_ref;
30 y_ref << 0.319465882659820, 9.730809352326228;
31
32 // Initialize RK with Butcher scheme
33 RK3Prey::RKIntegrator RK(A, b);
34
35 // Start convergence study
36 std::cout << std::setw(15) << "M" << std::setw(15) << "error" << std::setw(15)
37           << "rate" << std::endl;
38 std::vector<Eigen::VectorXd> sol;
39 double errold;
40 for (unsigned int i = 0; i < 8; ++i) {
41   sol = RK.solve(f, T, y0, M[i]);
42   double err = (sol.back() - y_ref).norm();
43   std::cout << std::setw(15) << M[i] << std::setw(15) << err;
44   if (i > 0) {
45     std::cout << std::setw(15) << log2(errold / err);
46   }
47   errold = err;
48   std::cout << std::endl;
49 }
```

M	error	est. rate
128	8.9601	
256	0.60584	3.88651
512	0.0688263	3.13791
1024	0.00838567	3.03696
2048	0.00103847	3.01347
4096	0.000129287	3.00581
8192	1.61305e-05	3.00271
16384	2.01448e-06	3.00131

In the table beside the column “rate” gives a “local” estimate of the rate α obtained from the quotient of two successive error norms:

$$\mathrm{err}(\tfrac{1}{2}h) : \mathrm{err}(h) \approx 2^{-\alpha} \quad \Leftrightarrow \quad \alpha \approx \frac{\log \mathrm{err}(\tfrac{1}{2}h) - \log \mathrm{err}(h)}{\log 2} \,.$$

Obviously, the 3-stage RK-SSM given in (6.2.4) is of order 3.

SOLUTION of (6-3.a):

Writing the explicit/forward Euler discrete evolution as $\boldsymbol{\Psi}^h(\mathbf{y}_0) := \mathbf{y}_0 + hf(\mathbf{y}_0)$ (and $p = 1$), we obtain:

$$\begin{aligned}\widetilde{\boldsymbol{\Psi}}^h(\mathbf{y}_0) &= \frac{1}{1-2^p}\left(\mathbf{y}_0 + hf(\mathbf{y}_0) - 2^p \cdot (\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0) + \frac{h}{2}f(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right)\\ &= \mathbf{y}_0 + \frac{1}{1-2^p}\left(h(1-2^{p-1})f(\mathbf{y}_0) - 2^{p-1}hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right)\\ &= \mathbf{y}_0 + \left(hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) .\end{aligned}$$

This is the evolution operator of the **explicit midpoint method**, a 2-stage explicit Runge-Kutta method, see [Lecture → Eq. (6.4.0.6)].

SOLUTION of (6-3.b):

The formula (6.3.1) can be implemented directly:

C++ code 6.3.2: Solution of (6-3.b)↝ GITHUB

```
2 template <class DiscEvlOp>
3 double PsiTilde(const DiscEvlOp& Psi, unsigned int p, double h, double y0) {
4   return (Psi(h, y0) - std::pow(2, p) * Psi(h / 2., Psi(h / 2., y0))) /
5          (1. - std::pow(2, p));
6 }
```

HINT 1 for (6-3.c): You can use `std::vector<double>::push_back()` to add elements to the end of a **std::vector**. ⌟

SOLUTION of (6-3.c):

Requested is a straightforward implementation of [Lecture → Eq. (6.3.1.5)].

C++ code 6.3.3: Solution of Sub-problem (6-3.c) ⇢ GITHUB

```
2  template <class DiscEvlOp>
3  std::vector<double> OdeIntEqui(const DiscEvlOp& Psi, double T, double y0,
4                                 int M) {
5    double h = T / M;
6    double t = 0.;
7    std::vector<double> Y;
8
9    Y.reserve(M + 1);
10   Y.push_back(y0);
11   double y = y0;
12
13   while (t < T) {
14     y = Psi(h, y);
15     Y.push_back(y);
16     t += std::min(T - t, h);
17   }
18
19   return Y;
20 }
```

HINT 1 for (6-3.d): The solution of the IVP Eq. (6.3.4) is $\tan(t)$. ⌟

SOLUTION of (6-3.d):

C++ code 6.3.5: Sub-problem (6-3.d): Implementation of `testcvpExtrapolatedEuler()`
→ GITHUB

```
double TestCvpExtrapolatedEuler() {
  double conv_rate;
  double T = 1.0;
  double y0 = 0.0;
  auto f = [](double y) -> double { return 1.0 + y * y; };

  auto Psi = [&f](double h, double y0) -> double { return y0 + h * f(y0); };
  unsigned p = 1;

  // lambda corresponding to \tilde{\psi}
  auto Psi_tilde = [&Psi, &f, &p](double h, double y0) -> double {
    return PsiTilde(Psi, p, h, y0);
  };

  // exact value
  double y_ex1 = std::tan(T);

  // values for convergence study
  Eigen::ArrayXd err(11);
  Eigen::ArrayXd M(11);

  std::cout << "Error table for equidistant steps:" << std::endl;
  std::cout << "M"
            << "\t"
            << "Error" << std::endl;

  for (int i = 0; i < 11; ++i) {
    M(i) = std::pow(2, i + 2);
    double yT = OdeIntEqui(Psi_tilde, T, y0, M(i)).back();
    err(i) = std::abs(yT - y_ex1);
    std::cout << M(i) << "\t" << err(i) << std::endl;
  }
  // compute fitted rate
  Eigen::VectorXd coeffs = polyfit(M.log(), err.log(), 1);
  conv_rate = -coeffs(0);
  return conv_rate;
}

/* SAM_LISTING_BEGIN_4 */
std::pair<std::vector<double>, std::vector<double>> SolveTangentIVP() {
  auto f = [](double y) -> double { return 1.0 + y * y; };
  double y0 = 0.0;

  double T = 1.5;
  unsigned p = 1;
  double h0 = 1. / 100.;

  auto Psi = [&f](double h, double y0) -> double { return y0 + h * f(y0); };
  // run adaptive algoritm
  return OdeIntSsCtrl(Psi, p, y0, T, h0, 10e-4, 10e-6, 10e-5);

```

```
}
/* SAM_LISTING_END_4 */

}  // namespace ODESolve
```

M	$\lvert y_N(1) - y_{ex}(1)\rvert$
4	0.0650468
8	0.0209801
16	0.00595508
32	0.00158261
64	0.000407501
128	0.000103355
256	2.60232e-05
512	6.52882e-06
1024	1.63508e-06
2048	4.09129e-07
4096	1.02327e-07

The empiric convergence rate is 2 ($O(h^2)$), which is evidence that the single-step method $\widetilde{\mathbf{\Psi}}^h$ from (6.3.1) enjoys order 2.
Conversely, the order of the underlying forward Euler method is $p = 1$.
Hence, the construction from (6.3.1) manages to raise the order of the method by 1.

HINT 1 for (6-3.e): Just adapt [Lecture → Code 6.5.2.18] using the discrete evolutions $\boldsymbol{\Psi}$ and $\widetilde{\boldsymbol{\Psi}}$ from (6.3.1) instead of the generic **Psilow** and **Psihigh**. ⌟

SOLUTION of (6-3.e):

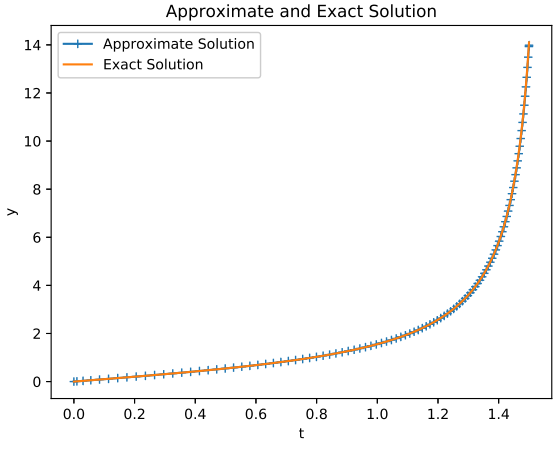
C++ code 6.3.6: Solution of Sub-problem (6-3.e). ➙ GITHUB

```
template <class DiscEvlOp>
std::pair<std::vector<double>, std::vector<double>> OdeIntSsCtrl(
    const DiscEvlOp& Psi, unsigned int p, double y0, double T, double h0,
    double reltol, double abstol, double hmin) {
  double h = h0;
  std::vector<double> Y, t;
  Y.push_back(y0);
  t.push_back(0.0);

  double y = y0;

  while (t.back() < T && h > hmin) {
    double y_high = PsiTilde(Psi, p, std::min(T - t.back(), h), y);
    double y_low = Psi(std::min(T - t.back(), h), y);
    double est = std::abs(y_high - y_low);
    double tol = std::max(reltol * std::abs(y), abstol);

    if (est < tol) {
      y = y_high;
      t.push_back(t.back() + std::min(T - t.back(), h));
      Y.push_back(y_high);
    }
    h = h * std::max(0.5, std::min(2., std::pow(tol / est, 1. / (p + 1))));
  }
  if (h < hmin) {
    std::cerr << "Warning: Failure at t=" << t.back()
              << ". Unable to meet integration tolerances without reducing the "
                 "step size below the smallest value allowed ("
              << hmin << ") at time t." << std::endl;
  }

  return {t, Y};
}
```

SOLUTION of (6-3.f):

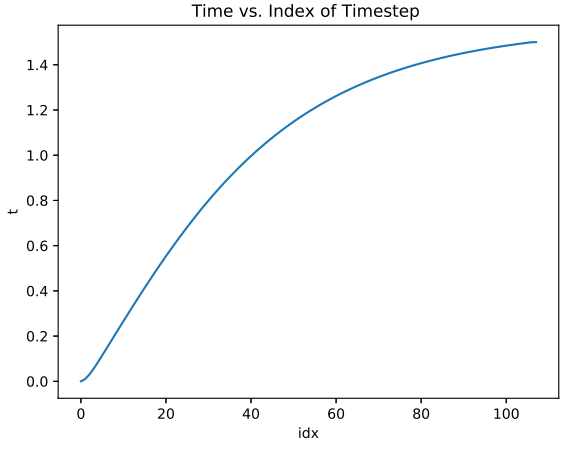
C++ code 6.3.7: Implementation of `SolveTangentIVP()` ↪ GITHUB

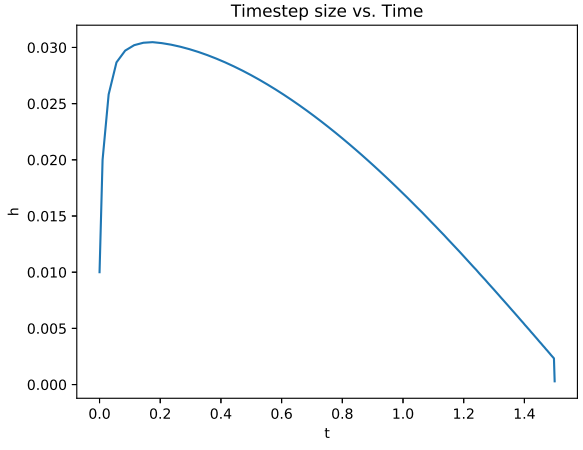
```
std::pair<std::vector<double>, std::vector<double>> SolveTangentIVP() {
  auto f = [](double y) -> double { return 1.0 + y * y; };
  double y0 = 0.0;

  double T = 1.5;
  unsigned p = 1;
  double h0 = 1. / 100.;

  auto Psi = [&f](double h, double y0) -> double { return y0 + h * f(y0); };
  // run adaptive algoritm
  return OdeIntSsCtrl(Psi, p, y0, T, h0, 10e-4, 10e-6, 10e-5);

}
```



◁ As in [Lecture → Ex. 6.5.1.4] the adaptive timestep control manages to detect the blow-up of the solution and reduces the timestep when approach the time $t_* = \pi/2$ from which the solution ceases to exist.

The following plots illustrate the effect of timestep control in this example.





HINT 1 for (6-4.a): What we face in (6.4.1) is an implicit ODE. First convert it into the form $\ddot{\mathbf{u}} = \mathbf{g}(\mathbf{u})$ for a suitable function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$. ⌟

HINT 2 for (6-4.a): The function $\mathbf{g}$ introduced in the previous hint can be expressed by means of the inverse of a suitable tridiagonal matrix. ⌟

HINT 3 for (6-4.a): See [Lecture → Rem. 6.1.3.5] for a recipe on how to convert a second-order ODE into a first-order system. ⌟

SOLUTION of (6-4.a):

The second-order IVP can be rewritten as a first-order one by introducing the derivative of $\mathbf{u} = [u_1, \ldots, u_n]$ as auxiliary vector-valued function $\mathbf{v}(t) := \dot{\mathbf{u}}(t)$

$$
\begin{aligned}
\dot{u}_i &= v_i \qquad i = 1, \ldots, n \,, \\
2\dot{v}_1 - \dot{v}_2 &= u_1(u_2 + u_1) \,, \\
-\dot{v}_{i-1} + 2\dot{v}_i - \dot{v}_{i+1} &= u_i(u_{i-1} + u_{i+1}) \qquad i = 2, \ldots, n-1 \,, \\
2\dot{v}_n - \dot{v}_{n-1} &= u_n(u_n + u_{n-1}) \,, \\
u_i(0) &= u_{0,i} \qquad i = 1, \ldots, n \,, \\
v_i(0) &= v_{0,i} \qquad i = 1, \ldots, n \,.
\end{aligned}
$$

The ODE system can then be written in the vector form

$$
\dot{\mathbf{u}} = \mathbf{v} \,, \qquad \mathbf{C}\dot{\mathbf{v}} = \mathbf{r}(u_1, \ldots, u_n) := \begin{bmatrix} u_1(u_2 + u_1) \\ \vdots \\ u_i(u_{i-1} + u_{i+1}) \\ \vdots \\ u_n(u_n + u_{n-1}) \end{bmatrix} \,,
$$

where $\mathbf{C} \in \mathbb{R}^{n,n}$ is:

$$
\mathbf{C} := \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} \,.
$$

In order to be able to use standard interfaces to RK-SSM for first-order ODEs, collect $\mathbf{u}$ and $\mathbf{v}$ in a $(2n)$-dimensional vector $\mathbf{y} := \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \end{bmatrix}$. Then the final system of first-order ODEs reads

$$
\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_{n+1} \\ \vdots \\ y_{2n} \\ \mathbf{C}^{-1}\mathbf{r}(y_1, \ldots, y_n) \end{bmatrix} \,.
$$

HINT 1 for (6-4.b): The general formulas are given in [Lecture → Def. 6.4.0.9] and their connection with Butcher schemes is made explicit in [Lecture → Eq. (6.4.0.11)]. ⌟

SOLUTION of (6-4.b):

The task boils down to specializing the formulas of [Lecture → Def. 6.4.0.9] for $\mathbf{f}(\mathbf{y}) = \mathbf{M}\mathbf{y}$:

$$\begin{aligned}
\mathbf{k}_1 &:= \mathbf{M}\mathbf{y}_n \,, \\
\mathbf{k}_2 &:= \mathbf{M}\left(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\right) \,, \\
\mathbf{k}_3 &:= \mathbf{M}\left(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\right) \,, \\
\mathbf{k}_4 &:= \mathbf{M}(\mathbf{y}_n + h\mathbf{k}_3) \,, \\
\mathbf{y}_{n+1} &:= \mathbf{y}_n + \frac{h}{6}\mathbf{k}_1 + \frac{h}{3}\mathbf{k}_2 + \frac{h}{3}\mathbf{k}_3 + \frac{h}{6}\mathbf{k}_4 \,, \\
\blacktriangleright \quad \mathbf{y}_{n+1} = \boldsymbol{\Psi}^h\mathbf{y}_n &:= \mathbf{y}_n + h\mathbf{M}\mathbf{y}_n + \frac{h^2}{2}\mathbf{M}^2\mathbf{y}_n + \frac{h^3}{6}\mathbf{M}^3\mathbf{y}_n + \frac{h^4}{24}\mathbf{M}^4\mathbf{y}_n \,,
\end{aligned} \tag{6.4.2}$$

where h is the size of the timestep and $\boldsymbol{\Psi}^h$ stands for the discrete evolution operator underlying the single-step method, see [Lecture → Def. 6.3.1.4].

Remark, *cf.* [Lecture → Rem. 7.1.0.21]. The solution of the ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ can be written as $\mathbf{y}(t) = \exp(\mathbf{M}$ with the **matrix exponential**

$$\exp(\mathbf{X}) := \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \,, \quad \mathbf{X} \in \mathbb{R}^{d,d} \,. \tag{6.4.3}$$

This means that $\mathbf{y}(t_{k+1}) = \exp(\mathbf{M}h)\mathbf{y}(t_k)$ and the evolution operator according to [Lecture → Def. 6.1.4.3 for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ has the form $\boldsymbol{\Phi}^h\mathbf{y} = \exp(\mathbf{M}h)\mathbf{y}$. Comparing with the formula for $\boldsymbol{\Psi}^h$ we notice that $\boldsymbol{\Psi}^h$ can be obtained by truncating the power series for $\boldsymbol{\Phi}^h$ after the fifth term.

SOLUTION of (6-4.c):

Using the Butcher scheme from [Lecture → Ex. 6.4.0.15], the implementation is straightforward from [Lecture → Def. 6.4.0.9] and [Lecture → Eq. (6.4.0.11)].

C++ code 6.4.4: Solution of Sub-problem (6-4.c) ➙ GITHUB

```
2  template <typename Function>
3  Eigen::VectorXd rk4step(Function &&f, double h, Eigen::VectorXd &y0) {
4    Eigen::VectorXd eval(y0.size());
5    Eigen::VectorXd k1 = f(y0);
6    Eigen::VectorXd k2 = f(y0 + h / 2 * k1);
7    Eigen::VectorXd k3 = f(y0 + h / 2 * k2);
8    Eigen::VectorXd k4 = f(y0 + h * k3);
9    eval = y0 + h / 6 * k1 + h / 3 * k2 + h / 3 * k3 + h / 6 * k4;
10   return eval;
11 }
```

SOLUTION of (6-4.d):

N	Error norm	approx. rate
2	0.249353	
4	0.0147421	4.08017
8	0.000783647	4.2336
16	4.19215e-05	4.22444
32	2.34881e-06	4.15769
64	1.375e-07	4.09442
128	8.29035e-09	4.05186
256	5.08553e-10	4.02696
512	3.15424e-11	4.01103
1024	2.05072e-12	3.94309

The rate estimate in the table is obtained based on the quotient of two subsequent error norms.
We observe algebraic convergence with rate 4 : this is expected, since the function $\mathbf{f}$ is smooth and the classical RK4 single-step method is known to be of order 4.

Note the use of EIGEN's sparse matrix format and sparse elimination solvers as explained in .

Also note the use of a lambda function to pass the right-hand-side function to the generic implementation of the RK-SSM in `rk4step()`.

C++ code 6.4.5: Solution of Sub-problem (6-4.d) ➙ GITHUB

```
int main() {
  // PARAMETERS
  double T = 1;
  int n = 5;

  // INITIAL VALUE
  Eigen::VectorXd y0(2 * n);
  for (int i = 0; i < n; ++i) {
    y0(i) = (i + 1.) / n;
    y0(i + n) = -1;
  }

  // SETUP
  double conv_rate = 0;
  std::cout << std::setw(8) << "M" << std::setw(20) << "Error" << std::endl;

  // IMPLEMENTATION OF RIGHT-HAND SIDE f
  // Build tridiagonal C matrix
  Eigen::SparseMatrix<double> C(n, n);
  C.reserve(Eigen::VectorXi::Constant(n, 3));
  C.insert(0, 0) = 2;
  for (int i = 1; i < n; ++i) {
    C.insert(i, i) = 2;
    C.insert(i, i - 1) = -1;
    C.insert(i - 1, i) = -1;
  }
  C.makeCompressed();
  // Compute the right-hand side f
  // The system of ODEs is y' = f(y) with y = [u;v],
  // and f(y) = f([u;v]) = [v;C^{-1}r(u)]
  Eigen::SparseLU<Eigen::SparseMatrix<double>> Csolver;
  Csolver.compute(C);
  auto f = [n, &Csolver](Eigen::VectorXd y) {
    Eigen::VectorXd fy(2 * n);
```

```
    fy.head(n) = y.tail(n);
    Eigen::VectorXd r(n);
    r(0) = y(0) * (y(1) + y(0));
    r(n - 1) = y(n - 1) * (y(n - 1) + y(n - 2));
    for (int i = 1; i < n - 1; ++i) {
      r(i) = y(i) * (y(i - 1) + y(i + 1));
    }
    fy.tail(n) = Csolver.solve(r);
    return fy;
  };

  // COMPUTE AN "EXACT" SOLUTION
  // Use N=2^12 steps to calculate an approximate "exact" solution.
  int N_exact = std::pow(2, 12);  // number of steps
  double h = T / N_exact;         // step size
  Eigen::VectorXd yT_exact = y0;  // initial value
  Eigen::VectorXd y_next;
  for (int step = 0; step < N_exact; step++) {
    y_next = SystemODE::rk4step(f, h, yT_exact);
    yT_exact = y_next;
  }

  // CONVERGENCE ANALYSIS
  // Calculate solution using N=2,...,2^kmax steps.
  int kmax = 10;
  Eigen::VectorXd Error(kmax);
  for (int k = 0; k < kmax; k++) {
    int M = std::pow(2, k + 1);  // number of steps
    double h = T / M;            // step size
    Eigen::VectorXd yT = y0;     // initial value
    // Take N RK4 steps:
    for (int step = 0; step < M; step++) {
      // yT is the solution at time t=h*step
      y_next = SystemODE::rk4step(f, h, yT);
      yT = y_next;
    }
    Error(k) = (yT - yT_exact).norm();
    std::cout << std::setw(8) << M << std::setw(20) << Error(k) << std::endl;
  }

  // Estimate convergence rate
  // Get natural logarithm of M by log(M) = log(2)*log2(N).
  Eigen::VectorXd logM =
      std::log(2) * Eigen::VectorXd::LinSpaced(kmax, 1, kmax);
  Eigen::VectorXd coeffs = polyfit(logM, Error.array().log(), 1);
  conv_rate = coeffs(0);

  std::cout << "Convergence rate: " << std::round(std::abs(conv_rate))
            << std::endl;

  return 0;
}
```

SOLUTION of (6-5.a):

The class **Ode45** can take **Eigen::MatrixXd** as **StateType**. Alternatively, one can transform Matrices to Vectors (and viceversa), using the **Eigen::Map** facility, see .

C++11-code 6.5.2: Implementation of `matode()`. ⇝ GITHUB

```
2  Eigen::MatrixXd matode(const Eigen::MatrixXd &Y0, double T) {
3    // Use the Ode45 class to find an approximation
4    // of the matrix IVP Y' = -(Y - Y')*Y at time T
5    Eigen::MatrixXd YT;
6    // Define the RHS
7    auto F = [](const Eigen::MatrixXd &M) { return -(M - M.transpose()) * M; };
8    Ode45<Eigen::MatrixXd> O(F);
9
10   // Set tolerances
11   O.options.atol = 1e-10;
12   O.options.rtol = 1e-8;
13
14   // Return only matrix at T, (solution is vector
15   // of pairs (y(t_k),t_k) for each step k
16   YT = O.solve(Y0, T).back().first;
17   return YT;
18 }
```

HINT 1 for (6-5.b): A time-dependent differentiable function is constant if and only if its derivative vanishes. ⌟

HINT 2 for (6-5.b): Use the product rule for differentiable matrix-valued functions $t \mapsto \mathbf{A}(t), \mathbf{B}(t)$

$$\frac{d}{dt}(\mathbf{A}(t)\mathbf{B}(t)) = \dot{\mathbf{A}}(t)\mathbf{B}(t) + \mathbf{A}(t)\dot{\mathbf{B}}(t) \,, \tag{6.5.3}$$

which is a special version of the general product rule . ⌟

SOLUTION of (6-5.b):

By the product rule and using the fact that $\vec{Y}$ is a solution of the IVP we obtain

$$\begin{aligned}\frac{d}{dt}(\mathbf{Y}^\top(t)\mathbf{Y}(t)) =& \frac{d}{dt}(\mathbf{Y}^\top(t))\mathbf{Y}(t) + \mathbf{Y}^\top(t)\frac{d}{dt}(\mathbf{Y}(t)) \\ =& (-(\mathbf{Y}(t) - \mathbf{Y}^\top(t))\mathbf{Y}(t))^\top \mathbf{Y}(t) + \mathbf{Y}^\top(t)(-(\mathbf{Y}(t) - \mathbf{Y}^\top(t))\mathbf{Y}(t)) \\ =& -\mathbf{Y}^\top(t)\mathbf{Y}^\top(t)\mathbf{Y}(t) + \mathbf{Y}^\top\mathbf{Y}(t)\mathbf{Y}(t) \\ & -\mathbf{Y}^\top\mathbf{Y}(t)\mathbf{Y}(t) + \mathbf{Y}^\top(t)\mathbf{Y}^\top(t)\mathbf{Y}(t) = 0.\end{aligned}$$

implying that the map is constant.

SOLUTION of (6-5.c):

Let $\mathbf{Y}_M$ be the output of `matode()`. We compute $\mathbf{Y}_0^\top \mathbf{Y}_0 - \mathbf{Y}_M^\top \mathbf{Y}_M$ and check if the (Frobenius) norm is smaller than $\|\mathbf{Y}\|_F$ (Frobenius norm) times machine precision `eps`. Even for an orthogonal matrix, we have to take into account round-off errors!

C++11-code 6.5.4: Implementation checking invariant. ↬ GITHUB

```
2  bool checkinvariant(const Eigen::MatrixXd &M, double T) {
3    // Check if Y'*Y is preserved at the time T by matode.
4    Eigen::MatrixXd N = matode(M, T);
5
6    if ((N.transpose() * N - M.transpose() * M).norm() <
7        10 * std::numeric_limits<double>::epsilon() * M.norm()) {
8      return true;
9    }
10
11   return false;
12 }
```

SOLUTION of (6-5.d):

Apart from realizing that the formal inverse matrix in (6.5.5) must be implemented as the solution of a linear system of equations, the implementation is straightforward.

C++ code 6.5.6: Implementation of `cvgDiscreteGradientMethod()` ↬ GITHUB

```
2  double cvgDiscreteGradientMethod() {
3    // Compute the fitted convergence rate of the Discrete
4    // gradient method. Also tabulate the values M and the errors.
5    double conv_rate = 0;
6    double T = 1.0;
7    // initial value
8    Eigen::MatrixXd Y0 = Eigen::MatrixXd::Zero(5, 5);
9    Y0(4, 0) = 1;
10   for (unsigned int i = 0; i < 4; ++i) {
11     Y0(i, i + 1) = 1;
12   }
13   // reference solution
14   Eigen::MatrixXd Y_ex = matode(Y0, T);
15
16   // define the rhs
17   auto F = [](const Eigen::MatrixXd &M) { return -(M - M.transpose()) * M; };
18
19   Eigen::MatrixXd I = Eigen::MatrixXd::Identity(5, 5);
20   Eigen::ArrayXd MM(8);
21   Eigen::ArrayXd err(8);
22
23   std::cout << "Error for equidistant steps:" << std::endl;
24   std::cout << "M"
25             << "\t"
26             << "Error" << std::endl;
27   for (unsigned int i = 0; i < 8; ++i) {
28     unsigned int M = 10 * std::pow(2, i);
29     double h = T / M;
30     MM(i) = M;
31     Eigen::MatrixXd Y = Y0;
32     for (unsigned int j = 0; j < M; ++j) {
33       Eigen::MatrixXd Ystar = Y + 0.5 * h * F(Y);
34
35       Eigen::MatrixXd Yinc = 0.5 * h * (Ystar - Ystar.transpose());
36       Y = (I + Yinc).lu().solve((I - Yinc) * Y);
37     }
38     err(i) = (Y - Y_ex).norm();
39     std::cout << M << "\t" << err(i) << std::endl;
40   }
41
42   // compute fitted rate
43   Eigen::VectorXd coeffs = polyfit(MM.log(), err.log(), 1);
44   conv_rate = -coeffs(0);
45   return conv_rate;
46 }
```

M	$\|\mathbf{Y}_M - \mathbf{Y}(T)\|_F$
10	0.00260457
20	0.000645272
40	0.000160726
80	4.00747e-05
160	9.9619e-06
320	2.44364e-06
640	5.84909e-07
1280	1.98872e-07

◁ Doubling the number of timesteps leads to a reduction of the error by roughly a factor of four.

This hints at *algbraic convergence* in the number of timesteps with rate 2: $\mathrm{err} = \mathrm{O}(\mathrm{M}^{-2}) = \mathrm{O}(\mathrm{h}^2)$ for $M \to \infty$, where $h > 0$ is the size of the uniform timestep.

We conclude that the discrete gradient method is a single-step method *of order* 2.

SOLUTION of (6-6.a):

Let us find an expression for the order of convergence. Set $M_k = 2^k$. We readily derive

$$\frac{E_{M_{k-1}}}{E_{M_k}} \approx \frac{C2^{-(k-1)r_k}}{C2^{-kr_k}} = 2^{r_k} \quad , \quad r_k \approx \log\left(\frac{E_{M_{k-1}}}{E_{M_k}}\right)/\log(2) \, .$$

A reasonable approximation of the order of convergence is given by

$$r \approx \frac{1}{\#K} \sum_{k \in K} r_k, \qquad K = \{k = 1, \ldots, 12 : E_{M_k} > 10^{-14}\}. \tag{6.6.2}$$

C++11-code 6.6.3: Implementation of `testCvgRKSSM` ↪ GITHUB

```
template <class Function>
void testCvgRKSSM(const Function &f, double T, double y0,
                  const Eigen::MatrixXd &A, const Eigen::VectorXd &b) {
  // Helper object carrying out the actual explicit RK-SSM
  RKIntegrator<double> rk(A, b);
  // Vector for collecting errors
  std::vector<double> error(15);
  // Vector for storing rate estimates
  std::vector<double> order(14);

  double sum = 0;
  int count = 0;
  bool test = true;
  // Reference numerical solution obtained with 2^15 timesteps
  std::vector<double> y_exact = rk.solve(f, T, y0, std::pow(2, 15));

  for (int k = 0; k < 12; k++) {
    // Number of timesteps
    int M = std::pow(2, k + 1);
    // Solve IVP
    std::vector<double> y1 = rk.solve(f, T, y0, M);
    // Error at final time
    error[k] = std::abs(y1[M] - y_exact[std::pow(2, 15)]);

    std::cout << std::left << std::setfill(' ') << std::setw(3)
              << "M = " << std::setw(7) << M << std::setw(8)
              << "Error = " << std::setw(13) << error[k];

    if (error[k] < 1e-14) {
      test = false;
    }
    if (k > 0 && test) {
      order[k - 1] = std::log2(error[k - 1] / error[k]);
      std::cout << std::left << std::setfill(' ') << std::setw(10)
                << "Approximated order = " << order[k - 1] << std::endl;
      sum += order[k - 1];
      count = k;
    } else
      std::cout << std::endl;
  }
  std::cout << "Average approximated order = " << sum / count << std::endl
```

```
43                 << std::endl;
44 }
```

HINT 1 for (6-6.b): For a scalar autonomous ODE the family of solutions can be determined by direct integration

$$\dot{y} = f(y) \quad \Rightarrow \quad \int_0^t \frac{\dot{y}(\tau)}{f(y(\tau))}\,\mathrm{d}\tau = t \quad \Rightarrow \quad \int_{y(0)}^{y(t)} \frac{1}{f(\sigma)}\,\mathrm{d}\sigma = t\,. \tag{6.6.6}$$

Thus, if $t \mapsto G(t)$ is a principal of f^{-1}, that is $\dot{G} = f^{-1}$, then

$$G(y(t)) = G(y(0)) + t \quad \Rightarrow \quad y(t) = G^{-1}(G(y(0)) + t)\,, \tag{6.6.7}$$

provided that G is monotone in the possible range of values of y. ⌟

HINT 2 for (6-6.b): For the logistic ODE the solution is given in [Lecture → Eq. (6.1.2.3)].

HINT 3 for (6-6.b): For (6.6.5) you have to distinguish cases depending on the sign of $1.1 - y(t)$. ⌟

SOLUTION of (6-6.b):

As far as (6.6.4) is concerned, the solution is $y(t) = (1 + e^{-t})^{-1}$ (see [Lecture $\rightarrow$ Eq. (6.1.2.3)]).

Let us now consider (6.6.5). Because of the absolute value on the right hand side of the differential equation, we have to distinguish two cases $y(t) < 1.1$ and $y(t) > 1.1$. Since the initial condition is given by $y(0) = 1 < 1.1$, we start with the case $y(t) < 1.1$. For $y(t) < 1.1$, the differential equation is $\dot{y} = 2.1 - y$. Separation of variables

$$\int_1^{y(t)} \frac{1}{2.1 - \tilde{y}} d\,\tilde{y} = \int_0^t d\,\tilde{t}$$

yields the solution

$$y(t) = 2.1 - 1.1e^{-t}, \quad \text{for } y(t) < 1.1.$$

For $y(t) > 1.1$, the differential equation is given by $\dot{y} = y - 0.1$ with initial condition $y(\ln(\frac{11}{10})) = 1.1$, where the initial time t^* was derived from the condition $y(t^*) = 2.1 - 1.1e^{-t^*} \stackrel{!}{=} 1.1$. Separation of variables yields the solution for this IVP

$$y(t) = \tfrac{10}{11}e^t + 0.1.$$

Together, the solution of the initial IVP is given by

$$y(t) = \begin{cases} 2.1 - 1.1e^{-t}, & \text{for } t \leq \ln(1.1) \\ \frac{10}{11}e^t + 0.1, & \text{for } t > \ln(1.1). \end{cases}$$

SOLUTION of (6-6.c):

The following code is self-explanatory. We encode the various RK-SSMs through their Butcher schemes.

C++11-code 6.6.8: Implementation of `cmpCvgRKSSM` ↪ GITHUB

```
void cmpCvgRKSSM() {
  // Construct data for Butcher schemes
  Eigen::MatrixXd A1 = Eigen::MatrixXd::Zero(1, 1);
  Eigen::VectorXd b1(1);
  b1 << 1;

  Eigen::MatrixXd A2 = Eigen::MatrixXd::Zero(2, 2);
  A2(1, 0) = 1;
  Eigen::VectorXd b2(2);
  b2 << .5, .5;

  Eigen::MatrixXd A3 = Eigen::MatrixXd::Zero(3, 3);
  A3(1, 0) = .5;
  A3(2, 0) = -1;
  A3(2, 1) = 2;
  Eigen::VectorXd b3(3);
  b3 << 1. / 6, 2. / 3, 1. / 6;

  Eigen::MatrixXd A4 = Eigen::MatrixXd::Zero(4, 4);
  A4(1, 0) = .5;
  A4(2, 1) = .5;
  A4(3, 2) = 1;
  Eigen::VectorXd b4(4);
  b4 << 1. / 6, 1. / 3, 1. / 3, 1. / 6;

  // First ODE
  std::cout << std::endl
            << "1. ODE y' = (1-y)y, y(0)=.5" << std::endl
            << std::endl;
  double T = 1;
  auto f = [](double y) { return (1. - y) * y; };
  double y0 = .5;

  std::cout << "Explicit Euler" << std::endl << std::endl;
  testCvgRKSSM(f, T, y0, A1, b1);
  std::cout << "Trapezoidal rule" << std::endl << std::endl;
  testCvgRKSSM(f, T, y0, A2, b2);
  std::cout << "RK order 3" << std::endl << std::endl;
  testCvgRKSSM(f, T, y0, A3, b3);
  std::cout << "Classical RK order 4" << std::endl << std::endl;
  testCvgRKSSM(f, T, y0, A4, b4);

  // Second ODE
  std::cout << std::endl
            << "2. ODE y' = |1.1 - y| + 1, y(0)=1" << std::endl
            << std::endl;
  auto f2 = [](double y) { return std::abs(1.1 - y) + 1.; };
  y0 = 1;

  std::cout << "Explicit Euler" << std::endl << std::endl;
```

```
52    testCvgRKSSM(f2, T, y0, A1, b1);
53    std::cout << "Trapezoidal rule" << std::endl << std::endl;
54    testCvgRKSSM(f2, T, y0, A2, b2);
55    std::cout << "RK order 3" << std::endl << std::endl;
56    testCvgRKSSM(f2, T, y0, A3, b3);
57    std::cout << "Classical RK order 4" << std::endl << std::endl;
58    testCvgRKSSM(f2, T, y0, A4, b4);
59  }
```

The code produces the following output:

```
1
2  1. ODE y' = (1-y)y, y(0)=.5
3
4  Explicit Euler
5  ------------------------------------------------------------
6  N = 2       Error = 0.0111282
7  N = 4       Error = 0.00577587    Approximated order = 0.946109
8  N = 8       Error = 0.00292368    Approximated order = 0.982252
9  N = 16      Error = 0.00146886    Approximated order = 0.993093
10 N = 32      Error = 0.00073573    Approximated order = 0.997446
11 N = 64      Error = 0.000367898   Approximated order = 0.999868
12 N = 128     Error = 0.000183685   Approximated order = 1.00208
13 N = 256     Error = 9.15057e-05   Approximated order = 1.0053
14 N = 512     Error = 4.53983e-05   Approximated order = 1.01122
15 N = 1024    Error = 2.23403e-05   Approximated order = 1.02299
16 N = 2048    Error = 1.08102e-05   Approximated order = 1.04726
17 N = 4096    Error = 5.04482e-06   Approximated order = 1.09951
18 Average approximated order = 1.00974
19
20 Trapezoidal rule
21 ------------------------------------------------------------
22 N = 2       Error = 0.00288256
23 N = 4       Error = 0.000677593   Approximated order = 2.08886
24 N = 8       Error = 0.000163389   Approximated order = 2.05211
25 N = 16      Error = 4.00683e-05   Approximated order = 2.02778
26 N = 32      Error = 9.91824e-06   Approximated order = 2.0143
27 N = 64      Error = 2.46712e-06   Approximated order = 2.00726
28 N = 128     Error = 6.15214e-07   Approximated order = 2.00367
29 N = 256     Error = 1.53601e-07   Approximated order = 2.0019
30 N = 512     Error = 3.83689e-08   Approximated order = 2.00118
31 N = 1024    Error = 9.58214e-09   Approximated order = 2.00152
32 N = 2048    Error = 2.38813e-09   Approximated order = 2.00447
33 N = 4096    Error = 5.89957e-10   Approximated order = 2.0172
34 Average approximated order = 2.02002
35
36 RK order 3
37 ------------------------------------------------------------
38 N = 2       Error = 0.000107796
39 N = 4       Error = 1.5402e-05    Approximated order = 2.80712
```

```
N = 8       Error = 2.03327e-06  Approximated order = 2.92124
N = 16      Error = 2.60614e-07  Approximated order = 2.96381
N = 32      Error = 3.29725e-08  Approximated order = 2.98258
N = 64      Error = 4.14608e-09  Approximated order = 2.99144
N = 128     Error = 5.19776e-10  Approximated order = 2.99579
N = 256     Error = 6.50568e-11  Approximated order = 2.99812
N = 512     Error = 8.12717e-12  Approximated order = 3.00088
N = 1024    Error = 1.00275e-12  Approximated order = 3.01879
N = 2048    Error = 1.13132e-13  Approximated order = 3.14789
N = 4096    Error = 6.55032e-15
Average approximated order = 2.98277

Classical RK order 4
------------------------------------------------------------
N = 2       Error = 1.11709e-05
N = 4       Error = 7.18023e-07  Approximated order = 3.95957
N = 8       Error = 4.51194e-08  Approximated order = 3.99221
N = 16      Error = 2.82329e-09  Approximated order = 3.9983
N = 32      Error = 1.76508e-10  Approximated order = 3.99957
N = 64      Error = 1.10368e-11  Approximated order = 3.99934
N = 128     Error = 6.95333e-13  Approximated order = 3.98848
N = 256     Error = 4.74065e-14  Approximated order = 3.87455
N = 512     Error = 3.9968e-15
N = 1024    Error = 3.77476e-15
N = 2048    Error = 8.88178e-16
N = 4096    Error = 1.33227e-15
Average approximated order = 3.97314

2. ODE y' = |1.1 - y| + 1, y(0)=1

Explicit Euler
------------------------------------------------------------
N = 2       Error = 0.296135
N = 4       Error = 0.176213     Approximated order = 0.748934
N = 8       Error = 0.104911     Approximated order = 0.74815
N = 16      Error = 0.0568369    Approximated order = 0.88427
N = 32      Error = 0.0303542    Approximated order = 0.904932
N = 64      Error = 0.0153476    Approximated order = 0.983887
N = 128     Error = 0.00770812   Approximated order = 0.993558
N = 256     Error = 0.00385358   Approximated order = 1.00018
N = 512     Error = 0.0019175    Approximated order = 1.00697
N = 1024    Error = 0.00094465   Approximated order = 1.02137
N = 2048    Error = 0.0004575    Approximated order = 1.04601
N = 4096    Error = 0.000213549  Approximated order = 1.09921
Average approximated order = 0.948861

Trapezoidal rule
------------------------------------------------------------
N = 2       Error = 0.0272722
```

```
N = 4       Error = 0.019933      Approximated order = 0.452272
N = 8       Error = 0.00315605    Approximated order = 2.65897
N = 16      Error = 0.00101849    Approximated order = 1.63169
N = 32      Error = 0.000278879   Approximated order = 1.86872
N = 64      Error = 4.51313e-05   Approximated order = 2.62744
N = 128     Error = 8.4172e-07    Approximated order = 5.74464
N = 256     Error = 2.80199e-06   Approximated order = -1.73504
N = 512     Error = 4.60735e-08   Approximated order = 5.92637
N = 1024    Error = 1.75031e-07   Approximated order = -1.9256
N = 2048    Error = 5.44107e-09   Approximated order = 5.00758
N = 4096    Error = 1.06517e-08   Approximated order = -0.969126
Average approximated order = 1.93526

RK order 3
------------------------------------------------------------
N = 2       Error = 0.0147288
N = 4       Error = 0.0028711     Approximated order = 2.35897
N = 8       Error = 0.000217491   Approximated order = 3.72257
N = 16      Error = 0.000773451   Approximated order = -1.83035
N = 32      Error = 3.17718e-05   Approximated order = 4.60549
N = 64      Error = 1.3791e-05    Approximated order = 1.20402
N = 128     Error = 3.99984e-06   Approximated order = 1.78571
N = 256     Error = 9.977e-07     Approximated order = 2.00327
N = 512     Error = 2.46644e-07   Approximated order = 2.01617
N = 1024    Error = 6.60285e-08   Approximated order = 1.90127
N = 2048    Error = 1.58186e-08   Approximated order = 2.06147
N = 4096    Error = 3.34817e-09   Approximated order = 2.24017
Average approximated order = 2.00625

Classical RK order 4
------------------------------------------------------------
N = 2       Error = 0.01995
N = 4       Error = 0.0021045     Approximated order = 3.24484
N = 8       Error = 0.000823604   Approximated order = 1.35346
N = 16      Error = 0.000651961   Approximated order = 0.337164
N = 32      Error = 3.42114e-05   Approximated order = 4.25224
N = 64      Error = 1.41017e-05   Approximated order = 1.27861
N = 128     Error = 4.03905e-06   Approximated order = 1.80378
N = 256     Error = 9.92776e-07   Approximated order = 2.02447
N = 512     Error = 2.50338e-07   Approximated order = 1.98759
N = 1024    Error = 6.55673e-08   Approximated order = 1.93283
N = 2048    Error = 1.58282e-08   Approximated order = 2.05048
N = 4096    Error = 3.34702e-09   Approximated order = 2.24154
Average approximated order = 2.04609
```

The convergence orders of the explicit Euler and Runge–Kutta 2 methods are as expected, but we do not see any relevant improvement in the convergence orders of RK3 and RK4. This is due to the fact that the right hand side of the IVP is not continuously differentiable: the convergence theory breaks down due to *lack of smoothness* of exact solutions.

SOLUTION of (6-7.a):

By construction, $\mathbf{\Phi}$ is a function with values in $\mathbb{R}^N$. Thus, $\mathsf{D}_{\mathbf{y}}\mathbf{\Phi}$ has values in $\mathbb{R}^{N,N}$, and so the state space of (6.7.2) is $\mathbb{R}^{N,N}$, a space of matrices.

SOLUTION of (6-7.b):

Writing $\mathbf{y} = [u, v]^T$, the map $\mathbf{f}$ associated to (6.7.4) is $\mathbf{f}(\mathbf{y}) = [(2-v)u, (u-1)v]^T$. Therefore

$$\mathrm{D}\mathbf{f}(\mathbf{y}) = \begin{bmatrix} 2-v & -u \\ v & u-1 \end{bmatrix}.$$

Thus, (6.7.4) becomes

$$\dot{\mathbf{W}} = \begin{bmatrix} 2-v(t) & -u(t) \\ v(t) & u(t)-1 \end{bmatrix} \mathbf{W}. \tag{6.7.5}$$

This is a non-autonomous ODE.

SOLUTION of (6-7.c):

Let $\mathbf{F}$ be defined by

$$\mathbf{F}(\mathbf{y}) := \mathbf{\Phi}(T_P, \mathbf{y}) - \mathbf{y}. \tag{6.7.6}$$

If the solution of the system (6.7.4) with initial value $\mathbf{y}_0$ has period T_P, then we have $\mathbf{\Phi}(T_P, \mathbf{y}_0) = \mathbf{y}_0$, so $\mathbf{F}(\mathbf{y}_0) = 0$.

SOLUTION of (6-7.d):

By definition of $\mathbf{W}$ we immediately obtain

$$\mathrm{D}\mathbf{F}(\mathbf{y}) = \mathbf{W}(T_P, \mathbf{y}) - \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix.

HINT 1 for (6-7.e): Study [Lecture → § 6.1.3.3] again. Also take a look at [Lecture → Fig. 285]. ⌟

SOLUTION of (6-7.e):

If $\mathbf{y}_0 \in \mathbb{R}^2_+$ is a solution, then every state on the trajectory $\mathbf{y}_0$ will also be a solution. Only if the trajectory starting in $\mathbf{y}_0$ collapses to a point, that is, if $\mathbf{y}_0$ is a stationary point, $\mathbf{f}(\mathbf{y}_0) = 0$, and there is no other trajectory with the prescribed period, we can expect uniqueness.

HINT 1 for (6-7.f): As in (6.7.3), both ODEs (for $\mathbf{\Phi}$ and $\mathbf{W}$) must be combined into a single autonomous differential equation on the state space $D \times \mathbb{R}^{d\times d}$. ⌟

SOLUTION of (6-7.f):

Writing $\mathbf{w} = [u, v, W_{11}, W_{21}, W_{12}, W_{22}]^T$, by (6.7.4) and (6.7.5) we have that

$$\begin{aligned} \dot{w}_1 &= (2 - w_2)w_1 & \dot{w}_3 &= (2 - w_2)w_3 - w_1 w_4 & \dot{w}_5 &= (2 - w_2)w_5 - w_1 w_6 \\ \dot{w}_2 &= (w_1 - 1)w_2 & \dot{w}_4 &= w_2 w_3 + (w_1 - 1)w_4 & \dot{w}_6 &= w_2 w_5 + (w_1 - 1)w_6 \end{aligned}$$

with initial conditions

$$w_1(0) = u_0, \quad w_2(0) = v_0, \quad w_3(0) = 1, \quad w_4(0) = 0, \quad w_5(0) = 0, \quad w_6(0) = 1,$$

since $\mathbf{W}(0, \mathbf{y}) = \mathrm{D}_{\mathbf{y}}\mathbf{\Phi}(0, \mathbf{y}) = \mathrm{D}_{\mathbf{y}}\mathbf{y} = \mathbf{I}$.

The class **Ode45** has to be instantiated with the right lambda function describing the right-hand-side function of the compound ODE.

C++11-code 6.7.7: Implementation of `PhiAndW()` ⇝ GITHUB

```
2  std::pair<Eigen::Vector2d, Eigen::Matrix2d> PhiAndW(double u0, double v0,
3                                                      double T) {
4    // Save the values of Phi and W at time T in PaW.first and PaW.second resp.
5    std::pair<Eigen::Vector2d, Eigen::Matrix2d> PaW;
6
7    auto f = [](const Eigen::VectorXd& w) {
8      Eigen::VectorXd temp(6);
9      temp(0) = (2. - w(1)) * w(0);
10     temp(1) = (w(0) - 1.) * w(1);
11     temp(2) = (2. - w(1)) * w(2) - w(0) * w(3);
12     temp(3) = w(1) * w(2) + (w(0) - 1.) * w(3);
13     temp(4) = (2. - w(1)) * w(4) - w(0) * w(5);
14     temp(5) = w(1) * w(4) + (w(0) - 1.) * w(5);
15     return temp;
16   };
17
18   Eigen::VectorXd w0(6);
19   w0 << u0, v0, 1., 0, 0, 1.;
20
21   // Construct ode solver with r.h.s
22   Ode45<Eigen::VectorXd> O(f);
23   // Set options
24   O.options.rtol = 1e-14;
25   O.options.atol = 1e-12;
26   // Solve ODE
27   auto sol = O.solve(w0, T);
28   // Extract needed component
29   Eigen::VectorXd wT = sol.back().first;
30
31   PaW.first << wT(0), wT(1);
32   PaW.second << wT(2), wT(4), wT(3), wT(5);
33   return PaW;
34 }
```

HINT 1 for (6-7.g): The correct solutions are $u(0) \approx 3.110$ and $v(0) = 2.081$. ⌟

SOLUTION of (6-7.g):

Remember that the Newton iteration for $\mathbf{F}(\mathbf{y}) = 0$ reads:

$$\vec{y}^{(k+1)} = \vec{y}^{(k)} + \mathrm{D}\vec{F}(\vec{x}^{(k)})^{-1}\vec{F}(\vec{x}^{(k)}) \,. \tag{6.7.8}$$

The evaluation of both $\mathrm{D}\mathbf{F}$ and $\mathbf{F}$ for a given initial state is possible through the function `PhiAndW()`.

C++11-code 6.7.9: Implementation of `initcondlv_main.cc`. ⇝ GITHUB

```
  Eigen::Vector2d y(3, 2);  // Initial guess
  double T = 5;             // Period

  // Compute Phi and W from initial guess
  std::pair<Eigen::Vector2d, Eigen::Matrix2d> PaW =
      InitCondLV::PhiAndW(y(0), y(1), T);
  Eigen::Vector2d F = PaW.first - y;  // Value of F at initial guess
  Eigen::Matrix2d DF;                 // Declare Jacobian of F

  // Until stopping condition
  while (F.norm() > 1e-5) {  // Residual based termination
    // Calculate Jacobian
    DF = PaW.second - Eigen::MatrixXd::Identity(2, 2);
    // Use Newton iteration
    y = y - DF.lu().solve(F);
    // Test current guess
    PaW = InitCondLV::PhiAndW(y(0), y(1), T);
    // Get value of F at the current guess
    F = PaW.first - y;
  }

  // Calculate the evolution of the Lotka-Volterra ODE up to
  // time 100, using the initial value y, found by the Newton iterations above.
  // If the evolution is indeed 5-periodic, then the solution at time 100
  // should have the value y. Print a warning message if this is not the case.
  PaW = InitCondLV::PhiAndW(y(0), y(1), 100);
  if ((y - PaW.first).norm() > 1e-5) {
    std::cout << "Warning: Solution not periodic, y(100) != y(0)" << std::endl;
  }

  // Testing obtained solution against reference "exact" solution
  Eigen::Vector2d ref(3.1098751029156, 2.08097564048345);
  double tol = 1.0e-8;
  double error = (ref - y).lpNorm<Eigen::Infinity>();
  if (std::abs(error) > tol) {
    std::cout
        << " Error w.r.t. to reference solution is GREATER than tol=1.0e-8."
        << std::endl;
  } else {
    std::cout
        << " Error w.r.t. to reference solution is SMALLER than tol=1.0e-8."
        << std::endl;
  }
```

HINT 1 for (6-8.a): Apply the chain rule, see , taking into account the ODE (6.8.1) in order to substitute derivatives. ⌟

SOLUTION of (6-8.a):

For the first time derivative of $\mathbf{y}$, we just use the differential equation: $\frac{d\mathbf{y}}{dt}(t) = \mathbf{y}'(t) = \mathbf{f}(\mathbf{y}(t))$. For the second derivative, we use the previous equation, apply the chain rule and then once again insert the ODE:

$$\frac{d^2\mathbf{y}}{dt^2}(t) = \frac{d}{dt}\mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}'(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)).$$

Here $\mathbf{Df}(\mathbf{y}(t))$ is the Jacobian of $\mathbf{f}$ evaluated at $\mathbf{y}(t)$.

HINT 1 for (6-8.b): To gain confidence, it is advisable to consider the scalar case $d = 1$ first, where $f : \mathbb{R} \to \mathbb{R}$ is a real valued function. ⌟

HINT 2 for (6-8.b): This time we have to apply both the product rule and the chain rule to the expression derived in the previous sub-problem. ⌟

HINT 3 for (6-8.b): Refresh yourself on the abstract concept of a derivative discussed in .

Relevant for the case $d > 1$ is the fact that the first derivative of $\mathbf{f}$ is a linear mapping $\mathbf{Df}(\mathbf{y}_0) : \mathbb{R}^n \to \mathbb{R}^n$. This linear mapping is applied by multiplying the argument with the Jacobian of $\mathbf{f}$.

Similarly, the second derivative is a *bilinear* mapping $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$. The i-th component of $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{v}, \mathbf{v})$ is given by

$$\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{v}, \mathbf{v})_i = \mathbf{v}^\top \mathbf{Hf}_i(\mathbf{y}_0)\mathbf{v},$$

where $\mathbf{Hf}_i(\mathbf{y}_0)$ is the Hessian of the i-th component of $\mathbf{f}$ evaluated at $\mathbf{y}_0$. ⌟

SOLUTION of (6-8.b):

We follow the hint and first have a look at the scalar case. Here, the Jacobian reduces to $f'(y(t))$. Thus, given the previous sub-problem, we have to calculate

$$\frac{d^3y}{dt^3}(t) = \frac{d}{dt}\left[f'(y(t))f(y(t))\right] .$$

Product rule and chain rule give

$$\frac{d}{dt}\left[f'(y(t))f(y(t))\right] = f''(y(t))y'(t)f(y(t)) + f'(y(t))f'(y(t))y'(t) .$$

Inserting the ODE $y'(t) = f(y(t))$ once again yields

$$\frac{d^3y}{dt^3}(t) = f''(y(t))f(y(t))^2 + f'(y(t))^2 f(y(t)) .$$

This is already equivalent to (6.8.3). The first term is quadratic in $f(y(t))$ and involves the second derivative of f, whereas the second term involves the first derivative of f in quadratic form.

To understand the formula for higher dimensions, we verify it component-wise. For each component $y_i(t)$ we have a function $f_i : \mathbb{R}^n \to \mathbb{R}$.

For the first derivative of $y_i(t)$, this is straightforward: $y_i'(t) = f_i(\mathbf{y}(t))$. Thus, the second derivative is:

$$\begin{aligned} y_i''(t) &= \frac{d}{dt}[f_i(\mathbf{y}(t))] = \sum_{j=1}^{n} \partial_{y_j} f_i(\mathbf{y}(t))\; y_j'(t) \\ &= [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{y}'(t) = [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{f}(\mathbf{y}(t)) . \end{aligned}$$

Building up all components gives us what we have already obtained in (6-8.a): $\mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t))$, with the Jacobian $\mathbf{Df}(\mathbf{y}(t))$ which contains the gradients of the components of f row-wise.

Now, we apply product rule to $y_i''(t)$ to obtain

$$y_i'''(t) = \left\{\frac{d}{dt}[\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top\right\} \cdot \mathbf{y}'(t) + [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{y}''(t) .$$

The second term of the sum again builds up to

$$\mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t))^2 \cdot \mathbf{f}(\mathbf{y}(t)) .$$

For the first term, we write the scalar product of the two vectors as a sum and then interchange the order of derivatives. This is possible as long as functions are sufficiently differentiable:

$$\left\{\frac{d}{dt}[\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top\right\} \cdot \mathbf{y}'(t) = \sum_{j=1}^{n} \left[\partial_{y_j} \frac{d}{dt} f_i(\mathbf{y}(t))\right] f_j(\mathbf{y}(t)) .$$

Now we apply the chain rule:

$$\begin{aligned}\sum_{j=1}^{n}\left[\partial_{y_j}\frac{d}{dt}f_i(\mathbf{y}(t))\right]f_j(\mathbf{y}(t)) &= \sum_{j=1}^{n}\left[\partial_{y_j}\sum_{l=1}^{n}\partial_{y_l}f_i(\mathbf{y}(t))y_l'(t)\right]f_j(\mathbf{y}(t)) \\ &= \sum_{j,l=1}^{n}\left[\partial_{y_j}\partial_{y_l}f_i(\mathbf{y}(t))\right]f_l(\mathbf{y}(t))f_j(\mathbf{y}(t)) \\ &= \mathbf{f}(\mathbf{y}(t))^{\top}\cdot\mathbf{H}f_i(\mathbf{y}(t))\cdot\mathbf{f}(\mathbf{y}(t))\end{aligned}$$

This is the desired result.

SOLUTION of (6-8.c):

For our particular example, recasting it into the standard form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ of an autonomous ODE, we have

$$\mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix} \quad , \quad \mathbf{f}(\mathbf{y}) = \begin{bmatrix} (\alpha_1 - \beta_1 y_2) y_1 \\ -(\alpha_2 - \beta_2 y_1) y_2 \end{bmatrix} , \tag{6.8.7}$$

and the derivatives

$$\mathbf{D}\mathbf{f}(\mathbf{y}) = \begin{bmatrix} \alpha_1 - \beta_1 y_2 & -\beta_1 y_1 \\ \beta_2 y_2 & -(\alpha_2 - \beta_2 y_1) \end{bmatrix},$$
$$\mathbf{H}f_1(\mathbf{y}) = \begin{bmatrix} 0 & -\beta_1 \\ -\beta_1 & 0 \end{bmatrix}, \qquad \mathbf{H}f_2(\mathbf{y}) = \begin{bmatrix} 0 & \beta_2 \\ \beta_2 & 0 \end{bmatrix} .$$

C++ code 6.8.8: Solution for SolvePredPreyTaylor ↪ GITHUB

```
Eigen::Vector2d PredPreyModel::f(const Eigen::Vector2d& y) const {
  return {(alpha1_ - beta1_ * y(1)) * y(0), (beta2_ * y(0) - alpha2_) * y(1)};
}

Eigen::Vector2d PredPreyModel::df(const Eigen::Vector2d& y,
                                  const Eigen::Vector2d& z) const {
  Eigen::Matrix2d Df;
  Df << alpha1_ - beta1_ * y(1), -beta1_ * y(0), beta2_ * y(1),
      -alpha2_ + beta2_ * y(0);
  return Df * z;
}

Eigen::Vector2d PredPreyModel::d2f(const Eigen::Vector2d& y,
                                   const Eigen::Vector2d& z) const {
  Eigen::Matrix2d H1, H2;
  H1 << 0, -beta1_, -beta1_, 0;
  H2 << 0, beta2_, beta2_, 0;
  return {z.transpose() * H1 * z, z.transpose() * H2 * z};
}

std::vector<Eigen::Vector2d> SolvePredPreyTaylor(const PredPreyModel& model,
                                                 double T,
                                                 const Eigen::Vector2d& y0,
                                                 unsigned int M) {
  std::vector<Eigen::Vector2d> res;
  res.reserve(M + 1);

  Eigen::Vector2d y = y0;
  double h = T / M;
  res.push_back(y);

  for (unsigned int k = 0; k < M; ++k) {
    // evaluate terms for taylor step.
    auto fy = model.f(y);
    auto dfyfy = model.df(y, fy);
    auto df2yfy = model.df(y, dfyfy);
    auto d2fyfy = model.d2f(y, fy);

    // evaluate taylor expansion to compute update
```

```
41      y = y + h * fy + 0.5 * h * h * dfyfy +
42          1.0 / 6.0 * h * h * h * (df2yfy + d2fyfy);
43
44      // save new state:
45      res.push_back(y);
46    }
47    return res;
48  }
```

SOLUTION of (6-8.d):

C++ code 6.8.9: Solution of (6-8.d) ➙ GITHUB

```
double TestCvgTaylorMethod() {
  // initialize parameters for the model:
  double T = 10;               // final time
  Eigen::Vector2d y0(100, 5);  // initial condition
  Eigen::Vector2d yex(0.319465882659820,
                      9.730809352326228);  // reference solution
  double alpha1 = 3.0;
  double alpha2 = 2.0;
  double beta1 = 0.1;
  double beta2 = 0.1;
  PredPreyModel model(alpha1, alpha2, beta1, beta2);

  // Initialize parameters for the convergence study
  unsigned int M0 = 128;    // Minimum number of timesteps
  unsigned int numRef = 8;  // Number of refinements

  // Convergence study
  Eigen::ArrayXd error(numRef);
  Eigen::ArrayXd M(numRef);
  for (unsigned int i = 0; i < numRef; ++i) {
    M(i) = std::pow(2, i) * M0;
    auto res = SolvePredPreyTaylor(model, T, y0, M(i));
    error(i) = (res.back() - yex).norm();
  }

  PrintErrorTable(M, error);

  // calculate linear regression line:  log(error)   c0 + c1*log(M)
  Eigen::MatrixXd A(numRef, 2);
  A.col(0) = Eigen::VectorXd::Ones(numRef);
  A.col(1) = M.log();
  Eigen::VectorXd logError = error.log();
  Eigen::Vector2d coeffs = A.householderQr().solve(logError);

  // estimated convergence rate:  -c1
  return -coeffs(1);
}
```

M	error	rate
128	6.49399	
256	0.533867	3.60455
512	0.0607558	3.13539
1024	0.00732692	3.05174
2048	0.000900452	3.02449
4096	0.000111607	3.01222
8192	1.38916e-05	3.00614
16384	1.73275e-06	3.00308

We see cubic algebraic convergence: $O(h^3)$.

SOLUTION of (6-8.e):

The error of the studied Runge-Kutta method and Taylor's method are practically identical. The obvious disadvantage of Taylor's method compared to Runge-Kutta methods is that the former involves rather complicated higher derivatives of f. If we want higher orders, those formulas get even more complicated, whereas explicit Runge-Kutta methods work with only a few evaluations of f itself, yielding results which are comparable.

Another fundamental drawback of the Taylor expansion method is that it cannot be applied to an ODE with $\mathbf{f}$ given in procedural form.

SOLUTION of (6-9.a):

By straightforward computation we find

$$\mathbf{grad}\, F(x,y) = \begin{bmatrix} 4(x^2+y^2)x - 3x^2 \\ 4(x^2+y^2)y - 3y^2 \end{bmatrix} \quad , \quad x,y \in \mathbb{R}\,. \tag{6.9.4}$$

Then we rotate the gradient counterclockwise using (6.9.2). This formula is easily explained: The linear mapping which performs this rotation is

$$\left[\boldsymbol{x} \mapsto \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \boldsymbol{x}\,, \quad \boldsymbol{x} \in \mathbb{R}^2 \right]$$

This is easily verified by looking at the unit vectors. The first unit vector pointing in x-direction $[1,0]^T$ is rotated, such that it points in y-direction, thus, it is mapped onto the second unit vector $[0,1]^T$. The second unit vector $[0,1]^T$ is further rotated in the $-x$-direction, thus, it is mapped on the opposite of the first unit vector $[-1,0]^T$.

Using (6.9.4), the differential equation (6.9.1) for F from (6.9.3) reads, with $\mathbf{y}(t) = \begin{bmatrix} x(t) \\ y(t) \end{bmatrix}$,

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \frac{1}{\sqrt{(4(x^2+y^2)x - 3x^2)^2 + (4(x^2+y^2)y - 3y^2)^2}} \cdot \begin{bmatrix} -4(x^2+y^2)y + 3y^2 \\ 4(x^2+y^2)x - 3x^2 \end{bmatrix}. \tag{6.9.5}$$

The time argument of x and y is suppressed for the sake brevity.

SOLUTION of (6-9.b):

The right-hand side function to be passed to the constructor of **Ode45** can conveniently be realized as a lambda function.

A conversion of the return value of the method `Ode45::solve()` into the correct return type is needed.

C++ code 6.9.6: Sub-problem (6-9.b): The function `computeIsolinePoints()` ⇝ GITHUB

```
2  template <typename GradientFunctor>
3  Eigen::Matrix<double, 2, Eigen::Dynamic> computeIsolinePoints(
4      GradientFunctor &&gradF, Eigen::Vector2d y0, double T) {
5    Eigen::Matrix<double, 2, Eigen::Dynamic> states;
6
7    // Right-hand-side vector field of isoline ODE
8    auto rhs = [gradF](Eigen::Vector2d x) -> Eigen::Vector2d {
9      Eigen::Vector2d gradFx = gradF(x);
10     return Eigen::Vector2d(-gradFx(1), gradFx(0)) / gradFx.norm();
11   };
12
13   // Adaptive explicit Runge-Kutta method
14   Ode45<Eigen::Vector2d> integrator(rhs);
15   // Set tolerances for timestep control (optional)
16   integrator.options.rtol = 1e-6;
17   integrator.options.atol = 1e-8;
18   // Perform explicit timestepping
19   std::vector<std::pair<Eigen::Vector2d, double>> sol = integrator.solve(y0, T);
20   // Convert output into requested format: points on isoline arranged into the
21   // columns of a matrix.
22   int M = sol.size() - 1;
23   states = Eigen::MatrixXd::Zero(2, M + 1);
24   for (int m = 0; m <= M; ++m) {
25     states.col(m) = sol[m].first;
26   }
27
28   return states;
29 }
```

HINT 1 for (6-9.c): It is recommended to call `computeIsolinePoints()` from Sub-problem (6-9.b). ⌟

SOLUTION of (6-9.c):

The normalized rotated gradient of F from (6.9.3) can be encapsulated in a lambda function.

C++ code 6.9.7: Sub-problem (6-9.c): The function `crookedEgg()` ⇝ GITHUB

```
Eigen::Matrix<double, 2, Eigen::Dynamic> crookedEgg() {
  auto gradF = [](Eigen::Vector2d x) -> Eigen::Vector2d {
    return 4.0 * x.squaredNorm() * x - 3.0 * x.cwiseAbs2();
  };
  Eigen::Vector2d y0(1.0, 0.0);
  double T = 4.0;
  return computeIsolinePoints(gradF, y0, T);
}
```

HINT 1 for (6-9.d): Of course, you may start with copying what you implemented for `computeIsoline`

HINT 2 for (6-9.d): Please remember that the numerical evaluation of difference quotients is affected by cancellation, see . That reference also tells you the “right” choice of h in (6.9.8). ⌟

SOLUTION of (6-9.d):

In we learned that the evaluation of difference quotients is affected by cancellation for very small span h. In order to curb cancellation it is recommended to use $h \approx \sqrt{\mathrm{EPS}}$, where EPS is the machine precision: $\mathrm{EPS} \approx 2 \cdot 10^{-16}$ for double precision floating point arithmetic, type **double**, see .

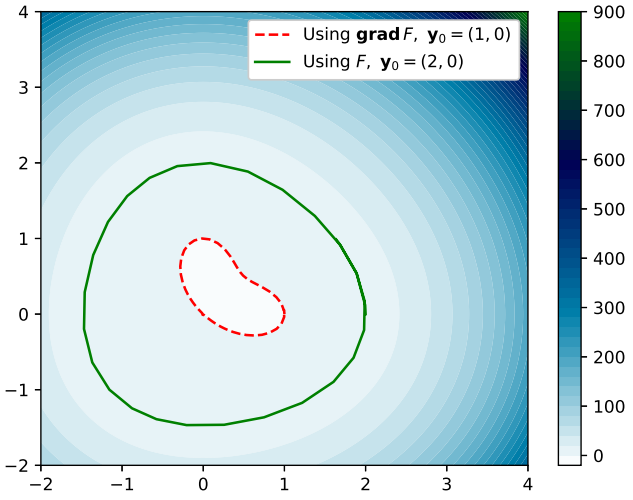
The implementation of this subproblem just boils down to augmenting the code of `computeIsolinePoi` by another lambda function supplying an approximation of $\mathbf{grad}\, F(\boldsymbol{x})$ based on two symmetric difference quotients in x_1 and x_2-directions.

C++ code 6.9.9: Sub-problem (6-9.b): The function `computeIsolinePointsDQ()`
➙ GITHUB

```
template <typename FFunctor>
Eigen::Matrix<double, 2, Eigen::Dynamic> computeIsolinePointsDQ(
    FFunctor &&F, Eigen::Vector2d y0, double T) {
  // Get gradient of F by symmetric difference quotients with
  // a span given by the root of the machine precision in order
  // to curb cancellation as much as possible.
  double h = std::sqrt(std::numeric_limits<double>::epsilon());
  auto gradF = [h, F](Eigen::Vector2d x) -> Eigen::Vector2d {
    Eigen::Vector2d dx(h, 0.0);
    Eigen::Vector2d dy(0.0, h);
    return Eigen::Vector2d(F(x + dx) - F(x - dx), F(x + dy) - F(x - dy)) /
           (2.0 * h);
  };

  return computeIsolinePoints(gradF, y0, T);
}
```

The span h of the symmetric difference quotient need only be in “the right ballpark”, roughly $h \approx 10^{-8}$.

SOLUTION of (6-9.e):



◁ Contour plot of F with isolines computed by `computeIsolinePoints()` (red, dashed) and `computeIsolinePointsDQ()` (green, solid).

Note that a wrong choice of the underlying ODE solver can corrupt the isonlines. In the case of **Ode45**, an appropriate choice of `rtol` and `atol` is needed, see Sub-problem (6-9.b).

C++ code 6.9.10: Sub-problem (6-9.e): The function `main()` ⇢ GITHUB

```
2   // Compute isolines using gradF and computeIsolinePoints(...)
3   Eigen::MatrixXd isolinePoints = ContourPlot::crookedEgg();
4
5   // Compute isolines using F and computeIsolinePointsDQ(...)
6   auto F = [](Eigen::Vector2d x) {
7     return Square(x.squaredNorm()) - Cube(x[0]) - Cube(x[1]);
8   };
9   Eigen::Vector2d y0(2.0, 0.0);
10  double T = 12.0;
11  Eigen::MatrixXd isolinePointsDQ =
12      ContourPlot::computeIsolinePointsDQ(F, y0, T);
13
14  // Write .csv file with 4 columns containg the x and y coordinates of the two
15  // isolines
16  std::ofstream csvFile;
17  csvFile.open("contourplot.csv");
18  csvFile << isolinePoints.format(CSVFormat) << std::endl;
19  csvFile << isolinePointsDQ.format(CSVFormat) << std::endl;
20  csvFile.close();
21  std::cout << "Generated " CURRENT_BINARY_DIR "/contourplot.csv" << std::endl;
22
23  // Call the Python script to plot the isolines
24  std::system("python3 " CURRENT_SOURCE_DIR
25              "/contourplot.py " CURRENT_BINARY_DIR
26              "/contourplot.csv " CURRENT_BINARY_DIR "/contourplot.eps");
```

SOLUTION of (7-1.a):

The solution is given in [Lecture → Rem. 7.3.3.6].

SOLUTION of (7-1.b):

We follow the developments of [Lecture → Rem. 7.3.3.9] and rewrite the stage equations as

$$F(\mathbf{g}) = \mathbf{0} \quad \text{with} \quad F(\mathbf{g}) := \mathbf{g} - h(\mathcal{A} \otimes \mathbf{I}) \begin{bmatrix} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_1) \\ \vdots \\ \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_s) \end{bmatrix} . \tag{7.1.1}$$

Then the corresponding Newton iteration reads

$$\mathbf{g}^{(k+1)} = \mathbf{g}^{(k)} - DF(\mathbf{g}^{(k)})^{-1} F(\mathbf{g}^{(k)}) . \tag{7.1.2}$$

We can write the function F in "vector-valued components":

$$F(\mathbf{g}) = F\left(\begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_s \end{bmatrix} \right) = \begin{bmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_s \end{bmatrix} - h \begin{bmatrix} \sum_{j=1}^{s} a_{1j} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_j) \\ \vdots \\ \sum_{j=1}^{s} a_{sj} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_j) \end{bmatrix} =: \begin{bmatrix} F_1(\mathbf{g}) \\ \vdots \\ F_s(\mathbf{g}) \end{bmatrix} .$$

Note that the components of F are functions $\mathbb{R}^{N \cdot s} \to \mathbb{R}^N$ Componentwise differentiation yields the Jacobian

$$DF(\mathbf{g}^{(k)}) = \left[\frac{\partial F_i}{\partial \mathbf{g}_k} \right]_{i,k=1}^{s} = \mathbf{I} - h \begin{bmatrix} a_{11} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_1) & \dots & a_{1s} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_s) \\ \vdots & & \vdots \\ a_{s1} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_1) & \dots & a_{ss} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_s) \end{bmatrix} . \tag{7.1.3}$$

Here $\dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{g}) \in \mathbb{R}^{N,N}$ are "partial-derivative Jacobi matrices".

SOLUTION of (7-1.c):

We implement the method discussed in Sub-problem (7-1.b). The main task is to wrap the function F from (7.1.1) and its Jacobian (7.1.3) into suitable lambda functions and pass those to `dampnewton()`.

C++ code 7.1.4: Implementation of implicitRKIntegrator ↪ GITHUB

HINT 1 for (7-1.d): Scan the particular implicit Runge-Kutta single step methods presented in [Lecture → Section 7.3].

⌟

SOLUTION of (7-1.d):

The Butcher scheme used corresponds to the Radau RK-SSM of order 3 (see [Lecture $\rightarrow$ Ex. 7.3.4.21] for the complete scheme). Thus, the method is A-stable and L-stable.

SOLUTION of (7-1.e):

C++11-code 7.1.7: Use of implicitRKIntegrator for solving an IVP for the Lotka-Volterra ODE
➥ GITHUB

```
  // Coefficients and handle for prey/predator model
  double alpha1 = 3.;
  double alpha2 = 2.;
  double beta1 = 0.1;
  double beta2 = 0.1;
  auto f = [&alpha1, &alpha2, &beta1, &beta2](const Eigen::VectorXd &y) {
    auto temp = y;
    temp(0) *= alpha1 - beta1 * y(1);
    temp(1) *= -alpha2 + beta2 * y(0);
    return temp;
  };

  auto Jf = [&alpha1, &alpha2, &beta1, &beta2](const Eigen::VectorXd &y) {
    Eigen::Matrix2d temp;
    temp << alpha1 - beta1 * y(1), -beta1 * y(0), beta2 * y(1),
        -alpha2 + beta2 * y(0);
    return temp;
  };

  // Final time for model
  double T = 10.;

  // Initial value for model
  Eigen::Vector2d y0(100.0, 5.0);

  // Array of number of steps (for convergence study)
  std::vector<unsigned int> N = {128,  256,  512,   1024,  2048,
                                 4096, 8192, 16384, 32768, 65536};

  // Exact value y(10) at final time T = 10 (approximated)
  Eigen::Vector2d yex(0.319465882659820, 9.730809352326228);
  // Start convergence study
  std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::setw(15)
            << "rate" << std::endl;
  double errold;
  for (unsigned int i = 0; i < N.size(); ++i) {
    auto res = integrator.solve(f, Jf, T, y0, N[i]);
    double err = (res.back() - yex).norm();
    std::cout << std::setw(15) << N[i] << std::setw(15) << err;
    if (i > 0) {
      std::cout << std::setw(15) << std::log2(errold / err);
    }
    errold = err;
    std::cout << std::endl;
  }
```

N	error norm
128	1.64554
256	0.184465
512	0.0237424
1024	0.00326135
2048	0.000503882
4096	5.79285e-05
8192	5.10432e-06
16384	6.37473e-07
32768	7.96485e-08
65536	9.95420e-09

◁ Norm of error at final time.

We observe an asymptotic (in the last few steps) decrease of the error norm by a factor of ≈ 8 whenever the number of (equidistant) timesteps is doubled. Thus we observe *algebraic convergence* with rate 3 in the timestep size h:

$$\text{errornorm} = O(h^3) \ .$$

We expect this because the 2-stage Radau single-step method is of order 3, see [Lecture → Ex. 7.3.4.21].

HINT 1 for (7-2.a): Examine the time derivative $\frac{d}{dt}\|\mathbf{y}(t)\|_2^2$ and use the product rule. ⌟

SOLUTION of (7-2.a):

Since $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$ for all $\mathbf{x}, \mathbf{y}$, we have

$$(\mathbf{a} \times \mathbf{y} + c(\mathbf{y} \times (\mathbf{a} \times \mathbf{y}))) \cdot \mathbf{y} = 0 \quad \Rightarrow \quad D_t \|\mathbf{y}(t)\|_2^2 = 2\dot{\mathbf{y}}(t) \cdot \mathbf{y}(t) = 0 \quad \Rightarrow \quad \|\mathbf{y}(t)\|_2^2 = \text{const. } \forall t \, .$$

Thus $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}(0)\|_2 = \|\mathbf{y}_0\|_2$.

SOLUTION of (7-2.b):

The right-hand side function $\mathbf{f}$ in (7.2.1) is a quadratic polynomial in the components of $\mathbf{y}$, hence smooth and defined on all of $\mathbb{R}^3$. Based on the product rule and appealing to the bilinearity of the cross product, we find

$$\mathrm{D}\mathbf{f}(\mathbf{y})\mathbf{h} = \mathbf{a}\times\mathbf{h} + c\mathbf{h}\times(\mathbf{a}\times\mathbf{y}) + c\mathbf{y}\times(\mathbf{a}\times\mathbf{h})\,.$$

The Jacobian $\mathrm{D}\mathbf{f}(\mathbf{y})$ is the matrix representation of this mapping $\mathbf{h}\mapsto\mathrm{D}\mathbf{f}(\mathbf{y})$. Using the definition of cross product, a simple but tedious calculation shows that

$$\mathrm{D}\mathbf{f}(\mathbf{y}) = \begin{bmatrix} -ca_2y_2 - ca_3y_3 & -a_3 + 2ca_1y_2 - ca_2y_1 & a_2 - ca_3y_1 + 2ca_1y_3 \\ a_3 - ca_1y_2 + 2ca_2y_1 & -ca_3y_3 - ca_1y_1 & -a_1 + 2ca_2y_3 - ca_3y_2 \\ -a_2 + 2ca_3y_1 - ca_1y_3 & a_1 - ca_2y_3 + 2ca_3y_2 & -ca_1y_1 - ca_2y_2 \end{bmatrix}.$$

Thus for $\mathbf{y} = \mathbf{a}$ we obtain

$$\begin{aligned} \mathrm{D}\mathbf{f}(\mathbf{a}) &= \begin{bmatrix} -c(a_2^2+a_3^2) & -a_3 + ca_1a_2 & a_2 + ca_3a_1 \\ a_3 + ca_1a_2 & -c(a_1^2+a_3^2) & -a_1 + ca_2a_3 \\ -a_2 + ca_1a_3 & a_1 + ca_2a_3 & -c(a_1^2+a_2^2) \end{bmatrix} \\ &= \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} + c\mathbf{a}\mathbf{a}^\top - c\|\mathbf{a}\|^2\mathbf{I}\,. \end{aligned}$$

The next step is an exercise in advanced linear algebra: We see that $\mathrm{D}\mathbf{f}(\mathbf{a})\mathbf{a} = \mathbf{0}$ and on the orthogonal complement $\mathbf{a}^\perp$ the matrix acts like a rotation (scaled with $\|\mathbf{a}\|$) minus $c\|\mathbf{a}\|\mathbf{I}$. This reveals that the spectrum (= set of eigenvalues) is given by

$$\sigma(\mathrm{D}\mathbf{f}(\mathbf{a})) = \{0, -c\|\mathbf{a}\|_2^2 \pm i\|\mathbf{a}\|_2\} = \{0, -c\pm i\}\,. \tag{7.2.2}$$

HINT 1 for (7-2.c): Recall from [Lecture → Section 7.2]:

An initial value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ will be stiff, if, for substantial periods of time,

$$\min\{\operatorname{Re}\lambda : \lambda \in \sigma(\mathrm{D}\mathbf{f}(\mathbf{y}(t)))\} \ll 0\,,$$
$$\textbf{and} \qquad \max\{\operatorname{Re}\lambda : \lambda \in \sigma(\mathrm{D}\mathbf{f}(\mathbf{y}(t)))\} \lesssim 0\,,$$

where $t \mapsto \mathbf{y}(t)$ is the solution trajectory and $\sigma(\mathbf{M})$ is the spectrum of the matrix $\mathbf{M}$, see .

SOLUTION of (7-2.c):

Appealing to (7.2.2) and applying the criteria [Lecture → Eq. (7.2.0.13)] and [Lecture → Eq. (7.2.0.14)] we find that at close to the state $\mathbf{y} = \mathbf{a}$ the ODE (7.2.1) is stiff for large $c > 0$.

SOLUTION of (7-2.d):

In the plot, we see that, on the one hand for the integrator **Ode45** the number of steps rises with c. On the other hand, the A-stable method uses roughly the same amount of steps, irregardless of the value of c chosen.

As **Ode45** is an explicit solver, it suffers from stability-induced step-size constraints for large $c > 0$. The A-stable integrator need not use smaller timesteps to satisfy the tolerance for big c.

This demonstrates [Lecture $\rightarrow$ Notion 7.2.0.7]: We see that the problem becomes the stiffer the larger the parameter c. This is expected from the analysis of Sub-problem (7-2.b).

SOLUTION of (7-2.e):

Recall the recursion formula for the implicit mid-point method applied to an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) .$$

Its specialization for (7.2.1) reads

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h} = \mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) + c\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) \times \left(\mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right)\right) .$$

SOLUTION of (7-2.f):

t	$\|y(t)\|$
0	1.73205
0.078125	1.73205
0.15625	1.73205
0.234375	1.73205
0.3125	1.73205
⋮	⋮

We observe that norm of the approximate states generated by the single step method *is constant*, $\|\mathbf{y}_k\|_2 = \|\mathbf{y}_0\|$ for all k, and equal to $1.73205 = \left\|[1,1,1]^\top\right\|_2$.

Our implementation relies on an auxiliary function that calls **implicitRKIntegrator** with parameters characteristic of the implicit midpoint method. The Butcher scheme for the implicit midpoint method can be found in [Lecture → Ex. 7.3.4.5].

C++-code 7.2.3: Example code - `solve_imp_mid` ↪ GITHUB

```
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solve_imp_mid(Function &&f, Jacobian &&Jf,
                                           double T, const Eigen::VectorXd &y0,
                                           unsigned int M) {
  std::vector<Eigen::VectorXd> res(M + 1);
  // Construct the implicit mid-point method with the class
  // implicit_RKIntegrator and execute the .solve() method.
  // Return the vector containing all steps including initial and final value.
  // Initialize implicit RK with Butcher scheme
  unsigned int s = 1;
  // Initialize coefficients for the implicit midpoint method
  Eigen::MatrixXd A(s, s);
  Eigen::VectorXd b(s);
  A << 1. / 2.;
  b << 1.;
  CrossProd::implicitRKIntegrator RK(A, b);
  res =
      RK.solve(std::forward<Function>(f), std::forward<Jacobian>(Jf), T, y0, M);
  return res;
}
```

The right-hand side function for (7.2.1) and its Jacobian as computed in Sub-problem (7-2.b) are realized as lambda functions.

C++-code 7.2.4: Implementation of `tab_crossprod` ↪ GITHUB

```
void tab_crossprod() {
  // Solve the cross-product ODE with the implicit RK method
  // defined in solve_imp_mid. Tabulate the norms of the results at all steps.
  double T = 10.;
  int M = 128;
  // set data
  double c = 1.;
  Eigen::Vector3d y0(1., 1., 1.);
  Eigen::Vector3d a(1., 0., 0.);

  // define rhs
```

```cpp
  auto f = [a, c](Eigen::Vector3d y) -> Eigen::Vector3d {
    return a.cross(y) + c * y.cross(a.cross(y));
  };
  // define Jacobian of rhs
  auto Jf = [a, c](Eigen::Vector3d y) -> Eigen::Matrix3d {
    Eigen::Matrix3d temp;
    temp << -c * (a(1) * y(1) + a(2) * y(2)),
        c * (2 * a(0) * y(1) - a(1) * y(0)) - a(2),
        a(1) + c * (2 * a(0) * y(2) - a(2) * y(0)),
        a(2) - c * (a(0) * y(1) - 2 * a(1) * y(0)),
        -c * (a(0) * y(0) + a(2) * y(2)),
        c * (2 * a(1) * y(2) - a(2) * y(1)) - a(0),
        -a(1) - c * (a(0) * y(2) - 2 * a(2) * y(0)),
        a(0) - c * (a(1) * y(2) - 2 * a(2) * y(1)),
        -c * (a(0) * y(0) + a(1) * y(1));
    return temp;
  };

  std::vector<Eigen::VectorXd> res_imp = solve_imp_mid(f, Jf, T, y0, M);

  std::cout << "1. Implicit midpoint method" << std::endl;
  std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
            << std::endl;

  for (int i = 0; i < M + 1; ++i) {
    std::cout << std::setw(10) << T * i / M << std::setw(15)
              << res_imp[i].norm() << std::endl;
  }
```

HINT 1 for (7-2.g): The linearization of increment equations is also discussed in [Lecture → Exp. 7.4.0.3] and [Lecture → Eq. (7.4.0.5)]. ⌟

SOLUTION of (7-2.g):

The linear implicit mid-point method is obtained by expanding the right-hand side of the increment equations of the implicit mid point method in its Taylor series in h around $h = 0$:

$$\mathbf{k}_1 = \mathbf{f}\left(\mathbf{y}_k + \frac{h}{2}\mathbf{k}_1\right) = \mathbf{f}(\mathbf{y}_k) + \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\mathbf{k}_1 + O(h^2)$$

and only taking the linear terms. Since the non-linear method is given by $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1$, the linearization reads

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1^{\text{lin.}} \quad , \qquad \mathbf{k}_1^{\text{lin.}} := \left(I - \tfrac{h}{2}D\mathbf{f}(\mathbf{y}_k)\right)^{-1}\mathbf{f}(\mathbf{y}_k) \ .$$

A step of the method just involves the solution of a single $N \times N$ linear system of equations.

SOLUTION of (7-2.h):

See file `crossprod.h`.

t	$\|y(t)\|$
0	1.73205
0.078125	1.73151
0.15625	1.731
0.234375	1.73054
0.3125	1.73015
0.390625	1.72981
0.46875	1.72954
0.546875	1.72932
0.625	1.72915
0.703125	1.72901
0.78125	1.7289
0.859375	1.72882
0.9375	1.72875
⋮	⋮

The sequence of the Euclidean norms of the states is not exactly constant: it starts at 1.73205 and converges to 1.72853. This is due to the approximation introduced with the linearization.

C++-code 7.2.5: `solve_lin_mid`↬ GITHUB

```
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solve_lin_mid(Function &&f, Jacobian &&Jf,
                                           double T, const Eigen::VectorXd &y0,
                                           unsigned int M) {
  std::vector<Eigen::VectorXd> res;
  // Implement the linear implicit mid-point method for
  // an autonomous ODE y' = f(y), y(0) = y0. Return the vector containing
  // all steps including initial and final value.
  // Initial step size
  double h = T / M;
  int d = y0.size();
  // Store initial data
  res.push_back(y0);

  // Initialize some memory to store temporary values
  Eigen::VectorXd ytemp1 = y0;
  Eigen::VectorXd ytemp2 = y0;
  // Pointers for efficient swapping of state vectors
  Eigen::VectorXd *yold = &ytemp1;
  Eigen::VectorXd *ynew = &ytemp2;
  Eigen::MatrixXd eye = Eigen::MatrixXd::Identity(3, 3);
  // Loop over all fixed steps
  for (unsigned int k = 0; k < M; ++k) {
    // Compute, save and swap next step
    *ynew = *yold + h * (eye - h / 2. * Jf(*yold)).lu().solve(f(*yold));
    res.push_back(*ynew);
    std::swap(yold, ynew);
  }

  return res;
}
```

C++-code 7.2.6: Solve the cross-product ODE with implicit RK ➙ GITHUB

```
  // Solve the cross-product ODE with the implicit RK method
  // defined in solve_lin_mid. Tabulate the norms of the results at all steps.
  std::vector<Eigen::VectorXd> res_lin = solve_lin_mid(f, Jf, T, y0, M);
  std::cout << "\n2. Linear implicit midpoint method" << std::endl;
  std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
            << std::endl;
  for (int i = 0; i < M + 1; ++i) {
    std::cout << std::setw(10) << T * i / M << std::setw(15)
              << res_lin[i].norm() << std::endl;
  }
```

HINT 1 for (7-3.a): Recall that in every step of an implicit RK-SSM we have to solve a non-linear system of equations for the increments, see [Lecture → Rem. 7.3.3.9].

⌟

SOLUTION of (7-3.a):

For SDIRK methods the Butcher matrix is lower triangular. In this case the increment equations/stage equations can be solved separately and sequentially, computing $\mathbf{k}_1 \to \mathbf{k}_2 \to \cdots \to \mathbf{k}_s$ in that order.

So instead of solving a big $sN \times sN$ non-linear system of equations for a generic implicit Runge-Kutta single-step method, we need only solve s smaller $N \times N$ non-linear systems.

SOLUTION of (7-3.b):

The solution is essentially given in [Lecture → Def. 7.3.3.1]. The increments $\mathbf{k}_i$ are given as solutions of the two $N \times N$ non-linear systems of equations,

$$\begin{aligned} \mathbf{k}_1 &= f(t_0 + h\gamma, \boldsymbol{y}_0 + h\gamma\mathbf{k}_1) \ , \\ \mathbf{k}_2 &= f(t_0 + h(1-\gamma), \boldsymbol{y}_0 + h(1-2\gamma)\mathbf{k}_1 + h\gamma\mathbf{k}_2) \ . \end{aligned}$$

SOLUTION of (7-3.c):

The stability function $S(z)$ of a method is derived by applying the method to the scalar, linear test equation

$$\dot{y}(t) = \lambda y(t) \ .$$

The solution can be written as

$$y_{k+1} = S(z) y_k \ ,$$

where $S(z)$ is the stability function and $z := h\lambda$.

In the case of the SDIRK method, we get the increment equations

$$\begin{aligned} k_1 &= \lambda(y_k + h\gamma k_1) \ , \\ k_2 &= \lambda(y_k + h(1-2\gamma)k_1 + h\gamma k_2) \ . \end{aligned}$$

Therefore:

$$\begin{aligned} k_1 &= \frac{\lambda}{1-h\lambda\gamma} y_k , \\ k_2 &= \frac{\lambda}{1-h\lambda\gamma}(y_k + h(1-2\gamma)k_1) \ , \end{aligned}$$

which leads to the recursion

$$y_{k+1} = y_k + \frac{h}{2}(k_1 + k_2) \ .$$

With $z := h\lambda$ and by plugging in k_1 and k_2 we arrive at:

$$y_{k+1} = \underbrace{\left(1 + \frac{z}{2(1-\gamma z)}\left(2 + \frac{z(1-2\gamma)}{1-\gamma z}\right)\right)}_{=:S(z)} y_k \ .$$

Hence:

$$S(z) = \frac{2(1-\gamma z)^2 + 2z(1-\gamma z)^2 + z^2(1-2\gamma)}{2(1-\gamma z)^2}$$

and after collecting the powers of z in the numerator we get

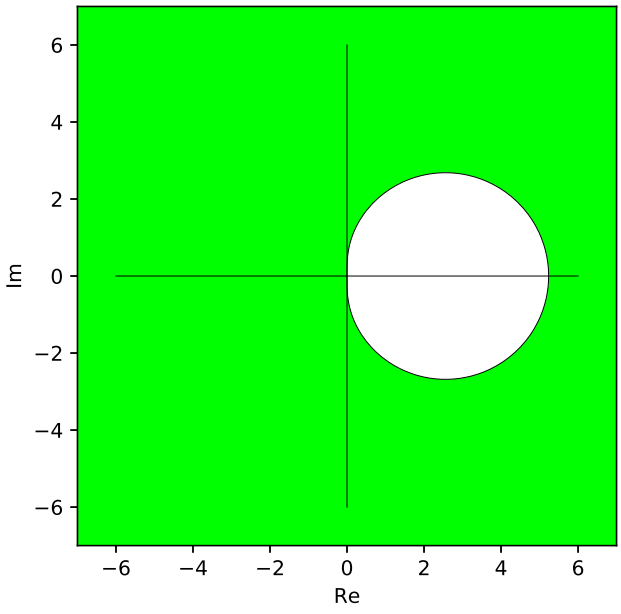
$$S(z) = \frac{1 + z(1-2\gamma) + z^2(\gamma^2 - 2\gamma + \frac{1}{2})}{(1-\gamma z)^2} \ .$$

Alternatively, you can directly appeal to the formula for the stability function of an implicit RK-SSM, in particular the determinant-based version:

Theorem [Lecture → Thm. 7.3.4.4]. Stability function of general Runge-Kutta methods

*The discrete evolution Ψ_λ^h of an s-stage Runge-Kutta single step method (→ **??**) with Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ (see (**??**)) for the ODE $\dot{y} = \lambda y$ is given by a multiplication with*

$$S(z) := \underbrace{1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}}_{\text{stability function}} = \frac{\det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T)}{\det(\mathbf{I} - z\mathfrak{A})} \ , \quad z := \lambda h \ , \quad \mathbf{1} = [1,\ldots,1]^T \in \mathbb{R}^s \ .$$



◁ Plot of the stability domain (green) of the 2-stage SDIRK method from (7.3.3).

- The imaginary axis is tangent to the boundary of the stability domain in $z = 0$
- We see that the left half plane $\mathbb{C}^-$ is included in the stability domain.

SOLUTION of (7-3.d):

Given the previous sub-problem, for $\gamma = 1$ the stability function is:

$$S_1(z) := \frac{1 - z - \frac{z^2}{2}}{(1-z)^2} \tag{7.3.5}$$

Verification of the A-stability of (7.3.5)**:**
By definition [Lecture → § 7.3.4.7], we need to show that $|S_1(z)| \leq 1$ for all $z \in \mathbb{C}^- := \{z \in \mathbb{C} \mid \operatorname{Re} z < 0\}$. In order to do this, we consider the stability function on the imaginary axis:

$$\begin{aligned} |S_1(iy)|^2 &= \frac{|1 - iy - (iy)^2/2|^2}{|1-iy|^4} = \frac{|1 - iy + y/2|^2}{|1-iy|^4} \\ &= \frac{(1 + y^2/2)^2 + y^2}{(1+y^2)^2} = \frac{1 + 2y^2 + y^4/4}{1 + 2y^2 + y^4} \leq 1, \quad y \in \mathbb{R} \end{aligned}$$

Since the only pole ($z = 1$) of the rational function $S_1(z)$ lies in the positive half plane of $\mathbb{C}$, the function S_1 is holomorphic in the left half plane. Furthermore, S_1 is bounded by 1 on the boundary of this half plane (i.e. on the imaginary axis). So, by the maximum principle for holomorphic functions (hint), S_1 is bounded on the entire left half plane by 1. This implies in particular that S_1 is A-stable.

Verification of the L-stability of (7.3.5)**:**
S_1 is not L-stable (*cf.* definition [Lecture → § 9.2.7.42]), because:

$$\lim_{\operatorname{Re} z \to -\infty} |S_1(z)| = \lim_{\operatorname{Re} z \to -\infty} \left| \frac{1 - z - z^2/2}{1 - 2z + z^2} \right| = \frac{1}{2} \neq 0$$

SOLUTION of (7-3.e):

Define $z_1 = y$, $z_2 = \dot{y}$, then the initial value problem

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0\,, \tag{7.3.6}$$

is equivalently to the *linear* first order system of ODEs

$$\begin{aligned} \dot{z}_1 &= z_2\,, \\ \dot{z}_2 &= -z_1 - z_2 \end{aligned} \quad \Leftrightarrow \quad \frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix}\begin{bmatrix} z_1 \\ z_2 \end{bmatrix}\,, \tag{7.3.7}$$

with initial values $z_1(0) = 1$ and $z_2(0) = 0$.

SOLUTION of (7-3.f):

The first-order ODE derived in Sub-problem (7-3.e) is of the form

$$\dot{\mathbf{z}} = \mathbf{A}\mathbf{z} \quad \text{with} \quad \mathbf{A} := \begin{bmatrix} 0 & 1 \\ -1 & -1 \end{bmatrix} . \tag{7.3.8}$$

From the course we recall (note that instead of $\mathbf{z}$ this uses the notation $\mathbf{y}$):

Definition [Lecture → Def. 7.3.3.1]. General Runge-Kutta single step method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an s-stage Runge-Kutta single step method (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$ is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{s} a_{ij}\mathbf{k}_j) , \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i .$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$ are called **increments**.

Shorthand notation for Runge-Kutta methods

Butcher scheme ▷

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} .$$

[Lecture → Eq. (7.3.3.3)]

Note that now, in contrast to [Lecture → Eq. (6.4.0.11)], $\mathfrak{A}$ can be a general $s \times s$-matrix.

Thus we can write the increment equations for (7.3.8) as

$$\begin{aligned} \mathbf{k}_1 &= \mathbf{A}\mathbf{z}_0 + h\gamma\mathbf{A}\mathbf{k}_1 , \\ \mathbf{k}_2 &= \mathbf{A}\mathbf{z}_0 + h(1-2\gamma)\mathbf{A}\mathbf{k}_1 + h\mathbf{A}\mathbf{k}_2 . \end{aligned}$$

This yields the explicit formulas

$$\begin{aligned} \mathbf{k}_1 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}\mathbf{A}\mathbf{z}_0 , \\ \mathbf{k}_2 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}(\mathbf{A}\mathbf{z}_0 + h(1-2\gamma)\mathbf{A}\mathbf{k}_1) . \end{aligned} \tag{7.3.9}$$

Note that a linear system of equations with the same coefficient matrix has to be solved to obtain both $\mathbf{k}_1$ and $\mathbf{k}_2$. Finally, we have the update equation

$$\mathbf{z}_1 = \mathbf{z}_0 + h(\tfrac{1}{2}\mathbf{k}_1 + \tfrac{1}{2}\mathbf{k}_2) .$$

C++11-code 7.3.10: Solution of (7-3.f) ➙ GITHUB

```
2 Eigen::Vector2d SdirkStep(const Eigen::Vector2d &z0, double h, double gamma) {
3   Eigen::Vector2d res;
4   // Compute one timestep of the SDIRK implicit RK-SSM for the linear
      ODE
5   // Matrix A for evaluation of f
6   Eigen::Matrix2d A;
```

```
  A << 0., 1., -1., -1.;
  // Precompute and reuse factorization
  auto A_lu = (Eigen::Matrix2d::Identity() - h * gamma * A).partialPivLu();
  Eigen::Vector2d az = A * z0;

  // Increments according to (7.3.9)
  Eigen::Vector2d k1 = A_lu.solve(az);
  Eigen::Vector2d k2 = A_lu.solve(az + h * (1 - 2 * gamma) * A * k1);

  // Updated state
  res = z0 + h * 0.5 * (k1 + k2);
  return res;
}
```

HINT 1 for (7-3.g): The exact solution of (7.3.2) is

$$y(t) = \tfrac{1}{3} e^{-\frac{1}{2}t} \left(3\cos(\tfrac{1}{2}\sqrt{3}t) + \sqrt{3}\sin(\tfrac{1}{2}\sqrt{3}t) \right) .$$

It can be computed by finding the roots of the characteristic polynomial associated with the second-order ODE. ⌟

SOLUTION of (7-3.g):

C++11-code 7.3.11: Timestepping for (7.3.2) with SDIRK implicit RK-SSM ➙ GITHUB

```
std::vector<Eigen::Vector2d> SdirkSolve(const Eigen::Vector2d &z0,
                                        unsigned int M, double T,
                                        double gamma) {
  // Solution vector
  std::vector<Eigen::Vector2d> res(M + 1);
  // Solve the ODE with uniform timesteps using the SDIRK method
  // Equidistant step size
  const double h = T / M;
  // Push initial data
  res[0] = z0;
  // Main loop
  for (unsigned int i = 1; i <= M; ++i) {
    res[i] = SdirkStep(res[i - 1], h, gamma);
  }
  return res;
}
```

C++11-code 7.3.12: Testing convergence of SDIRK method for an IVP for (7.3.2)➙ GITHUB

```
double CvgSDIRK() {
  double conv_rate;
  // Study the convergence rate of the method.
  // Initial data z0 = [y(0), y'(0)]
  Eigen::Vector2d z0;
  z0 << 1, 0;
  // Final time
  const double T = 10;
  // Parameter
  const double gamma = (3. + std::sqrt(3.)) / 6.;
  // Mesh sizes
  Eigen::ArrayXd err(10);
  Eigen::ArrayXd M(10);
  M << 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240;

  // Exact solution (only y(t)) given z0 = [y(0), y'(0)] and t
  auto yex = [&z0](double t) {
    return 1. / 3. * std::exp(-t / 2.) *
           (3. * z0(0) * std::cos(std::sqrt(3.) * t / 2.) +
            std::sqrt(3.) * z0(0) * std::sin(std::sqrt(3.) * t / 2.) +
            2. * std::sqrt(3.) * z0(1) * std::sin(std::sqrt(3.) * t / 2.));
  };

  // Store old error for rate computation
  double errold = 0;
  std::cout << std::setw(15) << "m" << std::setw(15) << "maxerr"
            << std::setw(15) << "rate" << std::endl;
  // Loop over all meshes
  for (unsigned int i = 0; i < M.size(); ++i) {
    int m = M(i);
    // Get solution
    auto sol = SdirkSolve(z0, m, T, gamma);
```

```
    // Compute error
    err(i) = std::abs(sol.back()(0) - yex(T));

    // Print table
    std::cout << std::setw(15) << m << std::setw(15) << err(i);
    if (i > 0) std::cout << std::setw(15) << std::log2(errold / err(i));
    std::cout << std::endl;

    // Store old error
    errold = err(i);
  }

  Eigen::VectorXd coeffs = polyfit(M.log(), err.log(), 1);
  conv_rate = -coeffs(0);
  return conv_rate;
}
```

N	error
20	0.000283567
40	5.05272e-05
80	7.73715e-06
160	1.07013e-06
320	1.40565e-07
640	1.80048e-08
1280	2.27798e-09
2560	2.86467e-10
5120	3.59161e-11
10240	4.49619e-12

◁ Norm of error at final time $T = 10$ on temporal mesh.

We observe an asymptotic (in the last steps) decrease of the error norm by a factor of ≈ 8 whenever the number of (equidistant) timesteps is doubled. Thus we observe *algebraic convergence* with rate 3 in the timestep size h:

$$\text{errornorm} = O(h^3) \,.$$

This hints that the SDIRK single-step method has order 3 .

The numerically estimated order of convergence is 3.

SOLUTION of (7-4.a):

Matching [Lecture → Eq. (7.4.0.9)] and (7.4.2) we find

- that we deal with a 2-stage method: $s = 2$,
- $d_{2,1} = a$,
- $a_{11} = a_{22} = a$, $a_{2,1} = \frac{1}{2} - a$,
- $b_1 = 0$, $b_2 = 1$.

SOLUTION of (7-4.b):

Give a formula for single step of size $h > 0$ of the RK-SSM for the scalar linear ODE $\dot{y} = \lambda y$ in terms of the method's stability function:

$$y_1 = S(h\lambda)y_0 \ .$$

SOLUTION of (7-4.c):

For a scalar ODE, the Jacobian is just the derivative w.r.t. y, whence $J = Df(y) = f'(y) = (\lambda y)' = \lambda$. The quantity $\mathbf{W}$ is, therefore a scalar quantity as well:

$$W = 1 - ahJ = 1 - ah\lambda$$

If we combine these formulas and assume h is small enough:

$$\begin{aligned}
y_1 &= y_0 + h\frac{f(y_0 + \frac{1}{2}hk_1) - ahJk_1}{W} \\
&= y_0 + h\frac{f(y_0 + \frac{1}{2}h\frac{f(y_0)}{W}) - ahJ\frac{f(y_0)}{W})}{W} \\
&= y_0 + h\frac{\lambda(y_0 + \frac{1}{2}h\frac{\lambda y_0}{1-ah\lambda}) - \frac{ah\lambda^2 y_0}{1-ah\lambda}}{1 - ah\lambda} \\
&= \frac{(1-ah\lambda)^2 + h\lambda(1-ah\lambda) + \frac{1}{2}h^2\lambda^2 - ah^2\lambda^2}{(1-ah\lambda)^2} y_0 \\
&= \frac{(1 + a^2z^2 - 2az) + z(1-az) + \frac{1}{2}z^2 - az^2}{(1-az)^2} y_0 \\
&= \frac{1 + (1-2a)z + (\frac{1}{2} - 2a + a^2)z^2}{(1-az)^2} y_0
\end{aligned}$$

Since $\frac{1}{2} - 2a + a^2 = 0$, it follows

$$S(z) = \frac{1 + (1-2a)z}{(1-az)^2} \,. \tag{7.4.3}$$

HINT 1 for (7-4.d): The idea behind this sub-problem is elucidated in [Lecture → Rem. 7.1.0.21]. Apply [Lecture → Lemma 7.1.0.23]. ⌟

SOLUTION of (7-4.d):

We compute:

$$\begin{aligned} S(0) &= 1, \\ S'(0) &= 1, \\ S''(0) &= 4a - 2a^2, \\ S'''(0) &= 18a^2 - 12a^3 \ , \end{aligned}$$

therefore

$$S(z) = 1 + z + (2a - a^2)z^2 + (3a^2 - 2a^3)z^3 + O(z^4) \quad \text{for} \quad z \to 0 \ .$$

We can also compute the expansion of $\exp$:

$$\exp(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + O(z^4) \quad \text{for} \quad z \to 0 \ .$$

Since $2a - a^2 = \frac{1}{2}$ but $3a^2 - 2a^3 = \frac{1}{2}(\sqrt{2} - 1) \neq \frac{1}{6}$, we have:

$$|S(z) - \exp(z)| = O(|z|^3) \ .$$

Note that this means that 3 is the order of the one-step error when applying the single-step method to solve initial-value problems for the ODE $\dot{y} = y$. Using [Lecture → Lemma 7.1.0.23], we deduce that the maximal order q of the scheme is $q = 2$.

SOLUTION of (7-4.e):

The implementation relies on EIGEN's direct solvers for dense linear systems.

C++11-code 7.4.4: Solution of Sub-problem (7-4.e) ⇝ GITHUB

```
template <class Func, class Jac>
std::vector<Eigen::VectorXd> SolveRosenbrock(Func &&f, Jac &&df,
                                             const Eigen::VectorXd &y0,
                                             unsigned int M, double T) {
  // Will contain all states computed by the ROW-SSM
  std::vector<Eigen::VectorXd> res(M + 1);
  res[0] = y0;              // Push initial data
  const double h = T / M;  // stepsize
  const double a = 1. / (std::sqrt(2) + 2.);
  // Some temporary variables
  Eigen::VectorXd k1, k2;
  Eigen::MatrixXd J, W;
  // Main loop: conduct M timesteps
  for (unsigned int i = 1; i <= M; ++i) {
    // Current state
    Eigen::VectorXd &yprev = res[i - 1];
    // Fetch Jacobian, which is supposed to be an Eigen matrix type
    J = df(yprev);
    W = Eigen::MatrixXd::Identity(J.rows(), J.cols()) - a * h * J;
    // LU-factorize Jacobian only once!
    auto W_lu = W.partialPivLu();
    // Compute the increments by solving linear systems
    k1 = W_lu.solve(f(yprev));
    k2 = W_lu.solve(f(yprev + 0.5 * h * k1) - a * h * J * k1);
    // Compute the next state and store it
    res[i] = yprev + h * k2;
  }
  return res;
}
```

SOLUTION of (7-4.f):

The Jacobian Df is:

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} \lambda(1-\|\mathbf{y}\|^2) - 2\lambda y_0^2 & -1 - 2\lambda y_1 y_0 \\ 1 - 2\lambda y_1 y_0 & \lambda(1-\|\mathbf{y}\|^2) - 2\lambda y_1^2 \end{bmatrix}$$

C++11-code 7.4.6: Solution of Sub-problem (7-4.e) ⇝ GITHUB

```
double CvgRosenbrock() {
  double cvgRate = 0.0;
  // Use polyfit to estimate the rate of convergence
  // for SolveRosenbrock.
  // Final time
  const double T = 10.0;
  // Timestep sizes h=2^{-k} for which discretization erorrs should be computed
  const Eigen::ArrayXd K = Eigen::ArrayXd::LinSpaced(7, 4, 10);
  // Initial state vector
  Eigen::Vector2d y0(1., 1.);

  // Parameter and useful matrix for f
  const double lambda = 1;
  Eigen::Matrix2d R;
  R << 0.0, -1.0, 1.0, 0.0;
  // Lambda functions giving the right-hand side function f
  // and its Jacobian df
  auto f = [&R, &lambda](Eigen::Vector2d y) {
    return R * y + lambda * (1.0 - y.squaredNorm()) * y;
  };
  auto df = [&lambda](Eigen::Vector2d y) {
    double x = 1 - y.squaredNorm();
    Eigen::Matrix2d J;
    J << lambda * x - 2 * lambda * y(0) * y(0), -1 - 2 * lambda * y(1) * y(0),
        1 - 2 * lambda * y(1) * y(0), lambda * x - 2 * lambda * y(1) * y(1);
    return J;
  };

  // timestep size h=2^{-12} (=> M = T*2^{12}) used to compute a reference
  // solution
  const int M_ref = T * std::pow(2, 12);
  // Reference solution
  std::vector<Eigen::VectorXd> solref = SolveRosenbrock(f, df, y0, M_ref, T);
  // Arrays for recording error norms and numbers of timesteps
  Eigen::ArrayXd Error(K.size());
  Eigen::ArrayXd M(K.size());
  // Loop over all temporal meshes
  for (unsigned int i = 0; i < K.size(); ++i) {
    // h = 2^{-k} => M = T*h = T*2^k
    M(i) = T * std::pow(2, K[i]);

    // Compute solution
    std::vector<Eigen::VectorXd> sol = SolveRosenbrock(f, df, y0, M(i), T);

    // Compute error
    double maxerr = 0;
    for (unsigned int j = 0; j < sol.size(); ++j) {
```

```
      maxerr = std::max(maxerr, (sol[j] - solref[j * M_ref / M(i)]).norm());
    }
    Error(i) = maxerr;
  }
  // Print Error table
  std::cout << std::setw(15) << "M" << std::setw(16) << "maxerr \n";
  for (int i = 0; i < M.size(); ++i) {
    std::cout << std::setw(15) << M(i) << std::setw(15) << Error(i)
              << std::endl;
  }
  // Estimate convergence rate using linear regression provided by the auxiliary
  // function polyfit()
  cvgRate = -polyfit(M.log(), Error.log(), 1)(0);
  return cvgRate;
}
```

M	maxerr
160	0.00181187
320	0.000439645
640	0.000108218
1280	2.68249e-05
2560	6.66039e-06
5120	1.64229e-06
10240	3.90643e-07

Least squares fitting of a linear function to $(\log h_i, \mathrm{err}(i))$ yields the estimated rate of algebraic convergence 2.02461 through the slope of the linear function, see .

We conjecture that the Rosenbrock single-step method (7.4.2) has order 2 , an integer order as all other commonly used single-step methods.

HINT 1 for (7-4.g): To investigate A-stability, calculate the complex norm of $S(z)$ on the imaginary axis $\mathrm{Re}\, z = 0$ and apply the ⌟

SOLUTION of (7-4.g):

We start by proving that the method is A-stable , meaning that $S(z) < 1, \forall z \in \mathbb{C}, \mathrm{Re}(z) < 0$. First of all, we can compute the complex norm of the stability function at $z = iy, y \in \mathbb{R}$ as:

$$|S(iy)|^2 = \frac{|1 + (1-2a)iy|^2}{|(1-aiy)^2|^2} = \frac{1 + (1-2a)^2y^2}{(1+a^2y^2)^2} = \frac{1 + (1-4a+4a^2)y^2}{(1+2a^2y^2+a^4y^4)} .$$

Notice that $1 - 4a + 4a^2 = 2a^2$, therefore:

$$|S(iy)|^2 = \frac{1 + 2a^2y^2}{1 + 2a^2y^2 + a^4y^4} < 1 .$$

The norm of the function S is bounded on the imaginary axis. Observe that $|S(z)| \to 0, |z| \to \infty$, which, follows from the fact that the degree of the denominator of S is bigger than the polynomial degree of the numerator. Notice that $1 - 2a = 1 - 2\frac{1}{\sqrt{(2)}+2} = \sqrt{2} - 2$.

The only pole of the function is at $z = 1/a$, therefore the function is holomorphic on the left complex plane.

Applying the theorem of the hint on concludes that the absolute value of the function is bounded by 1 on the left complex plane.

The L-stability follows immediately with A-stability and the fact that the absolute value of the function converges to zero as $|z| \to \infty$.

SOLUTION of (7-5.a):

We recall the notion of consistency of a single-step method:

Definition [Lecture → Def. 6.3.1.11]. Consistent single step methods

A single step method according to [Lecture → Def. 6.3.1.4] based on a discrete evolution of the form

$$\boldsymbol{\Psi}^h\mathbf{y} = \mathbf{y} + h\boldsymbol{\psi}(h,\mathbf{y}) \quad \text{with} \quad \begin{array}{l}\boldsymbol{\psi} : I \times D \to \mathbb{R}^N \text{ continuous,}\\ \boldsymbol{\psi}(0,\mathbf{y}) = \mathbf{f}(\mathbf{y})\ .\end{array} \quad [\text{Lecture} \to \text{Eq. (6.3.1.10)}]$$

is called **consistent** with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

In view of (7.5.1), consistency is equivalent to $\varphi\big(h\mathrm{D}\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0)$ for $h = 0$. By definition of the matrix exponential we have $e^{hZ} = \mathrm{Id} + hZ + O(h^2)$ as $h \to 0$. Therefore, for $h \to 0$,

$$\varphi(h\mathrm{D}\mathbf{f}(\mathbf{y}_0)) = (\exp(h\mathrm{D}\mathbf{f}(\mathbf{y}_0)) - \mathrm{Id})(h\mathrm{D}\mathbf{f}(\mathbf{y}_0))^{-1} = (\mathrm{D}\mathbf{f}(\mathbf{y}_0)) + O(h))\mathrm{D}\mathbf{f}(\mathbf{y}_0)^{-1} = \mathbf{I} + O(h)\ .$$

Whence

$$\varphi(\mathbf{O}) = \mathbf{I} \quad \Rightarrow \quad \varphi\big(h\mathrm{D}\mathbf{f}(\mathbf{y}_0)\big)\,\mathbf{f}(\mathbf{y}_0) = \mathbf{f}(\mathbf{y}_0)\ ,$$

as desired.

HINT 1 for (7-5.b): Recall [Lecture → Eq. (7.1.0.34)]; the solution of the IVP is $\mathbf{y}(t) = \exp(\mathbf{A}t)\mathbf{y}_0$. To facilitate formal calculations, you may assume that $\mathbf{A}$ is regular. ⌟

SOLUTION of (7-5.b):

Given $\mathbf{y}(0)$, a step of size h of the method gives

$$\begin{aligned}\mathbf{y}_1 &= \mathbf{y}(0) + h\ \varphi\big(h\ \mathrm{D}\mathbf{f}(\mathbf{y}(0))\big)\ \mathbf{f}(\mathbf{y}(0)) \\ &= \mathbf{y}(0) + h\ \varphi\big(h\ \mathbf{A}\big)\ \mathbf{A}\ \mathbf{y}(0) = \mathbf{y}(0) + h\ \big(\exp(\mathbf{A}h) - \mathrm{Id}\big)\ (h\mathbf{A})^{-1}\ \mathbf{A}\ \mathbf{y}(0) \\ &= \mathbf{y}(0) + \exp(\mathbf{A}h)\ \mathbf{y}(0) - \mathbf{y}(0) = \exp(\mathbf{A}h)\ \mathbf{y}(0) = \mathbf{y}(h)\ .\end{aligned}$$

Hence the discrete evolution exactly reproduces the exact evolution for (7.5.2).

SOLUTION of (7-5.c):

The discrete evolution $\boldsymbol{\Psi}^h$ associated to (7.5.1) is given by

$$\boldsymbol{\Psi}^h(\mathbf{y}) = \mathbf{y} + h\varphi(h\mathrm{D}\mathbf{f}(\mathbf{y}))\mathbf{f}(\mathbf{y}).$$

Thus, for a scalar linear ODE of the form $\dot{y} = \lambda y$ for $\lambda \in \mathbb{C}$ we have ($f(y) = \lambda y$, $Df(y) = \lambda$)

$$\boldsymbol{\Psi}^h(\mathbf{y}) = \mathbf{y} + h\frac{e^{h\lambda} - 1}{h\lambda}\lambda\mathbf{y} = e^{h\lambda}\mathbf{y}.$$

Thus, the stability function $S\colon \mathbb{C} \to \mathbb{C}$ is given by $S(z) = e^z$. Therefore, the region of stability of this method is

$$\mathcal{S}_{\boldsymbol{\Psi}} = \{z \in \mathbb{C} : |e^z| < 1\} = \{z \in \mathbb{C} : e^{\mathrm{Re}\, z} < 1\} = \{z \in \mathbb{C} : \mathrm{Re}\, z < 0\}.$$

It is illuminating to relate this to [Lecture → Rem. 11.4.3.22].

Alternatively, one may simply appeal to Sub-problem (7-5.b) to see that the method has the "ideal" region of stability as introduced in [Lecture → § 7.3.4.10].

SOLUTION of (7-5.d):

Given the auxiliary function `phim()`, the implementation of (7.5.1) is straightforward.

C++11-code 7.5.4: Code of `exponentialEulerStep` ↬ GITHUB

```
2 template <class Function, class Jacobian>
3 Eigen::VectorXd exponentialEulerStep(const Eigen::VectorXd &y0, Function &&f,
4                                       Jacobian &&df, double h) {
5   return y0 + h * phim(h * df(y0)) * f(y0);
6 }
```

HINT 1 for (7-5.e): The exact solution is

$$y(t) = \frac{y(0)}{y(0) + \big(1 - y(0)\big)e^{-t}}.$$

SOLUTION of (7-5.e):

M	error
2	0.00168674
4	0.000452805
8	0.000117603
16	2.99917e-05
32	7.57469e-06
64	1.90346e-06
128	4.77101e-07
256	1.1943e-07
512	2.98771e-08
1024	7.4717e-09
2048	1.86823e-09
4096	4.67095e-10
8192	1.16777e-10
16384	2.91974e-11
32768	7.29886e-12

Choosing a geometrically increasing sequence of equidistant timesteps is the most suitable approach to detecting conjectured algebraic convergence $|y(T) - y_M| = O(M^{-r})$ for $M \to \infty$.
In this experiment doubling the number of timesteps leads to a decrease of the error at final time by a factor of ≈ 4, which makes us conjecture

$$|y(T) - y_M| = O(M^{-2}) \quad \text{for} \quad M \to \infty \,.$$

In terms of the timestep size $h = \frac{1}{M}$ this means

$$\text{error}(h) = |y(T) - y_M| = O(h^2) \quad \text{for} \quad h \to 0 \,.$$

We conjecture that the exponential Euler method has order 2 .

C++11-code 7.5.5: Code for `testExpEulerLogODE()` ➙ GITHUB

```
2   // Final time
3   double T = 1.0;
4   // Initial value
5   Eigen::VectorXd y0(1);
6   y0 << 0.1;
7   // Function and Jacobian and exact solution
8   auto f = [](const Eigen::VectorXd &y) { return y(0) * (1.0 - y(0)); };
9   auto df = [](const Eigen::VectorXd &y) {
10    Eigen::MatrixXd dfy(1, 1);
11    dfy << 1.0 - 2.0 * y(0);
12    return dfy;
13  };
14  double exactyT = y0(0) / (y0(0) + (1.0 - y0(0)) * std::exp(-T));
15
16  // Container for errors
17  std::vector<double> error(15);
18
19  // Test many step sizes
20  for (int j = 0; j < 15; ++j) {
21    int M = std::pow(2, j + 1);
22    Eigen::VectorXd y = y0;
23    double h = T / M;
24    for (int k = 0; k < M; ++k) {
25      y = ExponentialIntegrator::exponentialEulerStep(y, f, df, h);
26    }
27
28    error[j] = std::abs(y(0) - exactyT);
29    std::cout << std::left << std::setfill(' ') << std::setw(3)
30              << "M = " << std::setw(7) << M << std::setw(8)
31              << "Error = " << std::setw(13) << error[j];
32    if (j > 0) {
33      std::cout << std::left << std::setfill(' ') << std::setw(10)
```

```
34                 << "Approximated order = " << std::log2(error[j - 1] / error[j])
35                 << std::endl;
36     } else
37       std::cout << std::endl;
38   }
```

HINT 1 for (7-6.a): Recall the stage form of implicit RK-SSM from [Lecture → Rem. 7.3.3.6]. ⌟

SOLUTION of (7-6.a):

Let us define $d_{i,j} := 0$ for $j \geq i$. Replacing $\mathbf{y}_1$ in the equation for $\mathbf{g}_i$, one obtains:

$$\mathbf{g}_i := (1 - v_i)\mathbf{y}_0 + v_i \left(\mathbf{y}_0 + h \sum_{j=1}^{s} b_j \mathbf{f}(\mathbf{g}_j) \right) + h \sum_{j=1}^{i-1} d_{i,j} \mathbf{f}(\mathbf{g}_j) = \mathbf{y}_0 + h \sum_{j=1}^{s} (v_i b_j + d_{i,j}) \mathbf{f}(\mathbf{g}_j),$$

$$\mathbf{y}_1 := \mathbf{y}_0 + h \sum_{j=1}^{s} b_j \mathbf{f}(\mathbf{g}_j)$$

Set $\mathbf{k}_i := \mathbf{f}(\mathbf{g}_i)$. Then, $\mathbf{k}_i = f(\mathbf{y}_0 + h \sum_{j=1}^{s} \overbrace{(v_i b_j + d_{i,j})}^{=:a_{i,j}} \mathbf{k}_j)$ and

$$\mathbf{y}_1 := \mathbf{y}_0 + h \sum_{j=1}^{s} b_j \mathbf{k}_j \tag{7.6.2}$$

Therefore, the Butcher scheme becomes

$$\begin{array}{c|c} \mathbf{c} & \mathbf{D} + \mathbf{v}\mathbf{b}^\top \\ \hline & \mathbf{b}^\top, \end{array} \quad ,$$

where $(\mathbf{D})_{i,j} = d_{i,j}$, $\mathbf{b} = [b_1, \ldots, b_s]^\top$, $\mathbf{v} = [v_1, \ldots, v_s]^\top$, and $c_i = \sum_{j=1}^{s} a_{i,j}$.

HINT 1 for (7-6.b): The stability function can be obtained by applying the single-step method to the scalar linear ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{R}$, see [Lecture → Ex. 7.1.0.9]. ⌟

SOLUTION of (7-6.b):

We follow the policy of the classical linear model problem analysis and apply the MIRK scheme to the ODE $y' = \lambda y$. Denote by $\mathbf{g} := [g_1, g_2]^\top$ and $z := h\lambda$, then:

$$\begin{aligned} \mathbf{g} &= (\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1 + z\mathbf{D}\mathbf{g} \\ (\mathbf{I} - z\mathbf{D})\mathbf{g} &= (\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1 \,, \end{aligned}$$

where $\mathbf{1} = [1, \ldots, 1]^\top$. Consider

$$y_1 = y_0 + z\mathbf{b}^\top \mathbf{g} = y_0 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}((\mathbf{1} - \mathbf{v})y_0 + \mathbf{v}y_1) \,.$$

Then

$$(1 - z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}\mathbf{v})y_1 = (1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}(\mathbf{1} - \mathbf{v}))y_0 \,,$$

whence

$$S(z) = \frac{1 + z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}(\mathbf{1} - \mathbf{v})}{1 - z\mathbf{b}^\top (\mathbf{I} - z\mathbf{D})^{-1}\mathbf{v}} \,. \tag{7.6.3}$$

Alternatively, you may use the formula from [Lecture $\rightarrow$ Thm. 7.3.4.4] together with the Butcher scheme derived in the previous sub-problem.

SOLUTION of (7-6.c):

In the scalar case we write for the autonomous ODE $\dot{y} = f(y)$. All vectors in (7.6.1) become more real numbers, which yields

$$\mathbf{F}\left(\begin{bmatrix} g_1 \\ g_2 \\ y_1 \end{bmatrix}\right) = \begin{bmatrix} g_1 - (1-v_1)y_0 - v_1 y_1 \\ g_2 - (1-v_2)y_0 - v_2 y_1 - h d_{2,1} f(g_1) \\ y_1 - y_0 - h\sum_{j=1}^{2} b_j f(g_j) \end{bmatrix} . \tag{7.6.4}$$

SOLUTION of (7-6.d):

Well, use (7.6.4) and compute the partial derivatives for all variables according to [Lecture → Eq. (0.3.2.16)]

$$\blacktriangleright \quad \mathrm{D}F\left(\begin{bmatrix} g_1 \\ g_2 \\ y_1 \end{bmatrix}\right) = \mathbf{I}_3 - \begin{bmatrix} 0 & 0 & v_1 \\ hd_{2,1}f'(g_1) & 0 & v_2 \\ hb_1 f'(g_1) & hb_2 f'(g_2) & 0 \end{bmatrix}.$$

SOLUTION of (7-6.e):

We just need to recall the general formula for a Newton step for the non-linear system of equations $\mathbf{F}(\mathbf{x}) = 0$:

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathrm{D}\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)}) \,.$$

Of course, the multiplication with the inverse of the Jacobi matrix indicates that a linear system has to be solved. This is done using EIGEN's built-in functions.

C++11-code 7.6.5: Solution code for `Newton2Steps()` ↪ GITHUB

```
template <class Func, class Jac>
Eigen::VectorXd Newton2Steps(Func &&F, Jac &&DF, Eigen::VectorXd z) {
  // First Newton step
  z = z - DF(z).lu().solve(F(z));
  // Second Newton step
  z = z - DF(z).lu().solve(F(z));
  return z;
}
```

SOLUTION of (7-6.f):

C++11-code 7.6.7: Solution code for `MIRKStep` ↪ GITHUB

```
template <class Func, class Jac>
double MIRKStep(Func &&f, Jac &&df, double y0, double h) {
  // Coefficients of MIRK
  const double v1 = 1.0;
  const double v2 = 344.0 / 2025.0;
  const double d21 = -164.0 / 2025.0;
  const double b1 = 37.0 / 82.0;
  const double b2 = 45.0 / 82.0;

  // F derived from MIRK scheme (vector valued)
  auto F = [&f, &y0, &v1, &v2, &d21, &b1, &b2,
            &h](Eigen::Vector3d z) -> Eigen::Vector3d {
    Eigen::Vector3d ret;
    ret << z(0) - (1 - v1) * y0 - v1 * z(2),
        z(1) - (1 - v2) * y0 - v2 * z(2) - h * d21 * f(z(0)),
        z(2) - y0 - h * (b1 * f(z(0)) + b2 * f(z(1)));
    return ret;
  };
  // Jacobian of F (matrix-valued)
  auto DF = [&df, &v1, &v2, &d21, &b1, &b2,
             &h](Eigen::Vector3d z) -> Eigen::Matrix3d {
    Eigen::Matrix3d M;
    M << 0, 0, v1, h * d21 * df(z(0)), 0, v2, h * b1 * df(z(0)),
        h * b2 * df(z(1)), 0;
    return Eigen::Matrix3d::Identity() - M;
  };
  // Initial Guess for Newton steps
  Eigen::Vector3d z(0.0, 0.0, y0);

  // Approximate z, s.t.  F(z) = 0
  z = Newton2Steps(F, DF, z);
  return z(2);
}
```

SOLUTION of (7-6.g):

C++11-code 7.6.8: Solution code for `MIRKSolve` ⇢ GITHUB

```
template <class Func, class Jac>
double MIRKSolve(Func &&f, Jac &&df, double y0, double T, unsigned int M) {
  // Step size
  const double h = T / M;
  // Will contain next step
  double y = y0;
  // Perform N quidistant steps
  for (unsigned int i = 0; i < M; ++i) {
    y = MIRKStep(f, df, y, h);
  }
  // Return final value at t = T
  return y;
}
```

SOLUTION of (7-6.h):

C++11-code 7.6.10: Solution code for `mirk_main.cc` ➙ GITHUB

```
int main() {
  // r.h.s
  auto f = [](double y) -> double { return 1 + y * y; };
  // Jacobian of f
  auto df = [](double y) -> double { return 2 * y; };
  // Initial data
  const double y0 = 0.;
  // Final time
  const double T = 1.;
  // Exact solution at t = T = 1
  const double yex = tan(T);

  std::cout << "Convergence MIRK for IVP y' = 1+y^2 " << std::endl;
  // Table header
  std::cout << "M"
            << "\t"
            << "yend"
            << "\t"
            << "err" << std::endl;
  for (unsigned int M = 4; M <= 512; M *= 2) {
    // Solve up to time T = 1, using M equidistant steps
    double yend = MIRK::MIRKSolve(f, df, y0, T, M);
    // Compute error
    double err = std::abs(yex - yend);
    // Print table
    std::cout << M << "\t" << yend << "\t" << err << std::endl;
  }
  return 0;
}
```

M	y_M	error norm
4	1.60432	0.0469136
8	1.56842	0.0110085
16	1.56009	0.00268694
32	1.55807	0.00066483
64	1.55757	0.000165414
128	1.55745	4.12584e-05
256	1.55742	1.0303e-05
512	1.55741	2.5742e-06

The error at final time is roughly reduced by a factor of four when halving the timestep size $h := \frac{1}{M}$.

The error converges like $O(h^2)$ for $h \to 0$. Thus, the experimental rate of convergence of the methods is 2 .

SOLUTION of (7-7.a):

Recall the definition:

Definition [Lecture → Def. 6.4.0.9]. Explicit Runge-Kutta single-step method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{i-1} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an **s-stage explicit Runge-Kutta single step method** (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, $\mathbf{f}: \Omega \to \mathbb{R}^N$, is defined by ($\mathbf{y}_0 \in D$)

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j)\,, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i\,.$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$, $i = 1,\ldots,s$, are called **increments**, $h > 0$ is the size of the timestep.

Also recall the shorthand notation for (explicit) Runge-Kutta methods

Butcher scheme ▷

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & 0 & \cdots & 0 \\ c_2 & a_{21} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{s,s-1} \quad 0 \\ \hline & b_1 & \cdots & b_{s-1} \quad b_s \end{array}\,.$$

(Note that the Butcher matrix $\mathfrak{A}$ is a strictly lower triangular $s \times s$-matrix)

[Lecture → Eq. (6.4.0.11)]

Given the right-hand side function $\mathbf{f}$ of the ODe $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ the implementation of an explicit Runge-Kutta single-step method is straightforward, because the increments $\mathbf{k}_i$ can be computed one after the other by plugging linear combinations of previous increments into $\mathbf{f}$.

C++-code 7.7.3: Solution of (7-7.a) ↪ GITHUB

```
2  Eigen::Vector2d PredPrey(Eigen::Vector2d y0, double T, unsigned int M) {
3    double h = T / M;
4    Eigen::Vector2d y = y0;
5
6    // Define right-hand-saide function for Lotka-Volterra ODE
7    auto f = [](Eigen::Vector2d y) -> Eigen::Vector2d {
8      return {(1 - y(1)) * y(0), (y(0) - 1) * y(1)};
9    };
10   // Main timstepping loop:  uniform stepsize
11   for (int j = 0; j < M; ++j) {
12     // Compute increments and updates according to [Lecture →
         Def. 6.4.0.9] for the method
13     // described by the Butcher scheme (7.7.1)
14     Eigen::Vector2d k1 = f(y);
15     Eigen::Vector2d k2 = f(y + h * k1);
16     Eigen::Vector2d k3 = f(y + (h / 4.) * k1 + (h / 4.) * k2);
17     y = y + (h / 6.) * k1 + (h / 6.) * k2 + (2. * h / 3.) * k3;
18   }
19
```

```
20    return y;
21  }
```

HINT 1 for (7-7.b): $\mathbf{y}_M$ for $M = 128$ is $[6.47563e-27, 40.7386]^\top$.

SOLUTION of (7-7.b):

♯steps	error	rate
4	inf	
8	inf	-nan
16	inf	-nan
32	2.2565	inf
64	0.148092	3.92952
128	0.0268461	2.46371
256	0.00421868	2.66985
512	0.000596288	2.82271
1024	7.93781e-05	2.9092
2048	1.02263e-05	2.95645
4096	1.28099e-06	2.99695
8192	1.43426e-07	3.15889

We observe that a doubling of the number of steps leads to a decrease of the error at final time by roughly a factor of 8.
Thus the scheme seems to be of order 3.
In case of big stepsize h the scheme is *unstable*. At least 32 steps are needed for the solution to be stable.

C++-code 7.7.4: Solution of (7-7.b) ↬ GITHUB

```
2  void SimulatePredPrey() {
3    // Parameters
4    double T = 1.0;
5    Eigen::Vector2d y0(100.0, 1.0);
6
7    // (Approximate) reference solution
8    Eigen::Vector2d y_ref = PredPrey(y0, T, std::pow(2, 14));
9
10   Eigen::ArrayXd error(12);
11   Eigen::ArrayXd M(12);
12   // Studying the error for geometrically increasing numbers of
       equidistant
13   // timesteps is the most appropriate approach to empirically exploring
14   // algebraic convergence.
15   M << 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192;
16
17   // Compute errors
18   for (int i = 0; i < M.size(); ++i) {
19     Eigen::Vector2d y = PredPrey(y0, T, M(i));
20     error(i) = (y - y_ref).norm();
21   }
22   // Print error table
23   PrintErrorTable(M, error);
24 }
25
26 void PrintErrorTable(const Eigen::ArrayXd& M, const Eigen::ArrayXd& error) {
27   std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::setw(15)
28             << "rate" << std::endl;
29   // Formatted output in C++
30   for (unsigned int i = 0; i < M.size(); ++i) {
31     std::cout << std::setw(15) << M(i) << std::setw(15) << error(i);
32     if (i > 0) {
33       std::cout << std::setw(15) << std::log2(error(i - 1) / error(i));
34     }
35     std::cout << std::endl;
36   }
```

```
37 }
```

SOLUTION of (7-7.c):

We obtain the stability function $S(z)$ by applying our method to the model problem $\dot{y}(t) = \lambda y$, $y(0) = y_0$ and by writing the result in the form $y_1 = S(z)y_0$, where $z := h\lambda$. For the increment, we obtain

$$\begin{aligned} k_1 &= \lambda y_0, \\ k_2 &= \lambda(y_0 + hk_1) = \lambda y_0(1+z), \\ k_3 &= \lambda(y_0 + \tfrac{h}{4}(k_1+k_2)) = \lambda y_0(1+\tfrac{1}{2}z+\tfrac{1}{4}z^2)\,, \end{aligned}$$

and for the update

$$\begin{aligned} y_1 &= y_0 + \frac{h\lambda y_0}{6}\left(1+1+z+4\left(1+\frac{1}{2}z+\frac{1}{4}z^2\right)\right) \\ &= y_0 \underbrace{\left(1+z+\tfrac{1}{2}z^2+\tfrac{1}{6}z^3\right)}_{=:S(z)}\,. \end{aligned}$$

In agreement with [Lecture → Cor. 7.1.0.20] the stability function of the 3-stage explicit RK-SSM is a *polynomial* of degree 3.

Note that the stability function agrees with the Taylor series of $z \mapsto \exp(z)$ in the first four terms, which implies

$$|S(z) - \exp(z)| = O(z^4) \quad \text{for} \quad z \to 0\,.$$

According to [Lecture → Lemma 7.1.0.23] this is necessary for an explicit Runge-Kutta single-step method of order 3.

SOLUTION of (7-8.a):

Since $\mathbf{grad}\, V$ is continuous, $t \mapsto \mathbf{y}(t)$ is a C^1-function. Therefore, also $t \mapsto V(\mathbf{y}(t))$ is continuously differentiable. This means that $\varphi(t) := V(\mathbf{y}(t))$ is non-increasing if and only if its *derivative is* ≤ 0. By the chain rule we compute

$$\dot{\varphi}(t) = \mathbf{grad}\, V(\mathbf{y}(t))^\top \dot{\mathbf{y}}(t) \overset{(7.8.1)}{=} -\|\mathbf{grad}\, V(\mathbf{y}(t))\|^2 \leq 0\,.$$

SOLUTION of (7-8.j):

Doubling the number of timesteps roughly reduces the error by a factor of 16, which hints that the SDIRK RK-SSM is of order 4 , because the order of a Runge-Kutta single-step method predicts its algebraic rate of convergence with respect to $h \sim N^{-1} \to 0$.

SOLUTION of (7-8.b):

For $N = 1$ the gradient boils down to the standard derivative. Thus we find

$$V(y) = \tfrac{1}{2}\lambda y^2 \quad \Rightarrow \quad \frac{dV}{dy}(y) = \lambda y \ .$$

SOLUTION of (7-8.c):

Note that thanks to the lower triangular shape of the Butcher matrix $\mathfrak{A}$ the increments can be computed sequentially $\mathbf{k}_1 \to \mathbf{k}_2 \to \cdots \to \mathbf{k}_5$.

$$
\begin{aligned}
\mathbf{k}_1 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{1}{4}\mathbf{k}_1\big)\Big) &= \mathbf{0}\,,\\
\mathbf{k}_2 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{1}{2}\mathbf{k}_1 + \frac{1}{4}\mathbf{k}_2\big)\Big) &= \mathbf{0}\,,\\
\mathbf{k}_3 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{17}{50}\mathbf{k}_1 - \frac{1}{25}\mathbf{k}_2 + \frac{1}{4}\mathbf{k}_3\big)\Big) &= \mathbf{0}\,,\\
\mathbf{k}_4 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{371}{1360}\mathbf{k}_1 - \frac{137}{2720}\mathbf{k}_2 + \frac{15}{544}\mathbf{k}_3 + \frac{1}{4}\mathbf{k}_4\big)\Big) &= \mathbf{0}\,,\\
\mathbf{k}_5 - \mathbf{f}\Big(\mathbf{y}_0 + h\big(\frac{25}{24}\mathbf{k}_1 - \frac{49}{48}\mathbf{k}_2 + \frac{125}{16}\mathbf{k}_3 - \frac{85}{12}\mathbf{k}_4 + \frac{1}{4}\mathbf{k}_5\big)\Big) &= \mathbf{0}\,.
\end{aligned}
$$

We tackle these equations from top to bottom. The highlighted increments are the unknowns in the equations, those printed in blue are known already.

Writing $\mathfrak{A} = \big[a_{i,j}\big]_{i,j=1}^{5}$ for the lower triangular Butcher matrix of the SDIRK method and using $a_{j,j} = \frac{1}{4}$, a more compact way to describe the increment equations is

$$
\mathbf{k}_j - \mathbf{f}(\mathbf{y}_0 + \tfrac{1}{4}h\mathbf{k}_j + h\sum_{\ell=1}^{j-1} a_{j,\ell}\mathbf{k}_\ell) = \mathbf{0}\,, \quad j = 1,2,3,4,5\,.
$$

SOLUTION of (7-8.d):

The stages defined in [Lecture → Eq. (7.3.3.7)] can be computed sequentially $\mathbf{g}_1 \to \mathbf{g}_2 \to \cdots \to \mathbf{g}_5$:

$$\begin{aligned}
\mathbf{g}_1 - h\Big(\frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_2 - h\Big(\frac{1}{2}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_3 - h\Big(\frac{17}{50}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) - \frac{1}{25}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_3)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_4 - h\Big(\frac{371}{1360}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) - \frac{137}{2720}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2) + \frac{15}{544}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_3) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_4)\Big) &= \mathbf{0}\,,\\
\mathbf{g}_5 - h\Big(\frac{25}{24}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_1) - \frac{49}{48}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_2) + \frac{125}{16}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_3) - \frac{85}{12}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_4) + \frac{1}{4}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_5)\Big) &= \mathbf{0}\,.
\end{aligned}$$

The generic form of a $N \times N$ non-linear system of equations to obtain another stage $\mathbf{g}$ is

$$\mathbf{g}_j - \tfrac{1}{4}h\mathbf{f}(\mathbf{y}_0+\mathbf{g}_j) = h\sum_{\ell=1}^{j-1} a_{j,\ell}\mathbf{f}(\mathbf{y}_0+\mathbf{g}_\ell)\,, \quad j=1,2,3,4,5\,, \tag{7.8.3}$$

Here $a_{j,\ell}$ denotes entries of the Butcher matrix $\mathfrak{A} \in \mathbb{R}^{5,5}$.

HINT 1 for (7-8.e): Consult to learn about ways to decide when to terminate a Newton iteration. ⌟

SOLUTION of (7-8.e):

The Newton iteration for the generic form (7.8.3) of the stage equation reads

$$\mathbf{g}^{(k+1)} = \mathbf{g}^{(k)} - \left(\mathbf{I}_N - \tfrac{1}{4}h\mathrm{D}f(\mathbf{y}_0 + \mathbf{g}^{(k)})\right)^{-1}(\mathbf{g}^{(k)} - \tfrac{1}{4}h\mathbf{f}(\mathbf{y}_0 + \mathbf{g}^{(k)}) - \mathbf{b}) \,. \tag{7.8.4}$$

where $\mathbf{b}$ is a known vector depending on previous stages.

It is smart to write an auxiliary function

```
template <typename Functor, typename Jacobian>
Eigen::VectorXd SolveGenStageEquation(
Functor &&f, Jacobian &&df,
const Eigen::VectorXd &y,
const Eigen::VectorXd &b, double h,
double rtol = 1E-6, double atol = 1E-8);
```

that solves (7.8.3) by means of the Newton iteration (7.8.4).

C++ code 7.8.5: Sub-problem (7-8.e): Auxiliary function `solveGenStageEquation()`
➥ GITHUB

```
2  // Use Newton method to approximate a stage.
3  template <typename Functor, typename Jacobian>
4  Eigen::VectorXd SolveGenStageEquation(Functor &&f, Jacobian &&df,
5                                        const Eigen::VectorXd &y,
6                                        const Eigen::VectorXd &b, double h,
7                                        double rtol = 1E-6, double atol = 1E-8) {
8    // Need to solve the equation lhs(g) = g - h*f(y+g)/4 - b = 0.
9    // lhs and its Jacobian Jlhs
10   auto lhs = [f, y, b, h](const Eigen::VectorXd &g) {
11     Eigen::VectorXd val = g - 0.25 * h * f(y + g) - b;
12     return val;
13   };
14   auto Jlhs = [df, y, h](const Eigen::VectorXd &g) {
15     // Jlhs(g) = Id - h*df(y+g)/4
16     int dim = y.size();
17     Eigen::MatrixXd Jval =
18         Eigen::MatrixXd::Identity(dim, dim) - 0.25 * h * df(y + g);
19     return Jval;
20   };
21
22   // Perform Newton iterations:
23   Eigen::VectorXd g = Eigen::VectorXd::Zero(y.size());  // initial guess g=0.
24   Eigen::VectorXd delta =
25       -Jlhs(g).lu().solve(lhs(g));  // Newton correction term.
26   int iter = 0;
27   int maxiter = 100;  // If correction based termination does not work.
28   while (delta.norm() > atol && delta.norm() > rtol * g.norm() &&
29          iter < maxiter) {
30     g = g + delta;
31     delta = -Jlhs(g).lu().solve(lhs(g));
32     iter++;
33   }
34   return g + delta;  // Perform the final step
35 }
```

C++ code 7.8.6: Sub-problem (7-8.e): Solution of stage equations for SDIRK method
➙ GITHUB

```cpp
// Compute the stages [g_1, ... , g_5] of the SDIRK method based on Newtons
// method
template <typename Func, typename Jac>
std::array<Eigen::VectorXd, 5> ComputeStages(Func &&f, Jac &&df,
                                             const Eigen::VectorXd &y, double h,
                                             double rtol = 1E-6,
                                             double atol = 1E-8) {
  std::array<Eigen::VectorXd, 5> G;  // array of stages
  int dim = y.size();
  Eigen::MatrixXd coeffs = ButcherMatrix();

  for (int i = 0; i < 5; i++) {
    // Calculate coefficients from previous stages:
    Eigen::VectorXd b = Eigen::VectorXd::Zero(dim);
    for (int j = 0; j < i; j++) {
      b += coeffs(i, j) * f(y + G[j]);
    }
    b *= h;
    // Compute stage i and store in G.at(i):
    G[i] = SolveGenStageEquation(f, df, y, b, h, rtol, atol);
  }

  return G;
}
```

SOLUTION of (7-8.f):

After computing the stages $\mathbf{g}_i$ or the increments $\mathbf{k}_i$, $i = 1, \ldots, 5$, by [Lecture → Def. 7.3.3.1]/[Lecture → Eq. (7.3.3.8)] we can evaluate

$$\Psi^h \mathbf{y}_0 = \mathbf{y}_0 + h \sum_{i=1}^{5} b_i \mathbf{k}_i \,,$$

$$\Psi^h \mathbf{y}_0 = \mathbf{y}_0 + h \sum_{i=1}^{5} b_i \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_i) \,,$$

where

$$[b_1, b_2, b_3, b_4, b_5] = \left[\frac{25}{24}, -\frac{49}{48}, \frac{125}{16}, -\frac{85}{12}, \frac{1}{4}\right] .$$

C++ code 7.8.7: Sub-problem (7-8.f): Implementation of SDIRK discrete evolution operator
➥ GITHUB

```
2  // Compute one step of the SDIRK scheme
3  template <typename Func, typename Jac>
4  Eigen::VectorXd DiscEvolSDIRK(Func &&f, Jac &&df, const Eigen::VectorXd &y,
5                                double h, double rtol = 1E-6,
6                                double atol = 1E-8) {
7    // The b weights are in the last row of coeffs.
8    Eigen::MatrixXd coeffs = ButcherMatrix();
9    int n_stages = coeffs.cols();
10   Eigen::VectorXd b = coeffs.row(n_stages);
11
12   // Vector after one step of the SDIRK scheme
13   Eigen::VectorXd Psi;
14
15   Psi = y;
16   // Compute array of stages
17   std::array<Eigen::VectorXd, 5> G = ComputeStages(f, df, y, h, rtol, atol);
18   for (int i = 0; i < n_stages; i++) {
19     Psi += h * b(i) * f(y + G[i]);
20   }
21
22   return Psi;
23 }
```

SOLUTION of (7-8.g):

Using the chain rule and

$$\mathrm{D}_{\mathbf{x}}\{\mathbf{x} \mapsto \|\mathbf{x}\|^2\}(\mathbf{y}) = 2\mathbf{y}\,,$$

we find

$$-\,\mathbf{grad}\, V(\mathbf{y}) = -2\cos(\|\mathbf{y}\|^2)\mathbf{y} - \lambda 2(\mathbf{d}^\top \mathbf{y})\mathbf{d}\,,$$

which agrees with the right-hand side function of the ODE. So we arrive at

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := -2\cos(\|\mathbf{y}\|^2)\mathbf{y} - \lambda 2(\mathbf{d}^\top \mathbf{y})\mathbf{d}\,. \tag{7.8.9}$$

HINT 1 for (7-8.h): Recall the stiffness criteria [Lecture → Eq. (7.2.0.13)] and [Lecture → Eq. (7.2.0.14)].

SOLUTION of (7-8.h):

Taking the cue from the criteria [Lecture → Eq. (7.2.0.13)] and [Lecture → Eq. (7.2.0.14)] we have to inverstigate the Jacobian $\mathrm{D}\mathbf{f}(\mathbf{0})$. Differentiating $\mathbf{f}$ from (7.8.9) yields

$$\mathrm{D}\mathbf{f}(\mathbf{y}) = 4\sin(\|\mathbf{y}\|^2)\mathbf{y}\mathbf{y}^\top - 2\cos(\|\mathbf{y}\|^2)\mathbf{I}_N - 2\lambda\mathbf{d}\mathbf{d}^\top \in \mathbb{R}^{N,N} . \tag{7.8.10}$$

Here we have used the product rule for the first term and the fact that the derivative of linear expressions are the expressions themselves. In particular, we get

$$\mathrm{D}\mathbf{f}(\mathbf{0}) = -2\mathbf{I}_N - 2\lambda\mathbf{d}\mathbf{d}^\top .$$

This matrix is a rank-1 perturbation of a scaled identity matrix and has the eigenvalues (spectrum)

$$\sigma(\mathrm{D}\mathbf{f}(\mathbf{0})) = \{-2, -2-2\lambda\} .$$

Thus, if $\lambda \gg 1$ we face a stiff problem for states close to zero.

HINT 1 for (7-8.i): Create suitable lambda functions for $\mathbf{f}$ and $D\mathbf{f}$. ⌟

SOLUTION of (7-8.i):

C++ code 7.8.11: Sub-problem (7-8.i): SDIRK solver for gradient flow problem ⇢ GITHUB

```
std::vector<Eigen::VectorXd> SolveGradientFlow(const Eigen::VectorXd &d,
                                               double lambda,
                                               const Eigen::VectorXd &y0,
                                               double T, unsigned int M) {
  // initialize solution vector
  std::vector<Eigen::VectorXd> sol(M + 1, Eigen::VectorXd::Zero(y0.size()));

  // Define the right hand side of the ODE y' = f(y), and the Jacobian of f.
  auto f = [d, lambda](const Eigen::VectorXd &y) {
    Eigen::VectorXd val =
        -2. * std::cos(y.squaredNorm()) * y - 2. * lambda * d.dot(y) * d;
    return val;
  };
  auto df = [d, lambda](const Eigen::VectorXd &y) {
    int dim = y.size();
    Eigen::MatrixXd term1 = 4. * std::sin(y.squaredNorm()) * y * y.transpose();
    Eigen::MatrixXd term2 =
        -2. * std::cos(y.squaredNorm()) * Eigen::MatrixXd::Identity(dim, dim);
    Eigen::MatrixXd term3 = -2. * lambda * d * d.transpose();
    Eigen::MatrixXd dfval = term1 + term2 + term3;
    return dfval;
  };

  // Split the interval [0,T] into M intervals of size h.
  double h = T / M;
  Eigen::VectorXd y = y0;
  sol[0] = y;
  // Evolve up to time T:
  for (int i = 1; i <= M; i++) {
    y = DiscEvolSDIRK(f, df, y, h);
    sol[i] = y;
  }
  return sol;
}
```

SOLUTION of (8-1.a):

In both cases the flux function depends on the state only. Matching with [Lecture → Eq. (8.2.1.6)]:

for (8.1.1) $\quad f(u) = \frac{1}{2}u^2$,

for (8.1.1) $\quad f(u) = \frac{2}{3}u^{3/2}$.

HINT 1 for (8-1.j): The solution is constant along the characteristics. In case characteristics intersect, there will be a shock that moves at variable speed, see [Lecture → Eq. (8.2.4.2)]. Get some ideas from [Lecture → Ex. 8.2.6.4]. ⌟

SOLUTION of (8-1.j):

We have $u_0(x) \geq 0$ and $f'(u) = \sqrt{u}$, which is non-negative and increasing. Therefore all information propagates to the right.

Since the flux function $f(u) =$ is convex, we expect to see a shock eventually developing from characteristics to the right of $x = \frac{1}{2}$, where the initial condition (8.1.5) is decreasing. This shock will start in $x = 1$ at $t = 0$ and move to the right with increasing speed.

Since u_0 is increasing in $[0, \frac{1}{2}]$, a rarefaction fan will develop in that zone. Its left edge will stay at $x = 0$ and its right edge will finally catch up with the shock.

[Lecture $\rightarrow$ Fig. 388] qualitatively describes the situation.

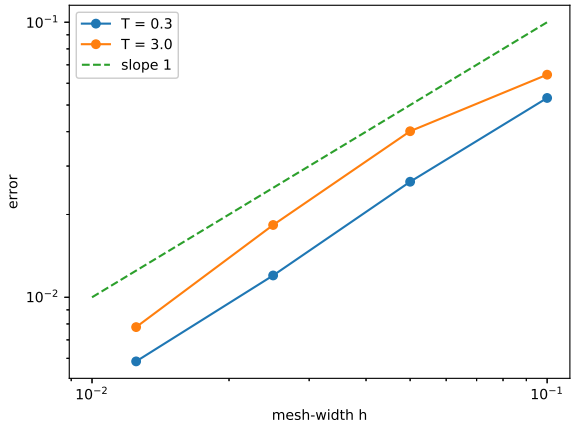
SOLUTION of (8-1.k):

C++11 code 8.1.9: Sub-problem (8-1.k), function `numexpBurgersGodunov()`, ↪ GITHUB

```
Eigen::VectorXd reduce(const Eigen::VectorXd &mu, unsigned int N) {
  Eigen::VectorXd mu_sub(N + 1);
  int fraction = mu.size() / N;
  for (int j = 0; j < N + 1; ++j) {
    mu_sub(j) = mu(j * fraction);
  }
  return mu_sub;
}

Eigen::Matrix<double, 3, 4> numexpBurgersGodunov() {
  const unsigned int N_large = 3200;
  Eigen::Vector2d T{0.3, 3.0};
  Eigen::Vector4i N{5 * 10, 5 * 20, 5 * 40, 5 * 80};
  Eigen::Vector4d h;
  for (int i = 0; i < 4; ++i) h(i) = 5.0 / N(i);

  Eigen::Matrix<double, 3, 4> result;
  result.row(0) = h.transpose();

  for (int k = 0; k < 2; ++k) {
    Eigen::VectorXd mu_ref = solveBurgersGodunov(T(k), N_large);
    Eigen::Vector4d error;
    for (int i = 0; i < 4; ++i) {
      Eigen::VectorXd mu = solveBurgersGodunov(T(k), N(i));
      Eigen::VectorXd mu_ref_sub = reduce(mu_ref, N(i));
      error(i) = h(i) * (mu - mu_ref_sub).lpNorm<1>();
    }
    result.row(k + 1) = error.transpose();
  }

  return result;
}
```

C++11 code 8.1.10: Sub-problem (8-1.k), in function `main()`, ↪ GITHUB

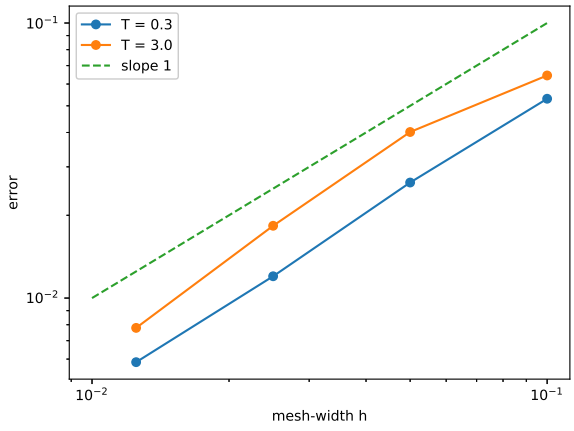
```
  Eigen::Matrix<double, 3, 4> result = BurgersEquation::numexpBurgersGodunov();

  // Write the result to a file that can be used for plotting.
  std::ofstream error_file;
  error_file.open("error.csv");
  error_file << result.format(BurgersEquation::CSVFormat) << std::endl;
  error_file.close();
  std::cout << "Generated " CURRENT_BINARY_DIR "/error.csv" << std::endl;
  std::system("python3 " CURRENT_SOURCE_DIR "/plot_error.py " CURRENT_BINARY_DIR
              "/error.csv " CURRENT_BINARY_DIR "/error.eps");
```



h	0.1	0.05	0.025	0.0125
$e(0.3,h)$	0.053	0.026	0.012	0.006
$e(3,h)$	0.064	0.040	0.018	0.007

◁ Plot of space-time L^1-error versus the spatial meshwidth = timestep.

SOLUTION of (8-1.l):



We follow [Lecture → § 3.2.2.5]. The error curves are almost straight lines in a doubly logarithmic plot, which indicates algebraic convergence,
From Fig. 93 we read off slope 1, which means convergence rate 1 for both final times.
Alternatively, from the table created in Subproblem (8-1.f) we see that halving meshwidth and timestep leads to a reduction of the error by roughly a factor of 2.

HINT 1 for (8-1.b): By monotonicity [Lecture → Thm. 8.2.7.1], $u(x,t) > 0$ for all $(x,t) \in \mathbb{R}\times]0,T[$. Then the proof boils down to clever use of the chain rule. ⌟

SOLUTION of (8-1.b):

Assume u solves (8.1.1), is positive, and is C^1-*smooth*. Then, by the chain rule,

$$\begin{aligned}\frac{\partial w}{\partial t}+\frac{\partial}{\partial x}\left(\frac{2}{3}w^{3/2}\right) &= 2u\frac{\partial u}{\partial t}+2u^2\frac{\partial u}{\partial x}\\ &= 2u\left(\frac{\partial u}{\partial t}+\frac{\partial}{\partial x}\left(\frac{1}{2}u^2\right)\right)=0.\end{aligned}$$

HINT 1 for (8-1.c): The flux function is the f in [Lecture $\rightarrow$ Eq. (8.2.2.1)]. It is convex, if $f''(u) > 0$ everywhere. ⌟

SOLUTION of (8-1.c):

The flux function is $f(w) = \frac{2}{3}w^{3/2}$, with $f''(w) = \frac{1}{2\sqrt{w}} > 0$ for $w > 0$. By monotonicity, we will never have $w = 0$ if $w_0 > 0$ everywhere, and of course $w < 0$ is impossible by definition.

HINT 1 for (8-1.d): You will need the Rankine-Hugoniot jump condition [Lecture → Eq. (8.2.4.2)] and may use [Lecture → Lemma 8.2.5.4]. ⌟

SOLUTION of (8-1.d):

We rely on the discussion of [Lecture → Section 8.2.5] and, in particular, [Lecture → Lemma 8.2.5.4]:

(I) For the case of the Burgers equation (8.1.1), we have $f'(u) = u$ and there are characteristics emanating from the negative x-axis with a velocity of $f'(2) = 2$, and characteristics emanating from the positive x-axis with a velocity of $f'(1) = 1$. Hence we will have a shock emanating from the origin with a velocity (the Rankine-Hugoniot velocity [Lecture → Eq. (8.2.5.3)]) of

$$\dot{s} = \frac{f(2) - f(1)}{2 - 1} = \frac{2^2}{2} - \frac{1^2}{2} = \frac{3}{2},$$

so the solution is

$$u(x,t) = \begin{cases} 1, & x/t > 3/2 \\ 2 & x/t < 3/2. \end{cases}$$

(I) The second case of (8.1.2) is similar. We have $f'(w) = \sqrt{w}$ and the characteristics emanating from the negative and positive x-axes will intersect. This time the shock moves with a velocity of

$$\dot{2} = \frac{f(4) - f(1)}{4 - 1} = \frac{2 \cdot 4^{3/2}/3 - 2 \cdot 1^{3/2}/3}{4 - 1} = \frac{14}{9},$$

so

$$w(x,t) = \begin{cases} 1, & x/t > 14/9 \\ 4 & x/t < 14/9. \end{cases}$$

HINT 1 for (8-1.e): Recall [Lecture → Lemma 8.2.5.10] about rarefaction solutions. ⌟

SOLUTION of (8-1.e):

Here, the characteristics diverge, so we get a rarefaction solution in the wedge between them. For (8.1.1), $f'(u) = u$, so using [Lecture $\rightarrow$ Lemma 8.2.5.10], we get

$$u(x,t) = \begin{cases} 2, & x/t > 2 \\ x/t & 1 < x/t < 2 \\ 1 & x/t < 1, \end{cases}$$

and for (8.1.2), $f'(w) = \sqrt{w}$, giving $g(x/t) = (x/t)^2$, so

$$w(x,t) = \begin{cases} 4, & x/t > 2 \\ (x/t)^2 & 1 < x/t < 2 \\ 1 & x/t < 1. \end{cases}$$

SOLUTION of (8-1.f):

The relationship $w = u^2$ is obviously violated for the shock solution and holds only for the rarefaction solution. However, this is not a logical dilemma, because a shock solution obviously fails to satisfy the C^1-condition of Sub-problem (8-1.b). Thus the transformation by the change rule cannot be applied when adopting the concept of weak solutions.

The problem sends a warning that manipulations of conservation laws using the formulas of classical calculus can be dangerous.

SOLUTION of (8-1.g):

The Godunov numerical flux is a two-point flux function such that the associated semi-discrete evolution for (8.1.2) in conservation form reads, see [Lecture → Eq. (8.3.2.8)],

$$\dot{\mu}_j(t) = -\frac{1}{h}\big[F(\mu_j, \mu_{j+1}) - F(\mu_{j-1}, \mu_j)\big] \ , \tag{8.1.3}$$

where F in this case is defined as $F(v,w) = f(u^{\downarrow}(v,w))$, where $u^{\downarrow}$ is defined as in [Lecture → Eq. (8.3.4.31)], because the flux function $f(u) = \frac{2}{3}u^{3/2}$ is convex.

We use the numbering of the different cases from [Lecture → Eq. (8.3.4.31)], [Lecture → Fig. 420]–[Lecture → Fig. 424]. Note that we will never encounter $f'(v) < 0$ or $f'(w) < 0$, so two of five cases (❷ and ❺) need not be taken into account. Moreover, the Rankine-Hugoniot speed $\dot{s}$ will always be positive, because the flux function is increasing. This eliminates case ❶. Finally we have

$$u^{\downarrow}(v,w) = v \quad \text{for all} \quad v, w > 0 \ .$$

This means that (8.1.3) becomes

$$\dot{\mu}_j(t) = -\frac{1}{h}\big[f(\mu_j) - f(\mu_{j-1})\big] = -\frac{2}{3h}\Big[\mu_j^{3/2} - \mu_{j-1}^{3/2}\Big] \ . \tag{8.1.4}$$

SOLUTION of (8-1.h):

The general formula for the fully discrete evolution operator for a 2-point numerical flux function in combination wit explicit Euler timestepping is given in [Lecture → Eq. (8.4.1.10)]. The particular 2-point numerical flux function has been derived in Sub-problem (8-1.g) as $F(v,w) = v$.

Alternative you may start from (8.1.4) and just apply the explicit Euler method.

C++11 code 8.1.6: Implementation of `solveBurgersGodunov()` for Sub-problem (8-1.h)
→ GITHUB

```
constexpr double PI = 3.14159265358979323846;

double Square(double x) { return x * x; }

double f(double x) { return 2.0 / 3.0 * std::sqrt(x * x * x); }

Eigen::VectorXd solveBurgersGodunov(double T, unsigned int N) {
  double h = 5.0 / N;            // meshwidth
  double tau = h;                // timestep = meshwidth by CFL condition
  int m = std::round(T / tau);   // no. of timesteps

  // initialize vector with initial nodal values
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N + 1, -1.0, 4.0);
  Eigen::VectorXd mu =
      x.unaryExpr([](double x) {
         return 0.0 <= x && x <= 1.0 ? Square(std::sin(PI * x)) : 0.0;
       }).eval();

  for (int i = 0; i < m; ++i) {
    for (int j = N; 0 < j; --j) {
      // Standard fully discrete evolution based on explicit Euler timestepping
      mu(j) = mu(j) - tau / h * (f(mu(j)) - f(mu(j - 1)));
    }
    // truncation to a finite vector. Only required on one side, because all
    // information flows from left to right.
    mu(0) = 0.0;  // Value of u0 to the left of x=0
  }

  return mu;
}
```

SOLUTION of (8-1.i):

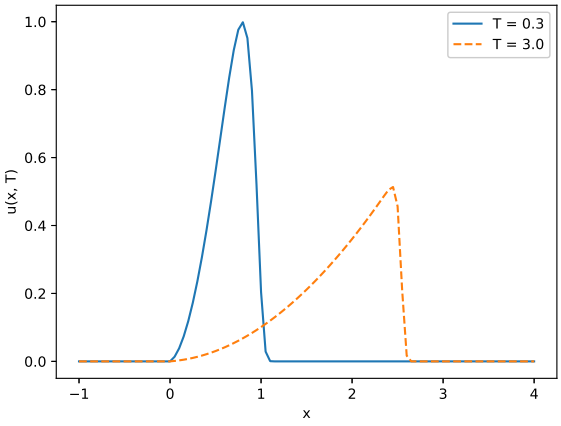
There are many ways to create a plot. One is to generate formatted output from the C++ code and then leave the actual creation of the plot to a PYTHON script ⇝ GITHUB.

C++11 code 8.1.7: Sub-problem (8-1.i), excerpt of `main()` ⇝ GITHUB

```
2   const unsigned int N = 100;
3   Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N + 1, -1.0, 4.0);
4   Eigen::VectorXd mu03 = BurgersEquation::solveBurgersGodunov(0.3, N);
5   Eigen::VectorXd mu30 = BurgersEquation::solveBurgersGodunov(3.0, N);
6
7   // Write the solutions to a file that can be used for plotting.
8   std::ofstream solution_file;
9   solution_file.open("solution.csv");
10  solution_file << x.transpose().format(BurgersEquation::CSVFormat)
11                << std::endl;
12  solution_file << mu03.transpose().format(BurgersEquation::CSVFormat)
13                << std::endl;
14  solution_file << mu30.transpose().format(BurgersEquation::CSVFormat)
15                << std::endl;
16  solution_file.close();
17  std::cout << "Generated " CURRENT_BINARY_DIR "/solution.csv" << std::endl;
18  std::system("python3 " CURRENT_SOURCE_DIR
19              "/plot_solution.py " CURRENT_BINARY_DIR
20              "/solution.csv " CURRENT_BINARY_DIR "/solution.eps");
```

PYTHON script 8.1.8: Sub-problem (8-1.i): Helper script for visualization ⇝ GITHUB

```
1  from matplotlib.pyplot import figure, legend, plot, savefig, xlabel, ylabel
2  from numpy import genfromtxt
3  from sys import argv
4
5  input_file = str(argv[1])
6  output_file = str(argv[2])
7
8  data = genfromtxt(input_file, delimiter=',')
9  x = data[0]
10 solution_short = data[1]
11 solution_long = data[2]
12
13 fig = figure()
14 plot(x, solution_short, '-', label='T = 0.3')
15 plot(x, solution_long, '--', label='T = 3.0')
16 xlabel('x')
17 ylabel('u(x, T)')
18 legend()
19 savefig(output_file)
20
21 print('Generated ' + output_file)
```



◁ Finite-volume solution of (8.1.2) at different times, computed on a mesh with $N = 100$ cells.

SOLUTION of (8-10.a):

We start from the integral form [Lecture → Eq. (8.2.3.3)] for the 1D scalar conservation law (8.10.1):

$$\int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t_{k+1})\,\mathrm{d}x - \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t_k)\,\mathrm{d}x = \int_{t_k}^{t_{k+1}} f(u(x_{j-\frac{1}{2}},t))\,\mathrm{d}t - \int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t\,. \tag{8.10.4}$$

Taking into account the definition of $\rho_j^{(k)}$ this means

$$h\big(\rho_j^{(k+1)} - \rho_j^{(k)}\big) = \int_{t_k}^{t_{k+1}} f(u(x_{j-\frac{1}{2}},t))\,\mathrm{d}t - \int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t\,,$$

and we get

$$\rho_j^{(k+1)} = \rho_j^{(k)} - \frac{\tau}{h}\left(\frac{1}{\tau}\int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t - \frac{1}{\tau}\int_{t_k}^{t_{k+1}} f(u(x_{j-\frac{1}{2}},t))\,\mathrm{d}t\right),$$

which means that (8.10.3) holds, if we set

$$\widetilde{f}_{\ell+\frac{1}{2}} := \frac{1}{\tau}\int_{t_k}^{t_{k+1}} f(u(x_{j+\frac{1}{2}},t))\,\mathrm{d}t\,, \quad \ell = j-1, j\,.$$

Hence, only the last option is correct.

SOLUTION of (8-10.b):

Using a 2-point numerical flux the spatially semi-discrete conservation form for (8.10.1) reads [Lecture → Eq. (8.3.2.8)]

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big) \;, \quad j \in \mathbb{Z} \;.$$

As discussed in [Lecture → Ex. 8.4.1.9], we apply explicit Euler timestepping to this "infinite-dimensional" ODE and get [Lecture → Eq. (8.4.1.10)]

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\big(F(\mu_j^{(k)},\mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)},\mu_j^{(k)})\big) \;,$$

which agrees with (8.10.5) when setting

$$f_{\ell+\frac{1}{2}} := F(\mu_\ell^{(k)},\mu_{\ell+1}^{(k)}) \;, \quad \ell \in \mathbb{Z} \;.$$

SOLUTION of (8-10.c):

For the scalar linear advection equation with advection velocity $\beta > 0$ the Godunov numerical flux boils down to the simple upwind numerical flux:

$$F_{\mathrm{GD}}(v,w) = \beta v\,, \quad v,w \in \mathbb{R}\,, \tag{8.10.8}$$

The Rankine-Hugoniot speed $\dot{s}$ agrees with β: $\dot{s} = \beta > 0$, which means that in (8.10.6d)

$$\theta = \frac{\mu_\ell^{(k)} - \mu_{\ell-1}^{(k)}}{\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}}\,.$$

Summing up, in the k-th timestep we use the flux terms

$$f_{\ell+\frac{1}{2}} = \beta\mu_\ell^{(k)} + \tfrac{1}{2}\beta(1-\gamma\beta)\varphi\Big(\frac{\mu_\ell^{(k)} - \mu_{\ell-1}^{(k)}}{\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}}\Big)\big(\mu_{\ell+1}^{(k)} - \mu_\ell^{(k)}\big)\,, \quad l \in \mathbb{Z}\,.$$

Plugging this into (8.10.5) yields

$$\begin{aligned}\mu_j^{(k+1)} = \mu_j^{(k)} &- \beta\gamma(\mu_j^{(k)} - \mu_{j-1}^{(k)}) - \\ &\tfrac{1}{2}\beta\gamma(1-\beta\gamma)\Bigg(\varphi\Big(\frac{\mu_j^{(k)} - \mu_{j-1}^{(k)}}{\mu_{j+1}^{(k)} - \mu_j^{(k)}}\Big)\big(\mu_{j+1}^{(k)} - \mu_j^{(k)}\big) - \\ &\qquad \varphi\Big(\frac{\mu_{j-1}^{(k)} - \mu_{j-2}^{(k)}}{\mu_j^{(k)} - \mu_{j-1}^{(k)}}\Big)\big(\mu_j^{(k)} - \mu_{j-1}^{(k)}\big)\Bigg)\,.\end{aligned} \tag{8.10.9}$$

We tacitly deal with "division by zero exceptions" is hinted above. Since $\varphi(\cdot)$ is always finite, for $\mu_{j+1}^{(k)} - \mu_j^{(k)}$ or $\mu_j^{(k)} - \mu_{j-1}^{(k)} = 0$ the ill-defined terms can just be skipped.

Remark. For $\varphi \equiv 1$ the recursion simplifies to

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \beta\gamma(\mu_j^{(k)} - \mu_{j-1}^{(k)}) - \tfrac{1}{2}\beta\gamma(1-\beta\gamma)\Big(\mu_{j+1}^{(k)} - 2\mu_j^{(k)} + \mu_{j-1}^{(k)}\Big)\,.$$

This is the well-known 2nd-order **Lax-Wendroff scheme** for linear advection.

HINT 1 for (8-10.d): To avoid division by zero when computing θ, use the provided function

```
inline double thetaquotient(double u, double v, double w) {
  const double denom = w-v;
  return (denom == 0.0)?1E17*(v-u):(v-u)/denom;
}
```

whose arguments stand for the three consecutive $\mu_*^{(k)}$-values entering the definition (8.10.6d) of θ. ⌟

SOLUTION of (8-10.d):

The implementation boils down to coding (8.10.9). We have to make special provision for the treatment of he terminal mesh cells:

- For $j = 0$ set $\mu_{-2}^{(k)} = \mu_{-1}^{(k)} = \mu_0^{(k)}$.
- For $j = 1$ set $\mu_{-1}^{(k)} = \mu_0^{(k)}$.
- For $j = N - 1$ set $\mu_N^{(k)} = \mu_{N-1}^{(k)}$.

C++ code 8.10.10: ➔ **GITHUB**

```
template <typename FLUXLIM = std::function<double(double)>>
Eigen::VectorXd fluxlimAdvection(
    double beta, const Eigen::VectorXd &mu0, double h, double tau,
    unsigned int nb_timesteps,
    FLUXLIM &&phi = [](double /*theta*/) { return 1.0; }) {
  if (beta < 0) {
    throw std::domain_error("fluxlimAdvection: negative beta!");
  }
  Eigen::VectorXd mu; // return vector

  // The length of the vector passing the initial conditions also
     determines teh
  // number of spatial (dual) cells
  int N = mu0.size();
  // Ratio of spatial and temporal timestep
  double gamma = tau / h;
  // Set initial conditions
  mu = mu0;
  // Constant continuation index mapping tool.  This is only one of many
     ways how
  // to implement the "endpoint conditions"
  auto idx_map = [N](int idx) {
    if (idx < 0) {
      return 0;
    } else if (idx > N - 1) {
      return N - 1;
    } else {
      return idx;
    }
  };

  Eigen::VectorXd mu_next(N);
  for (int k = 0; k < nb_timesteps; k++) {
    // Computing forward step
    for (int j = 0; j < N; j++) {
      // Implementation of (8.10.9).
      const double backward_diff = mu(idx_map(j)) - mu(idx_map(j - 1));
      const double shifted_backward_diff =
          mu(idx_map(j - 1)) - mu(idx_map(j - 2));
      const double forward_diff = mu(idx_map(j + 1)) - mu(idx_map(j));
      // Values of flux limiter
      double phi_1, phi_2;
      if (forward_diff == 0.0) {
```

```
        phi_1 = 0.0;
      } else {
        phi_1 = phi(backward_diff / forward_diff);
      }
      if (backward_diff == 0.0) {
        phi_2 = 0.0;
      } else {
        phi_2 = phi(shifted_backward_diff / backward_diff);
      }
      // Main recursion
      mu_next(j) = mu(idx_map(j)) - beta * gamma * backward_diff -
                   0.5 * beta * gamma * (1 - beta * gamma) *
                       (phi_1 * forward_diff - phi_2 * backward_diff);
    }
    mu = mu_next; // step forward (loop update)
  }
  return mu;
};
```

SOLUTION of (8-10.e):

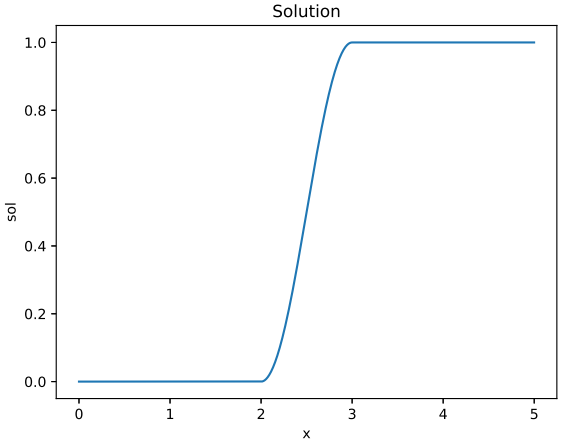
The quotients of successive error norms in the columns of the table evaluate to values that do not vary much between the rows. Thus, for both columns the data indicate **Algebraic convergence** of the error norms as $M \to \infty / \tau \to 0$.

Middle column: Algebraic convergence $O(\tau^{\alpha})$ with rate $\alpha < 1$
Right column: Algebraic convergence $O(\tau^2)$.

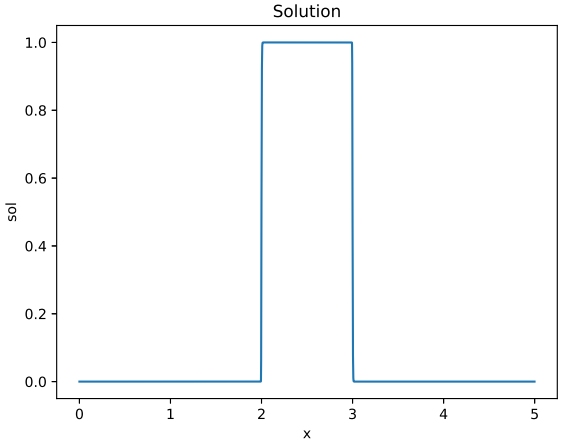
Convergence is much faster for the right column, which can only be possible for a smooth solution. Such a smooth solution can arise only from smooth u_0. Therefore we conclude

Middle column: u_0^B

Right column: u_0^A



FV-solution, $M = 1280$, $u_0 = u_0^A$



FV-solution, $M = 1280$, $u_0 = u_0^B$

SOLUTION of (8-10.f):

To evaluate $F_{\mathrm{GD}}(v,w)$ we have to distinguish several cases as in [Lecture → § 8.3.4.32] and examine the solutions of Riemann problems with left state v and right state w. The Burgers flux function is strictly convex.

❶ If $v > w$ the Riemann problem will have a shock solution and we get

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) = \frac{1}{2}v^2 & \text{, if } \frac{f(w)-f(v)}{w-v} = \frac{1}{2}(v+w) > 0\,, \\ f(w) = \frac{1}{2}w^2 & \text{, if } \frac{f(w)-f(v)}{w-v} = \frac{1}{2}(v+w) < 0\,. \end{cases} \tag{8.10.12}$$

❷ If $v < w$ the solution of the Riemann problem will be a rarefaction wave and we arrive at

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) = \frac{1}{2}v^2 & \text{, if } f'(v) = v > 0\,, \\ 0 & \text{, if } f'(v) = v < 0 < f'(w) = w\,, \\ f(w) = \frac{1}{2}w^2 & \text{, if } f'(w) = w < 0\,. \end{cases} \tag{8.10.13}$$

All these different cases are taken into account in the following lambda function where `f()` stands for the Burgers flux function $f(u) = \frac{1}{2}u^2$:

```
auto godnfnburgers = [f](double v, double w) -> double {
  if (v > w) {
    if (v + w > 0) return f(v); else  return f(w);
  } else {
    if (v > 0)  return f(v);
    else if (0 < w) return 0.0; else return f(w);
  }
};
```

SOLUTION of (8-10.g):

We implement (8.10.5) based on (8.10.6).

C++ code 8.10.14: ➙ **GITHUB**

```
template <typename FLUXLIM = std::function<double(double)>>
Eigen::VectorXd fluxlimBurgers(
    const Eigen::VectorXd &mu0, double h, double tau, unsigned int nb_timesteps,
    FLUXLIM &&phi = [](double /*theta*/) { return 1.0; }) {
  Eigen::VectorXd mu; // return vector
  int N = mu0.size(); // Number of sptial dual cells
  double gamma = tau / h;
  // Set initial conditions
  mu = mu0;
  // Flux function for Burgers equation
  auto f = [](double u) { return 0.5 * u * u; };
  // Godunov numerical flux
  auto godnfnburgers = [f](double v, double w) -> double {
    if (v > w) {
      if (v + w > 0) {
        return f(v);
      } else {
        return f(w);
      }
    } else {
      if (v > 0) {
        return f(v);
      } else if (0 < w) {
        return 0.0;
      } else {
        return f(w);
      }
    }
  };

  // Constant continuation index mapping tool
  auto idx_map = [N](int idx) {
    if (idx < 0) {
      return 0;
    } else if (idx > N - 1) {
      return N - 1;
    } else {
      return idx;
    }
  };
  // Rankine-Hugoniot speed
  auto s_dot = [f](double v, double w) {
    if (v == w) {
      return 0.0;
    } else {
      return (f(w) - f(v)) / (w - v);
    }
  };
  // The quantity θ from (8.10.6d)
  // Implementation avoid division by zero, see thetaquotient()
```

```
  auto theta = [s_dot](double x, double v, double w, double y) {
    if (s_dot(v, w) < 0) {
      if (y == w) {
        return 1.0e17 * (w - v);
      } else {
        return (w - v) / (y - w);
      }
    } else {
      if (v == w) {
        return 1.0e17 * (v - x);
      } else {
        return (v - x) / (w - v);
      };
    }
  };
  // See (8.10.6b)
  auto f_D = [&](int i) -> double {
    double diff = mu(idx_map(i + 1)) - mu(idx_map(i));
    double s_dot_abs = std::abs(s_dot(mu(idx_map(i)), mu(idx_map(i + 1))));
    return 0.5 * s_dot_abs * (1 - gamma * s_dot_abs) * diff *
           phi(theta(mu(idx_map(i - 1)), mu(idx_map(i)), mu(idx_map(i + 1)),
                     mu(idx_map(i + 2))));
  };
  // Auxility arrays:  must be declared outside the main timestepping
     loop
  Eigen::VectorXd mu_next(N);
  Eigen::VectorXd fluxdiff(N);
  for (int k = 0; k < nb_timesteps; k++) {
    // Computing modified flux difference vector, see (8.10.6a)
    for (int i = 0; i < N; i++) {
      fluxdiff(i) = godnfnburgers(mu(idx_map(i)), mu(idx_map(i + 1))) + f_D(i) -
                    godnfnburgers(mu(idx_map(i - 1)), mu(idx_map(i))) -
                    f_D(i - 1);
    }
    // Computing forward Euler step (8.10.5)
    for (int j = 0; j < N; j++) {
      mu_next(j) = mu(j) - gamma * fluxdiff(j);
    }
    mu = mu_next; // step forward (loop update)
  }
  return mu;
}
```

SOLUTION of (8-2.a):

Definition [Lecture → Def. 8.3.3.5]. Consistent numerical flux function

A numerical flux function $F : \mathbb{R}^{m_l+m_r} \mapsto \mathbb{R}$ is **consistent** with the flux function $f : \mathbb{R} \mapsto \mathbb{R}$, if

$$F(u,\ldots,u) = f(u) \quad \forall u \in \mathbb{R} .$$

We have

$$F_{\mathrm{EO}}(v,v) = \frac{1}{2}(f(v)+f(v)) - \frac{1}{2}\int_v^v |f'(\xi)|\,\mathrm{d}\xi = f(v) ,$$

so the flux is consistent.

HINT 1 for (8-2.b): Use the criterion of [Lecture → Cor. 8.3.5.6].

SOLUTION of (8-2.b):

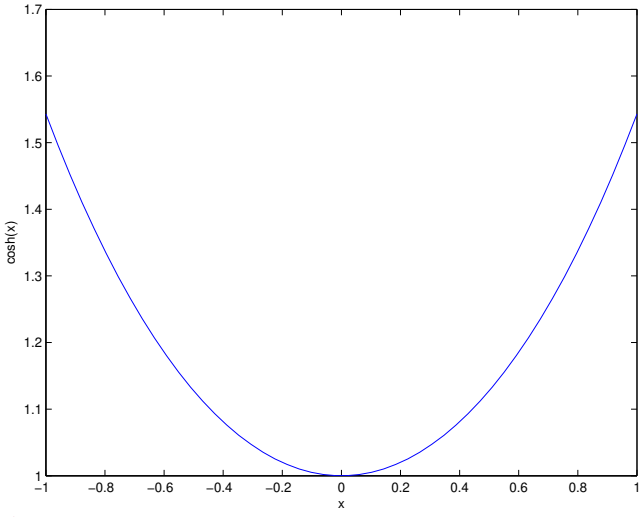
By the fundamental theorem of calculus we find

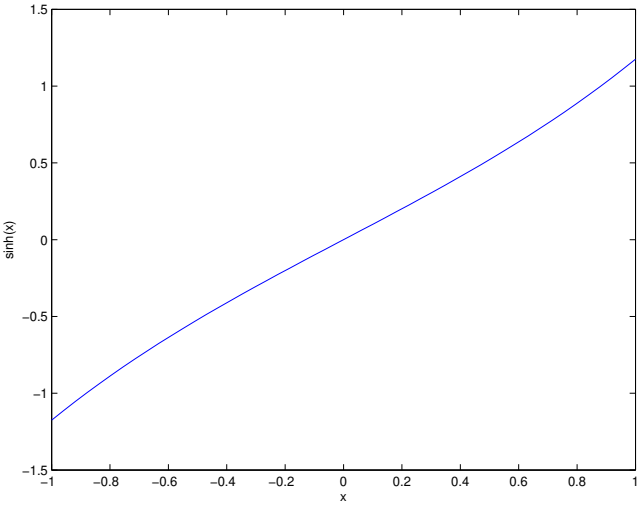
$$\frac{\partial F_{\mathrm{EO}}}{\partial v}(v,w) = \tfrac{1}{2}f'(v) + \tfrac{1}{2}|f'(v)| = \begin{cases} f'(v) & \text{, if } f'(v) \geq 0\,, \\ 0 & \text{, if } f'(v) < 0 \end{cases} \geq 0\,,$$

$$\frac{\partial F_{\mathrm{EO}}}{\partial w}(v,w) = \tfrac{1}{2}f'(w) - \tfrac{1}{2}|f'(w)| = \begin{cases} 0 & \text{, if } f'(w) \geq 0\,, \\ f'(w) & \text{, if } f'(w) < 0 \end{cases} \leq 0\,.$$

Hence, from [Lecture $\rightarrow$ Cor. 8.3.5.6] we conclude that F_{EO} is monotone.

HINT 1 for (8-2.d): The flux function $f(u) = \cosh(u)$ and its dervative $\sinh$ are plotted in Fig. 94 and Fig. 95. What is the “maximal speed of propagation”?





SOLUTION of (8-2.d):

The characteristic emanating from $]-1,0[$ can move to the left with maximal speed $\sinh(-1)$, and the characteristic emanating from $]1,0[$ can move to the right with maximal speed $\sinh(1)$. Thus, the support of $u(\cdot,t)$ will at least be contained in the interval

$$[0+t\sinh(-1), 1+t\sinh(1)] = [-t\sinh(1), 1+t\sinh(1)] \ .$$

The value of $\sinh(1)$ is about 1.1752.

SOLUTION of (8-2.e):

To simplify the integration, we can split the integral in two. One in the interval $[\min(v,0), \min(w,0)]$ and one in the interval $[\max(v,0), \max(w,0)]$. The first is guaranteed to be contained in $(-\infty, 0]$, where $|f'(x)| = -\sinh(x)$ and the second is contained in $[0,\infty)$, where $|f'(x)| = \sinh(x)$. The integration can then be performed analytically:

$$F_{\text{EO}}(v,w) = \tfrac{1}{2}(f(v)+f(w)) - \frac{1}{2}\cdot\begin{cases} f(w)-f(v) & \text{, if } v,w \geq 0\,, \\ f(v)+f(w)-2f(0) & \text{, if } v<0,\ w\geq 0\,, \\ 2f(0)-f(v)-f(w) & \text{, if } w<0,\ v\geq 0\,, \\ f(v)-f(w) & \text{, if } v,w<0\,. \end{cases}$$

$$= \begin{cases} f(v) & \text{, if } v,w \geq 0\,, \\ f(0) & \text{, if } v<0,\ w\geq 0\,, \\ f(v)+f(w)-f(0) & \text{, if } w<0,\ v\geq 0\,, \\ f(w) & \text{, if } v,w<0\,. \end{cases}$$

The following is a rather obfuscated implementation of this formula. Can you see, why it works?

C++11 code 8.2.3: Sub-problem (8-2.e): function `EngquistOsherNumFlux()` ⇝ GITHUB

```
double EngquistOsherNumFlux(double v, double w) {
  double result;
  auto Fv = [](double v) { return 0 <= v ? std::cosh(v) - 0.5 : 0.5; };
  result = Fv(v) + Fv(-w);
  return result;
}
```

SOLUTION of (8-2.g):

By the comparison principle from [Lecture → Thm. 8.2.7.1] we have

$$A \le u(x,t) \le B \quad \forall (x,t) \in \mathbb{R} \times \mathbb{R}_0^+ \ .$$

The maximal analytic domain of depdence follows from the formulas in [Lecture → § 8.4.2.4]:

$$D^-(\overline{x},\overline{t}) := \{(x,t) \in \mathbb{R} \times [0,\overline{t}] : \dot{s}_{\min}(\overline{t}-t) \le x - \overline{x} \le \dot{s}_{\max}(\overline{t}-t)\} \ . \tag{8.2.4}$$

with maximal speeds of propagation in either direction

$$\dot{s}_{\min} := \min\{f'(\xi) : \inf_{x\in\mathbb{R}} u_0(x) \le \xi \le \sup_{x\in\mathbb{R}} u_0(x)\} \ , \tag{8.2.5}$$

$$\dot{s}_{\max} := \max\{f'(\xi) : \inf_{x\in\mathbb{R}} u_0(x) \le \xi \le \sup_{x\in\mathbb{R}} u_0(x)\} \ . \tag{8.2.6}$$

Since $f'(u) = \sinh(u)$, which is increasing, we find:

$$\dot{s}_{\min} = \sinh(A) \quad \text{and} \quad \dot{s}_{\max} = \sinh(B) \ . \tag{8.2.7}$$

Hence, the maximal speed of propagation is

$$\dot{s} = \max\{|\sinh A|, |\sinh B|\} \ .$$

The Engquist-Osher numerical flux is a 2-point numerical flux and the explicit Euler method is a 1-stage explicit Runge-Kutta single-step method. Thus the resulting space-time stencil has width $m = 1$, see [Lecture → § 8.4.2.1] and [Lecture → Fig. 435]. According to [Lecture → Def. 8.4.2.5] the numerical domain of dependence is therefore

$$D_h^-(x_j,t_k) := \{(x_n,t_l) \in \mathbb{R} \times [0,t_k] : j-(k-l) \le n \le j+m(k-l)\} \ , \quad x_j = hj \ , \quad t_k = \tau k \ . \tag{8.2.8}$$

Thus we have by [Lecture → Eq. (8.4.2.12)]

$$\tau \le \frac{h}{\dot{s}} \quad \Longrightarrow \quad D^-(x_j,t_k) \subset \operatorname{convex}(D_h^-(x_j,t_k)) \ .$$

This means that the CFL-condition implies the timestep constraint

$$\tau \le \frac{h}{\max\{|\sinh A|, |\sinh B|\}} \ .$$

SOLUTION of (8-2.h):

We have to implement the formula [Lecture → Eq. (8.4.1.10)]

$$\mu^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\left(F_{\mathrm{EO}}(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F_{\mathrm{EO}}(\mu_{j-1}^{(k)}, \mu_j^{(k)})\right) ,$$

plus truncation of the computational domain assuming constant entries of the vector of cell values for indices outside of a finite range, see [Lecture → Code 8.3.2.9].

C++11 code 8.2.9: Sub-problem (8-2.h): function `solveCP()` ↬ GITHUB

```
2  Eigen::VectorXd solveCP(double a, double b, Eigen::VectorXd u0, double T) {
3    // Find the maximal speed of propagation
4    double A = u0.minCoeff();
5    double B = u0.maxCoeff();
6    double K = std::max(std::abs(std::sinh(A)), std::abs(std::sinh(B)));
7    // Set uniform timestep according to CFL condition
8    int N = u0.size();
9    double h = (b - a) / N;
10   double tau_max = h / K;
11   double timesteps = std::ceil(T / tau_max);
12   double tau = T / timesteps;
13
14   // Main timestepping loop
15   Eigen::VectorXd mu(N);
16   for (int i = 0; i < timesteps; ++i) {
17     mu.swap(u0);
18     double F_minus;
19     double F_plus = EngquistOsherNumFlux(mu(0), mu(0));
20     for (int j = 0; j < N - 1; ++j) {
21       F_minus = F_plus;
22       F_plus = EngquistOsherNumFlux(mu(j), mu(j + 1));
23       u0(j) = mu(j) - tau / h * (F_plus - F_minus);
24     }
25     F_minus = F_plus;
26     F_plus = EngquistOsherNumFlux(mu(N - 1), mu(N - 1));
27     u0(N - 1) = mu(N - 1) - tau / h * (F_plus - F_minus);
28   }
29
30   return u0;
31 }
```

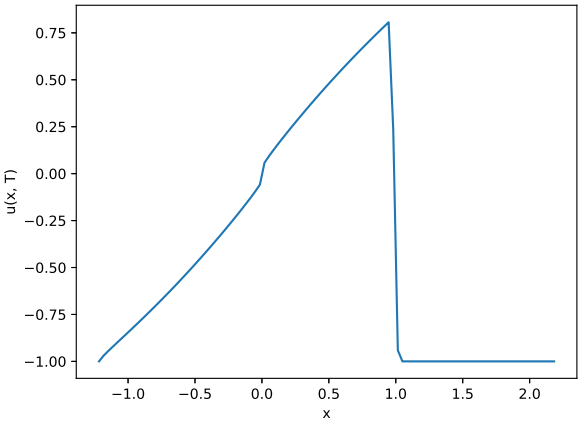
SOLUTION of (8-2.i):

C++11 code 8.2.10: Sub-problem (8-2.h): function `solveCP()` ⇝ GITHUB

```
const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
                                       Eigen::DontAlignCols, ", ", "\n");

double u0(double x) { return (0.0 < x && x <= 1.0) ? 1.0 : -1.0; }

int main() {
  unsigned int N = 100;
  double a = -1.2;
  double b = 2.2;
  double T = 1.0;
  double h = (b - a) / N;

  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, a - 0.5 * h, b - 0.5 * h);
  Eigen::VectorXd uinitial = x.unaryExpr(std::ref(u0));
  Eigen::VectorXd ufinal =
      EngquistOsherNumericalFlux::solveCP(a, b, uinitial, T);

  std::ofstream file;
  file.open("ufinal.csv");
  file << x.transpose().format(CSVFormat) << std::endl;
  file << ufinal.transpose().format(CSVFormat) << std::endl;
  file.close();

  std::cout << "Generated " CURRENT_BINARY_DIR "/ufinal.csv" << std::endl;
  std::system("python3 " CURRENT_SOURCE_DIR
              "/plot_solution.py " CURRENT_BINARY_DIR
              "/ufinal.csv " CURRENT_BINARY_DIR "/ufinal.eps");

  return 0;
}
```

PYTHON script 8.2.11: Sub-problem (8-2.h): PYTHON script for plotting solution ⇝ GITHUB

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel
from numpy import genfromtxt
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
x = data[0]
y = data[1]

figure()
plot(x, y)
xlabel('x')
ylabel('u(x,1)')
savefig(output_file)

print('Generated ' + output_file)
```



◁ Solution of Riemann problems at $T = 1$.

Note the remnants of a spurious expansion shock at state $u = 0$ for which $f'(u) = 0$ (“entry glitch”).
See also [Lecture → Ex. 8.3.4.36] for a similar observation.

SOLUTION of (8-3.a):

Our considerations are based on [Lecture → Section 8.2.6], where we studied the traffic flow equations as a specimen of a scalar conservation law with concave flux.

We see that $f(u) = \sin \pi u$, which is concave on $[0,1]$. Thus, the condition (i) should lead to a shock, and condition (ii) should lead to a rarefaction wave. The situation is the same as with the traffic flow conservation law discussed in [Lecture → Section 8.2.5].

For (i), we can apply [Lecture → Lemma 8.2.5.4] and use the Rankine-Hugoniot jump condition to determine the shock speed.

$$\dot{s} = \frac{\sin 0 - \sin \pi}{0 - 1} = 0,$$

and so we find that the shock doesn't move (stationary shock).

$$u(x,t) = \begin{cases} 0, & x < 0, \\ 1, & x > 0. \end{cases}$$

For the rarefaction wave in the case of (ii), we invoke [Lecture → Lemma 8.2.5.10] with $g(x) = (f')^{-1}(x) = \frac{1}{\pi} \arccos(x/\pi)$, hence we obtain the similarity solution

$$u(x,t) = \begin{cases} 1 & \text{for} \quad x < -\pi t\,, \\ \frac{1}{\pi} \arccos\left(\frac{x}{\pi t}\right) & \text{for} \quad -\pi < x/t < \pi\,, \\ 0 & \text{for} \quad x > \pi t\,. \end{cases}$$

HINT 1 for (8-3.j): Recall the finite volume interpretation of the coefficients of the vector argument `mu`.
⌟

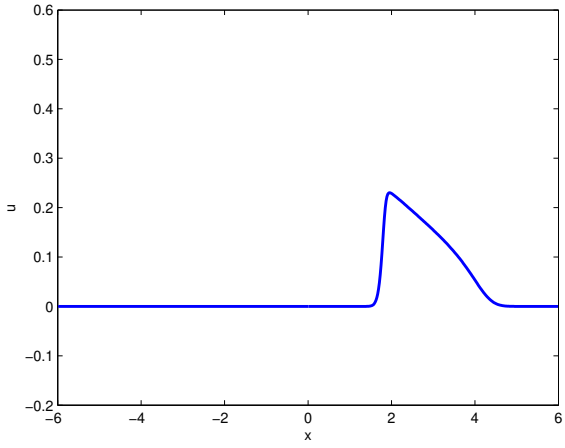
SOLUTION of (8-3.j):

The modification of the right-hand-side of the method-of-lines ODE just amounts to subtracting c times the vector $\vec{\mu}$ (suitably truncated).

C++ code 8.3.13: Sub-problem (8-3.i): function `sineClawReactionRhs()` ➙ GITHUB

```
2 Eigen::VectorXd sineClawReactionRhs(const Eigen::VectorXd &mu, double c) {
3   Eigen::VectorXd rhs(mu.size());
4   rhs = sineClawRhs(mu) - c * mu;
5   return rhs;
6 }
```

SOLUTION of (8-3.k):



◁ Solution of the Cauchy problem for (8.3.12) at $T = 1$

The extra reaction term causes substantial damping of the solution.
To understand why consider the Cauchy problem for the pure reaction equation $\frac{\partial u}{\partial t} = -cu$, whose solution is $u(x,t) = \exp(-ct)u_0(x)$.

SOLUTION of (8-3.b):

By [Lecture → Thm. 8.2.7.1] we know that $0 \leq u(x,t) \leq 1$ for all (x,t). Furthermore, for the flux function $f(u) = \sin(\pi u)$,

$$-\pi \leq f'(u) \leq \pi\,, \quad 0 \leq u \leq 1\,.$$

Hence the maximal speed of propagation is $-\pi$ to the left, and π to the right. Hence, by [Lecture → Cor. 8.2.7.5], the maximal support is

$$\{(x,t) : -\pi t \leq x \leq 1 + \pi t,\ t \geq 0\}\,.$$

HINT 1 for (8-3.c): Note that the flux function $f(u) = \sin(\pi u)$ is concave on $[0, 1]$. ⌟

SOLUTION of (8-3.c):

Note that the discussion in the beginning of [Lecture → Section 8.3.4.4] considers only convex flux functions. What you can use is the general formula [Lecture → Eq. (8.3.4.33)]. Alternatively, you may just flip the conditions $v < w$ and $v > w$ in the formulas for convex f.

Under the assumption that $0 \leq v, w \leq 1$, we end up with the following formula for the intermediate state:

$$u^{\downarrow}(v,w) = \begin{cases} v & \text{, if } v < w \text{ and } \dot{s} > 0, \\ v & \text{, if } v \geq w \text{ and } \cos \pi v > 0, \\ w & \text{, if } v < w \text{ and } \dot{s} < 0, \\ w & \text{, if } v \geq w \text{ and } \cos \pi w < 0, \\ \frac{1}{2} & \text{, if } v \geq w \text{ and } \cos \pi v \leq 0 \leq \cos \pi w \,, \end{cases} \qquad \dot{s} := \frac{\sin(\pi w) = \sin(\pi v)}{w - v} \,.$$

No simplifications are possible, as all conditions can occur.

C++11 code 8.3.3: Sub-problem (8-3.c): function `sineGodFlux()` ⇢ GITHUB

```
2  constexpr double PI = 3.14159265358979323846;
3
4  double f(double x) { return std::sin(PI * x); }
5
6  double sineGodFlux(double v, double w) {
7    double result;
8    // Rankine-Hugoniot speed.  No safeguards against cancellation are
       taken.  For v
9    // close to w the result will severely be affected by amplified
       round-off
10   // error.  However, this does not do any harm, because we are
       interested in the
11   // the sign alone.
12   double s = (v != w) ? (f(w) - f(v)) / (w - v) : PI * std::cos(PI * v);
13   // Treat all different cases separately
14   if (((v < w) && (s > 0.0)) || ((v >= w) && (std::cos(PI * v) > 0.0))) {
15     result = f(v);
16   } else if (((v < w) && (s < 0.0)) || ((v >= w) && (std::cos(PI * w) < 0.0))) {
17     result = f(w);
18   } else
19     result = f(0.5);
20
21   return result;
22 }
```

SOLUTION of (8-3.d):

C++11 code 8.3.4: Sub-problem (8-3.d): function `sineClawRhs()` ⇢ GITHUB

```
2  Eigen::VectorXd sineClawRhs(const Eigen::VectorXd &mu) {
3    int N = mu.size();
4    double h = 12.0 / N;
5    Eigen::VectorXd result(N);
6
7    double F_minus;
8    double F_plus = sineGodFlux(0.0, mu(0));
9    for (int j = 0; j < N - 1; ++j) {
10     F_minus = F_plus;
11     F_plus = sineGodFlux(mu(j), mu(j + 1));
12     result(j) = -1.0 / h * (F_plus - F_minus);
13   }
14   F_minus = F_plus;
15   F_plus = sineGodFlux(mu(N - 1), 0.0);
16   result(N - 1) = -1.0 / h * (F_plus - F_minus);
17
18   return result;
19 }
```

SOLUTION of (8-3.e):

The fully discrete evolution operator based of a 2-point flux and the explicity trapezoidal rule is given in [Lecture → Eq. (8.4.1.11)]:

$$\vec{\kappa} = \boldsymbol{\mu}^{(k)} + \tau \boldsymbol{\mathcal{L}}_h(\vec{\boldsymbol{\mu}}^{(k)}) \quad , \quad \vec{\boldsymbol{\mu}}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \frac{\tau}{2}\Big(\boldsymbol{\mathcal{L}}_h(\boldsymbol{\mu}^{(k)}) + \boldsymbol{\mathcal{L}}_h(\vec{\kappa})\Big) .$$

$$\blacktriangleright \quad \begin{cases} \kappa_j := (\vec{\kappa})_j = \mu_j - \dfrac{\tau}{h}\big(F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big) , \\ (\boldsymbol{\mathcal{H}}_h(\vec{\boldsymbol{\mu}}))_j = \mu_j - \dfrac{\tau}{2h}\big(F(\kappa_j,\kappa_{j+1}) - F(\kappa_{j-1},\kappa_j) + F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big) . \end{cases} \tag{8.3.6}$$

Given this formula the implementation should be straightforward. Take care to take into account the truncation of the state vector, see [Lecture → Code 8.3.2.9].

C++11 code 8.3.7: Sub-problem (8-3.e): function `explTrpzTimestepping()` ⇝ GITHUB

```
2  template <typename RHSFUNCTOR>
3  Eigen::VectorXd explTrpzTimestepping(RHSFUNCTOR &&g, const Eigen::VectorXd &y0,
4                                       double T, unsigned int M) {
5    double tau = T / M;
6    Eigen::VectorXd y = y0;
7
8    for (int j = 0; j < M; ++j) {
9      Eigen::VectorXd k0 = g(y);
10     Eigen::VectorXd k1 = g(y + tau * k0);
11     y = y + tau * 0.5 * (k0 + k1);
12   }
13
14   return y;
15 }
```

SOLUTION of (8-3.f):

C++11 code 8.3.8: Sub-problem (8-3.f): function `solveSineConsLaw()` ↪ GITHUB

```
template <typename RHSFUNCTOR>
Eigen::VectorXd solveSineConsLaw(RHSFUNCTOR &&g, unsigned int N,
                                 unsigned int M) {
  const double h = 12.0 / N;
  const double T = 1.0;
  Eigen::VectorXd result(N);

  Eigen::VectorXd x =
      Eigen::VectorXd::LinSpaced(N, -6.0 + 0.5 * h, 6.0 - 0.5 * h);
  Eigen::VectorXd u0 =
      x.unaryExpr([](double x) { return 0.0 <= x && x < 1.0 ? 1.0 : 0.0; });
  result = explTrpzTimestepping(std::forward<RHSFUNCTOR>(g), u0, T, M);

  return result;
}
```

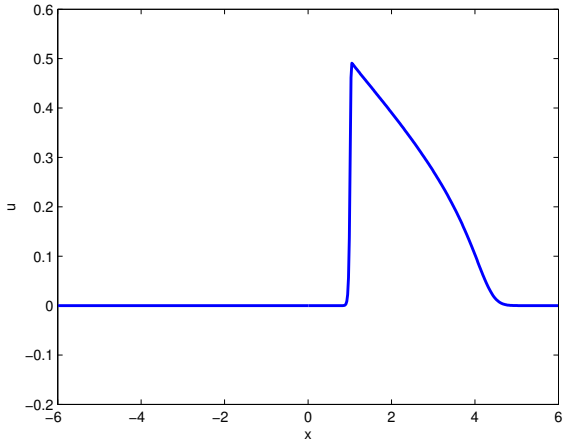
SOLUTION of (8-3.g):

C++11 code 8.3.9: Sub-problem (8-3.g): excerpt from `main()` with the code requested in the sub-problem ⇝ GITHUB

```
  unsigned int N = 600;
  unsigned int M = 200;
  double h = 12.0 / N;
  Eigen::VectorXd x =
      Eigen::VectorXd::LinSpaced(N, -6.0 + 0.5 * h, 6.0 - 0.5 * h);

  std::ofstream file;

  // without reaction term
  Eigen::VectorXd ufinal = solveSineConsLaw(&sineClawRhs, N, M);
  file.open("ufinal.csv");
  file << x.transpose().format(CSVFormat) << std::endl;
  file << ufinal.transpose().format(CSVFormat) << std::endl;
  file.close();

  std::cout << "Generated " CURRENT_BINARY_DIR "/ufinal.csv" << std::endl;
  std::system("python3 " CURRENT_SOURCE_DIR "/plot.py " CURRENT_BINARY_DIR
              "/ufinal.csv " CURRENT_BINARY_DIR "/ufinal.eps");
```

PYTHON script 8.3.10: PYTHON script for visualization ⇝ GITHUB

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel
from numpy import genfromtxt
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
x = data[0]
ufinal = data[1]

figure()
plot(x, ufinal, '-')
xlabel('x')
ylabel('u(x, T)')
savefig(output_file)

print('Generated ' + output_file)
```



The solution at final time $T = 1$.

SOLUTION of (8-3.h):

Numerically, the solution $\mu_j(t_k)$ depends on $\mu_{j-2}(t_{k-1})$ through $\mu_{j+2}(t_{k-1})$ (because of the double layered timestepping method). In the parlance of [Lecture $\rightarrow$ § 8.4.2.1] we deal with a stencil of with $m = 2$ due to the "stencil spread" induced by the 2-stage explicit Runge-Kutta timestepping employed. The numerical domain of dependence is thus a wedge with slope $2h/\tau$.

Analytically, the maximal speed of a characteristic is π.

We then need

$$\frac{2h}{\tau} \geq \pi \Longrightarrow \tau \leq \frac{2h}{\pi}.$$

HINT 1 for (8-3.i): You have to implement a bisection type approach based on monitoring a possible blow-up of solutions. To detect a blow-up, watch the maximal (in modulus) nodal value of the numerical solution at final time. Classify the run as affected by blow-up, in case the maximum is ≥ 2, ⌟

SOLUTION of (8-3.i):

One idea is to find the smallest timestep that keeps the values of the final solution bounded by within $[0,2]$ (try out a few experiments with many timesteps to see why). The algorithm works thus: we start with two values M_L and M_R which bound the critical value of M (thus, M_R is too many timesteps and M_L too few). We then investigate the midpoint $M_{\text{mid}} = (M_L + M_R)/2$. If this value of M causes no (or negligible) blow-up, we know it is safe, and otherwise it is not. In the first case, set $M_R \leftarrow M_{\text{mid}}$, and in the second, set $M_L \leftarrow M_{\text{mid}}$. Iterate until $M_L = M_R$.

C++ code 8.3.11: Sub-problem (8-3.i): function `findTimestep()` ⇝ GITHUB

```
2  bool blowup(const Eigen::VectorXd &mu) {
3    return mu.minCoeff() < 0.0 || mu.maxCoeff() > 2.0;
4  }
5
6  unsigned int findTimesteps() {
7    const unsigned int N = 600;
8    unsigned int ML = 100;
9    unsigned int MR = 200;
10
11   for (int i = 0; i < 100; ++i) {
12     if (ML == MR) {
13       return MR;
14     }
15     unsigned int M = (unsigned int)(0.5 * (ML + MR));
16     if (blowup(solveSineConsLaw(&sineClawRhs, N, M))) {
17       ML = M + 1;
18     } else {
19       MR = M;
20     }
21   }
22
23   return MR;
24 }
```

SOLUTION of (8-4.a):

The flow of information induced by (8.4.1) is described by the stencil of [Lecture → Fig. 435]. We face the situation of [Lecture → Fig. 439] and a stencil of width $m = 1$.

The maximal speed of propagation is the maximal modulus of $f'(u)$ taken over all possible states:

$$\dot{s} := \max\{f'(u) : \min_x u_0(x) \leq u \leq \max_{x\in\mathbb{R}} u_0(x)\} .$$

Thus, [Lecture → Eq. (8.4.2.12)] tell us that we have to demand the CFL-condition

$$\frac{\tau}{h} \leq \frac{1}{\dot{s}} .$$

SOLUTION of (8-4.j):

We implement the standard formula [Lecture → Eq. (8.4.1.10)]

$$\mu_j^{(k+1)} = \mu_j^{(k)} - \frac{\tau}{h}\left(F(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)}, \mu_j^{(k)})\right) , \tag{8.4.16}$$

for the numerical flux function F found in Sub-problem (8-4.i).

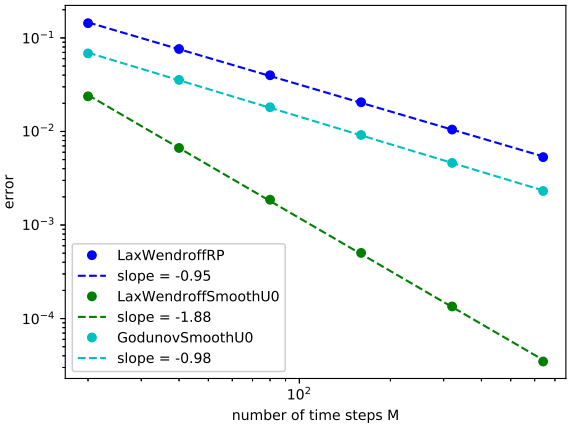
C++ code 8.4.17: Sub-problem (8-4.j): function `solveGodunov()` ↬ GITHUB

```
2  Eigen::VectorXd solveGodunov(const Eigen::VectorXd &u0, double T,
3                               unsigned int M) {
4    double tau = T / M;
5    double h = Constant::e * tau;
6    unsigned int N = u0.size();
7    Eigen::VectorXd mu = u0;
8
9    for (int i = 0; i < M; ++i) {
10     for (int j = N - 1; 0 < j; --j) {
11       mu(j) = mu(j) - tau / h * (std::exp(mu(j)) - std::exp(mu(j - 1)));
12     }
13   }
14   return mu;
15 }
```

SOLUTION of (8-4.k):

C++ code 8.4.18: Sub-problem (8-4.k): function `numexpGodunovSmoothU0()` ↪ GITHUB

```
Eigen::VectorXd numexpGodunovSmoothU0(const Eigen::VectorXi &M) {
  const double T = 1.0;
  const int M_size = M.size();
  Eigen::VectorXd error(M_size);

  for (int i = 0; i < M_size; ++i) {
    Eigen::VectorXd x = getXValues(T, M(i));
    Eigen::VectorXd u0 = x.unaryExpr(&smoothU0);
    Eigen::VectorXd uT = solveGodunov(u0, T, M(i));

    double tau = T / M(i);
    double h = Constant::e * tau;
    error(i) = h * (referenceSolution(x) - uT).lpNorm<1>();
  }
  return error;
}
```



◁ The double logarithmic error plot indicates algebraic convergence with rate 1 for the lowest-order finite volume method based on the monotone Godunov flux.

This matches the statement made in [Lecture → Section 8.4.4] that conservative finite volume methods based on monotone fluxes can achieve only first order convergence (“order barrier”).

Conversely, the Lax-Wendroff scheme achieves *second-order algebraic convergence*.

SOLUTION of (8-4.b):

For this case we have $f(u) = vu$, giving $f'(u) = v$ and $f'(u)^2 = v^2$:

$$\mu_j^{(k)} = \mu_j^{(k-1)} + \frac{\gamma v}{2}\left(\mu_{j+1}^{(k-1)} - \mu_{j-1}^{(k-1)}\right) + \frac{\gamma^2 v^2}{2}\left(\mu_{j+1}^{(k-1)} - 2\mu_j^{(k-1)} + \mu_{j-1}^{(k-1)}\right) . \tag{8.4.3}$$

Writing the recursion this way reveals that the Lax-Wendroff scheme pursues an idea similar to that motivating the Lax-Friedrichs/Rusanov flux [Lecture → § 8.3.4.7]; one adds artificial diffusionfor the sake of stabilization. Here the term $\mu_{j+1}^{(k-1)} - 2\mu_j^{(k-1)} + \mu_{j-1}^{(k-1)}$ is an h-weighted discrete diffusion operator.

SOLUTION of (8-4.c):

We can rewrite (8.4.3) as

$$\vec{\mu}^{(k)} = \boldsymbol{\mathcal{H}}(\vec{\mu}^{(k-1)}) \,,$$

$$\text{with} \qquad (\boldsymbol{\mathcal{H}}(\vec{\mu}))_j := \mu_j + \frac{\gamma v}{2}(\mu_{j+1} - \mu_{j-1}) + \frac{\gamma^2 v^2}{2}(\mu_{j+1} - 2\mu_j + \mu_{j-1}) \,.$$

Then

$$\begin{aligned} \boldsymbol{\mathcal{H}}(\vec{\boldsymbol{\psi}}_\xi) &= \left\{1 - \frac{\gamma v}{2}\left(e^{\imath\xi} - e^{-\imath\xi}\right) + \frac{\gamma^2 v^2}{2}\left(e^{\imath\xi} - 2 + e^{-\imath\xi}\right)\right\} \cdot \vec{\boldsymbol{\psi}}_\xi \\ &= \underbrace{\left\{1 - \gamma v \imath \sin(\xi) + (\gamma v)^2(\cos(\xi) - 1)\right\}}_{\hat{=}\text{ “eigenvalue”}} \cdot \vec{\boldsymbol{\psi}}_\xi \,. \end{aligned}$$

Thus, the necessary and sufficient condition for unconditional stability is

$$\begin{aligned} &\left|1 - \gamma v \imath \sin(\xi) + (\gamma v)^2(\cos(\xi) - 1)\right| \le 1 \quad \forall -\pi < \xi \le \pi \,, \\ \Leftrightarrow &\left(1 + (\gamma v)^2(\cos(\xi) - 1)\right)^2 + (\gamma v)^2 \sin(\xi)^2 \le 1 \quad \forall -\pi < \xi \le \pi \,, \\ \Leftrightarrow &\left(1 + (\gamma v)^2(t - 1)\right)^2 + (\gamma v)^2(1 - t^2) \le 1 \quad \forall -1 \le t \le 1 \,, \end{aligned} \tag{8.4.5}$$

We have to understand the behavior of the function

$$f_\alpha(t) := \left(1 + (\gamma v)^2(t - 1)\right)^2 + (\gamma v)^2(1 - t^2) \,, \quad -1 \le t \le 1 \,.$$

and have to find the values for $\alpha := \gamma v$ for which it stays below 1 on $[-1, 1]$. First we compute

$$f_\alpha(1) = 1 \quad \text{and} \quad f_\alpha(-1) = (1 - 2\alpha^2)^2 \,,$$

and find that there is no extremum in $]-1, 1[$. Thus we have to determine

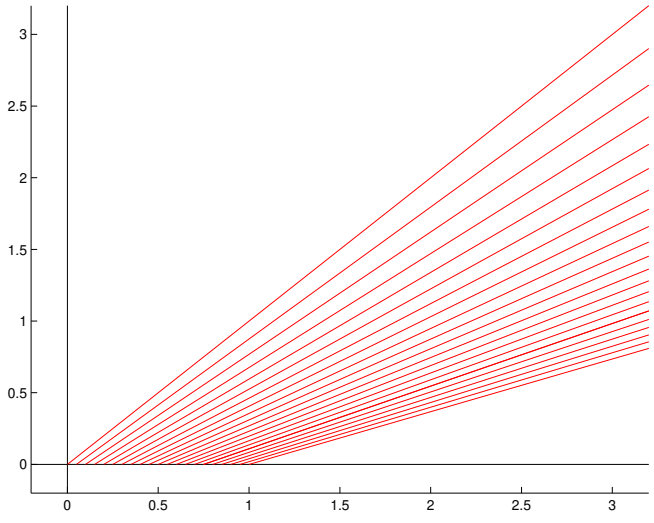
$$\alpha > 0\text{:} \quad (1 - 2\alpha^2)^2 \le 1 \quad \Leftrightarrow \quad \alpha^2 \le 1 \,.$$

Hence we, we have finally found the necessary and sufficient CFL-like condition for stability of the Lax-Wendroff full discrete discrete evolution (8.4.3)

$$\frac{\tau}{h} v \le 1 \quad \Leftrightarrow \quad \tau \le \frac{h}{v} \,.$$

This agrees with the geometric DFL-condition found in Sub-problem (8-4.a), because for $f(u) = vu$ the maximal speed of propagation is $s = v$, of course.

SOLUTION of (8-4.d):



There are characteristics emanating from the area $x \leq 0$ with slope $\gamma'(t) = e^A$ (where $u(x,t) = A$, and from the area $x \geq 1$ with slope $\gamma'(t) = e^B$ (where $u(x,t) = B$). On the wedge inbetween, we have $A \leq u(x,t) \leq B$, and indeed, this is the maximal set where the inequalities could be strict:

$$\{(x,t) \in \mathbb{R} \times [0,1] : \ x/t > e^A, \ (x-1)/t < e^B\} \ .$$

SOLUTION of (8-4.e):

The implementation boils down to coding (8.4.1) for $f(u) = f'(u) = e^u$. Note that the actual length of the spatial interval is needed only implicitly.

C++ code 8.4.7: Sub-problem (8-4.e): function `solveLaxWendroff()` ⇝ GITHUB

```
Eigen::VectorXd LaxWendroffRhs(const Eigen::VectorXd &mu, double gamma) {
  int N = mu.size();
  Eigen::VectorXd result(N);

  auto f = [](double x) { return std::exp(x); };
  auto df2 = [](double x) { return Square(std::exp(x)); };

  // Lax-Wendroff fully discrete evolution (8.4.1) for
  // f(u) = e^u.  Store the values of f(x) and (f'(x))^2 from the
  // previous iteration:

  // We extend mu to the left by mu(0):
  double mu_left = mu(0);

  // f'(1/2(mu_j + mu_{j-1}))^2:
  double df2_old = df2(0.5 * (mu(0) + mu_left));

  // f'(1/2(mu_{j+1} + mu_j))^2:
  double df2_new = df2(0.5 * (mu(1) + mu(0)));

  double f_old = f(mu_left);  // f(mu_{j-1})
  double f_mid = f(mu(0));    // f(mu_j)
  double f_new = f(mu(1));    // f(mu_{j+1})

  result(0) = mu(0) - 0.5 * gamma * (f_new - f_old) +
              0.5 * Square(gamma) *
                  (df2_new * (mu(1) - mu(0)) - df2_old * (mu(0) - mu_left));

  for (int j = 1; j < N - 1; ++j) {
    df2_old = df2_new;
    df2_new = df2(0.5 * (mu(j + 1) + mu(j)));
    f_old = f_mid;
    f_mid = f_new;
    f_new = f(mu(j + 1));
    result(j) =
        mu(j) - 0.5 * gamma * (f_new - f_old) +
        0.5 * Square(gamma) *
            (df2_new * (mu(j + 1) - mu(j)) - df2_old * (mu(j) - mu(j - 1)));
  }

  // We extend mu to the right by mu(N-1):
  double mu_right = mu(N - 1);

  df2_old = df2_new;
  df2_new = df2(0.5 * (mu_right + mu(N - 1)));

  f_old = f_mid;
  f_new = f(mu_right);
```

```
50
51    result(N - 1) = mu(N - 1) - 0.5 * gamma * (f_new - f_old) +
52                    0.5 * Square(gamma) *
53                        (df2_new * (mu_right - mu(N - 1)) -
54                         df2_old * (mu(N - 1) - mu(N - 2)));
55
56    return result;
57  }
58
59  Eigen::VectorXd solveLaxWendroff(const Eigen::VectorXd &u0, double T,
60                                   unsigned int M) {
61    double gamma = 1.0 / Constant::e;
62    Eigen::VectorXd mu = u0;
63    // Main timestepping loop
64    for (int j = 0; j < M; ++j) mu = LaxWendroffRhs(mu, gamma);
65    return mu;
66  }
```

HINT 1 for (8-4.f): We deal with a 1D scalar conservation law with a *convex* flux function $f(u) = e^u$ that is, with increasing $f'(u) = e^u$. Thus, initial data with $u_l < u_r$ will lead to a rarefaction entropy solution according to [Lecture $\rightarrow$ Lemma 8.2.5.10]. ⌟

SOLUTION of (8-4.f):

We rely on the formulas given in [Lecture → Lemma 8.2.5.10] with $u_l = 0$, $u_r = 1$, $f'(u_l) = 1$, $f'(u_r) = e$ $f'(u) = e^u$, $g(\xi) = \log(\xi)$, $\xi > 0$. This gives the concrete rarefaction solution of the Riemann problem:

$$u(x,t) = \begin{cases} 0 & \text{, if } x \leq t \text{ ,} \\ \log(x/t) & \text{, if } t < x < te \text{ ,} \\ 1 & \text{, if } x \geq et \text{ ,} \end{cases} \qquad x \in \mathbb{R} \text{ ,} \quad t \geq 0 \text{ .} \tag{8.4.9}$$

SOLUTION of (8-4.g):

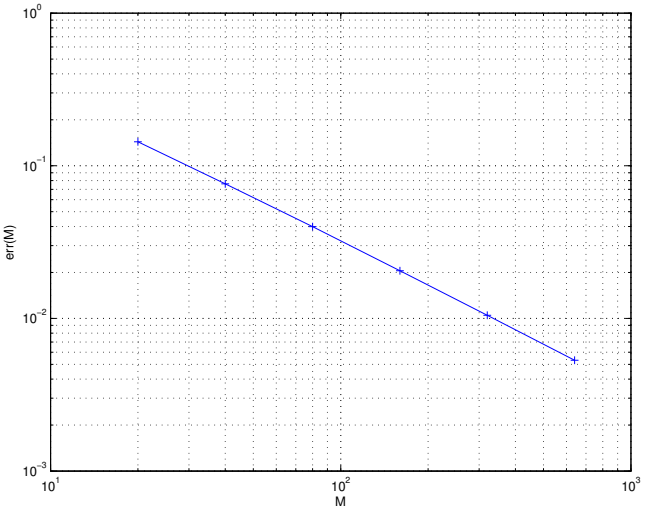
C++ code 8.4.11: Sub-problem (8-4.g): function `numexpLaxWendroffRP()` ⇝ GITHUB

```cpp
// Build spatial grid
Eigen::VectorXd getXValues(double T, unsigned int M) {
  double tau = T / M;
  double h = Constant::e * tau;
  int j_max = (int)(std::ceil((3.0 * T + 1.0) / h) + 0.5);
  int j_min = (int)(std::floor(-3.0 * T / h) - 0.5);
  unsigned int N = j_max - j_min + 1;
  return Eigen::VectorXd::LinSpaced(N, j_min * h, j_max * h);
}
Eigen::VectorXd numexpLaxWendroffRP(const Eigen::VectorXi &M) {
  const double T = 1.0;
  const int M_size = M.size();
  Eigen::VectorXd error(M_size);
  // Initial values for the Riemann problem
  auto u_initial = [](double x) { return 0.0 <= x ? 1.0 : 0.0; };
  // Exact solution (??) at time T = 1.0
  auto u_exact = [](double x) {
    return (x <= 1.0) ? 0.0 : ((Constant::e <= x) ? 1.0 : std::log(x));
  };
  for (int i = 0; i < M_size; ++i) {
    Eigen::VectorXd x = getXValues(T, M(i));
    Eigen::VectorXd u0 = x.unaryExpr(u_initial);
    Eigen::VectorXd uT = solveLaxWendroff(u0, T, M(i));

    double tau = T / M(i);
    double h = Constant::e * tau;
    error(i) = h * (x.unaryExpr(u_exact) - uT).lpNorm<1>();
  }
  return error;
}
```

PYTHON script 8.4.12: Sub-problem (8-4.g): PYTHON script for plotting ⇝ GITHUB

```python
from matplotlib.pyplot import figure, legend, loglog, savefig, xlabel, ylabel
from numpy import exp, genfromtxt, log, polyfit
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
M = data[0]

name = ['LaxWendroffRP', 'LaxWendroffSmoothU0', 'GodunovSmoothU0']
color = ['b', 'g', 'c']
n = len(name)

error = []
p = []
for i in range(n):
    error.append(data[i + 1])
```

```
19      p.append(polyfit(log(M), log(error[i]), 1))
20
21  fig = figure()
22  for i in range(n):
23      loglog(M, error[i], color[i] + 'o', label = name[i])
24      loglog(M, exp(p[i][0] * log(M) + p[i][1]), color[i] + '—', label='slope = ' +
            str(round(p[i][0], 2)))
25  xlabel('number of time steps M')
26  ylabel('error')
27  legend()
28  savefig(output_file)
29
30  print('Generated ' + output_file)
```



◁ Neatly aligned error points in a doubly logarithmic plot hint at *algebraic convergence*.

From the slope of the line we read off a rate of convergence of ≈ 1.

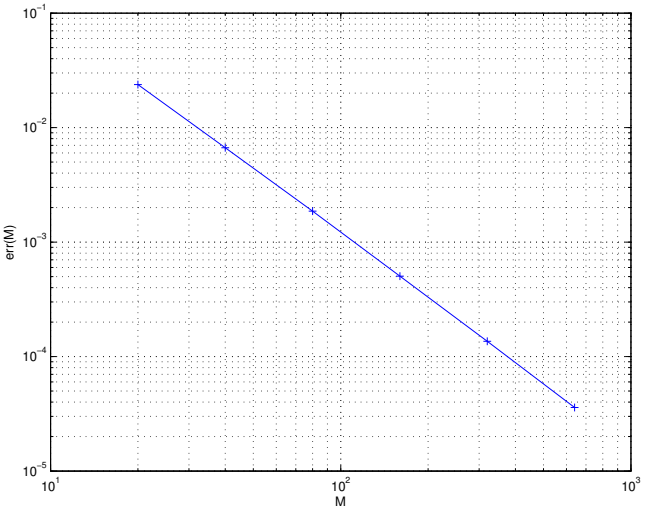
SOLUTION of (8-4.h):

C++ code 8.4.14: Sub-problem (8-4.h): Computation of reference solutuion ⇝ GITHUB

```
2  double smoothU0(double x) {
3    return (x < 0.0)
4               ? 0.0
5               : ((1.0 < x) ? 1.0 : Square(std::sin(0.5 * Constant::pi * x)));
6  }
7  Eigen::VectorXd referenceSolution(const Eigen::VectorXd &x) {
8    double T = 1.0;
9    // Reference solution on a very fine mesh
10   unsigned int M = 3200;
11
12   Eigen::VectorXd y = getXValues(T, M);
13   Eigen::VectorXd u0 = y.unaryExpr(&smoothU0);
14   Eigen::VectorXd u = solveLaxWendroff(u0, T, M);
15   int N = x.size();
16   Eigen::VectorXd u_ref(N);
17   // The vector u is larger than u_ref. Use eval() from above the "evaluate" u
18   // at the positions x(i) and thus obtain the reference solution u_ref.
19   for (int i = 0; i < N; ++i) {
20     u_ref(i) = eval(u, y, x(i));
21   }
22   return u_ref;
23 }
```

C++ code 8.4.15: Sub-problem (8-4.h): function `numexpLaxWendroffSmoothU0()` ⇝ GITHUB

```
2  Eigen::VectorXd numexpLaxWendroffSmoothU0(const Eigen::VectorXi &M) {
3    const double T = 1.0;
4    const int M_size = M.size();
5    Eigen::VectorXd error(M_size);
6
7    for (int i = 0; i < M_size; ++i) {
8      Eigen::VectorXd x = getXValues(T, M(i));
9      Eigen::VectorXd u0 = x.unaryExpr(&smoothU0);
10     Eigen::VectorXd uT = solveLaxWendroff(u0, T, M(i));
11
12     double tau = T / M(i);
13     double h = Constant::e * tau;
14     error(i) = h * (referenceSolution(x) - uT).lpNorm<1>();
15   }
16   return error;
17 }
```



Again we observe algebraic convergence with a slightly better rate of almost 2.

SOLUTION of (8-4.i):

The Godunov numerical flux function is a two-point flux function defined as

$$F(v,w) = f(u^{\downarrow}(v,w)) = e^{u^{\downarrow}(v,w)},$$

where $u^{\downarrow}$ is defined as in [Lecture $\rightarrow$ Eq. (8.3.4.31)]. There are five cases. However, the second and fifth can never occur, because f' is always positive in our case, and the first can never occur because $\dot{s} \geq 0$ always. So our Godunov flux function is quite simply

$$F(v,w) = e^{v}.$$

SOLUTION of (8-5.a):

$\dim V_h = \infty$, because it is defined on an infinite mesh!

SOLUTION of (8-5.j):

The modulus in the integral makes this problem a little awkward. In general, $f'(u) = 1 - 2u$, which changes sign at $u = 1/2$. Then, we can split the integral,

$$\int_v^w |f'(\xi)| \mathrm{d}\xi = \int_v^{1/2} |f'(\xi)| \mathrm{d}\xi - \int_w^{1/2} |f'(\xi)| \mathrm{d}\xi \,,$$

where we are now ensured that $f'(\xi)$ has constant sign in each of the terms on the right. Finally, we have

$$\int_s^{1/2} |f'(\xi)| \mathrm{d}\xi = \begin{cases} \frac{1}{4} - s(1-s), & \text{, if } s < \frac{1}{2} \,, \\ s(1-s) - \frac{1}{4}, & \text{, if } s \geq \frac{1}{2} \,, \end{cases}$$

or, expressed differently,

$$\int_s^{1/2} |f'(\xi)| \, \mathrm{d}\xi = \left(s(1-s) - \frac{1}{4} \right) \operatorname{sgn}\left(s - \frac{1}{2} \right) \,,$$

where sgn is the sign function.

C++ code 8.5.15: Sub-problem (8-5.j): function `Feo()` ⇝ GITHUB

```
double Feo(double v, double w) {
  double result;
  auto f = [](double u) { return u * (1.0 - u); };
  auto I = [](double s) {
    double y = 0.25 - s * (1.0 - s);
    return s < 0.5 ? y : -y;
  };
  result = 0.5 * (f(v) + f(w)) - 0.5 * (I(v) - I(w));
  return result;
}
```

SOLUTION of (8-5.k):

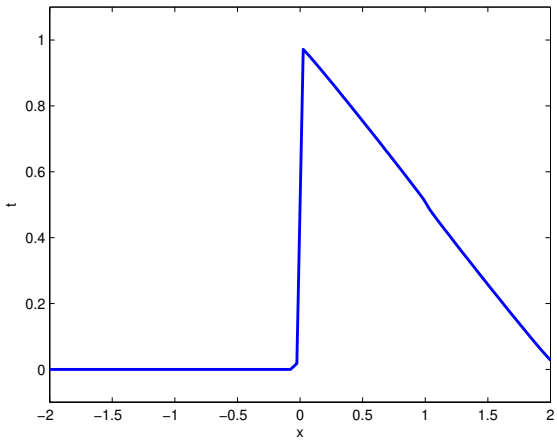
C++ code 8.5.17: Sub-problem (8-5.k): function `solveTrafficFlow()` ⇝ GITHUB

```
Solution solveTrafficFlow() {
  int Ml = 40;
  int Mr = 40;
  int N_half = Mr + Ml + 1;
  int N = 2 * N_half;

  double h = 0.05;
  double tau = h / 3;
  double T = 1.0;
  unsigned int m = (unsigned int)(T / tau);

  // Spatial mesh
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N_half, -2.0, 2.0);
  // Initial condition according to (8.5.16). Note that we just
  // initialize the cell averages; a more refined implementation would also
  // sample the local first moments of u_0.
  Eigen::VectorXd mu0 = Eigen::VectorXd::Zero(N);
  auto u0 = [](double x) { return 0.0 <= x && x <= 1.0 ? 1.0 : 0.0; };
  for (int i = 0; i < N_half; ++i) {
    mu0(2 * i) = u0(x(i));
  }
  // Flux function
  auto f = [](double u) { return u * (1.0 - u); };
  // Perform fully discrete evolution
  Eigen::VectorXd mu = dgcl(mu0, f, Feo, T, Ml, Mr, h, m);
  // Retrieve cell averages
  Eigen::VectorXd u(N_half);
  for (int i = 0; i < N_half; ++i) {
    u(i) = mu(2 * i);
  }

  return Solution(std::move(x), std::move(u));
}
```

PYTHON script 8.5.18: Sub-problem (8-5.k): Pyhton script for plotting solution at final time ⇝ GITHUB

```
from matplotlib.pyplot import figure, plot, savefig, xlabel, ylabel
from numpy import genfromtxt
from sys import argv

input_file = str(argv[1])
output_file = str(argv[2])

data = genfromtxt(input_file, delimiter=',')
x = data[0]
y = data[1]

figure()
plot(x, y)
```

```
14 xlabel('x')
15 ylabel('u(x,1)')
16 savefig(output_file)
17
18 print('Generated ' + output_file)
```



◁ Solution at final time $T = 1$.
The solution features a leading rarefaction fan and a trailing stationary shock.

HINT 1 for (8-5.b): Perform the usual steps in the derivation of a spatial variational formulation: test, integrate, and integrate by parts (in one spatial dimension). ⌟

SOLUTION of (8-5.b):

The important part is the integration by parts in the flux term.

$$\int_{x_{j-1}}^{x_j} v_h(x)\frac{\partial}{\partial x}\big(f(u(x,t))\,\mathrm{d}x = [f(u(x,t))v_h(x)]_{x_{j-1}}^{x_j} - \int_{x_{j-1}}^{x_j} f(u(x,t))\frac{\mathrm{d}v_h}{\mathrm{d}x}(x)\mathrm{d}x$$

$$= f(u(x_j,t))v_h(x_j-) - f(u(x_{j-1},t))v_h(x_{j-1}+) - \int_{x_{j-1}}^{x_j} f(u(x,t))\frac{\mathrm{d}v_h}{\mathrm{d}x}(x)(x)\mathrm{d}x\ .$$

SOLUTION of (8-5.c):

The *maximal speed of propagation* in either direction, and at any time, is c. Thus, at time $T = 1$, u is supported in $[-c, 1+c]$. We want $x_{-M_l-1} = (-M_l - 1)h \leq -c$, which implies $M_l + 1 \geq c/h$. Similarly, $M_r \geq (1+c)/h$. Thus

$$M_l = \left\lceil \frac{c}{h} \right\rceil - 1, \qquad M_r = \left\lceil \frac{1+c}{h} \right\rceil.$$

SOLUTION of (8-5.d):

The matrix $\mathbf{B}$ is actually just the mass matrix. We can see that two basis functions b_k^i and b_l^j have disjoint support if $i \neq j$, and since b_2^j has average value zero, we also see that b_1^i and b_2^i are orthogonal. This makes $\mathbf{B}$ fully diagonal. The entries corresponding to the constant basis functions are h, and the others are (using the $[0, h]$ interval to compute)

$$\int_0^h \left(x - \frac{h}{2}\right)^2 \mathrm{d}x = \frac{1}{3}\left(h - \frac{h}{2}\right)^3 - \frac{1}{3}\left(0 - \frac{h}{2}\right)^3 = \frac{h^3}{12}.$$

Thus,

$$\mathbf{B} = \begin{pmatrix} h & & & & \\ & \frac{h^3}{12} & & & \\ & & h & & \\ & & & \frac{h^3}{12} & \\ & & & & \ddots \end{pmatrix}.$$

SOLUTION of (8-5.e):

Pay attention to the efficient initialization of the sparse matrix in EIGEN. Since we know in advance that every row will have exactly one non-zero entry, we can reserve that space.

C++ code 8.5.9: Sub-problem (8-5.e): function `compBmat()` ↪ GITHUB

```
2  Eigen::SparseMatrix<double> compBmat(int Ml, int Mr, double h) {
3    const int N = 2 * (Ml + Mr + 1);
4    Eigen::SparseMatrix<double> A(N, N);
5    // Tell Eigen that the matrix is diagonal, which allows efficient
6    // initialization
7    A.reserve(Eigen::VectorXi::Ones(N));
8    for (int i = 0; i < N; i += 2) {
9      A.insert(i, i) = h;
10     A.insert(i + 1, i + 1) = h * h * h / 12.0;
11   }
12   return A;
13 }
```

HINT 1 for (8-5.f): If it helps, you can use the notation $\mu_{i,1}$ and $\mu_{i,2}$ to reference the elements in a vector $\vec{\boldsymbol{\mu}}$ corresponding to basis functions b_1^i and b_2^i respectively (e.g. as if $\boldsymbol{\mu}$ were a matrix). ⌟

SOLUTION of (8-5.f):

We need to evaluate the expression

$$-\sum_{j=\infty}^{\infty}\int_{x_{j-1}}^{x_j} f(u_h(x))\frac{\mathrm{d}v_h}{\mathrm{d}x}(x)\mathrm{d}x + \sum_{j=-\infty}^{\infty} F(u_h(x_j-),u_h(x_j+))(v_h(x_j-)-v_h(x_j+))$$

for $v_h = b_l^i$, $l = 1,2$.

For $l = 1$ (the constant basis functions), we see that the derivative is zero, and that $b_1^i(x_{i-1}+) = b_1^i(x_i-) = 1$, with all other grid-point evaluations being zero. The expression then collapses to

$$G(\vec{\mu})_{i,1} = F(u_h(x_i-),u_h(x_i+)) - F(u_h(x_{i-1}-),u_h(x_{i-1}+)) \ .$$

For $l = 2$ (the linear basis function with vanishing mean), the derivative is constant 1, and $b_2^i(x_{i-1}+) = -h/2$ and $b_2^i(x_i-) = h/2$. Thus, the integral term does *not* vanish, the numerical flux terms will be multiplied by $h/2$ and there is one sign change. We get

$$G(\vec{\mu})_{i,2} = -\int_{x_{i-1}}^{x_i} f(u_h(x))\mathrm{d}x + \frac{h}{2}[F(u_h(x_i-),u_h(x_i+)) + F(u_h(x_{i-1}-),u_h(x_{i-1}+))] \ .$$

Further, we have the formulas for u_h:

$$u_h(x_i-) = \mu_{i,1} + \frac{h}{2}\mu_{i,2}, \qquad u_h(x_{i-1}+) = \mu_{i,1} - \frac{h}{2}\mu_{i,2} \ .$$

HINT 1 for (8-5.g): Do not forget the rescaling of quadrature weights when using (8.5.10) on an interval of length h. ⌟

SOLUTION of (8-5.g):

The quadrature points relative to $[x_{i-1}, x_i]$ are

$$x_i + \frac{h}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right) .$$

And u_h evaluated at those points is (we benefit from nice cancellations)

$$u_h(x_i + \frac{h}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right)) = \mu_{i,1} + \mu_{i,2}\left[x_i + \frac{h}{2}\left(1 \pm \frac{1}{\sqrt{3}}\right) - \frac{1}{2}(2x_i + h)\right] = \mu_{i,1} \pm \frac{h}{2\sqrt{3}}\mu_{i,2} .$$

Our approximation for the integral in question is thus

$$\int_{x_{i-1}}^{x_i} f(u_h(x))\mathrm{d}x \approx \frac{h}{2}\left[f\left(\mu_{i,1} - \frac{h}{2\sqrt{3}}\mu_{i,2}\right) + f\left(\mu_{i,1} + \frac{h}{2\sqrt{3}}\mu_{i,2}\right)\right] ,$$

giving

$$\begin{aligned} G(\vec{\mu})_{i,2} = &- \frac{h}{2}\left[f\left(\mu_{i,1} - \frac{h}{2\sqrt{3}}\mu_{i,2}\right) + f\left(\mu_{i,1} + \frac{h}{2\sqrt{3}}\mu_{i,2}\right)\right] + \\ &+ \frac{h}{2}[F(u_h(x_i-), u_h(x_i+)) + F(u_h(x_{i-1}-), u_h(x_{i-1}+))] . \end{aligned}$$

The formula for $G(\vec{\mu})_{i,1}$ remains the same.

SOLUTION of (8-5.h):

C++ code 8.5.11: Sub-problem (8-5.h): function G() ⇝ GITHUB

```
template <typename FUNCTOR, typename NUMFLUX>
Eigen::VectorXd G(const Eigen::VectorXd &mu, FUNCTOR &&f, NUMFLUX &&F, int Ml,
                  int Mr, double h) {
  const int N_half = (Ml + Mr + 1);
  const int N = 2 * N_half;
  Eigen::VectorXd Gvec(N);
  double uN_xminus = 0.0;  // since we extend mu to the left by zero
  double uN_xplus = mu(0) - 0.5 * h * mu(1);
  double F_old;
  double F_new = F(uN_xminus, uN_xplus);
  const double w = h / (2.0 * std::sqrt(3.0));

  for (int i = 0; i < N_half - 1; ++i) {
    uN_xminus = mu(2 * i) + 0.5 * h * mu(2 * i + 1);
    uN_xplus = mu(2 * (i + 1)) - 0.5 * h * mu(2 * (i + 1) + 1);
    F_old = F_new;
    F_new = F(uN_xminus, uN_xplus);
    Gvec(2 * i) = F_new - F_old;

    double x_minus = mu(2 * i) - w * mu(2 * i + 1);
    double x_plus = mu(2 * i) + w * mu(2 * i + 1);
    double I = 0.5 * h * (f(x_minus) + f(x_plus));
    Gvec(2 * i + 1) = 0.5 * h * (F_new + F_old) - I;
  }

  uN_xminus = mu(2 * (N_half - 1)) + 0.5 * h * mu(2 * (N_half - 1) + 1);
  uN_xplus = 0.0;  // since we extend mu to the right by zero
  F_old = F_new;
  F_new = F(uN_xminus, uN_xplus);
  Gvec(2 * (N_half - 1)) = F_new - F_old;

  double x_minus = mu(2 * (N_half - 1)) - w * mu(2 * (N_half - 1) + 1);
  double x_plus = mu(2 * (N_half - 1)) + w * mu(2 * (N_half - 1) + 1);
  double I = 0.5 * h * (f(x_minus) + f(x_plus));
  Gvec(2 * (N_half - 1) + 1) = 0.5 * h * (F_new + F_old) - I;
  return Gvec;
}
```

SOLUTION of (8-5.i):

Applying the 2-stage explicit Runge-Kutta method to (8.5.8) entails the computation of the increment vector

$$\vec{\kappa} := \vec{\mu}^{(k)} - \tfrac{1}{2}\tau \mathbf{B}^{-1} G(\vec{\mu}^{(k)})$$

and the update

$$\vec{\mu}^{(k+1)} := \vec{\mu}^{(k)} + \tau \mathbf{B}^{-1} G(\vec{\kappa}) \,.$$

C++ code 8.5.13: Sub-problem (8-5.i): function `dgcl()` ↪ GITHUB

```
2  template <typename FUNCTOR, typename NUMFLUX>
3  Eigen::VectorXd dgcl(Eigen::VectorXd mu0, FUNCTOR &&f, NUMFLUX &&F, double T,
4                       int Ml, int Mr, double h, unsigned int m) {
5    Eigen::SparseMatrix<double> B = compBmat(Ml, Mr, h);
6    Eigen::SparseMatrix<double> Binv = B.cwiseInverse();
7
8    auto G_bound = [&f, &F, T, Ml, Mr, h](const Eigen::VectorXd &mu) {
9      return G(mu, std::forward<FUNCTOR>(f), std::forward<NUMFLUX>(F), Ml, Mr, h);
10   };
11   // Timestepping based on explicit midpoint rule, a 2-stage explicit
       Rune-Kutta
12   // single-step method
13   double tau = T / m;
14   for (int i = 0; i < m; ++i) {
15     // First compute the increment and then update the state vector, see
16     // [Lecture → Eq. (8.4.1.12)]
17     Eigen::VectorXd k = -Binv * G_bound(mu0);
18     mu0 = mu0 - tau * Binv * G_bound(mu0 + 0.5 * tau * k);
19   }
20   return mu0;
21 }
```

SOLUTION of (8-6.a):

From [Lecture → Thm. 8.2.7.1] we know that

$$-1 \leq \min_{x \in \mathbb{R}} u_0(x) \leq u(x,t) \leq \max_{x \in \mathbb{R}} u_0(x) \leq 1 \ .$$

Hence, all we need to know is $f|_{[-1,1]}$, that is, the spline model needs to cover only $[-1,1]$: $a = -1$, $b = 1$.

HINT 1 for (8-6.b): A function $f \in C^2([a,b])$ is strictly convex, if and only if its *second derivative* is strictly positive. ⌟

SOLUTION of (8-6.b):

Note that for a cubic spline f in the representation (8.6.4) its second derivative $s := f''$ is a piecewise linear and continuous function with $s(\xi_j) = M_j$, see (8.6.5). Obviously, such a function is strictly positive, if and only if all its function values at the knots ξ_j are positive

$$s(x) > 0 \quad \forall x \in [a,b] \quad \Leftrightarrow \quad s(\xi_j) > 0 \quad \forall j \in \{0,\ldots,n\} \, .$$

Hence, f is strictly convex, if and only if $M_j > 0$ for all $j \in \{0,\ldots,n\}$.

SOLUTION of (8-6.c):

Recall [Lecture → Thm. 8.2.7.3] and the accompanying discussion.

First we have to determine the maximal speeds of propagation in both directions:

$$\dot{s}_{\min} := \min\{f'(\xi) : \min_{x\in\mathbb{R}} u_0(x) \le \xi \le \max_{x\in\mathbb{R}} u_0(x)\} .$$

In light of (8.6.2) and the strict convexity of f, which implies that f' is *strictly increasing*, we conclude

$$\dot{s}_{\min} = f'(-1) \quad , \quad \dot{s}_{\max} = f'(1) . \tag{8.6.6}$$

Hence, if we know that $\operatorname{supp} u_0 \subset [A,B]$ (from `initsupp`), then

$$\operatorname{supp} u(\cdot,t) \subset [A + f'(-1)t, B + f'(1)t] , \tag{8.6.7}$$

see [Lecture → Fig. 395]. Thanks to the strict convexity of f we know $f'(-1) < f'(1)$ so that (8.6.7) always defines a valid interval. The numbers $f'(-1)$ and $f'(1)$ can be obtained by calling the `derivative` method of **UniformCubicSpline**.

We can easily convert (8.6.7) into a code.

C++ code 8.6.8: Implementation of `findSupport()` → GITHUB

```
2 Eigen::Vector2d findSupport(const UniformCubicSpline &f,
3                             Eigen::Vector2d initsupp, double t) {
4   Eigen::Vector2d result;
5   Eigen::Vector2d speed = {f.derivative(-1.0), f.derivative(1.0)};
6   result = initsupp + t * speed;
7   return result;
8 }
```

HINT 1 for (8-6.d): You may call the function `findSupport()` from Sub-problem (8-6.c). ⌟

SOLUTION of (8-6.d):

Note that the support of $\{x \mapsto u(x,t)\}$ may not necessarily contain the support of u_0! In fact, $\bigcup_{0\le t\le T} \operatorname{supp} u$ is the convex hull of $[-1,1]$ and $\operatorname{supp} u(\cdot,T)$. Thus, using (8.6.7) we find

$$\bigcup_{0\le t\le T} \operatorname{supp} u(\cdot,t) = [\min\{-1,-1+f'(-1)T\}, \max\{1,1+f'(1)T\}]\,,$$

which gives

$$m^- = \left\lfloor \frac{\min\{-1,-1+f'(-1)T\}}{h} \right\rfloor \quad , \quad m^+ = \left\lceil \frac{\max\{1,1+f'(1)T\}}{h} \right\rceil .$$

The two numbers $-1+f'(-1)T$ and $1+f'(1)T$ are returned by `findSupport()`.

C++ code 8.6.9: Implementation of `computeInitVec()` ↬ GITHUB

```
template <typename FUNCTOR>
Eigen::VectorXd computeInitVec(const UniformCubicSpline &f, FUNCTOR &&u0,
                               double h, double T) {
  Eigen::VectorXd mu0;
  const double A = -1.0;  // left bound of support
  const double B = 1.0;   // right bound of support
  // Here one could also call findSupport()
  const double AT = std::min(A, A + f.derivative(-1.0) * T);
  const double BT = std::max(B, B + f.derivative(1.0) * T);

  const int m_minus = (int)std::floor(AT / h);
  const int m_plus = (int)std::ceil(BT / h);
  const int N = m_plus - m_minus + 1;

  const Eigen::VectorXd x =
      Eigen::VectorXd::LinSpaced(N, m_minus * h, m_plus * h);
  mu0 = x.unaryExpr(std::forward<FUNCTOR>(u0));
  return mu0;
}
```

SOLUTION of (8-6.e):

As we have seen in [Lecture → Section 8.2.6], since f is strictly convex the entropy solution of a Riemann problem for (8.6.1) in the case of a downward jump $u_l \to u_r$, $u_l > u_r$, in the initial data u_0 is a **shock** moving with the Rankin-Hugoniot speed

$$\dot{s} := \frac{f(u_l) - f(u_r)}{u_l - u_r} .$$

Thus we find the solution

$$u(x,t) = \begin{cases} 1 & \text{for } x < \dot{s}\, t\,, \\ -1 & \text{for } x > \dot{s}\, t\,, \end{cases} \quad , \quad \dot{s} := \frac{f(u_l) - f(u_r)}{u_l - u_r} .$$

HINT 1 for (8-6.f): A simple bisection-type root finding algorithm is available in the function (also in `clempiricflux.cc`)

```
template <typename FUNCTOR>
double findRoots(double v, double w, FUNCTOR &&g,
                 double tol = 1.0E-6);
```

The functor `g` must pass a function $g : [v, w] \to \mathbb{R}$ with signature **std**`::function<`**double**`(`**double**`)>`, which changes sign in $[v, w]$. Th function returns an approximation of $\zeta \in [v, w]$ with $g(\zeta) = 0$. ⌟

SOLUTION of (8-6.f):

We write $F_{\mathrm{GD}} = F_{\mathrm{GD}}(v,w)$ for the Godunov numerical flux. The discussion follows [Lecture → § 8.3.4.32].

❶: If $v \geq w$, the local Riemann problem will have as an entropy solution a shock moving with the Rankine-Hugoniot speed

$$\dot{s} := \frac{f(u_l) - f(u_r)}{u_l - u_r} .$$

Hence,

$$F(v,w) = \begin{cases} f(v) & \text{, if } \dot{s} \geq 0 \,, \\ f(w) & \text{, if } \dot{s} < 0 \,. \end{cases} \tag{8.6.10}$$

❷: If $v < w$ we have to deal with a local rarefaction solution. The left edge of the rarefaction fan will move with speed $f'(v)$, the right edge with speed $f'(w)$.

$$F(v,w) = \begin{cases} f(v) & \text{, if } f'(v) > 0 \,, \\ f(z) & \text{, if } f'(v) < 0 < f'(w) \,, \\ f(w) & \text{, if } f'(w) < 0 \,. \end{cases} \tag{8.6.11}$$

where z is the unique zero of f' in $[v,w]$: $f'(z) = 0$. `findRoots()` can be used to determine z.

Computing the Godunov flux for a general flux function entails computing the zeros of a function ("root finding"). To find this zero, you may apply simple root-finding algorithms like the regula falsi or bisection merely require point evaluations of f', which are available through the `derivative()` method of **UniformCubicSpline**.

C++ code 8.6.12: Implementation of `findRoots()` ↬ GITHUB

```
2  template <typename FUNCTOR>
3  double findRoots(double v, double w, FUNCTOR &&g, double tol = 1.0E-6) {
4    double x = v;  // approximate root
5    const double len = w - v;
6    constexpr static const int maxN = 1000;
7    double gv = g(v), gw = g(w);
8    // Ensure that function changes sign
9    assert(gv * gw <= 0);
10   for (int N = 0; (std::abs(w - v) > tol * len) && N < maxN; N++) {
11     x = (v + w) / 2.0;
12     const double gx = g(x);
13     if (gv * gx < 0.0) {
14       // Sign change in left half of [v,w]
15       w = x;
16       gw = gx;
17     } else {
18       // Sign change in right half of [v,w]
19       v = x;
20       gv = gx;
21     }
22   }
23   return x;
24 }
```

C++ code 8.6.13: Evaluation operator for GodunovFlux ➙ GITHUB

```
double GodunovFlux::operator()(double v, double w) const {
  double result;
  if (v >= w)
    result = std::max(_f(v), _f(w));
  else {
    if (_f.derivative(v) > 0.0)
      result = _f(v);
    else if (_f.derivative(w) < 0.0)
      result = _f(w);
    else {
      auto df = [this](double x) { return _f.derivative(x); };
      double z = findRoots(v, w, df);
      result = _f(z);
    }
  }
  return result;
}
```

SOLUTION of (8-6.g):

Of course, we have to use [Lecture → Def. 6.2.6.31] and the conventions linking it with the Butcher scheme [Lecture → Eq. (6.2.6.33)]. From the (8.6.14) we get the coefficients

$$a_{11} = 0\,, \quad a_{12} = 0\,, \quad a_{21} = \frac{2}{3}\,, \quad a_{22} = 0\,,$$
$$b_1 = \frac{1}{4} \quad , \quad b_2 = \frac{3}{4}\,.$$

The coefficients c_i are not needed, because we deal with an autonomous ODE. For one step of the Ralston scheme starting from the state $\mathbf{y}_0$ and yielding the state $\mathbf{y}_1$ we obtain the increment equations

$$\mathbf{k}_1 := \mathbf{f}(\mathbf{y}_0) \quad , \quad \mathbf{k}_2 := \mathbf{f}(\mathbf{y}_0 + \frac{2}{3}\tau\mathbf{k}_1)\,,$$

and the update equation

$$\Psi^\tau\mathbf{y}_0 = \mathbf{y}_1 := \mathbf{y}_0 + \tau(\frac{1}{4}\mathbf{k}_1 + \frac{3}{4}\mathbf{k}_2)\,.$$

The implementation of these formulas is straightforward, because the Ralston scheme is an **explicit** Runge-Kutta single-step method.

C++ code 8.6.15: Implementation of `RalstonODESolver()` ↪ GITHUB

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd RalstonODESolver(FUNCTOR &&rhs, Eigen::VectorXd mu0, double tau,
4                                   int n) {
5    for (int i = 0; i < n; ++i) {
6      Eigen::VectorXd k1 = rhs(mu0);
7      Eigen::VectorXd k2 = rhs(mu0 + tau * 2.0 / 3.0 * k1);
8      mu0 = mu0 + 0.25 * tau * (k1 + 3.0 * k2);
9    }
10   return mu0;
11 }
```

SOLUTION of (8-6.h):

- First, we initialize an object of type **GodunovFlux** that will provide the numerical flux for our method.
- Then the right-hand-side of the ODE arising after spatial semi-discretization can be represented by a λ-function.
- The fully discrete scheme is then implemented using a formula similar to [Lecture → Eq. (8.5.3.2)] with coefficients given by the Butcher scheme (8.6.14).

  This 2-stage timestepping method leads to a 5-point spatial stencil in the sense of [Lecture → § 8.4.2.1]. Thus the geometric CFL-condition according to [Lecture → Def. 8.4.2.11] stipulates the timestep

$$\tau \leq \min\left\{ \frac{2h}{|f'(-1)|}, \frac{2h}{|f'(1)|} \right\}. \tag{8.6.16}$$

  However, depending on the Runge-Kutta single-step method employed, this condition may not ensure stability. Thus a safer option is the choice

$$\tau := \min\left\{ \frac{h}{|f'(-1)|}, \frac{h}{|f'(1)|} \right\}. \tag{8.6.17}$$

In the following implementation the right-hand side of the method-of-lines ODE is provided through a separate function, in order to be able to reuse `RalstonODESolver()`.

C++ code 8.6.18: Implementation of `solveCauchyProblem()` ↬ GITHUB

```
2  Eigen::VectorXd solveCauchyProblem(const UniformCubicSpline &f,
3                                     const Eigen::VectorXd &mu0, double h,
4                                     double T) {
5    Eigen::VectorXd muT(mu0.size());
6    double tau = std::min(h / std::abs(f.derivative(-1.0)),
7                          h / std::abs(f.derivative(1.0)));
8    double n = (int)std::floor(T / tau);
9    GodunovFlux godunovFlux(f);
10   auto rhs = [h, &godunovFlux](const Eigen::VectorXd &mu) {
11     return semiDiscreteRhs(mu, h, godunovFlux);
12   };
13   muT = RalstonODESolver(rhs, mu0, tau, n);
14   return muT;
15 }
```

The following function provides the implementation of the right-hand-side function $\mathbf{f}$ of the method-of-lines ODE arising from the conservative finite volume method outlined above.

C++ code 8.6.19: Implementation of right-hand side of the method-of-lines ODE ↬ GITHUB

```
2  template <typename FUNCTOR>
3  Eigen::VectorXd semiDiscreteRhs(const Eigen::VectorXd &mu0, double h,
4                                  FUNCTOR &&numFlux) {
5    int m = mu0.size();
6    Eigen::VectorXd mu1(m);
7    mu1(0) = -1.0 / h * (numFlux(mu0(0), mu0(1)) - numFlux(mu0(0), mu0(0)));
```

```
8     for (int j = 1; j < m - 1; ++j) {
9       mu1(j) =
10          -1.0 / h * (numFlux(mu0(j), mu0(j + 1)) - numFlux(mu0(j - 1), mu0(j)));
11    }
12    mu1(m - 1) =
13        -1.0 / h *
14        (numFlux(mu0(m - 1), mu0(m - 1)) - numFlux(mu0(m - 2), mu0(m - 1)));
15    return mu1;
16  }
```

SOLUTION of (8-7.a):

The flux function is defined for $u > 0$ and can be continuously extended to $u = 0$ through $f(0) := 0$. It is smooth on $]0, \infty[$. To check its curvature we examine the sign of the second derivative: By the product rule

$$f'(u) = \log u \quad , \quad f''(u) = u^{-1} > 0 \quad \text{for} \quad u > 0 \, .$$

SOLUTION of (8-7.j):

We observe that halving the timestep leads to a decrease of the error by a factor of 8, which is clear evidence of algebraic convergence with rate 3 : $\text{error} = O(\tau^3)$. The single-step method probably is of order 3.

SOLUTION of (8-7.k):

C++11 code 8.7.16: Code for `solveClaw()` ↬ GITHUB

```
template <typename U0_FUNCTOR>
Eigen::VectorXd solveClaw(U0_FUNCTOR &&u0, double T, unsigned int n) {
  // Set up spacial mesh and inital data.
  double a = 0.0;
  double b = 1.0;
  double h = (b - a) / n;
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(n, a + 0.5 * h, b - 0.5 * h);

  // Approximate dual cell averages at t=0
  Eigen::VectorXd mu = x.unaryExpr(u0);

  double alpha = mu.minCoeff();  // lower bound for initial data
  double beta = mu.maxCoeff();   // upper bound for initial data
  assert(alpha > 0.0 && beta > 0.0);

  // Set timestep tau according to the CFL-condition
  double tau =
      h / std::max(std::abs(std::log(alpha)), std::abs(std::log(beta)));
  // Semi-discretization (discretization in space)
  double h_inv = 1.0 / h;
  // Define right-hand-side for the SSP ODE solver
  auto semi_discrete_rhs =
      [h_inv](const Eigen::VectorXd &mu) -> Eigen::VectorXd {
    return -h_inv * slopelimfluxdiffper(mu, &logGodunovFlux, &limiterMC);
  };
  // Timestepping:  Solve the semi-discrete ODE
  int N = (int)(T / tau + 0.5);
  for (int i = 0; i < N; ++i) mu = sspEvolop(semi_discrete_rhs, mu, tau);

  return mu;
}
```

Note that the return type of the lambda function `semi_discrete_rhs` has to be specified explicitly. Otherwise a temporary object of the some EIGEN internal type will be returned, which causes problems later.

SOLUTION of (8-7.l):

For every point y_k we determine the corresponding cell of $\mathcal{M}$ and then perform local linear interpolation.

C++11 code 8.7.18: Code for `interpolate()` ↪ GITHUB

```
template <typename VECSOURCE, typename VECDEST>
void interpolate(const VECSOURCE &s, VECDEST &d) {
  // Determine number of cells
  const std::size_t n = s.size();
  const std::size_t N = d.size();
  // Mesh widths/cell sizes
  const double H = 1.0 / n;
  const double h = 1.0 / N;
  // Loop over nodes of destination mesh
  for (int j = 0; j < N; ++j) {
    // position of destinatioon node
    const double xd = h * (j + 0.5);
    // Index of source node to the right
    std::size_t k = (std::size_t)std::round(xd / H);
    // Position of nodes of source cell, in which xd is contained
    const double xsl = H * (k - 0.5);  // left node
    const double xsr = xsl + H;        // right node
    // Values at source nodes taking into account periodic continuation
    auto svl = s[(k < 1) ? (n - 1) : k - 1];
    auto svr = s[(k >= n) ? 0 : k];
    // Linear interpolation inside source cell
    d[j] = ((xsr - xd) * svl + (xd - xsl) * svr) / H;
  }
}
```

SOLUTION of (8-7.m):

C++11 code 8.7.19: Code for `studyCvgMUSCLSolution()` ↪ GITHUB

```
template <typename U0_FUNCTOR>
void studyCvgMUSCLSolution(U0_FUNCTOR &&u0, double T) {
  // For temporarily storing the number of cells and the associated
     error norms
  std::vector<std::tuple<std::size_t, double, double>> result{};
  constexpr int n_ref = 8192;
  // Compute reference solution
  std::cout << "Computing reference solution at T = " << T << " with " << n_ref
            << " cells" << std::endl;
  Eigen::VectorXd u_ref{solveClaw(std::forward<U0_FUNCTOR>(u0), T, n_ref)};
  // Compute solution on different meshes
  constexpr int lmin = 4;   // Coarsest mesh with 16 cells
  constexpr int lmax = 10;  // Finest mesh with 1024 cells
  for (int l = lmin; l <= lmax; ++l) {
    const std::size_t n = 1U << l;
    std::cout << "Computing with n = " << n << std::endl;
    // Compute solution
    Eigen::VectorXd u{solveClaw(std::forward<U0_FUNCTOR>(u0), T, n)};
    // Transfer solution to finest mesh
    Eigen::VectorXd u_prop = Eigen::VectorXd::Zero(u_ref.size());
    interpolate(u, u_prop);
    double linf_err = (u_prop - u_ref).lpNorm<Eigen::Infinity>();
    double l1_err = (u_prop - u_ref).lpNorm<1>() / n_ref;
    result.push_back({n, linf_err, l1_err});
  }
  std::cout << "n \t linf error \t l1 error" << std::endl;
  for (auto &data : result) {
    std::cout << std::get<0>(data) << " \t " << std::get<1>(data) << " \t "
              << std::get<2>(data) << std::endl;
  }
}
```

	$T = 0.2$		$T = 1.0$	
n	$L^\infty(]0,1[)$-norm	$L^1(]0,1[)$-norm	$L^\infty(]0,1[)$-norm	$L^1(]0,1[)$-norm
16	0.258554	0.0320086	0.382744	0.0279002
32	0.221642	0.0191052	0.389801	0.0131425
64	0.137047	0.00379365	0.473782	0.00757056
128	0.0792417	0.00235442	0.458947	0.00378283
256	0.0500682	0.00217349	0.47127	0.00212716
512	0.00907072	7.26122e-05	0.353872	0.000754736
1024	0.00252278	1.79136e-05	0.339504	0.000410959

As long as the solution is smooth convergence even in $L^\infty(]0,1[)$-norm can be observed. Once a shock appears, we can see only convergence in the $L^1(]0,1[)$-norm. From the data it is difficult to tell a rate of algebraic convergence. This is similar to what we saw in [Lecture → Exp. 8.4.4.1].

SOLUTION of (8-7.b):

Since the flux function f is convex, the structure of solutions of Riemann problems is the same as for the Burgers equation discussed in [Lecture $\rightarrow$ Section 8.2.6]. For $u_l > u_r$ we get a **shock solution**, a discontiunity moving with the Rankine-Hugoniot speed.

$$u(x,t) = \begin{cases} 2 = u_l & \text{for} \quad x < \dot{s}t \,, \\ 1 = u_r & \text{for} \quad x > \dot{s}t \,, \end{cases} \quad , \quad \dot{s} = \frac{f(u_l) - f(u_r)}{u_l - u_r} = 2(\log 2 - 1) + 1 \approx 0.3863 \,. \tag{8.7.2}$$

SOLUTION of (8-7.c):

The discussion for the Burgers equation in [Lecture → Section 8.2.6] reveals that in this case the entropy solution is a **rarefaction wave**. We use the formula given in [Lecture → Lemma 8.2.5.10] with $f'(u) = \log u$ and $g(\xi) = \exp \xi$, $u_l = 1$, $u_r = 2$:

$$f'(u_l) = 0 \quad , \quad f'(u_r) = log2 \approx 0.6931 > 0 \, .$$

This yields the continuous solution

$$u(x,t) = \begin{cases} 1 = u_l & \text{for} \quad x < 0 \, , \\ \exp(x/t) & \text{for} \quad 0 < x < t \log 2 \, , \\ 2 = u_r & \text{for} \quad x > t \log 2 \, . \end{cases} \tag{8.7.3}$$

SOLUTION of (8-7.d):

Of course, we have to assume admissible states $v, w > 0$. Since $u \mapsto f(u)$ is smooth and convex on $\mathbb{R}^+$, we can compute the intermediate state according to [Lecture → Eq. (8.3.4.31)] with $(f')^{-1}(0) = \exp 0 = 1$

$$u^{\downarrow}(v,w) = \begin{cases} w & \text{, if } v \geq w,\ \dot{s} \leq 0 \quad \text{or} \quad v < w,\ w < 1\,, \\ v & \text{, if } v > w,\ \dot{s} > 0 \quad \text{or} \quad v < w,\ v > 1\,, \\ 1 & \text{, if } v \leq 1 \leq w\,, \end{cases} \tag{8.7.4}$$

$$\text{with} \qquad \dot{s} = \begin{cases} \dfrac{v(\log v - 1) - w(\log w - 1)}{v - w} & \text{, if } v \neq w\,, \\ \log v & \text{, if } v = w\,. \end{cases} \tag{8.7.5}$$

Then $F_{\mathrm{GD}(v,w)} = f(u^{\downarrow}(v,w))$.

Remark. Note that due to cancellation (8.7.5) is prone to numerical instability in case $v \approx w$. However, we are only interested in the sign of $\dot{s}$, which will be correct even if we lose almost all the digits.

The implementation of (8.7.4) is straightforward.

C++11 code 8.7.6: Code for `logGodunovFlux()` ⇝ GITHUB

```
2  double logGodunovFlux(double v, double w) {
3    double godunov_numerical_flux;
4    auto f = [](double u) { return u * (std::log(u) - 1.0); };
5    auto df = [](double u) { return std::log(u); };
6    if (v >= w)
7      godunov_numerical_flux = std::max(f(v), f(w));
8    else {
9      if (df(v) > 0.0)
10       godunov_numerical_flux = f(v);
11     else if (df(w) < 0.0)
12       godunov_numerical_flux = f(w);
13     else
14       godunov_numerical_flux = f(1.0);  // note:  df(1.0) = 0.0
15   }
16   return godunov_numerical_flux;
17 }
```

SOLUTION of (8-7.e):

The scheme (8.7.7) has a spatial stencil of $m = 1$ as explained in [Lecture → § 8.4.2.1]. The CFL-conditions enforces the timestep constraint $\tau \leq h/s$, where $s > 0$ is the maximal speed of propagation

$$s := \sup\{|f'(\xi)| : \inf_{x\in\mathbb{R}} u_0(x) \leq \xi \leq \sup_{x\in\mathbb{R}} u_0(x)\} . \tag{8.7.8}$$

For (8.7.1) we have $f'(\xi) = \log \xi$ and we infer

$$s \leq \max\{|\log \alpha|, |\log \beta|\} \quad \blacktriangleright \quad \tau \leq \frac{h}{\max\{|\log \alpha|, |\log \beta|\}} .$$

Remark. The formula for the maximal speed of propagation (8.7.8) is relevant for the CFL-condition, because [Lecture → Lemma 8.3.5.8] tells us that the Godunov numerical flux is monoton according to [Lecture → Def. 8.3.5.5]. As a consequence of the arguments in [Lecture → § 8.3.5.1] we conclude that

$$\min_{\ell\in\mathbb{Z}} \mu_\ell^{(0)} \leq \mu_j^{(k)} \leq \max_{\ell\in\mathbb{Z}} \mu_\ell^{(0)} \quad \forall j \in \mathbb{Z}, k \in \mathbb{N}_0 .$$

SOLUTION of (8-7.f):

The information “compactly supported initial data u_0” hints that u_0 vanishes outside a bounded interval, that is, we have to deal with $u_0(x) = 0$.

The flux function f remains defined for the state $u = 0$ by setting $f(0) = 0$. However, for this state we encounter an “infinite speed of propagation”. As a consequence, the CFL-condition found in Subproblem (8-7.e) will enforce a vanishing timestep; the discrete evolution will stall or the discrete evolution (8.7.7) may suffer from blow-up.

SOLUTION of (8-7.g):

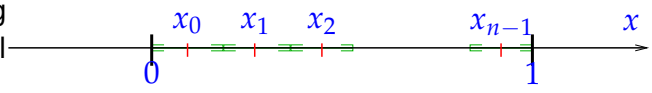
Note that the `slopes` functor yields the σ_j multiplied with the meshwidth h! In fact, the meshwidth is not even available in `slopelimfluxdiff()`.

C++11 code 8.7.11: Code for `limiterMC` ⇢ GITHUB

```
2  double limiterMC(double mu_left, double mu_center, double mu_right) {
3    double scaled_slope;
4
5    double sigma_l = 2.0 * (mu_center - mu_left);
6    double sigma_c = (mu_right - mu_left) / 2.0;
7    double sigma_r = 2.0 * (mu_right - mu_center);
8
9    std::initializer_list<double> sigma_list{sigma_l, sigma_c, sigma_r};
10
11   double min = std::min(sigma_list);
12   if (min > 0.0)
13     scaled_slope = min;
14   else {
15     double max = std::max(sigma_list);
16     if (max < 0.0)
17       scaled_slope = max;
18     else
19       scaled_slope = 0.0;
20   }
21
22   return scaled_slope;
23 }
```

HINT 1 for (8-7.h):

Mesh with n (dual) cells (draw in green) covering $]0,1[$. The nodes x_j are the centers of those dual cells.



SOLUTION of (8-7.h):

All it takes to adapt `slopelimfluxdiff` to the periodic setting is to replace μ_{-1} with μ_{n-1}, and μ_n with μ_0 whenever these dual-cell averages are referenced. This also applies to the values v_j^- and v_j^+ of the piecewise linear reconstruction at dual cell boundaries.

C++11 code 8.7.13: Implementation off `slopelimfluxdiffper()` ↪ GITHUB

```
2  // Function parameters:
3  // double h:  meshwidth of equidistant spatial grid
4  // Vector mu:  (finite) vector μ of cell averages
5  // Functor F: 2-point numerical flux function F = F(v,w)
6  // Functor slope:  slope reconstruction function
7  // hσ_j = slopes(μ_{j-1}, μ_j, μ_{j+1})
8  //
9  // returns a vector with differences of numerical fluxes
10 //
11 // Function that realizes -h· the right hand side operator
12 // L_h for the mehtod-of-lines ODE arising from conservative finite
13 // volume semidiscretization of the Cauchy problem for a 1D scalar
       conservation
14 // law [Lecture → Eq. (8.2.2.1)] in a 1-periodic setting.
15 template <typename FunctionF, typename FunctionSlopes>
16 Eigen::VectorXd slopelimfluxdiffper(const Eigen::VectorXd &mu, FunctionF &&F,
17                                     FunctionSlopes &&slopes) {
18   int n = mu.size();  // Number of active dual grid cells
19   Eigen::VectorXd sigma = Eigen::VectorXd::Zero(n);  // Vector of slopes
20   Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);     // Flux differences
21
22   // Computation of slopes σ_j, uses μ_{-1} = μ_{n-1},
23   // μ_n = μ_0, which amounts to constant extension of states
24   // beyond domain of influence [a,b] of non-constant intial data.  Same
25   // technique has been applied in [Lecture → Code 8.3.2.9]
26   sigma[0] = slopes(mu[n - 1], mu[0], mu[1]);  // @@
27   for (int j = 1; j < n - 1; ++j)
28     sigma[j] = slopes(mu[j - 1], mu[j], mu[j + 1]);
29   sigma[n - 1] = slopes(mu[n - 2], mu[n - 1], mu[0]);  // @@
30
31   // Compute linear reconstruction at endpoints of dual cells [Lecture →
       Eq. (8.5.1.2)]
32   Eigen::VectorXd nup = mu + 0.5 * sigma;
33   Eigen::VectorXd num = mu - 0.5 * sigma;
34
35   // Rely on periodicity to compute numerical fluxes at interval ends
36   fd[0] = F(nup[0], num[1]) - F(nup[n - 1], num[0]);  // @@
37   for (int j = 1; j < n - 1; ++j)
38     // see [Lecture → Eq. (8.3.2.8)]
39     fd[j] = F(nup[j], num[j + 1]) - F(nup[j - 1], num[j]);
40   fd[n - 1] = F(nup[n - 1], num[0]) -  // @@
41               F(nup[n - 2], num[n - 1]);
42   return fd;
43 }
```

The index changes are confined to Line 26, Line 29, Line 36, Line 40.

SOLUTION of (8-7.i):

Since the single-step method (8.7.14) is explicit, the implementation of the discrete evolution operator is straightforward.

C++11 code 8.7.15: Code for `sspEvolop()` ↪ GITHUB

```
2 template <typename FUNCTOR, typename State>
3 State sspEvolop(FUNCTOR &&f, State y, double tau) {
4   State y_tau;
5   State k = y + tau * f(y);
6   k = 0.75 * y + 0.25 * k + 0.25 * tau * f(k);
7   y_tau = 1.0 / 3.0 * y + 2.0 / 3.0 * k + 2.0 / 3.0 * tau * f(k);
8   return y_tau;
9 }
```

HINT 1 for (8-8.a): Integrate (8.8.1) over a cell $K \in \mathcal{M}$ and then integrate by parts using Green's formula [Lecture $\rightarrow$ Thm. 1.5.2.6]. ⌟

SOLUTION of (8-8.a):

Let us take (8.8.1) and integrate only over a cell K. Using Gauss' theorem [Lecture $\rightarrow$ Thm. 1.5.2.4], we then get

$$\int_K \frac{\partial u}{\partial t}(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} + \int_{\partial K} \boldsymbol{\beta}(\boldsymbol{x})\cdot\boldsymbol{n}_K(\boldsymbol{x})\,u(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0\,.$$

Since w_h is constant in K, we may multiply with $w_h\big|_K(\boldsymbol{x})$ and bring it under the integral. Summing over all K then yields (8.8.4).

SOLUTION of (8-8.j):

We can take the cue from [Lecture → Eq. (8.4.1.11)] and get

$$\vec{\kappa} = \boldsymbol{\mu}^{(k)} + \tau \mathbf{B}\vec{\boldsymbol{\mu}}^{(k)} \quad , \quad \vec{\boldsymbol{\mu}}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \frac{\tau}{2}\mathbf{B}\left(\boldsymbol{\mu}^{(k)} + \vec{\kappa}\right) . \tag{8.8.19}$$

SOLUTION of (8-8.k):

According to the discussion in [Lecture → § 8.4.2.10] we have to demand

$$\tau \leq m \frac{h_{\min}}{\dot{s}} ,$$

where m is the width of the "stencil", $h_{\min}$ the minimal distance of cell brycenters ($\boldsymbol{c}_j$ is the barycenter of $K_j \in \mathcal{M}$)

$$h_{\min} := \min\{\left\|\boldsymbol{c}_i - \boldsymbol{c}_j\right\| : \ K_i, K_j \in \mathcal{M} \text{ have a common edge}\} , \tag{8.8.20}$$

and $\dot{s}$ the maximal speed of propagation. Since $\|\boldsymbol{\beta}\| \leq 1$, we have $\dot{s} \leq 1$, see Sub-problem (8-8.b). Moreover, a two-stage explicit RK-SSM for timesteping means $m = 2$, because the spatial interactions encoded in $\mathbf{B}$ are only between neighboring cells. This yields the final threshold

$$\tau \leq \frac{1}{2} \cdot \frac{2h_{\min}}{\dot{s}} ,$$

where $\frac{1}{2}$ is a "safety factor", because violation of the CFL-condition is likely to lead to blow-up.

HINT 1 for (8-8.l): The function `getAdjacentCellPointers()` comes handy. ⌟

SOLUTION of (8-8.l):

Traverse all the cells, obtain pointers to its immediate neighbors from the array returned by `getAdjacent(`
compute the coordinates of the barycenters and their distance to the barycenter of the current cell. Take
the minimum over all these lengths.

C++ code 8.8.21: Sub-problem (8-8.l): function `computeHmin()` ⇝ GITHUB

SOLUTION of (8-8.m):

The implementation of (8.8.19) is straightforward after the matrix $\mathbf{B}$ has been initialized.

C++ code 8.8.22: Sub-problem (8-8.m): function `solveAdvection2D()` ⇝ GITHUB

SOLUTION of (8-8.n):

C++ code 8.8.26: Sub-problem (8-8.n): function `simulateAdvection()` ↬ GITHUB

HINT 1 for (8-8.o): See [Lecture → Ex. 3.1.4.4] to learn about the creation of sequences of meshes by regular refinement in LEHRFEM++. ⌟

SOLUTION of (8-8.o):

C++ code 8.8.27: Sub-problem (8-8.o): numerical experiment and computation of error norms. ⇝ GITHUB

◁ Doubly logarithmic plot of error norms $\|u_h(\cdot,1) - u(\cdot,1)\|_{L^2(\Omega)}$ versus the number N of mesh cells.

We observe algebraic convergence with rate 1.

SOLUTION of (8-8.p):

C++ code 8.8.28: Sub-problem (8-8.p): function `findCFLthreshold()` ⇢ GITHUB

SOLUTION of (8-8.q):

This amounts to insering a call to `findCFLthreshold()` in the loop that carries out the numerical simulation on the various refinement levels.

HINT 1 for (8-8.b): Remember that $\|\boldsymbol{\beta}(\boldsymbol{x})\| \leq 1$ for all $\boldsymbol{x}$. ⌟

HINT 2 for (8-8.b): You may quote the solution formula for the Cauchy problem for linear advection given in [Lecture → Thm. 8.1.1.8]:

$$u(\boldsymbol{x},t) = |\det(\mathrm{D}_{\boldsymbol{x}}\boldsymbol{\Phi})(\boldsymbol{x},t)|^{-1} u_0(\boldsymbol{\Phi}(\boldsymbol{x},-t)) \;, \quad (\boldsymbol{x},t) \in \mathbb{R}^d \times \mathbb{R} \;, \tag{8.8.5}$$

where $\Phi : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}^2$ is the flow induced by the velocity field $\boldsymbol{\beta}$ [Lecture → § 7.1.1.3], and $\mathrm{D}_{\boldsymbol{x}}\boldsymbol{\Phi}$ is its Jacobian. ⌟

SOLUTION of (8-8.b):

If $\boldsymbol{\Phi}$ is the flow map for $\boldsymbol{\beta}$, which satisfies

$$\frac{\partial \boldsymbol{\Phi}}{\partial t}(\boldsymbol{x},t) = \boldsymbol{\beta}(\boldsymbol{\Phi}(\boldsymbol{x},t)) \quad , \quad \boldsymbol{\Phi}(\boldsymbol{x},0) = \boldsymbol{x} \, , \quad \boldsymbol{x} \in \mathbb{R}^2 \, , \quad t \in \mathbb{R} \, , \tag{8.8.6}$$

then $t \mapsto \boldsymbol{\Phi}(\boldsymbol{x},t)$, $t \geq 0$, describes the trajectory of a "particle" with initial position $\boldsymbol{x}$. By the fundamental theorem of calculus

$$\boldsymbol{\Phi}(\boldsymbol{x},t) - \boldsymbol{x} = \int_0^t \boldsymbol{\beta}(\boldsymbol{\Phi}(\boldsymbol{x},s)) \, \mathrm{d}s \quad \overset{\|\boldsymbol{\beta}\| \leq 1}{\Longrightarrow} \quad \|\boldsymbol{\Phi}(\boldsymbol{x},t) - \boldsymbol{x}\| \leq |t| \, . \tag{8.8.7}$$

Physically speaking, this expresses that a particle with speed ≤ 1 cannot cover a distance larger than t in time t.

Thus, taking (8.8.5) for granted, it is clear that,

$$u_0(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} : \ \|\boldsymbol{x}\| > 1 \quad \Longrightarrow \quad u(\boldsymbol{x},t) = 0 \quad \forall \boldsymbol{x} : \ \|\boldsymbol{x}\| > 1 + t \, .$$

Thus, we conclude that

$$\operatorname{supp}(\{\boldsymbol{x} \mapsto u(\boldsymbol{x},t)\}) \subset \overline{B}_{1+t} := \{\boldsymbol{x} \in \mathbb{R}^2 : \ \|\boldsymbol{x}\| \leq R := 1 + t\} \, .$$

SOLUTION of (8-8.c):

If $e \subset \partial\Omega$, the second argument of F_e^K is never used in the definition (8.8.8).

SOLUTION of (8-8.d):

In (8.8.9) test with $w_h = b_h^j$, which gives, since the supports of the basis functions agree with the cells and do not overlap,

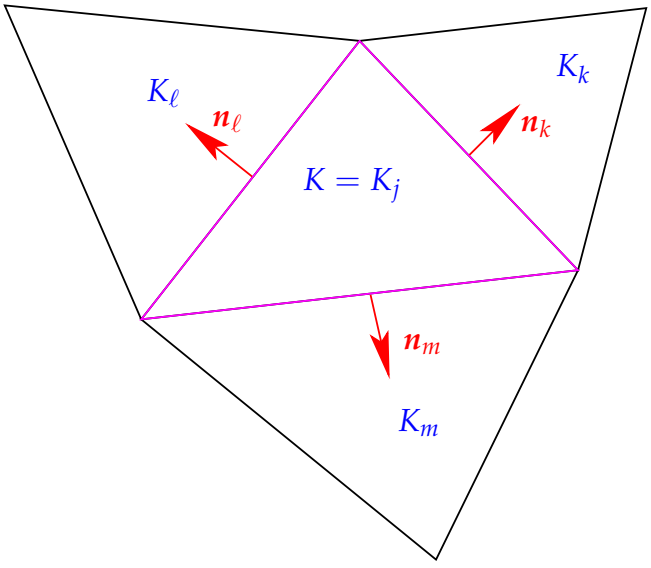
$$|K|\dot{\mu}_j(t) = -\sum_{e\in\mathcal{E},\, e\subset\partial K} |e| F_e^j(\mu_j, \mu_j^+)\,, \tag{8.8.11}$$

where F_e^j is the upwind numerical flux for cell K_j and edge e, and μ_j^+ is the value of the degree of freedom associated with the cell K^+ (which depends on e!). More explicitly, we have

$$|K|\dot{\mu}_j(t) = -\sum_{\nu\in\{\ell,m,k\}} |e_\nu| F_\nu^j(\mu_j, \mu_\nu)\,. \tag{8.8.12}$$

For the sake of compact notations we introduce the widely used symbols

$$(\xi)_+ := \begin{cases} \xi & \text{, if } \xi \geq 0\,, \\ 0 & \text{, if } \xi < 0\,, \end{cases} \quad , \quad (\xi)_- := \begin{cases} 0 & \text{, if } \xi \geq 0\,, \\ \xi & \text{, if } \xi < 0\,. \end{cases} \tag{8.8.13}$$



For $K = K_k$ we also write e_ν, $\nu \in \{\ell, m, k\}$, for the common edge of K_j and K_ν, $\boldsymbol{m}_\nu$ for the midpoint of e_ν and $\boldsymbol{n}_\nu$ for the associated unit normal vector pointing into the exterior of K.

With these notations, (8.8.12) can be converted into

$$\dot{\mu}_j(t) = -\sum_{\nu\in\{\ell,m,k\}} \frac{|e_\nu|}{|K|}\big((\boldsymbol{n}_\nu\cdot\boldsymbol{\beta}(\boldsymbol{m}_\nu))_+\mu_j + (\boldsymbol{n}_\nu\cdot\boldsymbol{\beta}(\boldsymbol{m}_\nu))_-\mu_\nu\big) = 0\,. \tag{8.8.14}$$

SOLUTION of (8-8.e):

Each cell caries a degree of freedom and can give rise to a non-zero diagonal entry of $\mathbf{B}$. From (8.8.14) we conclude that every edge contributes at most one non-zero off-diagonal entry of $\mathbf{B}$. Hence

$$\mathrm{nnz}(\mathbf{B}) \leq \sharp\mathcal{M} + \sharp\mathcal{E} \ .$$

From (8.8.14) we also learn that a row of $\mathbf{B}$ can contain at most 5 (= number of edges of a quadrilateral $+1$) non-zero entries. The same for every column of $\mathbf{B}$.

SOLUTION of (8-8.f):

There is exactly one global shape function associated with each cell.

```
lf::assemble::UniformFEDofHandler dof_handler(
      mesh_p, {{lf::base::RefEl::kPoint(), 0},
               {lf::base::RefEl::kSegment(), 0},
               {lf::base::RefEl::kTria(), 1},
               {lf::base::RefEl::kQuad(), 1}});
```

HINT 1 for (8-8.g): Writing λ_ℓ, $\ell = 1,2,3$, for the barycentric coordinates of a triangular cell K, we have the following formula for the exterior unit normals at its edges:

$$\boldsymbol{n}^j = -\frac{\mathbf{grad}\,\lambda_j}{\|\mathbf{grad}\,\lambda_j\|}\,, \tag{8.8.15}$$

see [Lecture → Fig. 74] for the numbering convention. Thus [Lecture → Code 2.4.5.11] can be used to compute the normals.

This trick can also be used for quadrilaterals after splitting them into two triangles. ⌟

SOLUTION of (8-8.g):

We implement a loop over all cells. For triangles we call `gradbarycoordinates()` from [Lecture → Code 2.4.5.11] with the vertex coordinates and normalize the columns of the matrix it gives. For quadrilaterals we do this twice for the vertices $1,2,3$ and $1,3,4$, making sure that we get the numbering right.

C++ code 8.8.16: Sub-problem (8-8.g): function `computeCellNormals()` → GITHUB

```
std::shared_ptr<
    lf::mesh::utils::CodimMeshDataSet<Eigen::Matrix<double, 2, Eigen::Dynamic>>>
computeCellNormals(std::shared_ptr<const lf::mesh::Mesh> mesh_p) {
  // Appears only in mastersolution

  // Initialize datastruture for the result
  lf::mesh::utils::CodimMeshDataSet<Eigen::Matrix<double, 2, Eigen::Dynamic>>
      result(mesh_p, 0);

  // Compute normal vectors
  for (const lf::mesh::Entity *cell : mesh_p->Entities(0)) {
    const lf::geometry::Geometry *geo_p = cell->Geometry();
    const Eigen::MatrixXd corners = lf::geometry::Corners(*geo_p);

    if (corners.cols() == 3) {
      Eigen::Matrix<double, 2, Eigen::Dynamic> normal_vectors(2, 3);

      // Compute normal vectors
      Eigen::Matrix<double, 2, 3> bary_coord = gradbarycoordinates(corners);

      // Reorder computed normal vectors
      // normal_vectors[0] -> normal_vector of edge[0]
      normal_vectors.col(0) = -(bary_coord.col(2)).normalized();
      normal_vectors.col(1) = -(bary_coord.col(0)).normalized();
      normal_vectors.col(2) = -(bary_coord.col(1)).normalized();

      // Save the matrix in our datastructure
      result(*cell) = normal_vectors;
    } else {  // corners.cols() == 4
      Eigen::Matrix<double, 2, Eigen::Dynamic> normal_vectors(2, 4);

      // Split the quadrilateral into two triangles (1,2,3) and (1,3,4)
      Eigen::Matrix<double, 2, 3> tria_1;
      Eigen::Matrix<double, 2, 3> tria_2;
      tria_1 << corners.col(0), corners.col(1), corners.col(2);
      tria_2 << corners.col(0), corners.col(2), corners.col(3);

      // Compute normal vectors
      Eigen::Matrix<double, 2, 3> bary_coord_1 = gradbarycoordinates(tria_1);
      Eigen::Matrix<double, 2, 3> bary_coord_2 = gradbarycoordinates(tria_2);

      // Reorder computed normal vectors
      // normal_vectors[0] -> normal_vector of edge[0]
      normal_vectors.col(0) = -(bary_coord_1.col(2)).normalized();
      normal_vectors.col(1) = -(bary_coord_1.col(0)).normalized();
      normal_vectors.col(2) = -(bary_coord_2.col(0)).normalized();
      normal_vectors.col(3) = -(bary_coord_2.col(1)).normalized();
```

```
49
50          // Save the matrix in our datastructure
51          result(*cell) = normal_vectors;
52        }
53      }
54      return std::make_shared<lf::mesh::utils::CodimMeshDataSet<
55          Eigen::Matrix<double, 2, Eigen::Dynamic>>>(result);
56
57  }
```

HINT 1 for (8-8.h): You may initialize an auxiliary object of type

```
lf::mesh::utils::CodimMeshDataSet<std::array<const
   lf::mesh::Entity*,2>>
```

that provides the pointers to the one or two cells adjacent to any edge. ⌟

SOLUTION of (8-8.h):

We loop over the cells and first initialize the auxiliary array recommended in the hint. The we visit the cells once more and collect the pointers from the auxiliary array.

C++ code 8.8.17: Sub-problem (8-8.h): function `getAdjacentDofIndex()` ⇢ GITHUB

HINT 1 for (8-8.i): Since from Sub-problem (8-8.e) we know reasonably sharp upper bounds for the maximal number of non-zero entries in the rows/columns of $\mathbf{B}$, we can directly initialize the sparse matrix after communicating a bound for $\mathrm{nnz}(\mathbf{B})$ via the `reserve()` member function ➙ documentation. ⌟

SOLUTION of (8-8.i):

We use the formula (8.8.14) inside a loop over the cells of the mesh.

SOLUTION of (8-9.a):

The unknowns for a conservative finite volume method are the averages over dual cells

$$\mu_j(t) \approx \frac{1}{h} \int_{x_{j-\frac{1}{2}}}^{x_{j+\frac{1}{2}}} u(x,t)\,\mathrm{d}x\,, \quad t \in [0,T]\,, \quad x_{j+\frac{1}{2}} := \frac{1}{2}(x_j + x_{j+1})\,.$$

This can be interpreted as the use of a *piecewise constant* approximation of $x \mapsto u(x,t)$ on a mesh dual to $\mathcal{M}$. In particular, we have $u(x_j,t) \approx \mu_j(t)$.

To derive the semi-discrete form we employ the considerations from [Lecture → Section 8.3.2], see [Lecture → Eq. (8.3.2.4)], [Lecture → Eq. (8.3.2.5)]. We integrate the PDE of (8.9.1) over a dual cell $[x_{j-1/2}, x_{j+1/2}]$, apply integration by parts, and replace the ambiguous boundary terms with the numerical flux:

$$\frac{d}{dt} \int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\,\mathrm{d}x + [f(u)]_{x_{j-1/2}}^{x_{j+1/2}} = \int_{x_{j-1/2}}^{x_{j+1/2}} s(u(x,t))\,\mathrm{d}x\,, \quad j \in \mathbb{Z}\,,$$

$$\blacktriangleright \quad h\frac{d\mu_j}{dt}(t) + F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t)) = hs(\mu_j(t))\,, \quad j \in \mathbb{Z}\,,$$

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big) + s(\mu_j(t))\,, \quad j \in \mathbb{Z}\,, \tag{8.9.2}$$

which is generalization of [Lecture → Eq. (8.3.2.8)].

SOLUTION of (8-9.b):

Starting from the code for `fluxdiff()`, all we habe to add is the extra term $s(\mu_j(t))$ in (8.9.2):

C++ code 8.9.3: Sub-problem (8-9.c): Implementation of `fluxdiffsource` ↬ GITHUB

```
template <typename FunctionF, typename SourceFunction>
Eigen::VectorXd fluxdiffsource(const Eigen::VectorXd &mu, FunctionF &&F,
                               SourceFunction &&s, double h) {

  unsigned n = mu.size();
  Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);

  // constant continuation to the left by mu[0]
  fd[0] = F(mu[0], mu[1]) - F(mu[0], mu[0]) - h * s(mu[0]);
  for (unsigned j = 1; j < n - 1; ++j) {
    fd[j] = F(mu[j], mu[j + 1]) - F(mu[j - 1], mu[j]) - h * s(mu[j]);
  }
  // constant continuation to the right by mu[n - 1]
  fd[n - 1] =
      F(mu[n - 1], mu[n - 1]) - F(mu[n - 2], mu[n - 1]) - h * s(mu[n - 1]);

  return fd;
}
```

HINT 1 for (8-9.c): Adapt the idea of the proof of [Lecture → Lemma 8.3.5.9]. ⌟

SOLUTION of (8-9.c):

Since $\mu_j(0)$ are cell averages of u_0, we conclude that $\mu_j(0) \geq 0$.

Let $t_0 \in [0,T]$ be the *smallest time*, when there is a $k \in \mathbb{Z}$ such that $\mu_k(t_0) = 0$. Clearly we have $\mu_j(t_0) \geq 0$ for all $j \in \mathbb{Z}$. From the semi-discrete evolution (8.9.2) we get

$$\frac{d\mu_k}{dt}(t_0) = -\frac{1}{h}\big(F(0,\mu_{k+1}(t_0)) - F(\mu_{k-1}(t_0),0)\big) + s(0)\ . \tag{8.9.4}$$

The monotonicity of the numerical flux function combined with the fact that $\mu_{k+1}(t_0) \geq 0$ and $\mu_{k-1}(t_0) \geq 0$ implies

$$\begin{aligned} \text{[decreasing in second argument]:}\quad & F(0,\mu_{k+1}(t_0)) \leq F(0,0) = f(0)\ , \\ \text{[increasing in first argument]:}\quad & F(\mu_{k-1}(t_0),0) \geq F(0,0) = f(0)\ . \end{aligned}$$

Hence, we infer from (8.9.4) and the assumption $s(0) = 0$ that

$$\frac{d\mu_k}{dt}(t_0) \geq -\frac{1}{h}\big(F(0,0) - F(0,0)\big) + 0 = 0\ .$$

This means that the value $\mu_k(t)$ cannot be decreasing at $t = t_0$. In particular, it cannot drop below zero.

SOLUTION of (8-9.d):

We have

$$f'(u) = \exp(u) - 1 \quad , \quad f''(u) = \exp(u) > 0 \, , \quad u \in \mathbb{R} \, .$$

Hence, the flux function f is *strictly convex*. We face a situation as in the case of Burgers equation and an upward jump of the data in the Riemann problem will give rise to a shock solution, the discontinuity moving with the Rankine-Hugoniot speed:

$$u(x,t) = \begin{cases} 1 & \text{for} \quad x < \dot{s}t \, , \\ -1 & \text{for} \quad x \geq \dot{s}t \, , \end{cases} \qquad \dot{s} = \frac{f(1) - f(-1)}{2} = \tfrac{1}{2}(e - e^{-1}) - 1 \, .$$

SOLUTION of (8-9.e):

Taking the cue from the insights gained for the Burgers equation we know that the entropy solution will be rarefaction wave. Its left/right edges will move with speeds

$$s_l := f'(-1) = e^{-1} - 1 < 0 \quad , \quad s_r := f'(1) = e - 1 > 0 \, ,$$

respectively. Hence, we face a transsonic rarefaction situation, *cf.* [Lecture → Ex. 8.3.4.22].

We have to recast the general formula of [Lecture → Lemma 8.2.5.10] for this special case. We have

$$g(v) := (f')^{-1}(v) = \log(1+v) \, , \quad v > -1 \, ,$$

so that

$$u(x,t) = \begin{cases} -1 & \text{for} \quad x < (e^{-1}-1)t \, , \\ \log(1+\frac{x}{t}) & \text{for} \quad (e^{-1}-1)t \le x \le (e-1)t \, , \\ 1 & \text{for} \quad x > (e-1)t \, , \end{cases} \qquad x \in \mathbb{R}, \; t > 0 \; .$$

SOLUTION of (8-9.f):

In [Lecture → § 8.3.4.32] we derived the following formulas for the Godunov numerical 2-point flux in the case of a *strictly convex* flux function f and $v < w$:

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) & \text{, if } f'(v) > 0\,, \\ f(z) & \text{, if } f'(v) < 0 < f'(w)\,, \\ f(w) & \text{, if } f'(w) < 0\,, \end{cases} \tag{8.9.9}$$

where z is the unique zero of f': $f'(z) = 0$.

In our concrete setting with $f'(u) = e^u - 1$, we have $z = 0$ and $f'(w) > 0 \quad \Leftrightarrow \quad w > 0$. Thus (8.9.9) becomes

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) & \text{, if } v > 0\,, \\ f(0) = 1 & \text{, if } v \le 0 \le w\,, \\ f(w) & \text{, if } w < 0\,, \end{cases} \tag{8.9.10}$$

In the case $v \ge w$ we learn from [Lecture → § 8.3.4.32]:

$$F_{\mathrm{GD}}(v,w) = \begin{cases} f(v) & \text{, if } \quad f(w) < f(v)\,, \\ f(w) & \text{, if } \quad f(w) \ge f(v)\,. \end{cases} \tag{8.9.11}$$

So we first test, whether $v < w$ and then implement either (8.9.10) or (8.9.11).

C++ code 8.9.12: Sub-problem (8-9.f): Implementation of `godnfn` ⇢ GITHUB

```
2  double godnfn(double v, double w) {
3    auto f = [](double u) { return std::exp(u) - u; };
4    if (v < w) {
5      if (0.0 < v)
6        return f(v);
7      if (w < 0.0)
8        return f(w);
9      return 1.0; // = f(0.0)
10   } else {
11     return f(w) < f(v) ? f(v) : f(w);
12   }
13 }
```

HINT 1 for (8-9.g):

1. As regards the CFL condition the source term is irrelevant.
2. We can expect $0 \leq u(x,t) \leq 1$ for all $(x,t) \in \mathbb{R} \times \mathbb{R}^+$.

HINT 2 for (8-9.g):

N	computed $m(3)$
10	0.00039
20	0.01353
40	0.03582
80	0.03688
160	0.04712
320	0.04696
640	0.04911
1280	0.04904

If you run the executable `ConsLawWithSource_mysolution` generated by the `make` command, it should produce the table of approximations of $m(3)$ shown beside.

⌟

SOLUTION of (8-9.g):

Since $0 \leq u(x,t) \leq 1$ for all $(x,t) \in \mathbb{R} \times \mathbb{R}^+$ the maximal speed of propagation in the concrete situation is

$$s_{\max} = \max\{|f'(0)|, |f'(1)|\} = e - 1 \approx 1.71828 \,.$$

Then we can apply the considerations of [Lecture → Section 8.4.2]. Since we rely on a two-point numerical flux and a standard conservative finite volume method we can apply [Lecture → Eq. (8.4.2.12)] with $m = 1$ and get the CFL timestep constraint

$$\tau \leq \frac{h}{e-1} \,.$$

Since $h = 15/N$ we conclude that we can still choose the timestep $\tau = \frac{15}{N(e-1)}$. Of course, the timestep should be compatible with the duration $T = 3$ of the simulation so that we should use

$$M := \lceil 3/\tau \rceil$$

timesteps, where $\lceil \xi \rceil$ stands for the smallest integer $\geq \xi$.

C++ code 8.9.14: Sub-problem (8-9.g): Implementation of `traceMass()` ➙ GITHUB

```
template <typename U0Functor>
Eigen::VectorXd traceMass(U0Functor &&u0, unsigned int N) {
  // Spacial boundaries
  double a = -5.0;
  double b = 10.0;

  // Compute μ(t = 0)
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N, a, b);
  Eigen::VectorXd mu = x.unaryExpr(std::forward<U0Functor>(u0));

  // Get timestep size and number by CLF condition
  double h = (b - a) / N;
  double s_max = std::exp(1.0) - 1.0; // since 0 ≤ u_0(x) ≤ 1
  double tau = h / s_max;             // stencil m = 1
  double T = 3.0;
  // Total number of timesteps
  int M = (int)std::ceil(T / tau);
  tau = 3.0 / M; // Timestep size

  // Compute solution and total masses at different times
  auto totalMass = [h](const Eigen::VectorXd &mu) { return (mu * h).sum(); };
  Eigen::VectorXd m(M + 1);
  for (int i = 0; i < M; ++i) {
    m(i) = totalMass(mu);
    auto s = [](double u) { return -u; };
    mu = mu + tau * (-1.0 / h) * fluxdiffsource(mu, &godnfn, s, h);
  }
  m(M) = totalMass(mu);

  return m;
}
```

SOLUTION of (8-9.h):

Since u_0 has compact support and scalar conservation laws feature a finite speed of propagation, $u(\cdot,t)$ will have compact support for all times $t \geq 0$. Integrate the conservation law in space over $\mathbb{R}$ and notice that the contribution of the term $\frac{\partial}{\partial x} f(u)$ will vanish thanks to integration by parts. We end up with

$$\frac{d}{dt}\int_{\mathbb{R}} u(x,t)\,\mathrm{d}x = \int_{\mathbb{R}} s(u(x,t))\,\mathrm{d}x \overset{(8.9.6)}{=} -\int_{\mathbb{R}} u(x,t)\,\mathrm{d}x\,.$$

Hence, the total mass solves the ODE

$$\frac{dm}{dt}(t) = -m(t) \quad \Rightarrow \quad m(t) = m(0)\exp(-t)\,, \quad t \geq 0\,.$$

Since $m(0) = 1$ we obtain $m(t) = e^{-t}$.

SOLUTION of (9-1.a):

We follow the considerations of [Lecture → Section 9.2.2]: Time is treated as a mere parameter. Since we are dealing with a Dirchlet problem (in space) with homogeneous Dirichlet boundary conditions, the trial and test space will be $H_0^1(\Omega)$. Thus the spatial variational formulation seeks a function-space-valued function of time $u : [0, T] \to H_0^1(\Omega)$ such that

$$\int_\Omega \frac{\partial u}{\partial t}(\boldsymbol{x}, t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}, t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}, t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \tag{9.1.4}$$

for all $v \in H_0^1(\Omega)$, Note that the test function does *not* depend on time, whereas the trial function does.

HINT 1 for (9-1.j): First you have to compute the increments $\vec{\kappa}_i$, $i = 1,2$ by solving (9.1.15) using EIGEN's direct sparse elimination solver. To that end you have to build the system matrix as an object of type **Eigen::SparseMatrix<double>** by using the Kronecker-product formula, see [Lecture → (9.2.7.9)]. The matrices $\mathbf{A}$ and $\mathbf{M}$ should have been initialized in the constructor.

Alternatively, you could already have initialized the two $2N \times 2N$-matrices

$$\mathbf{M}_{\otimes} := \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \mathbf{M} \quad , \quad \mathbf{A}_{\otimes} := \begin{bmatrix} \frac{5}{12} & -\frac{1}{12} \\ \frac{3}{4} & \frac{1}{4} \end{bmatrix} \otimes \mathbf{A}$$

in the constructor and rely on EIGEN's sparse matrix arithmetic to form $\widehat{\mathbf{M}} + \tau \widehat{\mathbf{A}}$ first, and then factorize it.

⌟

SOLUTION of (9-1.j):

C++11 code 9.1.23: Implementation of member function `discreteEvolutionOperator()` of Radau3MOLTimestepper. ↪ GitLab

```
Eigen::VectorXd Radau3MOLTimestepper::discreteEvolutionOperator(
    double time, double tau, const Eigen::VectorXd &mu) const {
  Eigen::VectorXd discrete_evolution_operator(dofh_.NumDofs());
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs(dofh_.NumDofs());
  LF_VERIFY_MSG(N_dofs == mu.size(),
                "Dimension mismatch between the number of degrees of freedom "
                "and the dimension of the argument vector.");

  // Building the linear system for the implicitely defined increments
  // Assembling the right hand side using block initialization
  Eigen::VectorXd linSys_rhs(2 * N_dofs);
  Eigen::VectorXd rhs_subtraction_term = A_ * mu;  // precomputation
  linSys_rhs << rhsVectorheatSource(dofh_, time + c_[0] * tau) -
                    rhs_subtraction_term,
      rhsVectorheatSource(dofh_, time + tau) - rhs_subtraction_term;

  // Implicit Runge-Kutta methods lead to systems of equations that must be
  // solved in order to obtained the increments.
  Eigen::SparseMatrix<double> linSys_mat;
  // Assembling the system right hand side matrix using the (unfortunately
  // officially not supported) Eigen Kronecker product
  linSys_mat = M_Kp_ + tau * A_Kp_;
  LF_VERIFY_MSG(linSys_mat.rows() == linSys_mat.cols(),
                "The linSys_mat Eigen matrix is not squared.");
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(linSys_mat);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");

  // Solve linear system using Eigen's sparse direct elimination
  Eigen::VectorXd k_vec = solver.solve(linSys_rhs);
  LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");

  // Compute action of the discrete evolution operator on argument vec
  discrete_evolution_operator = mu + tau * (b_[0] * k_vec.topRows(N_dofs) +
                                            b_[1] * k_vec.bottomRows(N_dofs));
  return discrete_evolution_operator;
}
```

SOLUTION of (9-1.k):

C++11 code 9.1.24: Implementation of `solveHeatEvolution()` ↪ GitLab

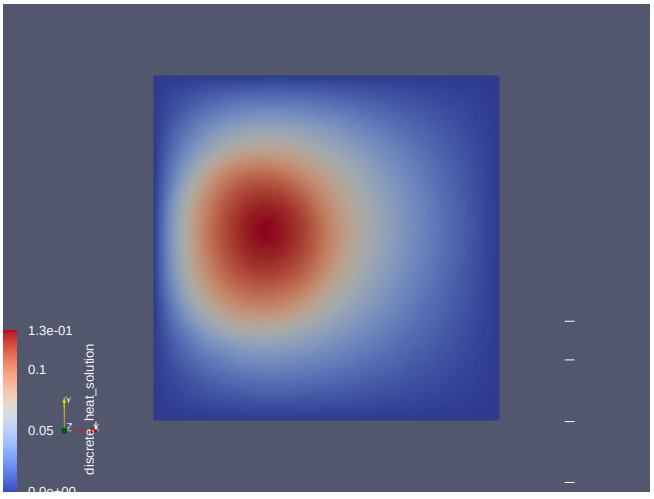
```
Eigen::VectorXd solveHeatEvolution(const lf::assemble::DofHandler &dofh,
                                   unsigned int m, double final_time) {
  Eigen::VectorXd discrete_heat_sol(dofh.NumDofs());
  double tau = final_time / m;                          // step size
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());  // dim. of FE space

  std::cout << "*********************************************************"
            << std::endl;
  std::cout << "\n>>> SolveHeatEvolution: m = " << m << ", N = " << N_dofs
            << std::endl;
  /* Setting up the problem information */
  // Precomputing the required data for the Runge-Kutta method
  // Assemble the Runge-Kutta Radau IIA 2-stages method solver (order 3)
  Radau3MOLTimestepper radau_solver(dofh);
  // Starting with the zero initial condition vector
  Eigen::VectorXd discrete_solution_cur =
      radau_solver.discreteEvolutionOperator(0.0, tau,
                                             Eigen::VectorXd::Zero(N_dofs));

  std::cout << "\n>> Iterating the action of discreteEvolutionOperator"
            << std::endl;
  /* Evolving the parabolic heat system */
  // While less elegant, we use a current and next step solution vector in the
  // iteration to stay away from potential harming aliasing effects of putting
  // an Eigen::Vector on both sides of an assignment statement.
  Eigen::VectorXd discrete_solution_next;
  for (int i = 1; i < m; i++) {
    discrete_solution_next = radau_solver.discreteEvolutionOperator(
        i * tau, tau, discrete_solution_cur);
    discrete_solution_cur = discrete_solution_next;
  }
  discrete_heat_sol = discrete_solution_cur;
  return discrete_heat_sol;
}
```

HINT 1 for (9-1.l): Study the documentation of **lf::io::VtkWriter** in order to learn how to generate data files suitable for visualization with PARAVIEW. ⌟

SOLUTION of (9-1.l):

C++11 code 9.1.25: `main()` function for simulation transient heat conduction ⇢ GitLab

```
  // Generate the linear lagrange FE data
  // Finite element space
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());

  // Solve heat evolution with zero initial and boundary conditions
  double final_time = 1.0;
  unsigned int m = 50;
  Eigen::VectorXd discrete_heat_solution =
      solveHeatEvolution(dofh, m, final_time);
  LF_ASSERT_MSG(
      discrete_heat_solution.size() == N_dofs,
      "Size of discrete solution and dimension of FE space mismatch.");

  // Output results to vtk file
  lf::io::VtkWriter vtk_writer(mesh_p, "discrete_heat_solution.vtk");
  // Write nodal data taking the values of the discrete solution at the vertices
  auto nodal_data = lf::mesh::utils::make_CodimMeshDataSet<double>(mesh_p, 2);
  for (int global_idx = 0; global_idx < N_dofs; global_idx++) {
    nodal_data->operator()(dofh.Entity(global_idx)) =
        discrete_heat_solution[global_idx];
  };
  vtk_writer.WritePointData("discrete_heat_solution", *nodal_data);
```



◁ Visualization of the finite element/2-stage Radau RK-SSM method-of-lines solution of the transient heat conduction problem (9.1.1) at $T = 1$.

Visualization with PARAVIEW

SOLUTION of (9-1.m):

In the case of a uniform timestep $\tau > 0$ the linear system of equations (9.1.15) for the increments is the same for every timestep. Thus, it can be built and LU-factorized once and the LU-factors can be reused in every timestep, which will significantly reduce the amount of work.

SOLUTION of (9-1.b):

Recall

Definition [Lecture → Def. 7.3.3.1]. General Runge-Kutta method

For coefficients $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, the discrete evolution $\Psi^{s,t}$ of an s-stage Runge-Kutta single step method (RK-SSM) for the ODE $\dot{\mathbf{u}} = \mathbf{f}(t,\mathbf{u})$, is defined by

$$\mathbf{k}_i := \mathbf{f}(t + c_i\tau, \mathbf{u} + \tau \sum_{j=1}^{s} a_{ij}\mathbf{k}_j) \;, \quad i = 1,\ldots,s \quad , \quad \Psi^{t,t+\tau}\mathbf{u} := \mathbf{u} + \tau \sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

The $\mathbf{k}_i \in V_0$ are called increments.

and from [Lecture → (7.3.3.3)] how this single-step method can compactly be described through a Butcher scheme, and apply this knowledge to $\dot{y} = -y =: f(y)$. Concretely, applying the Radau scheme with timestep size $\tau > 0$, we get the increment equations

$$\kappa_1 = -y - \tau\frac{5}{12}\kappa_1 + \tau\frac{1}{12}\kappa_2 \;, \tag{9.1.5}$$

$$\kappa_2 = -y - \tau\frac{3}{4}\kappa_1 - \tau\frac{1}{4}\kappa_2 \;. \tag{9.1.6}$$

Solving this 2×2 linear system, we get

$$\kappa_1 = \frac{-(1+\frac{1}{3}\tau)y}{(1+\tau\frac{5}{12})(1+\tau\frac{1}{4}) + \frac{\tau^2}{16}}, \qquad \kappa_2 = \frac{-(1-\frac{1}{3}\tau)y}{(1+\tau\frac{5}{12})(1+\tau\frac{1}{4}) + \frac{\tau^2}{16}} \tag{9.1.7}$$

Finally, updating the solution, we get

$$\begin{aligned} y^{(j+1)} &= y^{j} + \tau\frac{3}{4}\kappa_1 + \tau\frac{1}{4}\kappa_2 \\ &= y^{(j)}\left(1 - \frac{\tau(1+\frac{\tau}{6})}{(1+\tau\frac{5}{12})(1+\tau\frac{1}{4}) + \frac{\tau^2}{16}}\right) . \end{aligned} \tag{9.1.8}$$

The discrete evolution operator amounts to multiplication with a single number:

$$\Psi^{t,t+\tau}y = \left(1 - \frac{\tau(1+\frac{\tau}{6})}{(1+\tau\frac{5}{12})(1+\tau\frac{1}{4}) + \frac{\tau^2}{16}}\right) \cdot y \;. \tag{9.1.9}$$

Note that $\Psi^{t,t+\tau}$ depends only on τ.

Remark. Actually, what we have computed above is the stability function $S(z)$ for the 2-stage Radau RK-SSM. A general formula for the stability functions of implicit RK-SSMs given through their Buchter schemes is presented in [Lecture → (9.2.7.45b)], its derivation can be found in .

SOLUTION of (9-1.c):

We just implement (9.1.9) for $\tau = \frac{5}{m}$ and let it run for m steps computing

$$y^{(j)} := \Psi^{t_j, t_j+\tau} y^{(j-1)} , \quad t_j := \tau j , \quad j = 1, \ldots, m ,$$

where $\Psi^{t,t+\tau} \in \mathbb{R}$ is given in (9.1.9).

C++11 code 9.1.11: Implementation of `twoStageRadauTimeSteppingLinScalODE()`
➥ GitLab

```
2  std::vector<double> twoStageRadauTimesteppingLinScalODE(unsigned int m) {
3    std::vector<double> sol_vec;
4    double step_size = 5.0 / m;  // Timestep "tau"
5    sol_vec.push_back(1.0);      // Initial value
6
7    // Discrete evolution operator: For the two-stage Radau method applied to the
8    // scalar linear ODE (d/dt)y = -y, this turns out to be a scalar valued
9    // function depending on the step_size. Since we take equidistant step sizes
10   // in this example, the action of the evolution operator is simply
11   // multiplication by a constant double
12   double evolution_op =
13       (1 - (step_size * (1 + step_size / 6.0)) /
14                ((1 + step_size * 5.0 / 12.0) * (1 + step_size / 4.0) +
15                 step_size * step_size / 16.0));
16   // Compute discrete evolution by applying the evolution operator at each step
17   for (int i = 1; i < m + 1; i++) {
18     sol_vec.push_back(evolution_op * sol_vec.at(i - 1));
19   }
20   return sol_vec;
21 }
```

SOLUTION of (9-1.d):

We produce $N = 10$ solutions by solving the initial value problem (9.1.10) with $m = 2 \cdot 10^k$, $k = 0, ..., N-1$, and tabulate m versus

$$\epsilon_m := \max_{j=1,...,m} |y^{(j)} - y(\tau j)| \,, \quad \tau := \frac{5}{m} \,,$$

where $y(t) = e^{-t}$ is the exact solution of the initial-value problem (9.1.10) and $\left(y^{(j)}\right)_{j=0}^{m}$ the sequence produced by the single-step method.

C++11 code 9.1.12: Implementation of `testConvergenceTwoStageRadauLinScalODE`
➥ GitLab

```
2  void testConvergenceTwoStageRadauLinScalODE() {
3    constexpr int nIter = 10;       // total number of iterations
4    double max_norm_errors[nIter];  // errors vector for all approx. sols
5    double rates[nIter - 1];        // The rates of convergence
6    double avg_rate = 0.0;  // The average rate of convergence over all iterations
7
8    // Error between the approx solutions as given by the two stage Radau method
9    // and the exact solution vector computed from the anlytic formula vector
10   // computed from the anlytic formula
11   double diff;  // temporary variable used to compute error at various nodes
12   std::vector<double> approx_sol_vec;
13   for (int k = 0; k < nIter; k++) {
14     unsigned int m = 10 * std::pow(2, k);  // number of equidistant steps
15     double step_size = 5.0 / m;            // time step 'tau'
16     // Creating exact solution vector. This vector is created by evaluating the
17     // exact solution using the analytic formula y(t) = exp(-t) at the
18     // equidistant nodes of the time steps.
19     std::vector<double> exact_sol_vec;
20     for (int i = 0; i < m + 1; i++) {
21       exact_sol_vec.push_back(std::exp(-i * step_size));
22     }
23     // Computing approximate solution
24     approx_sol_vec = twoStageRadauTimesteppingLinScalODE(m);
25     // Computing the error in the maximum norm
26     max_norm_errors[k] = 0;
27     for (int i = 0; i < m + 1; i++) {
28       diff = std::abs(approx_sol_vec.at(i) - exact_sol_vec.at(i));
29       max_norm_errors[k] =
30           (diff > max_norm_errors[k]) ? diff : max_norm_errors[k];
31     }
32   }
33   // Computing rates of convergence
34   for (int k = 0; k < nIter - 1; k++) {
35     rates[k] = log2(max_norm_errors[k] / max_norm_errors[k + 1]);
36     avg_rate += rates[k];
37   }
38   avg_rate = avg_rate / (nIter - 1);
```

m_ℓ	error ϵ_ℓ	estimated rate
10	5.20365e-05	$*$
20	6.86977e-06	2.92119
40	8.84939e-07	2.95661
80	1.12381e-07	2.97718
160	1.41621e-08	2.98829
320	1.77756e-09	2.99407
640	2.22655e-10	2.99701
1280	2.78612e-11	2.99848
2560	3.48362e-12	2.9996
5120	4.36833e-13	2.99543

We observe that halfing the timestep τ roughly leads to a decrease of the error by a factor of 8, which is evidence for *algebraic convergence* of *order* 3. This is the order of the 2-stage Radau RK-SSM as predicted by theory.

Formula for the estimated rate α, *cf.* [Lecture → § 3.2.2.5]:

$$\alpha \approx -\frac{\log \epsilon_\ell - \log \epsilon_{\ell-1}}{\log m_\ell - \log m_{\ell-1}} \,.$$

SOLUTION of (9-1.e):

We write $\vec{\kappa}$ for the basis expansion coefficient vectors corresponding to the increments. From [Lecture → § 9.2.7.6] we know that these satisfy

$$\mathbf{M}\vec{\kappa}_1 + \tau\frac{5}{12}\mathbf{A}\vec{\kappa}_1 + \tau\frac{-1}{12}\mathbf{A}\vec{\kappa}_2 = \vec{\varphi}(t_j + \frac{1}{3}\tau) - \mathbf{A}\vec{\mu}^{(j)} \,, \tag{9.1.13}$$

$$\mathbf{M}\vec{\kappa}_2 + \tau\frac{3}{4}\mathbf{A}\vec{\kappa}_1 + \tau\frac{1}{4}\mathbf{A}\vec{\kappa}_2 = \vec{\varphi}(t_j + \tau) - \mathbf{A}\vec{\mu}^{(j)} \,, \tag{9.1.14}$$

which gives rise to the $2N \times 2N$ linear system of equations, $N := \dim \mathcal{S}^0_{1,0}(\mathcal{M})$,

$$\begin{bmatrix} \mathbf{M} + \tau\frac{5}{12}\mathbf{A} & \tau\frac{-1}{12}\mathbf{A} \\ \tau\frac{3}{4}\mathbf{A} & \mathbf{M} + \tau\frac{1}{4}\mathbf{A} \end{bmatrix} \begin{bmatrix} \vec{\kappa}_1 \\ \vec{\kappa}_2 \end{bmatrix} = \begin{bmatrix} \vec{\varphi}(t_j + \frac{1}{3}\tau) - \mathbf{A}\vec{\mu}^{(j)} \\ \vec{\varphi}(t_j + \tau) - \mathbf{A}\vec{\mu}^{(j)} \end{bmatrix} . \tag{9.1.15}$$

Remark. We can rewrite this linear system of equations using the Kronecker product of matrices as in [Lecture → § 9.2.7.6], [Lecture → (9.2.7.9)]:

$$\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \mathbf{M} + \tau \begin{bmatrix} \frac{5}{12} & -\frac{1}{12} \\ \frac{3}{4} & \frac{1}{4} \end{bmatrix} \otimes \mathbf{A} \right) \begin{bmatrix} \vec{\kappa}_1 \\ \vec{\kappa}_2 \end{bmatrix} = \begin{bmatrix} \vec{\varphi}(t_j + \frac{1}{3}\tau) - \mathbf{A}\vec{\mu}^{(j)} \\ \vec{\varphi}(t_j + \tau) - \mathbf{A}\vec{\mu}^{(j)} \end{bmatrix} . \tag{9.1.16}$$

HINT 1 for (9-1.f): Study the implementation of **LinFEElemVecProvider** ⇢ GITHUB and [Lecture → Code 2.7.4.38].

SOLUTION of (9-1.f):

Refer to the inline comments for Code 9.1.18, Code 9.1.19, and Code 9.1.20. Explanations on how to define a class providing element vectors are given in [Lecture → Ex. 2.7.4.33].

C++ code 9.1.18: Sub-problem (9-1.f): Class compatible with ELEMENT_VECTOR_PROVIDER for the computation of element vectors ⇝ GitLab

```
template <typename FUNCTOR>  // lambda predicate
class TrapRuleLinFEElemVecProvider {
 public:
  /**
   * @brief Constructor storing the right hand side function
   */
  explicit TrapRuleLinFEElemVecProvider(FUNCTOR f) : f_(f) {}

  /**
   * @brief Default implement: all cells are active
   */
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) { return true; }

  /**
   * @brief Main method for computing the element vector
   * @param cell current cell for which the element vector is desired
   * @returns The element vector corresponding to the given cell
   *
   * The implementation uses simple vertex based quadrature and an approximation
   * of the volume of a cell just using the integration element at the
   * barycenter.*/
  Eigen::Vector3d Eval(const lf::mesh::Entity &tria);

 private:
  // f_ provides the evaluation of the source function at coordinates
  FUNCTOR f_;
};
```

The key member function is `Eval()`

C++ code 9.1.19: Sub-problem (9-1.f): `Eval()` method returning element vector ⇝ GitLab

```
template <typename FUNCTOR>
Eigen::Vector3d TrapRuleLinFEElemVecProvider<FUNCTOR>::Eval(
    const lf::mesh::Entity &tria) {
  Eigen::Vector3d ElemVec;
  // Throw error in case no triangular cell
  LF_VERIFY_MSG(tria.RefEl() == lf::base::RefEl::kTria(),
                "Unsupported cell type " << tria.RefEl());
  // Obtain vertex coordinates of the triangle in a 2x3 matrix
  const auto corners{lf::geometry::Corners(*(tria.Geometry()))};
  // Compute the scaling factor for the local load vector
  const double area_third = lf::geometry::Volume(*(tria.Geometry())) / 3.0;
  LF_ASSERT_MSG((corners.cols() == 3) && (corners.rows() == 2),
                "Invalid vertex coordinate " << corners.rows() << "x"
                                             << corners.cols() << " matrix");
  ElemVec = Eigen::Vector3d(area_third * f_(corners.col(0)),
```

```
                                area_third * f_(corners.col(1)),
                                area_third * f_(corners.col(2)));
  return ElemVec;
}
```

C++ code 9.1.20: Sub-problem (9-1.f): Function `rhsVectorheatSource()` → GitLab

```
Eigen::VectorXd rhsVectorheatSource(const lf::assemble::DofHandler &dofh,
                                    double time) {
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
  // Right-hand side vector has to be set to zero initially
  Eigen::VectorXd phi(N_dofs);
  // Functor for computing the source function at 2d coordinates
  auto f = [time](Eigen::Vector2d x) -> double {
    const double PI = 3.14159265358979323846;
    Eigen::Vector2d v(std::cos(time * PI), std::sin(time * PI));
    return ((x - 0.5 * v).norm() < 0.5) ? 1.0 : 0.0;
  };
  auto mesh_p = dofh.Mesh();  // pointer to current mesh
  phi.setZero();

  /* Assembling right-hand side source vector */
  // Initialize object taking care of local computations on all cells.
  TrapRuleLinFEElemVecProvider<decltype(f)> elvec_builder(f);
  // Computing right hand side vector
  // Invoke assembly on cells (codim == 0 as first agrument)
  lf::assemble::AssembleVectorLocally(0, dofh, elvec_builder, phi);

  /* Enforce the zero Dirichlet boundary conditions */
  // Obtain an array of boolean flags for the vertices of the mesh: 'true'
  // indicates that the vertex lies on the boundary. This predicate will
  // guarantee that the computations are carried only on the boundary vertices
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 2)};

  // Assigning zero to the boundary values of phi
  for (const lf::mesh::Entity *vertex : mesh_p->Entities(2)) {
    if (bd_flags(*vertex)) {
      auto dof_idx = dofh.GlobalDofIndices(*vertex);
      LF_ASSERT_MSG(
          dofh.NumLocalDofs(*vertex) == 1,
          "Too many global indices were returned for a vertex entity!");
      phi(dof_idx[0]) = 0.0;
    }
  }
  return phi;
}
```

SOLUTION of (9-1.h):

The function `dropMatrixRowsColumns()`

(i) sets those rows and columns of the matrix **A** to zero, for which the index predicate provided by the first components of `selectvals` returns **true**.

(ii) set the diagonal entries corresponding to those rows and columns to the value 1.

HINT 1 for (9-1.i): In the constructor we should at least assemble the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrices $\mathbf{A}$ and $\mathbf{M}$. This can be done using the LEHRFEM++ library class **lf::uscalfe::ReactionDiffusionElementMatrixP** (supplied with suitable coefficient functions) together with the function `lf::assemble::assembleMat` which will yield the matrices in triplet format as objects of type **lf::assemble::COOMatrix**.

We must not forget to manipulate the **lf::assemble::COOMatrix** objects in order to enforce zero Dirichlet boundary conditions. To do so, we can use `dropMatrixRowsColumns()` from Sub-problem (9-1.h).

⌟

SOLUTION of (9-1.i):

Note that in Code 9.1.22 the class **lf::uscalfe::ReactionDiffusionElementMatrixProvider** is not used. Instead it relies on simpler types compliant with **ENTITY_MATRIX_PROVIDER**, namely **lf::uscalfe::LinearFE** and another class **LinFEMassMatrixProvider** for the computation of element mass matrices. The latter class is also implemented in `radauthreetimestepping.cc`.

C++11 code 9.1.22: Implementation of constructor of Radau3MOLTimestepper. ➔ GitLab

```
2  Radau3MOLTimestepper::Radau3MOLTimestepper(const lf::assemble::DofHandler &dofh)
3      : dofh_(dofh) {
4    std::cout << "\n>> Constructing SRadau3MOLTimestepper " << std::endl;
5    auto mesh_p = dofh.Mesh();  // pointer to current mesh
6
7    // Instantiating Galerkin matrices to be pre-computed
8    // Dimension of finite element space
9    const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
10   // Matrices in triplet format holding Galerkin matrices, zero initially.
11   lf::assemble::COOMatrix<double> A_COO(N_dofs,
12                                         N_dofs);  // element matrix Laplace
13   lf::assemble::COOMatrix<double> M_COO(N_dofs,
14                                         N_dofs);  // element mass matrix
15
16   std::cout << "> Initializing the Galerking local matrices builders"
17             << std::endl;
18   // Initialize classes containing the information required for the
19   // local computations of the Galerkin matrices. Simple implementations of
20   // LinFEMassMatrixProvider and TrapRuleLinFEElemVecProvider adapted to this
21   // particular problem was written to spare some of the overhead calculations
22   // involved in the use of the more general LehrFEM++ matrices providers.
23   lf::uscalfe::LinearFELaplaceElementMatrix elLapMat_builder;
24   LinFEMassMatrixProvider elMassMat_builder;
25
26   std::cout << "> Assembling Galerking matrices in COO format" << std::endl;
27   // Compute the Galerkin matrices
28   // Invoke assembly on cells (co-dimension = 0 as first argument)
29   // Information about the mesh and the local-to-global map is passed through
30   // a Dofhandler object, argument 'dofh'. This function call adds triplets to
31   // the internal COO-format representation of the sparse matrices A and M.
32   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elLapMat_builder, A_COO);
33   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elMassMat_builder, M_COO);
34
35   // Enforcing zero Dirichlet boundary conditions
36   // Obtain an array of boolean flags for the vertices of the mesh: 'true'
37   // indicates that the vertex lies on the boundary.
38   auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 2)};
39   // Index predicate for the selectvals FUNCTOR of dropMatrixRowsColumns
40   auto bdy_vertices_selector = [&bd_flags, &dofh](unsigned int idx) -> bool {
41     return bd_flags(dofh.Entity(idx));
42   };
43   dropMatrixRowsColumns(bdy_vertices_selector, A_COO);
44   dropMatrixRowsColumns(bdy_vertices_selector, M_COO);
45
46   std::cout << "> Converting triplets to sparse matrices" << std::endl;
47   // Creating the private Galerkin stiffness and mass matrices
48   A_ = A_COO.makeSparse();
```

```
49    Eigen::SparseMatrix<double> M = M_COO.makeSparse();
50
51    // Runge-Kutta matrices defining the 2-stage Radau timestepping. In the
52    // Butcher tableau, this corresponds to c = (1/3 1)^T (top-left column
53    // vector), b^T = (3/4 1/4) (bottom-right row vector), U_11 = 5/12, U_12 =
54    // -1/12, U_21 = 3/4, U_22 = 1/4 (top-right block); values are fixed in
55    // time
56    // clang-format off
57      U_ << 5.0/12.0, -1.0/12.0,
58                0.75,      0.25;
59      c_ << 1.0/3.0, 1.0;
60      b_ << 0.75, 0.25;
61    // clang-format on
62    // Precomputing the kronecker products involved in the implicit linear system
63    // for the increments of the RADAU-2 method
64    M_Kp_ = Eigen::kroneckerProduct(Eigen::Matrix<double, 2, 2>::Identity(), M);
65    A_Kp_ = Eigen::kroneckerProduct(U_, A_);
66  }
```

SOLUTION of (9-10.a):

As in [Lecture → § 9.3.2.13] we differentiate $t \mapsto \mathcal{E}(t)$ using the product rule for symmetric bilinear forms:

$$\frac{d\mathcal{E}}{dt}(t) = \int_\Omega \rho(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, u(\boldsymbol{x},t)\, \mathrm{d}\boldsymbol{x} + \int_\Omega \frac{\partial \mathbf{j}}{\partial t}(\boldsymbol{x},t) \cdot \mathbf{j}(\boldsymbol{x},t)\, \mathrm{d}\boldsymbol{x}\,. \tag{9.10.6}$$

Next, we consider (9.10.3) with $v = u(\cdot,t)$ and $\mathbf{q} := \mathbf{j}(\cdot,t)$, and *add both equations*. Two integrals will cancel and we end up with

$$\int_\Omega \rho(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, u(\boldsymbol{x},t)\, \mathrm{d}\boldsymbol{x} + \int_\Omega \frac{\partial \mathbf{j}}{\partial t}(\boldsymbol{x},t) \cdot \mathbf{j}(\boldsymbol{x},t)\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x},t)\, u(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \overset{f\equiv 0}{=} 0\,.$$

Matching this with (9.10.6), we conclude $\dfrac{d\mathcal{E}}{dt}(t) = 0$, which means that $t \mapsto \mathcal{E}(t)$ is a constant function.

HINT 1 for (9-10.j): Use the matrices $\widetilde{\mathbf{M}}_V$, $\widetilde{\mathbf{M}}_Q$ to compute the integrals (9.10.30) for finite-element functions. ⌟

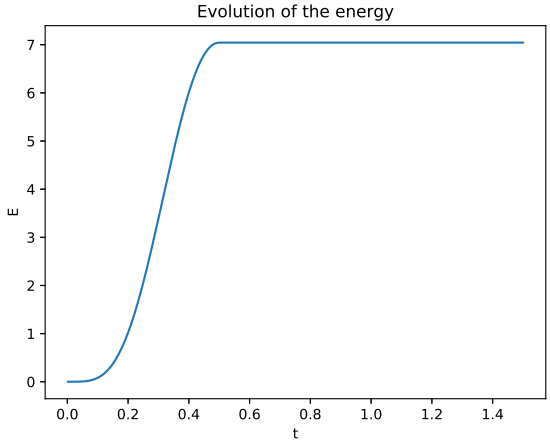
SOLUTION of (9-10.j):

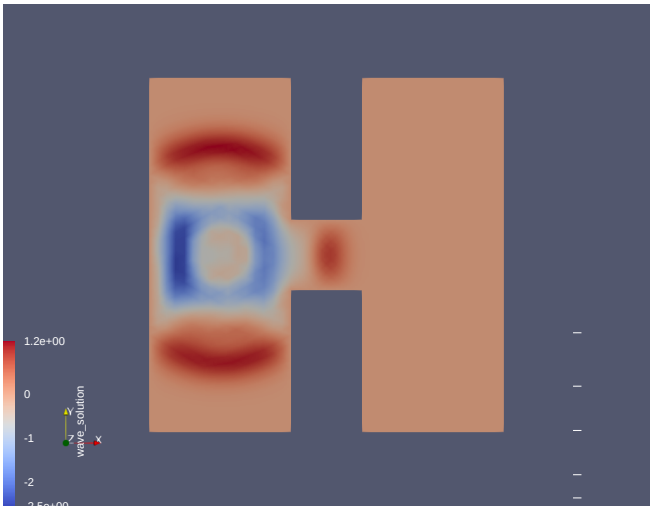
See Code 9.10.29.

Supplement. Computed total energy and solution at final time $T = 1.5$ for

$$f(\boldsymbol{x}, t) := \begin{cases} 15\sin(2\pi t) & \text{, if } \left\| \boldsymbol{x} - \begin{bmatrix} 1 \\ 2.5 \end{bmatrix} \right\| < \frac{1}{2} \quad \text{and} \quad t < \frac{1}{2} \text{ ,} \\ 0 & \text{else.} \end{cases}$$

These results are generated by invoking the main executable.





HINT 1 for (9-10.b): Eliminate **j** after differentiating the first equation of (9.10.3) with respect to time. ⌟

SOLUTION of (9-10.b):

First, we differentiate the first equation of (9.10.3) with respect to time and get

$$\int_\Omega \rho(x)\frac{\partial^2 u}{\partial t^2}(x,t)\,v(x)\,\mathrm{d}x - \int_\Omega \frac{\partial \mathbf{j}}{\partial t}(x,t)\cdot\mathbf{grad}\,v(x)\,\mathrm{d}x = \int_\Omega \frac{\partial f}{\partial t}(x,t)\,v(x)\,\mathrm{d}x \tag{9.10.8}$$

for all $v \in H_0^1(\Omega)$. We continue with the obvious fact

$$\mathbf{grad}\,H_0^1(\Omega) \subset (L^2(\Omega))^2\,,$$

which permits us to use $\mathbf{q} := \mathbf{grad}\,v$, $v \in H_0^1(\Omega)$, as test function in the second equation of (9.10.3):

$$\int_\Omega \frac{\partial \mathbf{j}}{\partial t}(x,t)\cdot\mathbf{grad}\,v(x)\,\mathrm{d}x + \int_\Omega \mathbf{grad}\,v(x)\cdot\mathbf{grad}\,u(x,t)\,\mathrm{d}x = 0 \quad \forall v \in H_0^1(\Omega)\,. \tag{9.10.9}$$

Thus we can replace the highlighted term in (9.10.8) and get

$$\int_\Omega \rho(x)\frac{\partial^2 u}{\partial t^2}(x,t)\,v(x)\,\mathrm{d}x + \int_\Omega \mathbf{grad}\,v(x)\cdot\mathbf{grad}\,u(x,t)\,\mathrm{d}x = \int_\Omega \frac{\partial f}{\partial t}(x,t)\,v(x)\,\mathrm{d}x \quad \forall v \in H_0^1(\Omega)\,,$$

which means, for $w, v \in H_0^1(\Omega)$,

$$\mathsf{m}(w,v) := \int_\Omega \rho(x)w(x)\,v(x)\,\mathrm{d}x\,, \tag{9.10.10}$$

$$\mathsf{a}(w,v) := \int_\Omega \mathbf{grad}\,w(x)\cdot\mathbf{grad}\,v(x)\,\mathrm{d}x\,, \tag{9.10.11}$$

$$\ell(t)(v) := \int_\Omega \frac{\partial f}{\partial t}(x,t)\,v(x)\,\mathrm{d}x\,. \tag{9.10.12}$$

SOLUTION of (9-10.c):

As in [Lecture → Section 2.2.2] we plug basis expansion for $u_h(t) \in V_h$ and $\mathbf{j}_h(t) \in Q_h$ into (9.10.3) and use single basis functions as test functions v and $\mathbf{q}$.

$$
\begin{aligned}
(\mathbf{M}_V)_{i,j} &= \int_\Omega \rho(\boldsymbol{x}) b_h^j(\boldsymbol{x}) b_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\,, \quad i,j = 1,\ldots,N\,,\\
(\mathbf{M}_Q)_{i,j} &= \int_\Omega \mathbf{q}_h^j(\boldsymbol{x}) \cdot \mathbf{q}_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\,, \quad i,j = 1,\ldots,M\,,\\
(\mathbf{B})_{i,j} &= \int_\Omega \mathbf{q}_h^i(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^j(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\,, \quad i = 1,\ldots,M\,, \quad j = 1,\ldots,N\,,\\
(\vec{\boldsymbol{\varphi}}(t))_j &= \int_\Omega f(\boldsymbol{x},t)\, b_h^j(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\,, \quad j = 1,\ldots,N\,.
\end{aligned}
$$

SOLUTION of (9-10.d):

The basis functions of Q_h are all associated with cells of the mesh. As we see from (9.10.15), every cell owns two basis functions:

```
lf::assemble::UniformFEDofHandler dof_handler(
mesh_p, {{lf::base::RefEl::kPoint(), 0},
  {lf::base::RefEl::kSegment(), 0},
  {lf::base::RefEl::kTria(), 2},
  {lf::base::RefEl::kQuad(), 2}});
```

HINT 1 for (9-10.e): The matrix $\mathbf{M}_Q$ has a particularly simple sparsity pattern so that the number of non-zero entries per row/column is precisely known in advance. ⌟

SOLUTION of (9-10.e):

The matrix $\mathbf{M}_Q$ is diagonal and a diagonal entry associated with cell $K \in \mathcal{M}$ has the value $|K|$ (area of K). This is an immediate consequence of the choide (9.10.15) of global shape functions for Q_h.

The implementation is straightforward using a loop over all cells of the mesh.

C++ code 9.10.17: Implementation of `computeMQ()` ➙ GitLab

```
2  Eigen::SparseMatrix<double> computeMQ(const lf::assemble::DofHandler &dofh_Q) {
3    // TOOLS AND DATA
4    // Dimension of finite element space
5    const lf::uscalfe::size_type N_dofs_Q(dofh_Q.NumDofs());
6    // Pointer to current mesh
7    std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh_Q.Mesh();
8    // Sparse matrix to be returned
9    Eigen::SparseMatrix<double> M_Q(N_dofs_Q, N_dofs_Q);
10   // ASSEMBLY
11   // First option:  assembly via triplet format
12   lf::assemble::COOMatrix<double> M_Q_COO(N_dofs_Q, N_dofs_Q);
13   // Loop over all cells
14   for (const lf::mesh::Entity *entity : mesh_p->Entities(0)) {
15     const double area = lf::geometry::Volume(*(entity->Geometry()));
16     auto global_idx = dofh_Q.GlobalDofIndices(*entity);
17     M_Q_COO.AddToEntry(global_idx[0], global_idx[0], area);
18     M_Q_COO.AddToEntry(global_idx[1], global_idx[1], area);
19   }
20   M_Q = M_Q_COO.makeSparse();
21   std::cout << "Assembly: M_Q finished" << std::endl;
22   // Allternative option:  reserve() method plus direct initialization
       of
23   // diagonal entries.
24
25   return M_Q;
26 }
```

HINT 1 for (9-10.f): The use of the 2D local trapezoidal quadrature rule boils down to the **mass lumping** approach, which leads to a particulary simple structure of $\widetilde{\mathbf{M}}_V$, see [Lecture $\rightarrow$ Rem. 9.3.4.18]. ⌟

SOLUTION of (9-10.f):

Thanks to the use of the 2D trapezoidal quadrature rule, the matrix $\widetilde{\mathbf{M}}_V$ is *diagonal* and every diagonal entry is associated with a node of the mesh and has the following value:

$$(\mathbf{M}_V)_{p,p} = \tfrac{1}{3}\rho(p) \sum_{p \in \overline{K}} |K| \,, \quad p \in \mathcal{N}(\mathcal{M}) \,. \tag{9.10.19}$$

This formula can be implemented directly. Note that we can circumvent the triplet/COO intermediate format, because the number of non-zero entries for row/column is known a priori and allows the use of EIGEN's `SparseMatrix::reserve()` method.

C++ code 9.10.20: Alternative implementation of `computeMV()` ⇝ GitLab

```
2  template <typename RHOFUNCTION>
3  Eigen::SparseMatrix<double> computeMV_alt(
4      std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space_V,
5      RHOFUNCTION &&rho) {
6    const lf::mesh::Mesh &mesh{*(fe_space_V->Mesh())};
7    const lf::assemble::DofHandler &dofh_V{fe_space_V->LocGlobMap()};
8    const lf::uscalfe::size_type N_dofs_V(dofh_V.NumDofs());
9
10   // Reserve space for a sparse diagonal matrix
11   Eigen::SparseMatrix<double> M_V(N_dofs_V, N_dofs_V);
12   M_V.reserve(Eigen::VectorXi::Constant(N_dofs_V, 1));
13
14   // Determine area of mesh neighborhood of every node
15   const auto areas{areasOfAdjacentCells(fe_space_V->Mesh())};
16   // Loop over all nodes
17   for (const lf::mesh::Entity *node : mesh.Entities(2)) {
18     // Obtain position vector of node
19     const Eigen::Vector2d node_pos{
20         lf::geometry::Corners(*(node->Geometry())).col(0)};
21     // Evaluate coefficient function at node
22     const double rho_val = rho(node_pos);
23     // Find global index of dof associated with the node
24     const std::span<const lf::assemble::gdof_idx_t> node_dofs(
25         dofh_V.GlobalDofIndices(*node));
26     LF_ASSERT_MSG(node_dofs.size() == 1, "A node has to carry exatly one dof!");
27     // Global index of tent function sitting on the node
28     const lf::assemble::gdof_idx_t dof_no = node_dofs[0];
29     M_V.insert(dof_no, dof_no) = areas(*node) * rho_val / 3.0;
30   }
31   return M_V;
32 }
```

C++ code 9.10.21: Implementation of `areasOfAdjacentCells()` that computes the sum of the areas of all cells adjacent to a node ⇝ GitLab

```
2  // Auxiliary function: Determine combined areas of cells adjacent to the nodes
3  // of a mesh
4  lf::mesh::utils::CodimMeshDataSet<double> areasOfAdjacentCells(
5      std::shared_ptr<const lf::mesh::Mesh> mesh_p) {
6    const lf::mesh::Mesh &mesh{*mesh_p};
7    // The areas are stored in a node-indexed (codim == 2) array
```

```
  lf::mesh::utils::CodimMeshDataSet<double> areas(mesh_p, 2, 0.0);
  // Loop over all cells (codim == 0 entities)
  for (const lf::mesh::Entity *cell : mesh.Entities(0)) {
    const double area = lf::geometry::Volume(*(cell->Geometry()));
    // Loop over nodes of a cell (relative codim == 2 sub-entities)
    const std::span<const lf::mesh::Entity *const> cell_nodes{
        cell->SubEntities(2)};
    for (const lf::mesh::Entity *node : cell_nodes) {
      LF_ASSERT_MSG(node->RefEl() == lf::base::RefEl::kPoint(),
                    "Illegal topological type for node");
      areas(*node) += area;
    }
  }
  return areas;
}
```

Alternatively, we may use LEHRFEM++'s cell-oriented assembly utilities to build $\mathbf{M}_V$

C++ code 9.10.22: Implementation of `computeMV()` ↪ GitLab

```
template <typename RHOFUNCTION>
Eigen::SparseMatrix<double> computeMV(
    std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space_V,
    RHOFUNCTION &&rho) {
  // TOOLS AND DATA
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_V->Mesh();
  const lf::assemble::DofHandler &dofh_V{fe_space_V->LocGlobMap()};
  const lf::uscalfe::size_type N_dofs_V(dofh_V.NumDofs());
  // For returning the matrix
  Eigen::SparseMatrix<double> M_V(N_dofs_V, N_dofs_V);
  auto zero_mf = lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> double { return 0.0; });
  auto rho_mf = lf::mesh::utils::MeshFunctionGlobal(
      [rho](Eigen::Vector2d x) -> double { return rho(x); });

  // ASSEMBLY
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> M_V_COO(N_dofs_V, N_dofs_V);
  // ReactionDiffusionElementMatrixProvider<SCALAR,DIFF_COEFF,REACTION_COEFF,
  // quadrule>
  // Tell this ENTITY_MATRIX_PROVIDER object to use the local trapezoidal rule
  // Lehrfempp requires the specification of a quadrature rule for
  // quadrilateral. A generic rule for QUAD that is NOT used in this problem is
  // passed. Alternatively : see computeMV_alt
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double, decltype(zero_mf),
                                                      decltype(rho_mf)>
      elmat_builder(fe_space_V, zero_mf, rho_mf,
                    {{lf::base::RefEl::kTria(), make_TriaQR_TrapezoidalRule()},
                     {lf::base::RefEl::kQuad(), lf::quad::make_QuadQR_P4O4()}});
  lf::assemble::AssembleMatrixLocally(0, dofh_V, dofh_V, elmat_builder,
                                      M_V_COO);
  M_V = M_V_COO.makeSparse();
  std::cout << "Assembly: M_V finished" << std::endl;

  return M_V;
}
```

Note that the problem statement demands that we use the 2D trapezoidal rule on triangles (9.10.18). This particular quadrature rule is not available in LEHRFEM++'s collection of predefined quadrature rules in `make_quad_rule.h`. Therefore we have to define it ourselves:

C++ code 9.10.23: lf::quad::QuadRule type for 2D trapezoidal rule on a triangle ➪ GitLab

```
2  // An auxiliary function providing a QuadRule object for the local
       trapezoidal
3  // rule on a triangle
4  lf::quad::QuadRule make_TriaQR_TrapezoidalRule() {
5    // A 3-point quadrature rule
6    Eigen::MatrixXd points(2, 3);
7    Eigen::VectorXd weights(3);
8    // Quadrature nodes, reference coordinates of vertices of reference
       triangle
9    points(0, 0) = 0.0;
10   points(1, 0) = 0.0;
11   points(0, 1) = 1.0;
12   points(1, 1) = 0.0;
13   points(0, 2) = 0.0;
14   points(1, 2) = 1.0;
15   // Quadrature weights on reference triangle
16   weights(0) = 0.16666666666666665741;
17   weights(1) = 0.16666666666666665741;
18   weights(2) = 0.16666666666666665741;
19   // Quadrature rule is exact for polynomials of degree 1
20   return lf::quad::QuadRule(lf::base::RefEl::kTria(), std::move(points),
21                             std::move(weights), 1);
22 }
```

HINT 1 for (9-10.g): In the file `linfelaplacelemmatprovider.cc` you find the implementation of the class

```
class LinFELaplaceElemMatProvider {
  public:
  LinFELaplaceElemMatProvider() = default;
  virtual bool isActive(const lf::mesh::Entity & /*cell*/) { return
     true; }
  Eigen::Matrix<double, 3, 3> Eval(const lf::mesh::Entity &tria);
};
```

which is designed to compute the element matrices for the bilinear form $(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}x$ and the finite-element space $\mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh.

Large parts of the code of the `Eval()` method of that class can be reused in `BElemMatProvider::Eva`

⌟

SOLUTION of (9-10.g):

We have to compute the element matrix for the bilinear form

$$(v, \mathbf{q}) \mapsto \int_\Omega \mathbf{grad}\, v(\boldsymbol{x}) \cdot \mathbf{q}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$$

on $\mathcal{S}_1^0(\mathcal{M}) \times \mathcal{Q}_h$. In light of the choice of basis functions (9.10.15), the entries of the 2×3 element matrix $\mathbf{B}_K$ for a triangular cell $K \in \mathcal{M}$ are

$$(\mathbf{B})_{i,\ell} = \int_K (\mathbf{grad}\, \lambda_\ell(\boldsymbol{x}))_i \, \mathrm{d}\boldsymbol{x} = |K|(\mathbf{grad}\, \lambda_\ell)_i\,, \quad i = 1,2\,, \quad \ell = 1,2,3\,, \tag{9.10.24}$$

where λ_1, λ_2, and λ_3 are the barycentric coordinate functions for K, whose gradients are constant. These gradients are computed in `LinFELaplaceElemMatProvider::Eval()`. The formula (9.10.24) is implemented in the following code:

C++ code 9.10.25: `Eval()` method of BElemMatProvider ⇝ GitLab

```
2  Eigen::Matrix<double, 2, 3> BElemMatProvider::Eval(
3      const lf::mesh::Entity &tria) {
4    // Obtain vertex coordinates of the triangle in a 2x3 matrix
5    const auto vertices = lf::geometry::Corners(*(tria.Geometry()));
6    LF_ASSERT_MSG((vertices.cols() == 3) && (vertices.rows() == 2),
7                  "Invalid vertex coordinate " << vertices.rows() << "x"
8                                               << vertices.cols() << " matrix");
9    // Matrix for returning result
10   Eigen::Matrix<double, 2, 3> elmat;
11   // Compute the gradients of the barycentric coordinate functions
12   // and store them in the columns of a 2x3 matrix grad_bary_coords
13   Eigen::Matrix<double, 3, 3> X;  // temporary matrix
14   X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
15   X.block<3, 2>(0, 1) = vertices.transpose();  // area of triangular cell
16   const double area = 0.5 * std::abs(X.determinant());
17   // This matrix contains the barycentric coordinate functions in its
         columns
18   auto inv = X.inverse().eval();
19   const Eigen::Matrix<double, 2, 3> grad_bary_coords{inv.block<2, 3>(1, 0)};
20   // Since the local shape function for the finite element space Q_h are
21   // Cartesian coordinate vectors, we just need to scale the components
         of the
22   // gradients of the barycentric coordinate functions with the area of
         the
23   // triangle.
24   elmat = area * grad_bary_coords;
25   return elmat;
26 }
```

HINT 1 for (9-10.h): In order to “truncate” it manipulate the matrix in triplet/COO format, as it is returned by `assembleMatrixLocally()`. Use LEHRFEM++'s utility function `flagEntitiesOnBoundary(` to determine which nodes of the mesh are located on the boundary. The method `setZero()` of **lf::assembl** may be useful. ⌟

SOLUTION of (9-10.h):

Of course, we first build the full Galerkin matrix using **BElemMatProvider** and `assembleMatrixLocall` and store in in and **lf::assemble::COOMatrix** object.

Then we obtain boundary flags for the nodes of the mesh and use `dofh_V` to map the boundary flags to the indices of the bases functions. Finally, we loop over all triplets, as obtained by calling the `lf::assembl` method, and set the value field of those, whose second index corresponds to one associated with a boundary node, to zero.

C++ code 9.10.26: Building the matrix $\widetilde{\mathbf{B}}$: `computeB` ↪ GitLab

```
2  Eigen::SparseMatrix<double> computeB(const lf::assemble::DofHandler &dofh_V,
3                                       const lf::assemble::DofHandler &dofh_Q) {
4    // TOOLS AND DATA
5    auto mesh_p = dofh_V.Mesh();
6    // Dimension of finite element space
7    const lf::uscalfe::size_type N_dofs_Q(dofh_Q.NumDofs());
8    const lf::uscalfe::size_type N_dofs_V(dofh_V.NumDofs());
9    // Sparse matrix to be returned
10   Eigen::SparseMatrix<double> B(N_dofs_Q, N_dofs_V);
11
12   // ASSEMBLY
13   lf::assemble::COOMatrix<double> B_COO(N_dofs_Q, N_dofs_V);
14   BElemMatProvider BElemMat_builder;
15   lf::assemble::AssembleMatrixLocally(0, dofh_V, dofh_Q, BElemMat_builder,
16                                       B_COO);
17
18   // Node-index array of flags marking boundary nodes
19   auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 2)};
20   // Set columns of the matrix to zero using a predicate that evaluates
       to true,
21   // if the column index is that of a d.o.f.  associated with a node on
       the
22   // boundary
23   B_COO.setZero([&bd_flags, &dofh_V](lf::assemble::gdof_idx_t /*i*/,
24                                      lf::assemble::gdof_idx_t j) -> bool {
25     return bd_flags(dofh_V.Entity(j));
26   });
27
28   // Convert from triplet to CRS format
29   B = B_COO.makeSparse();
30
31   return B;
32 }
```

SOLUTION of (9-10.i):

For the sake of efficient computation the LU-decomposition of the two (diagonal!) matrices $\mathbf{M}_V$ and $\mathbf{M}_Q$ should be done outside the timestepping loop. Alternatively the diagonal can be extracted in order to use it for componentwise division. However the sparse elimination solver of EIGEN can be expected to realize that it is supposed to solve a diagonal linear system.

In the body of the main timestepping loop (loop variable j) the vectors can be computed in the following order:

$$\text{from} \quad \vec{\mu}^{(j)}, \vec{\kappa}^{(j+\frac{1}{2})} \quad \Longrightarrow \quad \vec{\mu}^{(j+1)} \quad \Longrightarrow \quad \vec{\kappa}^{(j+\frac{3}{2})} .$$

C++ code 9.10.29: Code for `leapfrogMixedWave()` ➙ GitLab

```
template <typename RHOFUNCTION, typename FFUNCTION,
          typename RECORDER = std::function<void(double, double)>>
std::pair<Eigen::VectorXd, Eigen::VectorXd> leapfrogMixedWave(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_V,
    const lf::assemble::UniformFEDofHandler &dofh_Q, RHOFUNCTION &&rho,
    FFUNCTION &&f, double T, unsigned int nb_timesteps,
    RECORDER &&rec = [](double, double) {}) {
  // Size of timestep
  double stepsize = T / nb_timesteps;
  // Index mapper for linear Lagrangian FE space
  const lf::assemble::DofHandler &dofh_V{fe_space_V->LocGlobMap()};
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs_Q(dofh_Q.NumDofs());
  const lf::uscalfe::size_type N_dofs_V(dofh_V.NumDofs());

  // Zero INITIAL CONDITIONS
  Eigen::VectorXd mu_init = Eigen::VectorXd::Zero(N_dofs_V);
  Eigen::VectorXd kappa_init = Eigen::VectorXd::Zero(N_dofs_Q);

// PRECOMPUTATIONS: essential for efficiency
  std::cout << "Precomputing Galerkin Matrices :" << std::endl;
  // Galerkin matrix M̃_V
  Eigen::SparseMatrix<double> M_V = computeMV(fe_space_V, rho);
  std::cout << "Creating SparseLU solver for M_V: ";
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_MV;
  solver_MV.compute(M_V);
  LF_VERIFY_MSG(solver_MV.info() == Eigen::Success, "LU decomposition failed");
  std::cout << " Done" << std::endl;
  // Galerkin matrix M̃_Q
  Eigen::SparseMatrix<double> M_Q = computeMQ(dofh_Q);
  std::cout << "Creating SparseLU solver for M_Q: ";
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_MQ;
  solver_MQ.compute(M_Q);
  LF_VERIFY_MSG(solver_MQ.info() == Eigen::Success, "LU decomposition failed");
  std::cout << "Done" << std::endl;
  // Galerkin matrix B̃
  Eigen::SparseMatrix<double> B = computeB(dofh_V, dofh_Q);

  // EVOLUTION
  // Vectors for returning result
  Eigen::VectorXd mu;
```

```
43    Eigen::VectorXd kappa;
44    std::cout << "Evolution using the leapfrog timestepping scheme." << std::endl;
45    Eigen::VectorXd mu_cur = mu_init;          // μ⃗^(j)
46    Eigen::VectorXd kappa_cur = kappa_init;    // κ⃗^(j+1/2)
47    Eigen::VectorXd mu_next;                   // μ⃗^(j+1)
48    Eigen::VectorXd kappa_next;                // κ⃗^(j+3/2)
49    Eigen::VectorXd kappa_avg;
50    for (int j = 0; j < nb_timesteps; j++) {
51      // Right hand side vector φ⃗(τ(j+1/2))
52      const Eigen::VectorXd rhs{computeRHS(fe_space_V, f, stepsize * (j + 0.5))};
53      // Update of μ⃗
54      mu_next = solver_MV.solve(stepsize * (rhs + B.transpose() * kappa_cur) +
55                                M_V * mu_cur);
56      // Update of κ⃗
57      kappa_next = solver_MQ.solve(M_Q * kappa_cur - stepsize * B * mu_next);
58      // Compute approximate total energy according to (9.10.5)
59      // "Kinetic energy" at time τ(j+1)
60      const double en1 = 0.5 * mu_next.dot(M_V * mu_next);
61      // "Potential energy" at time τ(j+1)
62      kappa_avg = 0.5 * (kappa_cur + kappa_next);
63      const double en2 = 0.5 * kappa_avg.dot(M_Q * kappa_avg);
64      // Record contributions to energy, see Sub-problem (9-10.j)
65      rec(en1, en2);
66      // Prepare next timestep
67      mu_cur = mu_next;
68      kappa_cur = kappa_next;
69    }  // end timestepping loop
70    mu = mu_cur;
71    kappa = kappa_avg;
72    return {mu, kappa};
73  }
```

It is also acceptable if `kappa_cur` is returned as a basis expansion coefficient vector for a finite-element approximation of $\mathbf{j}(T)$.

SOLUTION of (9-11.a):

Essential condition are enforced on u through $u(t)|_{\partial\Omega} = g(t)\mathbf{1}$. These will give rise to Dirichlet boundary conditions in space. Otherwise the derivation of the strong form is standard and we get

$$\begin{aligned} \dot{u} - \Delta u &= 0 && \text{in } \Omega\times]0,T[\ , \\ u &= g(t)\mathbf{1} && \text{on } \partial\Omega\times]0,T[\ , \\ u &= 0 && \text{on } \Omega\times\{0\}\ , \end{aligned} \tag{9.11.2}$$

SOLUTION of (9-11.b):

$$
\begin{aligned}
(\mathbf{M}_{00})_{i,j} &= \int_\Omega b_h^j(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} && i,j \in \{1,\ldots,N_0\}\,, \\
(\mathbf{M}_{0,\partial})_{i,j} &= \int_\Omega b_h^{j+N_0}(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} && i \in \{1,\ldots,N_0\}\,, \quad j \in \{1,\ldots,N_\partial\}\,, \\
(\mathbf{A}_{00})_{i,j} &= \int_\Omega \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} && i,j \in \{1,\ldots,N_0\}\,, \\
(\mathbf{A}_{0,\partial})_{i,j} &= \int_\Omega \mathbf{grad}\, b_h^{j+N_0}(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} && i \in \{1,\ldots,N_0\}\,, \quad j \in \{1,\ldots,N_\partial\}\,, \\
\left(\vec{\boldsymbol{\beta}}_\partial\right)_j &= 1 && j \in \{1,\ldots,N_\partial\}\,.
\end{aligned}
$$

The formula for $\vec{\boldsymbol{\beta}}_\partial$ is a consequence of the cardinal basis property, because summing all basis functions will yield the constant function $\equiv 1$.

HINT 1 for (9-11.c): Of course, in a LEHRFEM++ code you must never assume that the global shape functions are ordered “interior first” as was taken for granted above. ⌟

HINT 2 for (9-11.c): You can use LEHRFEM++'s expansive finite-element assembly facilities: **lf::uscalfe:** and `lf::assemble::AssembleMatrixLocally()`. ⌟

HINT 3 for (9-11.c): Remember the method `lf::assemble::COOMatrix::setZero()`.

SOLUTION of (9-11.c):

We follow [Lecture → Ex. 2.7.4.22].

C++ code 9.11.6: Auxiliary function `initMbig()` ↬ GitLab

```
2  lf::assemble::COOMatrix<double> initMbig(
3      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space) {
4    const lf::assemble::DofHandler &dofh = fe_space->LocGlobMap();
5    // Diffusion coefficient =0, reaction coefficient = 1
6    lf::mesh::utils::MeshFunctionConstant alpha(0.0), gamma(1.0);
7    lf::uscalfe::ReactionDiffusionElementMatrixProvider entity_matrix_provider(
8        fe_space, alpha, gamma);
9    // Compute mass matrix for full finite element space
10   lf::assemble::COOMatrix<double> M =
11       lf::assemble::AssembleMatrixLocally<lf::assemble::COOMatrix<double>,
12                                           decltype(entity_matrix_provider)>(
13           0, dofh, entity_matrix_provider);
14   // Find mesh nodes on the boundary
15   const lf::mesh::utils::CodimMeshDataSet<bool> bd_flags =
16       lf::mesh::utils::flagEntitiesOnBoundary(fe_space->Mesh(), 2);
17   // Predicate for selecting matrix rows induced by test functions associated
18   // with nodes on the boundary
19   auto pred = [&bd_flags, &dofh](int i, int j) {
20     return bd_flags(dofh.Entity(i));
21   };
22   // Set the corresponding triplets to zero using LehrFEM++ helper function
23   M.setZero(pred);
24
25   return M;
26 }
```

HINT 1 for (9-11.d): The hints given for Sub-problem (9-11.c) still apply: You can again make use of **lf::uscalfe::ReactionDiffusionElementMatrixProvider** and `lf::assemble::AssembleMatrixLo`

SOLUTION of (9-11.d):

C++ code 9.11.7: Utility function `initAbig()` ↪ GitLab

```
lf::assemble::COOMatrix<double> initAbig(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space) {
  const lf::assemble::DofHandler &dofh = fe_space->LocGlobMap();
  // Diffusion coefficient =1, reaction coefficient = 0
  lf::mesh::utils::MeshFunctionConstant alpha(1.0), gamma(0.0);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider entity_matrix_provider(
      fe_space, alpha, gamma);
  // Compute Galerkin matrix for -Laplacian on full FE space
  lf::assemble::COOMatrix<double> A =
      lf::assemble::AssembleMatrixLocally<lf::assemble::COOMatrix<double>,
                                          decltype(entity_matrix_provider)>(
          0, dofh, entity_matrix_provider);
  // Find mesh nodes on the boundary
  const lf::mesh::utils::CodimMeshDataSet<bool> bd_flags =
      lf::mesh::utils::flagEntitiesOnBoundary(fe_space->Mesh(), 2);
  // Predicate for selecting matrix rows induced by test functions associated
  // with nodes on the boundary
  auto pred = [&bd_flags, &dofh](int i, int j) {
    return bd_flags(dofh.Entity(i));
  };
  // Set the corresponding triplets to zero using LehrFEM++ helper function
  A.setZero(pred);
  // Set "boundary block" to the identity matrix
  for (int i = 0; i < dofh.NumDofs(); ++i) {
    if (bd_flags(dofh.Entity(i))) A.AddToEntry(i, i, 1.0);
  }

  return A;
}
```

SOLUTION of (9-11.e):

Unraveling what is behind $\widetilde{\mathbf{M}}$ and $\widetilde{\mathbf{A}}$ we find

$$\widetilde{\mathbf{M}} + \xi \widetilde{\mathbf{A}} = \begin{bmatrix} \mathbf{M}_{00} + \xi \mathbf{A}_{00} & \mathbf{M}_{0,\partial} + \xi \mathbf{A}_{0,\partial} \\ \mathbf{O} & \xi \mathbf{I} \end{bmatrix} .$$

The matrices $\mathbf{M}_{00}$ and $\mathbf{A}_{00}$ are Galerkin matrices for the bilinear forms

$$(u,v) \mapsto \int_\Omega u\, v \,\mathrm{d}\boldsymbol{x} \quad \text{and} \quad (u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} , \quad u,v \in H^1(\Omega) ,$$

and the finite-element space $\mathcal{S}^0_{p,0}(\mathcal{M}) := \mathcal{S}^0_p(\mathcal{M}) \cap H^1_0(\Omega)$. Both are symmetric and positive definite ($\mathbf{A}_{00}$ by the arguments of [Lecture → § 1.2.3.36]), and so will be the diagonal block $\mathbf{M}_{00} + \xi \mathbf{A}_{00}$. Hence, both diagonal blocks of $\widetilde{\mathbf{M}} + \xi \widetilde{\mathbf{A}}$ are invertible, so that the whole matrix is invertible, because it is block upper triangular.

You may also argue that

$$\det(\widetilde{\mathbf{M}} + \xi \widetilde{\mathbf{A}}) = \det(\mathbf{M}_{00} + \xi \mathbf{A}_{00}) \cdot \det(\xi \mathbf{I}) .$$

SOLUTION of (9-11.f):

Remember (9.11.4): The **operator** `(double t)` returns a fixed vector $\vec{\beta}_\partial$, see (9.11.4), multiplied with $g(t)$. Hence, $\vec{\beta}_\partial$ should be precomputed in the constructor and stored in a data member of the class.

C++ code 9.11.8: Class definition of RHSProvider ⇢ GitLab

```
2  class RHSProvider {
3   public:
4    // Disabled constructors
5    RHSProvider() = delete;
6    RHSProvider(const RHSProvider &) = delete;
7    RHSProvider(RHSProvider &&) = delete;
8    RHSProvider &operator=(const RHSProvider &) = delete;
9    RHSProvider &operator=(const RHSProvider &&) = delete;
10   // Main constructor; precomputations to be done here
11   RHSProvider(const lf::assemble::DofHandler &dofh,
12               std::function<double(double)> g);
13   // Destructor
14   virtual ~RHSProvider() = default;
15   /* Class member functions */
16   // Evaluation operator for right-hand-side vector
17   Eigen::VectorXd operator()(double t) const;
18
19  private:
20   std::function<double(double)> g_;
21   Eigen::VectorXd zero_one_;
22 };
```

C++ code 9.11.9: Implementation of some methods of RHSProvider ⇢ GitLab

```
2  RHSProvider::RHSProvider(const lf::assemble::DofHandler &dofh,
3                           std::function<double(double)> g)
4      : g_(std::move(g)) {
5    // Finde nodes on the boundary
6    const lf::mesh::utils::CodimMeshDataSet<bool> bd_flags =
7        lf::mesh::utils::flagEntitiesOnBoundary(dofh.Mesh(), 2);
8    int N = dofh.NumDofs();
9    // Initialize the fixed vector, components for degrees of freedom associated
10   // with nodes on the boundary are set to 1, all other to 0
11   zero_one_ = Eigen::VectorXd(N);
12   for (int i = 0; i < N; ++i) {
13     zero_one_(i) = bd_flags(dofh.Entity(i)) ? 1.0 : 0.0;
14   }
15 }
16
17 Eigen::VectorXd RHSProvider::operator()(double t) const {
18   // Just rescale the stored vector
19   return g_(t) * zero_one_;
20 }
```

HINT 1 for (9-11.g): Consult [Lecture → § 9.2.7.6] and [Lecture → (9.2.7.7)]. ⌟

SOLUTION of (9-11.g):

Following the considerations of [Lecture → § 9.2.7.6], the increment equations for the Gauss-Lobatto IIIC implicit Runge-Kutta single step method (9.11.10) with timestep $\tau > 0$ are

$$\begin{aligned}\widetilde{\mathbf{M}}\vec{\kappa}_1 + \tfrac{1}{2}\tau\widetilde{\mathbf{A}}\vec{\kappa}_1 - \tfrac{1}{2}\tau\widetilde{\mathbf{A}}\vec{\kappa}_2 &= \vec{\varphi}(t_j) - \widetilde{\mathbf{A}}\vec{\mu}^{(j)} \ ,\\ \widetilde{\mathbf{M}}\vec{\kappa}_2 + \tfrac{1}{2}\tau\widetilde{\mathbf{A}}\vec{\kappa}_1 + \tfrac{1}{2}\tau\widetilde{\mathbf{A}}\vec{\kappa}_2 &= \vec{\varphi}(t_{j+1}) - \widetilde{\mathbf{A}}\vec{\mu}^{(j)} \ ,\end{aligned}$$

where $t_j = j\tau$ in the case of uniform timestep, $j = 0, \ldots, M-1$. Hence the increments $\vec{\kappa}_1$ and $\vec{\kappa}_2$ can be obtained as the solution of the $2N \times 2N$ linear system of equations ($N := \dim \mathcal{S}_1^0(\mathcal{M})$)

$$\begin{bmatrix} \widetilde{\mathbf{M}} + \frac{1}{2}\tau\widetilde{\mathbf{A}} & -\frac{1}{2}\tau\widetilde{\mathbf{A}} \\ \frac{1}{2}\tau\widetilde{\mathbf{A}} & \widetilde{\mathbf{M}} + \frac{1}{2}\widetilde{\mathbf{A}} \end{bmatrix} \begin{bmatrix} \vec{\kappa}_1 \\ \vec{\kappa}_2 \end{bmatrix} = \begin{bmatrix} \vec{\varphi}(t_j) - \widetilde{\mathbf{A}}\vec{\mu}^j \\ \vec{\varphi}(t_{j+1}) - \widetilde{\mathbf{A}}\vec{\mu}^j \end{bmatrix} \ . \tag{9.11.11}$$

HINT 1 for (9-11.h): Of course, the implementation of this function should rely on the functions `initMbig()` from Sub-problem (9-11.c), `initAbig()` from Sub-problem (9-11.d), and the class **RHSPr** from Sub-problem (9-11.f). ⌟

HINT 2 for (9-11.h): First build the matrix for the linear system of equations obtained in Sub-problem (9-11 in triplet/COO format using the triplet vectors for $\widetilde{\mathbf{A}}$ and $\widetilde{\widetilde{\mathbf{M}}}$. ⌟

SOLUTION of (9-11.h):

The system matrix from (9.11.11) should be constructed and *outside the main timestepping loop* using the triplet/COO format of the matrices $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{M}}$ as made available through `initMbig()` from Sub-problem (9-11.c) and `initAbig()` from Sub-problem (9-11.d). Its (sparse) LU-factorization should also be computed before entering the main timestepping loop, using EIGEN's built-in functions.

Having computed the increments by solving (9.11.11), the update equation for a single timestep is

$$\vec{\boldsymbol{\mu}}^{(j+1)} = \vec{\boldsymbol{\mu}}^{(j)} + \tfrac{1}{2}\tau\vec{\boldsymbol{\kappa}}_1 + \tfrac{1}{2}\tau\vec{\boldsymbol{\kappa}}_2 \,. \tag{9.11.12}$$

Hence, inside the body of the main timestepping loop

- we have to compute $\vec{\boldsymbol{\varphi}}(t_{j+1})$ using an **RHSProvider** object ($\vec{\boldsymbol{\varphi}}(t_j)$ can be reused),
- we have to evaluate $\widetilde{\mathbf{A}}\vec{\boldsymbol{\mu}}^{(j)}$ once,
- we have to solve the linear system of equations (9.11.11) (just backward substitution using the precomputed LU-factors),
- and have to combine the resulting increments according to (9.11.12).

C++ code 9.11.13: Implementation of the main timestepping function ↪ GitLab

```
2  template <typename GFUNCTION>
3  Eigen::VectorXd evolveIBVPGaussLobatto(
4      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space,
5      double T, unsigned int M, GFUNCTION &&g) {
6    // timestep size
7    const double tau = T / M;
8
9    const lf::assemble::DofHandler &dofh = fe_space->LocGlobMap();
10   const int N = dofh.NumDofs();
11
12   // Coefficient vector, initial value = 0
13   Eigen::VectorXd mu = Eigen::VectorXd::Zero(N);
14
15   // Build left-hand side sparse block matrix
16   lf::assemble::COOMatrix<double> lhs(2 * N, 2 * N);
17   // Arrange blocks by triplet manipulations
18   // First: Two copies of \tilde{M} on the diagonal
19   lf::assemble::COOMatrix<double> COO_M = initMbig(fe_space);
20   for (const Eigen::Triplet<double> &triplet : COO_M.triplets()) {
21     lhs.AddToEntry(triplet.row(), triplet.col(), triplet.value());
22     lhs.AddToEntry(triplet.row() + N, triplet.col() + N, triplet.value());
23   }
24   // Scaled copies of \tilde{A} added to diagonal and set as off-diagonal blocks
25   lf::assemble::COOMatrix<double> COO_A = initAbig(fe_space);
26   for (const Eigen::Triplet<double> &triplet : COO_A.triplets()) {
27     const int row = triplet.row();
28     const int col = triplet.col();
29     const double value = 0.5 * tau * triplet.value();
30     lhs.AddToEntry(row, col, value);
31     lhs.AddToEntry(row, col + N, -value);
32     lhs.AddToEntry(row + N, col, value);
33     lhs.AddToEntry(row + N, col + N, value);
34   }
```

```
35    // Use SparseLU for non-symmetric but square left-hand side matrix
36    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
37    solver.compute(lhs.makeSparse());
38
39    // Get Sparse A to compute \Blue{A\mu} below
40    Eigen::SparseMatrix<double> A = COO_A.makeSparse();
41    // Helper class for computuing source term
42    RHSProvider rhs_provider(dofh, std::forward<GFUNCTION>(g));
43
44    // Main timesteping loop : initializations
45    // Get time-dependent load vector for t=0
46    Eigen::VectorXd phi = rhs_provider(0.0);
47
48    // The actual timestepping loop
49    for (unsigned int j = 0; j < M; ++j) {
50      // Compute time-dependent right-hand side vector
51      Eigen::VectorXd rhs(2 * N);
52      Eigen::VectorXd A_mu = A * mu;
53      rhs.head(N) = phi - A_mu;
54      phi = rhs_provider(tau * (j + 1));
55      rhs.tail(N) = phi - A_mu;
56
57      // Compute increments
58      Eigen::VectorXd k = solver.solve(rhs);
59
60      // Perform a Gauss-Lobatto step to compute \Blue{\vec{\mu}^{j+1}}
61      mu = mu + 0.5 * tau * (k.head(N) + k.tail(N));
62    }
63    return mu;
64  }
```

SOLUTION of (9-11.i):

According to [Lecture → Thm. 9.2.8.5] we can expect the spatial and temporal errors to add up

$$\left(\tau \sum_{j=1}^{M_k} |u(\tau j) - u_k(\tau j)|^2_{H^1(\Omega)}\right)^{\frac{1}{2}} \leq C(h_k + \tau_k^2) \tag{9.11.14}$$

where $C > 0$ must not depend on h_k, τ_k. Thus, in order to balance both contributions we have to choose

$$\tau_k \sim h_k^{\frac{1}{2}} \ . \tag{9.11.15}$$

When we refine the mesh ($h_{k+1} = \frac{1}{2} h_k$), the timestep should be reduced by a factor of $\sqrt{2}$.

Supplement. The table lists the error (9.11.14) for a sequence of "uniform" meshes of the unit square $\Omega =]0,1[^2$ with the specified meshwidths.

The setting slightly differs from the specification of the sub-problem, since a non-zero initial value was used

$$u_0(\boldsymbol{x}) = \sin(2\pi x_1)\sin(2\pi x_2) + 1 \ , \quad \boldsymbol{x} \in \Omega \ .$$

The Dirichlet boundary data were set to $g(t) \equiv 1$. This yields the exact solution

$$u(\boldsymbol{x}, t) = e^{-4\pi^2} \sin(2\pi x_1)\sin(2\pi x_2) + 1 \ ,$$

tending to the equilibrium solution $u(\boldsymbol{x}, t) \equiv 1$.

h	error norm (9.11.14)
0.5	0.812285
0.25	0.313529
0.125	0.179342
0.0625	0.076974
0.03125	0.047418
0.015625	0.024303

We observe the expected asymptotic **algebraic convergence** of the error norm with rate ≈ 1 as $h \to 0$.

With (9.11.15) this agrees with the prediction of $O(h_k)$ for $h_k \to 0$ made in (9.11.14).

SOLUTION of (9-12.a):

Applying "reverse" integration by parts by means of Green's first formula [Lecture → Thm. 1.5.2.7] to both bilinear forms and noting that boundary terms do not arise for functions in $H_0^1(\Omega)$, we end up with the PDE

$$\operatorname{div}(\beta \,\mathbf{grad}\, \frac{\partial u}{\partial t}) + \operatorname{div}(\alpha \,\mathbf{grad}\, u) = 0 \quad \text{in} \quad \Omega \times]0,T[\ .$$

The temporal derivative can also be put in front:

$$\frac{\partial}{\partial t} \operatorname{div}(\beta \,\mathbf{grad}\, u) + \operatorname{div}(\alpha \,\mathbf{grad}\, u) = 0 \quad \text{in} \quad \Omega \times]0,T[\ .$$

These equations have to be supplemented with the boundary conditions and initial conditions

$$u = 0 \quad \text{on} \quad \partial\Omega \times]0,T[\quad , \quad u(\cdot,0) = u_0 \quad \text{in} \quad \Omega \ . \tag{9.12.3}$$

SOLUTION of (9-12.b):

The assumption on the coefficients means that $\mathsf{a} = \lambda \mathsf{b}$. Thus, (9.12.1) becomes

$$u :]0,T[\to H_0^1(\Omega): \quad \mathsf{b}(\dot{u}(t) + \lambda u(t), v) = 0 \quad \forall v \in H_0^1(\Omega) \, .$$

Since, thanks to the Poincaré-Friedrichs inequality [Lecture → Thm. 1.3.4.17], b is an inner product on $H_0^1(\Omega)$, we conclude

$$\dot{u}(\boldsymbol{x},t) + \lambda u(\boldsymbol{x},t) = 0 \, , \quad (\boldsymbol{x},t) \in \Omega \times]0,T[\, .$$

For every $\boldsymbol{x} \in \Omega$ this is a scalar linear ordinary differential equation with the solution

$$u(\boldsymbol{x},t) = e^{-\lambda t} u_0(\boldsymbol{x}) \, , \quad (\boldsymbol{x},t) \in \Omega \times]0,T[\, .$$

HINT 1 for (9-12.c): [Lecture → Lemma 9.2.3.8] might be useful. ⌟

SOLUTION of (9-12.c):

The evolution problem (9.12.1) is simply an abstract parabolic evolution problem of the form [Lecture → (9.2.2.9)]. Therefore the abstract estimates of [Lecture → Section 9.2.3] apply, in particular

Lemma [Lecture → Lemma 9.2.3.8]. Decay of solutions of parabolic evolutions

For $f \equiv 0$ the solution $u(t)$ of [Lecture → (9.2.2.4)] satisfies

$$\|u(t)\|_m \leq e^{-\gamma t}\|u_0\|_m \quad , \quad \|u(t)\|_a \leq e^{-\gamma t}\|u_0\|_a \quad \forall t \in]0,T[\, ,$$

where $\gamma > 0$ is the constant from [Lecture → (9.2.3.6)], and $\|\cdot\|_a$, $\|\cdot\|_m$ stand for the energy norms induced by $\mathsf{a}(\cdot,\cdot)$ and $\mathsf{m}(\cdot,\cdot)$, respectively.

when we replace $\|\cdot\|_m$ with $\|\cdot\|_b$.

We still have to determine the constant $\gamma > 0$, which is the largest number such that

$$\mathsf{a}(v,v) = \int_\Omega \alpha(\boldsymbol{x})\|\mathbf{grad}\, v(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \geq \gamma \mathsf{b}(v,v) = \int_\Omega \beta(\boldsymbol{x})\|\mathbf{grad}\, v(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \, , \quad v \in H^1_0(\Omega) \tag{9.12.4}$$

Clearly, choosing

$$\gamma := \min_{\boldsymbol{x}\in\Omega} \frac{\alpha(\boldsymbol{x})}{\beta(\boldsymbol{x})} \, , \tag{9.12.5}$$

will make (9.12.4) hold and this is also the largest possible value, as we can see by plugging in some function support in a neighborhood of that point where the ration (9.12.5) attains its minimum.

Summing up, we have

$$\|u(t)\|_b \leq e^{-\gamma t}\|u_0\|_b \quad , \quad \|u(t)\|_a \leq e^{-\gamma t}\|u_0\|_a \, ,$$

with γ from (9.12.5). Also acceptable would be cruder bound

$$\gamma := \min_{\boldsymbol{x}\in\Omega} \alpha(\boldsymbol{x}) : \max_{\boldsymbol{x}\in\Omega} \beta(\boldsymbol{x}) \, .$$

SOLUTION of (9-12.d):

The relationship $\alpha = \lambda\beta$ implies $\mathbf{A} = \lambda\mathbf{B}$. Since $\mathbf{A}$ is invertible, (9.12.6) becomes

$$\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \lambda\vec{\boldsymbol{\mu}}(t) = 0 \,,$$

which amounts to N decoupled scalar linear autonomous ordinary differential equations.

From [Lecture → Thm. 7.3.4.4] we known that a RK-SSM with stability function $z \mapsto S(z)$, when applied to the initial-value problem

$$\dot{y} = \lambda y \quad , \quad y(0) = y_0$$

with constant timestep $\tau > 0$ produces the sequence

$$\left(y^{(k)}\right)_{k=0}^{M} , \quad y^{(k)} := S(\lambda\tau)^k y_0 \,.$$

Hence, in the current setting we have

$$\vec{\boldsymbol{\mu}}^{(j)} = S(-\lambda\tau)^j \vec{\boldsymbol{\mu}}_0 \,.$$

This sequence will remain bounded, if $-\lambda\tau \in \mathcal{S}_\Psi$, where is the region of stability of the RK-SSM according to [Lecture → Def. 7.1.0.51]. In particular we observe that this timestep constraint does not depend on choice of $V_{0,h}$ at all!

HINT 1 for (9-12.e): The file `sobolevevolutionproblem.h` contains the useful **auxiliary function**

```
template <typename SCALAR, typename MESHFUNCTION>
  Eigen::SparseMatrix<SCALAR> getFEMatrixDirichlet(
    std::shared_ptr<const lf::fe::ScalarFESpace<SCALAR>> fe_space_p,
    const MESHFUNCTION &mf_coeff);
```

When passed a pointer to an object describing a Lagrangian finite-element space $V_h \subset H^1(\Omega)$ of dimension $K \in \mathbb{N}$ and with basis $\{b_h^1, \ldots, b_h^K\}$ in `fe_space_p` and a coefficient function $\rho \in C^0(\overline{\Omega})$ as a **MeshFunction**-compatible object in `mf_coeff`, it computes a Galerkin matrix $\mathbf{G} \in \mathbb{R}^{K,K}$ aware of zero Dirichlet boundary conditions according to

$$(\mathbf{G})_{i,j} = \begin{cases} T_{i,j} & \text{, if } b_h^i \textit{ and } b_h^j \text{ located in the interior of } \Omega\,, \\ 1 & \text{, if } i=j \text{ and } b_h^i \text{ is located on } \partial\Omega\,, \\ 0 & \text{, if } i=j, \text{ and } b_h^i \textit{ or } b_h^j \text{ located on } \partial\Omega\,, \end{cases} \qquad 1 \le i,j \le K\,, \tag{9.12.7}$$

$$\text{with} \quad T_{i,j} := \int_\Omega \rho(\boldsymbol{x})\, \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,.$$

Here we have labeled a basis function "located on $\partial\Omega$" or "located in the interior of Ω", if it is associated with a mesh entity contained in $\partial\Omega$ or the interior of Ω, respectively. ⌟

SOLUTION of (9-12.e):

The ODE of (9.12.6) can be recast into the standard form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ of an autonomous ODE in $\mathbb{R}^N$ with

$$\mathbf{y} \quad \leftrightarrow \quad \vec{\boldsymbol{\mu}} \quad , \quad \mathbf{f}(\mathbf{y}) \quad \leftrightarrow \quad -\mathbf{B}^{-1}\mathbf{A}\vec{\boldsymbol{\mu}} \, .$$

Hence, the concrete incarnation of

Definition [Lecture → Def. 6.4.0.9]. Explicit Runge-Kutta single-step method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{i-1} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an **s-stage explicit Runge-Kutta single step method** (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, $\mathbf{f} : \Omega \to \mathbb{R}^N$, is defined by ($\mathbf{y}_0 \in D$)

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{i-1} a_{ij}\mathbf{k}_j) \, , \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i \, .$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$, $i = 1,\ldots,s$, are called **increments**, $h > 0$ is the size of the timestep.

for (9.12.6) reads, when replacing $\mathbf{k}_i \leftrightarrow \vec{\boldsymbol{\kappa}}_i$, $h \leftrightarrow \tau$,

$$\begin{aligned}
\vec{\boldsymbol{\kappa}}_1 &:= -\mathbf{B}^{-1}\mathbf{A}\vec{\boldsymbol{\mu}}_0 \, , \\
\vec{\boldsymbol{\kappa}}_2 &:= -\mathbf{B}^{-1}\mathbf{A}(\vec{\boldsymbol{\mu}}_0 + \tau a_{21}\vec{\boldsymbol{\kappa}}_1) \, , \\
&\vdots \qquad\qquad \vdots \\
\vec{\boldsymbol{\kappa}}_s &:= -\mathbf{B}^{-1}\mathbf{A}(\vec{\boldsymbol{\mu}}_0 + \tau a_{s1}\vec{\boldsymbol{\kappa}}_1 + \cdots + \tau a_{s,s-1}\vec{\boldsymbol{\kappa}}_{s-1}) \, , \\
\vec{\boldsymbol{\mu}}_1 &:= \vec{\boldsymbol{\mu}}_0 + \tau b_1\vec{\boldsymbol{\kappa}}_1 + \cdots + \tau b_s\vec{\boldsymbol{\kappa}}_s \, .
\end{aligned}$$

C++ code 9.12.8: Implementation of `(solveRKSobEv)` ➙ GitLab

```
2  template <typename MESHFUNCTION_BETA, typename MESHFUNCTION_ALPHA>
3  Eigen::VectorXd solveRKSobEvl(
4      std::shared_ptr<const lf::fe::ScalarFESpace<double>> fe_space_p,
5      const MESHFUNCTION_BETA &beta, const MESHFUNCTION_ALPHA &alpha,
6      const Eigen::VectorXd &mu0, double T, const Eigen::MatrixXd &RK_Mat,
7      const Eigen::VectorXd &b, unsigned int M) {
8    LF_ASSERT_MSG(mu0.size() == (fe_space_p->LocGlobMap()).NumDofs(),
9                  "Wrong length of coefficient vector");
10   // Build Galerkin matrices taking into account homogeneous Dirichlet
       boundary
11   // conditions
12   Eigen::SparseMatrix<double> B =
13       getFEMatrixDirichlet<double, MESHFUNCTION_BETA>(fe_space_p, beta);
14   Eigen::SparseMatrix<double> A =
15       getFEMatrixDirichlet<double, MESHFUNCTION_ALPHA>(fe_space_p, alpha);
16   Eigen::VectorXd muj(mu0);  // current state vector
17   // LU-decompose B outside timestepping loop, which is important for
       efficient
18   // implementation
19   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
20   solver.compute(B);
21   LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
22
```

```
  int s = RK_Mat.cols();  // Number of stages
  LF_ASSERT_MSG(s == RK_Mat.rows(), "Butcher matrix must be square!");
  LF_ASSERT_MSG(s == b.size(), "s weights required!");
  // timestep size
  double tau = T / M;
  // Main timestepping loop
  Eigen::VectorXd mu_next(mu0.size());            // next state vector
  Eigen::VectorXd tmp(mu0.size());                // Summation vector
  std::vector<Eigen::VectorXd> incs(s, mu_next);  // RK increments
  // The r.h.s.  vector field for MOL ODE in standard form
  // Invoking solver.solve() amounts to the application of the
  // inverse of the matrix B.
  auto Vf = [&solver, &A](const Eigen::VectorXd &y) -> Eigen::VectorXd {
    const Eigen::VectorXd fy = -solver.solve(A * y);
    LF_VERIFY_MSG(solver.info() == Eigen::Success, "Solving LSE failed");
    return fy;
  };
  for (int k = 0; k < M; ++k) {
    // First increment vector
    incs[0] = Vf(muj);
    // Compute increments successively, which is possible for an
       explicit RK-SSM
    // Simultaneous assemble next state vector
    mu_next = muj + (tau * b[0]) * incs[0];
    for (int i = 1; i < s; ++i) {
      // Weighted sum of already computed increments
      tmp.setZero();
      for (int j = 0; j < i; ++j) {
        tmp += RK_Mat(i, j) * incs[j];
      }
      incs[i] = Vf(muj + tau * tmp);
      // Update state by weighed sum of increments
      mu_next += (tau * b[i]) * incs[i];
    }
    // Efficient way to set current state to new state
    std::swap(muj, mu_next);
  }
  return muj;
}
```

SOLUTION of (9-13.a):

The approach is standard and discussed in [Lecture → Section 9.2.2], see [Lecture → § 9.2.2.2].

As usual, we start by multiplying by a test function and integrating over the domain

$$\int_{-1}^{1} \frac{\partial^2 u(x,t)}{\partial t^2} w(x)\,\mathrm{d}x - \int_{-1}^{1} \frac{\partial}{\partial x}\left(\gamma(x)\frac{\partial u(x,t)}{\partial x}\right) w(x)\,\mathrm{d}x = 0, \quad \forall w \in H^1(]-1,1[)\ . \tag{9.13.3}$$

Then use integration by parts [Lecture → § 1.5.1.5] in the second term, which introduces "boundary contributions",

$$\int_{-1}^{1} \frac{\partial^2 u(x,t)}{\partial t^2} w(x)\,\mathrm{d}x + \int_{-1}^{1} \left(\gamma(x)\frac{\partial u(x,t)}{\partial x}\right) \frac{\partial w(x)}{\partial x}\,\mathrm{d}x - \left[\gamma(x)\frac{\partial u(x,t)}{\partial x} w(x)\right]_{x=-1}^{1} = 0\ , \tag{9.13.4}$$

$\forall v \in H^1(]-1,1[)$. Since the last term vanishes due to the homogeneous Neumann boundary conditions (9.13.2), we finally obtain: seek $u :]0,T[\to H^1(]-1,1[)$

$$\int_{-1}^{1} \frac{\partial^2 u(x,t)}{\partial t^2} w(x)\,\mathrm{d}x + \int_{-1}^{1} \left(\gamma(x)\frac{\partial u(x,t)}{\partial x}\right) \frac{\partial w(x)}{\partial x}\,\mathrm{d}x = 0, \quad \forall w \in H^1(]-1,1[)\ . \tag{9.13.5}$$

SOLUTION of (9-13.j):

We start from (9.13.19)

$$\begin{aligned} \mathbf{M}_1^1 \delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} &+ \mathbf{M}_\sigma^1 \frac{\vec{\boldsymbol{\mu}}^{(k+1)} + \vec{\boldsymbol{\mu}}^{(k)}}{2} &+ \mathbf{B}_1 \frac{\vec{\boldsymbol{\nu}}^{(k+1)} + \vec{\boldsymbol{\nu}}^{(k)}}{2} &= \vec{\boldsymbol{\varphi}}_0 \,, \\ \mathbf{M}_1^0 \delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} &+ \mathbf{M}_\sigma^0 \frac{\vec{\boldsymbol{\nu}}^{(k+1)} + \vec{\boldsymbol{\nu}}^{(k)}}{2} &- \mathbf{B}_1^\top \frac{\vec{\boldsymbol{\mu}}^{(k+1)} + \vec{\boldsymbol{\mu}}^{(k)}}{2} &= 0 \,, \end{aligned} \tag{9.13.31}$$

and the same equations for the previous timestep

$$\begin{aligned} \mathbf{M}_1^1 \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})} &+ \mathbf{M}_\sigma^1 \frac{\vec{\boldsymbol{\mu}}^{(k)} + \vec{\boldsymbol{\mu}}^{(k-1)}}{2} &+ \mathbf{B}_1 \frac{\vec{\boldsymbol{\nu}}^{(k)} + \vec{\boldsymbol{\nu}}^{(k-1)}}{2} &= \vec{\boldsymbol{\varphi}}_0 \,, \\ \mathbf{M}_1^0 \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})} &+ \mathbf{M}_\sigma^0 \frac{\vec{\boldsymbol{\nu}}^{(k)} + \vec{\boldsymbol{\nu}}^{(k-1)}}{2} &- \mathbf{B}_1^\top \frac{\vec{\boldsymbol{\mu}}^{(k)} + \vec{\boldsymbol{\mu}}^{(k-1)}}{2} &= 0 \,. \end{aligned} \tag{9.13.32}$$

We subtract corresponding equations:

$$\begin{aligned} \mathbf{M}_1^1 \left(\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} - \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}\right) &+ \mathbf{M}_\sigma^1 \frac{\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}}{2} &+ \mathbf{B}_1 \frac{\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}}{2} &= \vec{\boldsymbol{\varphi}}_0 \,, \\ \mathbf{M}_1^0 \left(\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} - \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}\right) &+ \mathbf{M}_\sigma^0 \frac{\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}}{2} &- \mathbf{B}_1^\top \frac{\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}}{2} &= 0 \,. \end{aligned}$$

We multiply the first equation from the left with $\left(\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}\right)^\top$ and the second equation from the left with $\left(\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}\right)^\top$, add the resulting equations and observe the cancellation of all terms involving the matrix $\mathbf{B}_1$.

$$\begin{aligned} &\left(\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}\right)^\top \mathbf{M}_1^1 \left(\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} - \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}\right) \\ &\qquad + \left(\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}\right)^\top \mathbf{M}_1^0 \left(\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} - \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}\right) \\ &\qquad = -\left(\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}\right)^\top \mathbf{M}_\sigma^1 \frac{\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}}{2} \\ &\qquad\qquad - \left(\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}\right)^\top \mathbf{M}_\sigma^0 \frac{\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}}{2} \end{aligned}$$

For any *symmetric* matrix $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, we have the "binomial formula"

$$\left(\vec{\boldsymbol{\xi}} + \vec{\boldsymbol{\eta}}\right) \mathbf{T} \left(\vec{\boldsymbol{\xi}} - \vec{\boldsymbol{\eta}}\right) = \vec{\boldsymbol{\xi}}^\top \mathbf{T} \vec{\boldsymbol{\xi}} - \vec{\boldsymbol{\eta}}^\top \mathbf{T} \vec{\boldsymbol{\eta}} \quad \forall \vec{\boldsymbol{\xi}}, \vec{\boldsymbol{\eta}} \in \mathbb{R}^n \,.$$

Since $\mathbf{M}_1^1 = \operatorname{diag}(\frac{1}{2}, 1, \ldots, 1, \frac{1}{2})$ and $\mathbf{M}_1^0 = h\mathbf{I}$ this lets us conclude

$$\begin{aligned} &E_k - E_{k-1} = \\ &- \left(\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}\right)^\top \mathbf{M}_\sigma^1 \frac{\delta\vec{\boldsymbol{\mu}}^{(k+\frac{1}{2})} + \vec{\boldsymbol{\mu}}^{(k-\frac{1}{2})}}{2} + \left(\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} - \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}\right)^\top \mathbf{M}_\sigma^0 \frac{\delta\vec{\boldsymbol{\nu}}^{(k+\frac{1}{2})} + \delta\vec{\boldsymbol{\nu}}^{(k-\frac{1}{2})}}{2} \end{aligned}$$

≤ 0 ,

because the matrices $\mathbf{M}_\sigma^1$ and $\mathbf{M}_\sigma^0$ are symmetric and positive definite (s.p.d.).

Remark. The sequence $(E_k)_k$ cannot grow.

SOLUTION of (9-13.b):

We set $L = 0$ in (9.13.6) and get.

$$\int_{-1}^{1} \frac{\partial u}{\partial t}\, w \, \mathrm{d}x + \int_{-1}^{1} \sigma(x) u \, w \, \mathrm{d}x + \int_{-1}^{1} v \, \frac{\partial w}{\partial x} \, \mathrm{d}x = \int_{-1}^{1} v_0 \, w \, \mathrm{d}x,$$
$$\int_{-1}^{1} \frac{\partial v}{\partial t}\, q \, \mathrm{d}x + \int_{-1}^{1} \sigma(x) v \, q \, \mathrm{d}x - \int_{-1}^{1} \gamma(x) \frac{\partial u}{\partial x}\, q \, \mathrm{d}x = 0 \,.$$

Taking into account that $\sigma(x) = 0$ for $-1 \le x \le 1$ this boils down to

$$\int_{-1}^{1} \frac{\partial u}{\partial t}\, w \, \mathrm{d}x + \int_{-1}^{1} v \, \frac{\partial w}{\partial x} \, \mathrm{d}x = \int_{-1}^{1} v_0 \, w \, \mathrm{d}x, \tag{9.13.7a}$$
$$\int_{-1}^{1} \frac{\partial v}{\partial t}\, q \, \mathrm{d}x - \int_{-1}^{1} \gamma(x) \frac{\partial u}{\partial x}\, q \, \mathrm{d}x = 0 \tag{9.13.7b}$$

for all $w \in H^1(]-1,1[)$ and $q \in L^2(]-1,1[)$.

Remark. The variational problem (9.13.7) corresponds to the following PDE in strong form

$$\frac{\partial u}{\partial t} - \frac{\partial v}{\partial x} = v_0 \,, \quad \frac{\partial v}{\partial t} - \gamma(x) \frac{\partial u}{\partial x} = 0 \quad \text{in} \quad]-1,1[\,. \tag{9.13.8}$$

More generally, the variational equations (9.13.6) encode the following first-order PDE system in space and time

$$\begin{array}{rcrcl} \dfrac{\partial u}{\partial t} & + & \sigma(x) u & - & \dfrac{\partial v}{\partial x} \\ \dfrac{\partial v}{\partial t} & + & \sigma(x) v & - & \gamma(x) \dfrac{\partial u}{\partial x} \end{array} \begin{array}{c} = \\ = \end{array} \begin{array}{l} v_0 \,, \\ 0 \end{array} \quad \text{on} \quad]-1-L, 1+L[\times]0,T[\,. \tag{9.13.9}$$

HINT 1 for (9-13.c): Test the right equation in (9.13.10) with a function in $L^2(]-1,1[)$. ⌟

SOLUTION of (9-13.c):

Following the hint, we begin with

$$\int_{-1}^{1} \frac{\partial v}{\partial t} q \, \mathrm{d}x - \int_{-1}^{1} \gamma(x) \frac{\partial u}{\partial x} q \, \mathrm{d}x = 0, \quad \forall q \in L^2(]-1,1[) \tag{9.13.11}$$

and get (9.13.7b). Consider now the variational formulation (9.13.5) and integrate over time

$$\int_{-1}^{1} \frac{\partial u(x,t)}{\partial t} w(x) \, \mathrm{d}x - \int_{-1}^{1} v_0(x) w(x) \, \mathrm{d}x + \int_{-1}^{1} v(x,t) \frac{\partial w(x)}{\partial x} \, \mathrm{d}x = 0 \tag{9.13.12}$$

for all $v \in H^1(]-1,1[)$, which yields (9.13.7a), because

$$\int_0^t \frac{\partial^2 u}{\partial t^2}(x,\tau) \, \mathrm{d}\tau = \frac{\partial u}{\partial t}(x,t) - \frac{\partial u}{\partial t}(x,0) = \frac{\partial u}{\partial t}(x,t) - v_0(x) \; .$$

HINT 1 for (9-13.d): Differentiate both equations of (9.13.6) with respect to time t and then choose suitable test functions. ⌟

SOLUTION of (9-13.d):

As recommended in the hint we differentiate both equations of (9.13.6) with respect to time t and then

- test the first equation with $w(x) := \frac{\partial u}{\partial t}(x,t)$,
- and the second equation with $q(x) := \frac{1}{\gamma(x)}\frac{\partial v}{\partial t}(x,t)$,

for fixed t, which yields

$$\begin{aligned}\int_{-1-L}^{1+L} \frac{\partial^2 u}{\partial t^2}(x,t)\,\frac{\partial u}{\partial t}(x,t)\,\mathrm{d}x \quad &+ \int_{-1-L}^{1+L} \sigma(x)\frac{\partial u}{\partial t}(x,t)\,\frac{\partial u}{\partial t}(x,t)\,\mathrm{d}x \\ &+ \int_{-1-L}^{1+L} \frac{\partial v}{\partial t}(x,t)\,\frac{\partial^2 u}{\partial x \partial t}(x,t)\,\mathrm{d}x = 0\,, \\ \int_{-1-L}^{1+L} \frac{\partial^2 v}{\partial t^2}(x,t)\,\frac{1}{\gamma(x)}\frac{\partial v}{\partial t}(x,t)\,\mathrm{d}x \quad &+ \int_{-1-L}^{1+L} \sigma(x)\frac{\partial v}{\partial t}(x,t)\,\frac{1}{\gamma(x)}\frac{\partial v}{\partial t}(x,t)\,\mathrm{d}x \\ &- \int_{-1-L}^{1+L} \frac{\partial^2 u}{\partial x \partial t}(x,t)\,\frac{\partial v}{\partial t}(x,t)\,\mathrm{d}x = 0\,.\end{aligned} \tag{9.13.14}$$

Adding both equations the highlighted terms cancels and this yields

$$\begin{aligned}\int_{-1-L}^{1+L} \frac{\partial^2 u}{\partial t^2}(x,t)\,\frac{\partial u}{\partial t}(x,t)\,\mathrm{d}x + \int_{-1-L}^{1+L} \sigma(x)\frac{\partial u}{\partial t}(x,t)\,\frac{\partial u}{\partial t}(x,t)\,\mathrm{d}x + \int_{-1-L}^{1+L} \frac{\partial^2 v}{\partial t^2}(x,t)\,\frac{1}{\gamma(x)}\frac{\partial v}{\partial t}(x,t)\,\mathrm{d}x \\ + \int_{-1-L}^{1+L} \sigma(x)\frac{\partial v}{\partial t}(x,t)\,\frac{1}{\gamma(x)}\frac{\partial v}{\partial t}(x,t)\,\mathrm{d}x = 0\,.\end{aligned} \tag{9.13.15}$$

Compare this with the temporal derivative of the energy function obtained by applying the chain rule,

$$\frac{dE}{dt}(t) = \int_{-1-L}^{1+L} \frac{\partial^2 u}{\partial t^2}(x,t)\frac{\partial u}{\partial t}(x,t) + \frac{1}{\gamma(x)}\frac{\partial^2 v}{\partial t^2}(x,t)\frac{\partial v}{\partial t}(x,t)\,\mathrm{d}x\,,$$

to conclude from (9.13.15)

$$\frac{dE}{dt}(t) = -\int_{-1-L}^{1+L} \sigma(x)\left|\frac{\partial u}{\partial t}(x,t)\right|^2 + \frac{\sigma(x)}{\gamma(x)}\left|\frac{\partial v}{\partial t}(x,t)\right|^2 \mathrm{d}x \le 0\,.$$

We also see that the energy cannot increase in the course of the evolution.

SOLUTION of (9-13.e):

Using the matrices provided in the problem statement, the discrete variational problem gives rise to the following algebraic equations

$$\begin{aligned} \mathbf{M}_1^1 \frac{\vec{\boldsymbol{\mu}}^{(k+1)} - \vec{\boldsymbol{\mu}}^{(k)}}{\tau} + \mathbf{M}_\sigma^1 \frac{\vec{\boldsymbol{\mu}}^{(k+1)} + \vec{\boldsymbol{\mu}}^{(k)}}{2} + \mathbf{B}_1 \frac{\vec{\boldsymbol{\nu}}^{(k+1)} + \vec{\boldsymbol{\nu}}^{(k)}}{2} &= \vec{\boldsymbol{\varphi}}_0 \,, \\ \mathbf{M}_1^0 \frac{\vec{\boldsymbol{\nu}}^{(k+1)} - \vec{\boldsymbol{\nu}}^{(k)}}{\tau} + \mathbf{M}_\sigma^0 \frac{\vec{\boldsymbol{\nu}}^{(k+1)} + \vec{\boldsymbol{\nu}}^{(k)}}{2} - \mathbf{B}_\gamma^\top \frac{\vec{\boldsymbol{\mu}}^{(k+1)} + \vec{\boldsymbol{\mu}}^{(k)}}{2} &= 0 \,, \end{aligned} \tag{9.13.19}$$

Collecting all terms involving $\vec{\boldsymbol{\mu}}^{(k+1)}$ and $\vec{\boldsymbol{\nu}}^{(k+1)}$ on the left, the others on the right-hand side, we end up with

$$\begin{bmatrix} \frac{1}{\tau}\mathbf{M}_1^1 + \frac{1}{2}\mathbf{M}_\sigma^1 & \frac{1}{2}\mathbf{B}_1 \\ -\frac{1}{2}\mathbf{B}_\gamma^\top & \frac{1}{\tau}\mathbf{M}_1^0 + \frac{1}{2}\mathbf{M}_\sigma^0 \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\mu}}^{(k+1)} \\ \vec{\boldsymbol{\nu}}^{(k+1)} \end{bmatrix} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}_0 + \left(\frac{1}{\tau}\mathbf{M}_1^1 - \frac{1}{2}\mathbf{M}_\sigma^1\right)\vec{\boldsymbol{\mu}}^{(k)} - \frac{1}{2}\mathbf{B}_1\vec{\boldsymbol{\nu}}^{(k)} \\ \frac{1}{2}\mathbf{B}_\gamma^\top\vec{\boldsymbol{\mu}}^{(k)} + \left(\frac{1}{\tau}\mathbf{M}_1^0 - \frac{1}{2}\mathbf{M}_\sigma^0\right)\vec{\boldsymbol{\nu}}^{(k)} \end{bmatrix} . \tag{9.13.20}$$

SOLUTION of (9-13.f):

(i) The finite element space $\mathcal{S}_1^0(\mathcal{M})$ equipped with the tent-function bases [Lecture → Section 2.3.2] induces the two local shape functionson $]x_{j-1}, x_j[$

$$b_j^1(x) := \frac{x_j - x}{h} \quad \text{and} \quad b_j^2(x) := \frac{x - x_{j-1}}{h}, \quad h := x_j - x_{j-1}.$$

Since $b_j^1(x_{j-1}) = 1$, $b_j^1(x_j) = 0$, $b_j^2(x_{j-1}) = 0$, and $b_j^2(x_j) = 1$, when using the quadrature formula (9.13.21) we get a *diagonal* 2×2 element matrix

$$\mathbf{M}_{\mu,j}^1 = \frac{1}{2}h \begin{bmatrix} \mu(x_{j-1}) & 0 \\ 0 & \mu(x_j) \end{bmatrix}.$$

(ii) For the finite element space $\mathcal{S}_0^{-1}(\mathcal{M})$ we have the single local shape function $x \mapsto 1$ associated with every cell, which gives the 1×1 element matrix

$$\mathbf{M}_{\mu,j}^0 = \tfrac{1}{2}h\big[\mu(x_{j-1}) + \mu(x_j)\big].$$

(iii) We rely on the local shape functions as introduced above and take into account [Lecture → (2.3.2.6)]

$$\frac{db_j^1}{dx}(x) = -\frac{1}{h} \quad \text{and} \quad \frac{db_j^2}{dx}(x) = \frac{1}{h}.$$

This gives the 2×1 element matrix

$$\mathbf{B}_{\mu,j} = \begin{bmatrix} -\frac{1}{2}(\mu(x_{j-1}) + \mu(x_j)) \\ \\ \frac{1}{2}(\mu(x_{j-1}) + \mu(x_j)) \end{bmatrix} \in \mathbb{R}^{2,1}.$$

Note, that the size h of the cell cancels in this case.

SOLUTION of (9-13.g):

We set $h := \frac{2+2L}{N}$ for the meshwidth of the underlying equidistant mesh. Based on Sub-problem (9-13.f) we perform "assembly on paper" and get

$$\mathbf{M}_1^1 = h\,\mathrm{diag}(\tfrac{1}{2}, 1, \dots, 1, \tfrac{1}{2}) \in \mathbb{R}^{N+1,N+1}\,, \tag{9.13.23a}$$

$$\mathbf{M}_\sigma^1 = h\,\mathrm{diag}(\tfrac{1}{2}\sigma(x_0), \sigma(x_1), \dots, \sigma(x_{N-1}), \tfrac{1}{2}\sigma(x_N)) \in \mathbb{R}^{N+1,N+1}\,, \tag{9.13.23b}$$

$$\mathbf{M}_1^0 = h\mathbf{I}_N \in \mathbb{R}^{N,N}\,, \tag{9.13.23c}$$

$$\mathbf{M}_\sigma^0 = \tfrac{1}{2}h\,\mathrm{diag}(\sigma(x_0) + \sigma(x_1), \dots, \sigma(x_{N-1}) + \sigma(x_N)) \in \mathbb{R}^{N,N}\,, \tag{9.13.23d}$$

$$\mathbf{B}_1 = \begin{bmatrix} -1 & & & & \\ 1 & -1 & & & \\ & 1 & \ddots & & \\ & & \ddots & \ddots & \\ & & & -1 & \\ & & & 1 & -1 \\ & & & & 1 \end{bmatrix} \in \mathbb{R}^{N+1,N}\,, \tag{9.13.23e}$$

$$\mathbf{B}_\gamma = \begin{bmatrix} -\frac{1}{2}(\gamma_0+\gamma_1) & & & & \\ \frac{1}{2}(\gamma_0+\gamma_1) & -\frac{1}{2}(\gamma_1+\gamma_2) & & & \\ & \frac{1}{2}(\gamma_1+\gamma_2) & \ddots & & \\ & & \ddots & \ddots & \\ & & & -\frac{1}{2}(\gamma_{N-2}+\gamma_{N-1}) & \\ & & & \frac{1}{2}(\gamma_{N-2}+\gamma_{N-1}) & -\frac{1}{2}(\gamma_{N-1}+\gamma_N) \\ & & & & \frac{1}{2}(\gamma_{N-1}+\gamma_N) \end{bmatrix} \tag{9.13.23f}$$

$$\mathbf{B}_\gamma^\top = \begin{bmatrix} -\frac{1}{2}(\gamma_0+\gamma_1) & \frac{1}{2}(\gamma_0+\gamma_1) & & & & \\ & -\frac{1}{2}(\gamma_1+\gamma_2) & \frac{1}{2}(\gamma_1+\gamma_2) & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & \ddots & \ddots \\ & & & -\frac{1}{2}(\gamma_{N-2}+\gamma_{N-1}) & \frac{1}{2}(\gamma_{N-2}+\gamma_{N-1}) & \\ & & & & -\frac{1}{2}(\gamma_{N-1}+\gamma_N) & \frac{1}{2}(\gamma_{N-1}- \end{bmatrix}$$

with $\gamma_j := \gamma(x_j)$, $j = 0, \dots, N$. For the system matrix from (9.13.20) set

$$\mathbf{A} := \begin{bmatrix} \frac{1}{\tau}\mathbf{M}_1^1 + \frac{1}{2}\mathbf{M}_\sigma^1 & \frac{1}{2}\mathbf{B}_1 \\ -\frac{1}{2}\mathbf{B}_\gamma^\top & \frac{1}{\tau}\mathbf{M}_1^0 + \frac{1}{2}\mathbf{M}_\sigma^0 \end{bmatrix} \in \mathbb{R}^{2N+1,2N+1}\,. \tag{9.13.24}$$

Using (9.13.23), we find

$$(\mathbf{A})_{i,j} = \begin{cases} \frac{h}{2\tau} + \frac{h}{4}\sigma(x_0) & \text{, if } \quad i = j = 1 \text{ ,} \\ \frac{h}{\tau} + \frac{h}{2}\sigma(x_{i-1}) & \text{, if } \quad i = j \text{ and } 1 < i,j \leq N \text{ ,} \\ \frac{h}{2\tau} + \frac{h}{4}\sigma(x_N) & \text{, if } \quad i = j = N+1 \text{ ,} \\ \frac{h}{\tau} + \frac{h}{4}(\sigma(x_{i-N-2}) + \sigma(x_{i-N-1})) & \text{, if } \quad i = j \text{ and } N+2 \leq i,j \leq 2N+1 \text{ ,} \\ -\frac{1}{2} & \text{, if } \quad 1 \leq i < N+1 \text{ and } j = i+N+1 \text{ ,} \\ \frac{1}{2} & \text{, if } \quad 1 < i \leq N+1 \text{ and } j = i+N \text{ ,} \\ \frac{1}{4}(\gamma_{j-1} + \gamma_j) & \text{, if } \quad 1 \leq j < N+1 \text{ and } i = j+N+1 \text{ ,} \\ -\frac{1}{4}(\gamma_{j-2} + \gamma_{j-1}) & \text{, if } \quad 1 < j \leq N+1 \text{ and } i = j+N \text{ ,} \\ 0 & \text{else.} \end{cases}$$

Using (9.13.23) the right-hand side vector of (9.13.20)

$$\vec{\boldsymbol{\beta}}^{(k)} := \begin{bmatrix} \vec{\boldsymbol{\varphi}}_0 + \left(\frac{1}{\tau}\mathbf{M}_1^1 - \frac{1}{2}\mathbf{M}_\sigma^1\right)\vec{\boldsymbol{\mu}}^{(k)} - \frac{1}{2}\mathbf{B}_1\vec{\boldsymbol{\nu}}^{(k)} \\ \\ \frac{1}{2}\mathbf{B}_\gamma^\top\vec{\boldsymbol{\mu}}^{(k)} + \left(\frac{1}{\tau}\mathbf{M}_1^0 - \frac{1}{2}\mathbf{M}_\sigma^0\right)\vec{\boldsymbol{\nu}}^{(k)} \end{bmatrix} \in \mathbb{R}^{2N+1}$$

can be rewritten as

$$\vec{\boldsymbol{\beta}}^{(k)} = \begin{bmatrix} \vec{\boldsymbol{\varphi}}_0 \\ \mathbf{0} \end{bmatrix} - \mathbf{R}\begin{bmatrix} \vec{\boldsymbol{\mu}}^{(k)} \\ \vec{\boldsymbol{\nu}}^{(k)} \end{bmatrix}, \quad \mathbf{R} := \begin{bmatrix} -\frac{1}{\tau}\mathbf{M}_1^1 + \frac{1}{2}\mathbf{M}_\sigma^1 & \frac{1}{2}\mathbf{B}_1 \\ -\frac{1}{2}\mathbf{B}_\gamma^\top & -\frac{1}{\tau}\mathbf{M}_1^0 + \frac{1}{2}\mathbf{M}_\sigma^0 \end{bmatrix}. \tag{9.13.25}$$

Note that the matrix $\mathbf{R}$ features the same building blocks as $\mathbf{A}$. The entries of the vector $\vec{\boldsymbol{\varphi}}_0 \in \mathbb{R}^{N+1}$ are

$$(\vec{\boldsymbol{\varphi}}_0)_i = \begin{cases} \frac{1}{2}hv_0(x_0) & \text{for} \quad i = 1 \text{ ,} \\ hv_0(x_{i-1}) & \text{for} \quad 1 < i \leq N \text{ ,} \\ \frac{1}{2}hv_0(x_N) & \text{for} \quad i = N+1 \end{cases} \quad \approx \int_{-1-L}^{1+L} v_0(x) b_h^{i-1}(x)\,\mathrm{d}x \text{ ,} \quad i = 1,\ldots,N+1 \text{ .}$$

Next, we introduce a single vector for the unknown expansion coefficients

$$\vec{\boldsymbol{\zeta}}^{(k)} := \begin{bmatrix} \vec{\boldsymbol{\mu}}^{(k)} \\ \vec{\boldsymbol{\nu}}^{(k)} \end{bmatrix} \in \mathbb{R}^{2N+1} \quad \Rightarrow \quad \begin{aligned} \mu_i &= \left(\vec{\boldsymbol{\zeta}}^{(k)}\right)_i, & i &= 1,\ldots,N+1 \text{ ,} \\ \nu_j &= \left(\vec{\boldsymbol{\zeta}}^{(k)}\right)_{j+N+1}, & j &= 1,\ldots,N \text{ .} \end{aligned}$$

Then the discrete evolution over one timestep $k \to k+1$ can be stated as

$$\vec{\boldsymbol{\zeta}}^{(k+1)} := \mathbf{A}^{-1}\left(\begin{bmatrix} \vec{\boldsymbol{\varphi}}_0 \\ \mathbf{0} \end{bmatrix} - \mathbf{R}\vec{\boldsymbol{\zeta}}^{(k)}\right). \tag{9.13.26}$$

C++ code 9.13.27: Implementation of `solve1DWavePML()`

```
2 template <typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
3 Eigen::VectorXd solve1DWavePML(
4     const Eigen::VectorXd &zeta_0, const Eigen::VectorXd &gamma,
5     const Eigen::VectorXd &sigma, unsigned int M, double T,
```

```
  RECORDER &&rec = [](Eigen::VectorXd & /*zeta*/) -> void {}) {
// Grid resolution parameter N = number of grid nodes - 1
const unsigned int N = gamma.size() - 1;
assert(N == sigma.size() - 1);
assert(2 * N + 1 == zeta_0.size());
// Vector of basis expansion coefficients, contains
// ζ⃗^(k)
Eigen::VectorXd zeta{zeta_0};
// Obtain initial velocities at grid nodes by linear interpolation
Eigen::VectorXd v0 =
    (Eigen::VectorXd(N + 1) << 0.0,
     0.5 * (zeta.segment(N + 1, N - 1) + zeta.segment(N + 2, N - 1)), 0.0)
        .finished();
// Discretization parameters
const double L = L_default;          // default width of PML layer
const double h = (2.0 + 2 * L) / N;  // meshwidth
const double tau = T / M;            // size of timestep
// I. Initialize sparse matrices A,R ∈ ℝ^{2N+1,2N+1}.
Eigen::SparseMatrix<double> A(2 * N + 1, 2 * N + 1);
Eigen::SparseMatrix<double> R(2 * N + 1, 2 * N + 1);
// We know that the matrix A has at most 3 non-zero entries per
   row/column
A.reserve(Eigen::VectorXi::Constant(2 * N + 1, 3));
R.reserve(Eigen::VectorXi::Constant(2 * N + 1, 3));
// First initialize diagonal
A.insert(0, 0) = h / (2 * tau) + 0.25 * h * sigma[0];
R.insert(0, 0) = -h / (2 * tau) + 0.25 * h * sigma[0];
for (unsigned int i = 1; i < N; ++i) {
  A.insert(i, i) = h / tau + 0.5 * h * sigma[i];
  R.insert(i, i) = -h / tau + 0.5 * h * sigma[i];
}
A.insert(N, N) = h / (2 * tau) + 0.25 * h * sigma[N];
R.insert(N, N) = -h / (2 * tau) + 0.25 * h * sigma[N];
for (unsigned int i = 0; i < N; ++i) {
  A.insert(i + N + 1, i + N + 1) =
      h / tau + 0.25 * h * (sigma[i] + sigma[i + 1]);
  R.insert(i + N + 1, i + N + 1) =
      -h / tau + 0.25 * h * (sigma[i] + sigma[i + 1]);
  A.insert(i, i + N + 1) = -0.5;
  A.insert(i + 1, i + N + 1) = 0.5;
  R.insert(i, i + N + 1) = -0.5;
  R.insert(i + 1, i + N + 1) = 0.5;
}
for (unsigned int j = 0; j < N; ++j) {
  A.insert(j + N + 1, j) = 0.25 * (gamma[j] + gamma[j + 1]);
  A.insert(j + N + 1, j + 1) = -0.25 * (gamma[j] + gamma[j + 1]);
  R.insert(j + N + 1, j) = 0.25 * (gamma[j] + gamma[j + 1]);
  R.insert(j + N + 1, j + 1) = -0.25 * (gamma[j] + gamma[j + 1]);
}
// For the sake efficiency Precompute LU factorization of A
Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
solver.compute(A);
if (solver.info() != Eigen::Success) {
  throw std::runtime_error("LU decomposition of A failed");
}
// II. Main timestepping loop
rec(zeta);
```

```
62    for (unsigned int k = 0; k < M; ++k) {
63      // Initialize right-hand side vector β⃗^(k) of size
64      // 2N+1
65      Eigen::VectorXd beta = -R * zeta;
66      beta[0] += 0.5 * h * v0[0];
67      for (unsigned int i = 1; i < N; ++i) beta[i] += h * v0[i];
68      beta[N] += 0.5 * h * v0[N];
69      Eigen::VectorXd zeta_old = zeta;
70      zeta = solver.solve(beta);  // ζ⃗^(k+1)
71      rec(zeta);                  // Provide state vector to monitor object
72    }
73    return zeta;
74  }
```

SOLUTION of (9-13.h):

From the d'Alembert solution [Lecture → Ex. 11.6.1.23] for the Cauchy problem for the 1D wave equation with constant coefficients we know that the exact solution is

$$u(x,t) = \frac{1}{2}(u_0(x-t) + u_0(x+t)) \,, \quad x \in \mathbb{R} \,, \quad t \geq 0 \,.$$

Its graph is two humps moving with speed 1 in opposite directions.

This behavior is very well captured by the numerical solution in $[-1,1]$. Inside the PML layers the solution decays quickly and away from the support $(x,t) \mapsto u(x,t)$ in $]-1,1[\times]0,T[$ the numerical solution is fairly small. This means that there are hardly any spurious reflections from the PML layers, which perform well as absorbing boundary conditions.

HINT 1 for (9-13.i): In C++ a lambda function defined as

```
auto rec = [&](/*args*/ ){/*code*/};
```

has read and write access to all variables in the local scope. ⌟

SOLUTION of (9-13.i):

C++ code 9.13.30: Implementation of `solve1DWavePML()`

```
std::vector<double> trackEnergy(const Eigen::VectorXd &zeta_0,
                                const Eigen::VectorXd &gamma,
                                const Eigen::VectorXd &sigma, unsigned int M,
                                double T) {
  // Grid resolution parameter N = number of grid nodes - 1
  const unsigned int N = gamma.size() - 1;
  const double L = L_default;
  const double h = (2.0 + 2 * L) / N;
  const double tau = T / M;
  // Storage for energies
  std::vector<double> en{};
  // Temporary storage for solution
  Eigen::VectorXd zeta_old(2 * N + 1);
  // Flag telling recorder lambda function that we are at the first
     timestep
  bool first_step = true;
  // Recorder lambda function
  auto rec = [&](const Eigen::VectorXd &zeta) -> void {
    if (first_step) {
      first_step = false;
    } else {
      auto delta = zeta - zeta_old;
      // Norm of discrete temporal derivative of u-component
      double mu_d_norm = 0.5 * delta[0] * delta[0];
      for (unsigned int i = 1; i < N; ++i) {
        mu_d_norm += delta[i] * delta[i];
      }
      mu_d_norm += 0.5 * delta[N] * delta[N];
      mu_d_norm *= 0.5 * h / (tau * tau);
      // Norm of discrete temporal derivative of v-component
      double nu_d_norm = 0.0;
      for (unsigned int i = 0; i < N; ++i) {
        nu_d_norm += (delta[i + N + 1] * delta[i + N + 1]) / gamma[i];
      }
      nu_d_norm *= 0.5 * h / (tau * tau);
      en.push_back(nu_d_norm + mu_d_norm);
    }
    zeta_old = zeta;
  };
  (void)solve1DWavePML(zeta_0, gamma, sigma, M, T, rec);
  return en;
}
```

SOLUTION of (9-14.a):

Note that (9.14.1) features homogeneous Neumann boundary conditions, that is, natural spatial boundary conditions, which means that the spatial variational formulation is posed on the full space $H^1(\Omega)$. Thus from [Lecture → Section 9.2.2] in combination with [Lecture → Ex. 1.8.0.10] we learn that the spatial variational formulation of (9.14.1) reads: seek $t \mapsto u(t) \in H^1(\Omega)$ such that

$$\underbrace{\int_\Omega \rho(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}_{=:\mathsf{m}(\dot{u},v)} + \underbrace{\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = 0 \quad \forall v \in H^1(\Omega)\,. \tag{9.14.3}$$

Appealing to the formulas from [Lecture → Section 9.2.4] we arrive at

$$\begin{aligned} \mathbf{M} &= \left[\int_\Omega \rho(\boldsymbol{x}) b_h^j(\boldsymbol{x}) b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right]_{i,j=1}^N, \\ \mathbf{A} &= \left[\int_\Omega \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right]_{i,j=1}^N, \\ \vec{\boldsymbol{\varphi}}(t) &= \mathbf{0}\,. \end{aligned}$$

HINT 1 for (9-14.b):

Definition [Lecture → Def. 7.3.3.1]. General Runge-Kutta method

For coefficients $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, the discrete evolution $\boldsymbol{\Psi}^{s,t}$ of an s-stage Runge-Kutta single step method (RK-SSM) for the ODE $\dot{\mathbf{u}} = \mathbf{f}(t,\mathbf{u})$, is defined by

$$\mathbf{k}_i := \mathbf{f}(t + c_i\tau, \mathbf{u} + \tau \sum_{j=1}^{s} a_{ij}\mathbf{k}_j) \;, \quad i = 1,\ldots,s \quad , \quad \boldsymbol{\Psi}^{t,t+\tau}\mathbf{u} := \mathbf{u} + \tau \sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

The $\mathbf{k}_i \in V_0$ are called increments.

Widely used is the **Butcher-scheme** notation for the defining coefficients introduced in [Lecture → Section 7.3.3]:

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \;\widehat{=}\; \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \ldots & a_{1s} \\ c_2 & a_{21} & \ddots & & a_{2s} \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & \ldots & & a_{ss} \\ \hline & b_1 & b_2 & \ldots & b_s \end{array} \quad , \quad \mathbf{c},\mathbf{b} \in \mathbb{R}^s, \quad \mathfrak{A} \in \mathbb{R}^{s,s} \;. \qquad [\text{Lecture} \rightarrow (7.3.3.3)]$$

⌟

SOLUTION of (9-14.b):

For a general RK-SSM the equations defining a single timestep for (9.14.2) have been derived in [Lecture → § 9.2.7.6]. We specialize those formulas to the RK-SSM (9.14.4).

To begin with, we remind of the following notations and correspondences:

- state vector $\mathbf{u}(t)$ $\quad\leftrightarrow\quad$ basis expansion coefficient vector $\vec{\boldsymbol{\mu}}(t)$
- Runke-Kutta increments $\mathbf{k}_1, \mathbf{k}_2$ $\quad\leftrightarrow\quad$ $\vec{\boldsymbol{\kappa}}_1, \vec{\boldsymbol{\kappa}}_2 \in \mathbb{R}^N$

The MOL-ODE (9.14.2) then corresponds to the ODE in standard form $\dot{\mathbf{u}}(t) = \mathbf{f}(t, \mathbf{u}(t))$ with

$$\mathbf{f}(t, \mathbf{u}(t)) \quad\leftrightarrow\quad \mathbf{f}(t, \vec{\boldsymbol{\mu}}) := -\mathbf{M}^{-1}\mathbf{A}\vec{\boldsymbol{\mu}} + \mathbf{M}^{-1}\vec{\boldsymbol{\varphi}}(t) \,.$$

So the defining formulas for an RK-SSM yield

$$\begin{aligned}
\vec{\boldsymbol{\kappa}}_1 &= -\mathbf{M}^{-1}\mathbf{A}\left(\vec{\boldsymbol{\mu}^{(k)}} + \tau\tfrac{5}{12}\vec{\boldsymbol{\kappa}}_1 - \tau\tfrac{1}{12}\vec{\boldsymbol{\kappa}}_2\right) + \mathbf{M}^{-1}\vec{\boldsymbol{\varphi}}(t_k + \tfrac{1}{3}\tau) \,,\\
\vec{\boldsymbol{\kappa}}_2 &= -\mathbf{M}^{-1}\mathbf{A}\left(\vec{\boldsymbol{\mu}^{(k)}} + \tau\tfrac{3}{4}\vec{\boldsymbol{\kappa}}_1 + \tau\tfrac{1}{4}\vec{\boldsymbol{\kappa}}_2\right) + \mathbf{M}^{-1}\vec{\boldsymbol{\varphi}}(t_k + \tau) \,.
\end{aligned}$$

This is equivalent to the $2N \times 2N$ linear system of equations

$$\begin{bmatrix} \mathbf{M} + \frac{5}{12}\tau\mathbf{A} & -\frac{1}{12}\tau\mathbf{A} \\ \frac{3}{4}\tau\mathbf{A} & \mathbf{M} + \frac{1}{4}\tau\mathbf{A} \end{bmatrix} \begin{bmatrix} \vec{\boldsymbol{\kappa}}_1 \\ \vec{\boldsymbol{\kappa}}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{A}\vec{\boldsymbol{\mu}}^{(k)} + \vec{\boldsymbol{\varphi}}(t_k + \frac{1}{3}\tau) \\ -\mathbf{A}\vec{\boldsymbol{\mu}}^{(k)} + \vec{\boldsymbol{\varphi}}(t_{k+1}) \end{bmatrix} .$$

This has to be supplemented with the update equation

$$\vec{\boldsymbol{\mu}}^{(k+1)} = \vec{\boldsymbol{\mu}}^{(k)} + \tau\tfrac{3}{4}\,\vec{\boldsymbol{\kappa}}_1 + \tau\tfrac{1}{4}\,\vec{\boldsymbol{\kappa}}_2 \,.$$

SOLUTION of (9-15.a):

From [Lecture → Section 7.3.3]/ we recall the terminology for RK-SSMs described by the **Butcher scheme** $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$:

$\mathfrak{A}$ strict lower triangular matrix ➤ explicit Runge-Kutta method, [Lecture → Def. 6.4.0.9]/

$\mathfrak{A}$ non-zero entry on or above the diagonal ➤ implicit Runge-Kutta method (IRK)

Hence,

- (B) is definitely explicit, but
- for (A) we cannot decide: it is implicit, if $a_{ii} \neq 0$ for at least one $i \in \{1, \ldots, s\}$, otherwise also explicit.

SOLUTION of (9-15.b):

The scalar Newton iteration for solving $\tilde{f}(u) = 0$ for a function $\tilde{f}(u) \in C^1(\mathbb{R})$ is

$$u^{(n+1)} = u^{(n)} - \frac{\tilde{f}(u^{(n)})}{\tilde{f}'(u^{(n)})} \quad \text{provided that the derivative } \tilde{f}'(u^{(n)}) \neq 0 \, .$$

In (9.15.4), we can define $\tilde{f}(v_i)$ as

$$\tilde{f}(v_i) := \underbrace{u_0 + \tau \sum_{j=1}^{i-1} a_{i,j} k_j + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j} \widehat{k}_j}_{\text{independent of } v_i} + \tau a_{i,i} g(v_i) - v_i \, .$$

Hence, the Newton iteration is

$$v_i^{(n+1)} = v_i^{(n)} - \frac{1}{\tau a_{i,i} g'(v_i^{(n)}) - 1} \left(u_0 + \tau \sum_{j=1}^{i-1} a_{i,j} k_j + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j} \widehat{k}_j + \tau a_{i,i} g(v_i^{(n)}) - v_i^{(n)} \right) .$$

SOLUTION of (9-15.c):

Rewrite $\mathbf{v}_i$ in (9.15.4) as

$$\mathbf{v}_i := \mathbf{u}_0 + \tau \sum_{j=1}^{i-1} a_{i,j}\mathbf{k}_j + \tau a_{i,i}\mathbf{g}(\mathbf{v}_i) + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j}\widehat{\mathbf{k}}_j \,. \tag{9.15.7}$$

Then, by replacing $\mathbf{g}(\mathbf{u}) := -\mathbf{A}\mathbf{u}$ with $\mathbf{A}$ given and s.p.d., you get

$$\mathbf{v}_i = (\mathbf{I} + \tau a_{i,i}\mathbf{A})^{-1}\left(\mathbf{u}_0 + \tau \sum_{j=1}^{i-1} a_{i,j}\mathbf{k}_j + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j}\widehat{\mathbf{k}}_j\right) \,.$$

Remark. For practical IMEX RK-SSMs we have $a_{i,i} > 0$. In this case, for s.p.d. $\mathbf{A}$ also the matrix $\mathbf{I} + \tau a_{i,i}\mathbf{A}$ is s.p.d. and, hence, invertible.

SOLUTION of (9-15.d):

C++ code 9.15.11: Sub-problem (9-15.d): Implementation of `IMEXError` ⇢ GitLab

```cpp
double IMEXError(int M) {
  const double tau = 1. / M;
  // We store the error for each intermediate step we take
  std::vector<double> err(M + 1);
  err[0] = 0;
  // We store each intermediate step we take
  std::vector<double> y_rk(M + 1);
  // Initialize y_0 with 1/4
  y_rk[0] = 0.25;
  // Compute gamma given by the RKS
  const double gamma = (3.0 + std::sqrt(3)) / 6.0;

  // Store the increment vectors
  Eigen::Vector2d k;
  Eigen::Vector3d k_hat;

  // Store the Butcher Matrices in Eigen::Matrices for ease of use
  Eigen::Matrix2d a;
  a << gamma, 0.0, 1.0 - 2.0 * gamma, gamma;
  const Eigen::Vector2d b{0.5, 0.5};
  Eigen::Matrix3d a_hat;
  a_hat << 0., 0., 0., gamma, 0., 0., gamma - 1., 2.0 * (1.0 - gamma), 0.;
  const Eigen::Vector3d b_hat{0.0, 0.5, 0.5};

  // Define the functions g and f
  auto f = [](double x) { return -x * x; };
  auto g = [](double x) { return x; };

  // Define the exact solutions
  auto y_exact = [](double t) { return std::exp(t) / (3 + std::exp(t)); };

  // Keep some temporary variables
  double tmp = 0.0;
  double u;

  // In this for loop we execute the explicit update from y_0 to y_1
  for (unsigned int step = 1; step < M + 1; ++step) {
    k_hat[0] = f(y_rk[step - 1]);
    // Accumualte the terms all in one variable tmp
    for (unsigned int i = 0; i < 2; ++i) {
      for (unsigned int j = 0; j < i; ++j) {
        tmp += a(i, j) * k[j] + a_hat(i + 1, j) * k_hat[j];
      }
      // Don't forget the additional term
      tmp += a_hat(i + 1, i) * k_hat[i];
      // Compute u
      u = (y_rk[step - 1] + tau * tmp) / (1 - tau * a(i, i));
      // Keep track of the increment vectors
      k[i] = g(u);
      k_hat[i + 1] = f(u);
      // Reset our temporary variable
      tmp = 0;
```

```
54      }
55      // Store the intermediate result and error
56      y_rk[step] = y_rk[step - 1] + tau * (b.dot(k) + b_hat.dot(k_hat));
57      err[step] = std::abs(y_rk[step] - y_exact(step * tau));
58    }
59    // Return the maximal error we made during the time stepping
60    return *std::max_element(err.begin(), err.end());
61  }
```

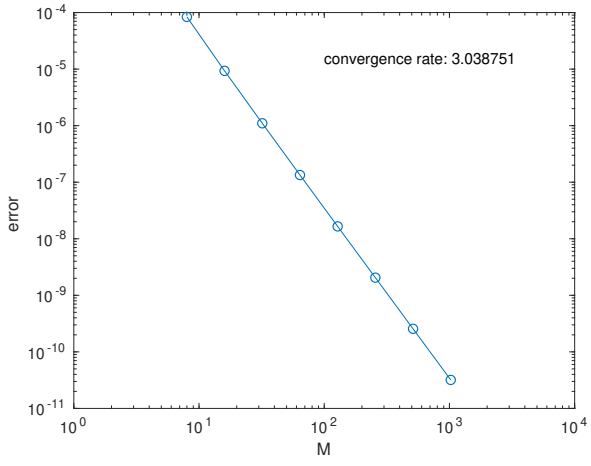


Table of error:

M	error
2^3	8.38411e-05
2^4	9.30965e-06
2^5	1.10098e-06
2^6	1.33980e-07
2^7	1.65278e-08
2^8	2.05249e-09
2^9	2.55726e-10
2^{10}	3.19141e-11

◁ Log-log plot of `err` vs. M. The error curve is almost a line and from its slope we can read off algebraic convergence with a rate ~ 3 .

SOLUTION of (9-15.e):

It boils down to "undoing" integration by parts using Green's formula. The linear terms are the standard terms occurring in the variational formulation of a scalar linear parabolic boundary value problem with spatial radiation/impedance boundary conditions, see [Lecture → (1.8.0.8)]. The non-linear term is of order zero and is not affected by integration by parts. We first convert (9.15.12) into

$$\int_\Omega \left(\frac{\partial u}{\partial t} - \Delta u + \sinh(u) \right) v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} (u + \mathbf{grad}\, u \cdot \boldsymbol{n}) v \,\mathrm{d}S(\boldsymbol{x}) = \int_{\partial\Omega} v \,\mathrm{d}S \quad \forall v \in C^\infty(\overline{\Omega}) \,.$$

Next, we

1. test with compactly supported $v \in C_0^\infty(\Omega)$ to extract the PDE based on [Lecture → Lemma 1.5.3.4], and
2. then we test with $v \in C^\infty(\overline{\Omega})$, use the PDWE to cancel all integral over Ω and apply [Lecture → Lemma 1.5.3.16] to tease out the boundary conditions.

We arrive at

$$\begin{aligned} \text{PDE:} \quad & \frac{\partial u}{\partial t} - \Delta u + \sinh(u) = 0 \quad \text{in} \quad \Omega \,, \\ \text{boundary condtion:} \quad & \mathbf{grad}\, u \cdot \boldsymbol{n} + u = 1 \quad \text{on} \quad \partial\Omega \,, \\ \text{initial condition:} \quad & u(0) = u_0 \,. \end{aligned}$$

SOLUTION of (9-15.f):

From [Lecture → Section 9.2.4] it is immediate:

$$
\begin{aligned}
(\mathbf{M})_{ij} &= \int_{\Omega} b_h^i \, b_h^j \, \mathrm{d}\boldsymbol{x} \;, \\
(\mathbf{A})_{ij} &= \int_{\Omega} \mathbf{grad}\, b_h^i \cdot \mathbf{grad}\, b_h^j \, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} b_h^i \, b_h^j \, \mathrm{d}S \;, \\
(\vec{\boldsymbol{\varphi}})_i &= \int_{\partial\Omega} b_h^i \, \mathrm{d}S \;,
\end{aligned}
$$

with $i, j = 1, \ldots, N$.

SOLUTION of (9-15.g):

The vector $\mathbf{r}(\vec{\mu})$ has the entries

$$(\mathbf{r}(\vec{\mu}))_i = \sum_{K\in\mathcal{M}} \tfrac{1}{3}|K| \sum_{\ell=1}^{3} \sinh\Big(\sum_{k=1}^{N}\Big((\vec{\mu})_k b_h^k(a_\ell^K)\Big)\Big)\, b_h^i(a_K^\ell)\ .$$

Here, a_ℓ^K, $\ell = 1,2,3$, are the vertices of the triangle K. Since the tent functions are a cardinal basis of $\mathcal{S}_1^0(\mathcal{M})$ with respect to the nodes of the mesh, this can be simplified to

$$(\mathbf{r}(\vec{\mu}))_i = \sinh\big((\vec{\mu})_i\big) \sum_{x^i\in K} \tfrac{1}{3}|K|\ , \quad i = 1,\ldots,N\ , \tag{9.15.15}$$

where $\{x^i\}$ is the set of nodes of the mesh, endowed with the same numbering as the set of global shape functions. From this formula we can also see that $\mathbf{r}(\vec{\mu})$ can be built by *cell-oriented local assembly* [Lecture → Section 2.4.6] using the element vectors

$$\vec{\rho}_K = \tfrac{1}{3}|K| \begin{bmatrix} \sinh\big((\vec{\mu})_{i_1}\big) & \sinh\big((\vec{\mu})_{i_2}\big) & \sinh\big((\vec{\mu})_{i_3}\big) \end{bmatrix}\ ,$$

where i_ℓ, $\ell = 1,2,3$ is the global index number of the ℓ-th vertes of the triangle K. For details see [Lecture → Section 2.4.6].

Remark. The formula (9.15.15) means that the i-th component of $\mathbf{r}(\vec{\mu})$ depends only on $(\vec{\mu})_i$.

C++ code 9.15.16: Sub-problem (9-15.g): Implementation of `AssemblerLocalFunc`

```
2  struct AssemblerLocalFunc {
3    template <typename BUILDER_DATA, typename ELEMENT>
4    static Eigen::Vector3d Eval(const BUILDER_DATA& data, const ELEMENT& el) {
5      Eigen::Vector3d res;
6      for (int i = 0; i < 3; ++i) {
7        res[i] = std::sinh(data.u_loc[i]);
8      }
9      const double vol = lf::geometry::Volume(*el.Geometry());
10     return res * vol / 3.;
11   };
12 };
```

C++ code 9.15.17: Sub-problem (9-15.g): Implementation of `compNonlinearTerm`

```
2  Eigen::VectorXd compNonlinearTerm(
3      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_test,
4      const Eigen::VectorXd& u) {
5    // Get the DofHandler
6    const lf::assemble::DofHandler& dofh = fe_test->LocGlobMap();
7    // Get the mesh
8    const std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_test->Mesh();
9    // Initialize the vector that will hold the nonLinearTerm
10   Eigen::VectorXd nonLinearTerm = Eigen::VectorXd::Zero(dofh.NumDofs());
11   // Define the data structure , that will be passed on to the
12   // localVectorAssembler
13   struct data_t {
```

```
    Eigen::VectorXd u;
    Eigen::Vector3d u_loc;
    explicit data_t(Eigen::VectorXd u) : u(std::move(u)){};
  } data(u);

  // Run over all Elements in the mesh
  for (const lf::mesh::Entity* element : mesh_p->Entities(0)) {
    LF_ASSERT_MSG(lf::base::RefEl::kTria() == element->RefEl(),
                  "compNonlinearTerm is only Implemented for triangles");
    // Get the indices of the corners of the current triangle
    auto indicies = dofh.GlobalDofIndices(*element);
    // Store the coefficients of the basis expansion in u_loc
    data.u_loc[0] = u[indicies[0]];
    data.u_loc[1] = u[indicies[1]];
    data.u_loc[2] = u[indicies[2]];
    // Compute the local nonlinearTerms
    Eigen::Vector3d local_non_linear_term =
        AssemblerLocalFunc::Eval<data_t, const lf::mesh::Entity>(data,
                                                                 *element);
    nonLinearTerm[indicies[0]] += local_non_linear_term[0];
    nonLinearTerm[indicies[1]] += local_non_linear_term[1];
    nonLinearTerm[indicies[2]] += local_non_linear_term[2];
  }
  // Return the entire nonLinearTerm
  return nonLinearTerm;
}
```

HINT 1 for (9-15.h): You can also use a `compGalerkinMatrix()` to initialize the right-hand-side vector `phi_`! ⌟

SOLUTION of (9-15.h):

We start by rewriting (9.15.13) as follows

$$\dot{\vec{\mu}} = \mathbf{M}^{-1}\vec{\varphi} - \mathbf{M}^{-1}\mathbf{A}\vec{\mu} - \mathbf{M}^{-1}\mathbf{r}(\vec{\mu})$$

Then define

$$\mathbf{f}(\vec{\mu}) := -\mathbf{M}^{-1}\mathbf{r}(\vec{\mu}) \,,$$
$$\mathbf{g}(\vec{\mu}) := \mathbf{M}^{-1}\vec{\varphi} - \mathbf{M}^{-1}\mathbf{A}\vec{\mu} \,.$$

We then plug $\mathbf{g}(\vec{\mu})$ into (9.15.7) and obatain an explicit update rule for the basis expansion coefficient vector $\vec{\eta}_i$ corresponding to $\mathbf{v}_i$, $i = 1, 2 \ldots, s$:

$$\begin{aligned}
\vec{\eta}_i &= \vec{\mu}_0 + \tau \sum_{j=1}^{i-1} a_{i,j}\mathbf{k}_j + \tau a_{i,i}\mathbf{g}(\vec{\eta}_i) + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j}\widehat{\mathbf{k}}_j \\
&= \left(\mathbf{I} + \tau a_{i,i}\mathbf{M}^{-1}\mathbf{A}\right)^{-1} \left(\vec{\mu}_0 + \tau \sum_{j=1}^{i-1} a_{i,j}\vec{\kappa}_j + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j}\widehat{\vec{\kappa}}_j + \tau a_{i,i}\mathbf{M}^{-1}\vec{\varphi}\right) \,, \qquad (9.15.18) \\
\vec{\kappa}_j &:= \mathbf{M}^{-1}\vec{\varphi} - \mathbf{M}^{-1}\mathbf{A}\vec{\eta}_j \quad , \quad \widehat{\vec{\kappa}}_{j+1} = -\mathbf{M}^{-1}\mathbf{r}(\vec{\eta}_j) \,.
\end{aligned}$$

Pulling out $\mathbf{M}^{-1}$, an equivalent expression is

$$\begin{aligned}
\vec{\eta}_i &= \left(\mathbf{M} + \tau a_{i,i}\mathbf{A}\right)^{-1} \left(\mathbf{M}\vec{\mu}_0 + \tau \sum_{j=1}^{i-1} a_{i,j}\vec{\kappa}_j + \tau \sum_{j=1}^{i} \widehat{a}_{i+1,j}\widehat{\vec{\kappa}}_j + \tau a_{i,i}\vec{\varphi}\right) \,, \qquad (9.15.19) \\
\vec{\kappa}_j &:= \vec{\varphi} - \mathbf{A}\vec{\eta}_j \quad , \quad \widehat{\vec{\kappa}}_{j+1} = -\mathbf{r}(\vec{\eta}_j) \,.
\end{aligned}$$

In terms of implementation the matrices $\mathbf{M} + \tau a_{ii}\mathbf{A}$ should be built in the constructor and their LU-decompositions should also be computed and stored there, for later re-use in every timestep.

C++ code 9.15.20: Sub-problem (9-15.h): Implementation of `IMEXTimestep` ↪ GitLab

```
IMEXTimestep::IMEXTimestep(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_test,
    const double tau, const Eigen::VectorXd& butcher_matrix_diag)
    : tau_(tau) {
  // Define some helper functions that we can pass to compGalerkinMatrix as
  // alpha, beta and gamma
  const lf::mesh::utils::MeshFunctionConstant<double> mf_one(1.);
  const lf::mesh::utils::MeshFunctionConstant<double> mf_zero(0.);
  auto const_one = [](const Eigen::VectorXd& /*x*/) -> double { return 1.0; };
  auto const_zero = [](const Eigen::VectorXd& /*x*/) -> double { return 0.0; };
  // Flag the edges located on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(fe_test->Mesh(), 1)};
  auto edges_predicate = [&bd_flags](const lf::mesh::Entity& edge) -> bool {
    return bd_flags(edge);
  };
  // Compute the rhs vector
  const lf::fe::ScalarLoadEdgeVectorProvider<double, decltype(mf_one),
                                             decltype(edges_predicate)>
```

```
      LocalVectorAssembler(fe_test, mf_one, edges_predicate);
  // Compute the LHS time independent Matrices
  M_ = compGalerkinMatrix(fe_test->LocGlobMap(), const_zero, const_one,
                          const_zero);
  A_ = compGalerkinMatrix(fe_test->LocGlobMap(), const_one, const_zero,
                          const_one);
  phi_ = lf::assemble::AssembleVectorLocally<Eigen::VectorXd>(
      1, fe_test->LocGlobMap(), LocalVectorAssembler);

  // Do some pre-computations to initialize the sparse solvers
  solver_A_.analyzePattern(A_);
  solver_A_.factorize(A_);
  LF_ASSERT_MSG(solver_A_.info() == Eigen::Success,
                "Solver did not manage to factorize A");
  solver_M_.analyzePattern(M_);
  solver_M_.factorize(M_);
  LF_ASSERT_MSG(solver_M_.info() == Eigen::Success,
                "Solver did not manage to factorize M");
  // Build and factorize matrices M + τa_{i,i}A
  for (unsigned int i = 0; i < solver_MplustauA_.size(); ++i) {
    MplustauA_[i] = (M_ + tau_ * butcher_matrix_diag[i] * A_);
    solver_MplustauA_[i].analyzePattern(MplustauA_[i]);
    solver_MplustauA_[i].factorize(MplustauA_[i]);
    LF_ASSERT_MSG(solver_MplustauA_[i].info() == Eigen::Success,
                  "Solver did not manage to factorize MplustauA_[" << i << "]");
  }

}
```

C++ code 9.15.21: Sub-problem (9-15.h): Implementation of `compTimestep` ⇢ GitLab

```
void IMEXTimestep::compTimestep(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO1<double>> fe_test,
    Eigen::VectorXd& y) const {
  // Define gamma
  const double gamma = (3.0 + std::sqrt(3)) / 6.0;
  const int N = fe_test->LocGlobMap().NumDofs();

  // Define the increment Vectors and store the in the columns of a
  // Eigen::Matrix
  Eigen::MatrixXd kappa(N, 2);
  Eigen::MatrixXd kappa_hat(N, 3);

  // Define the butcher Matrices
  Eigen::Matrix2d a;
  a << gamma, 0.0, 1.0 - 2.0 * gamma, gamma;
  const Eigen::Vector2d b{0.5, 0.5};
  Eigen::Matrix3d a_hat;
  a_hat << 0., 0., 0., gamma, 0., 0., gamma - 1., 2.0 * (1.0 - gamma), 0.;
  const Eigen::Vector3d b_hat{0.0, 0.5, 0.5};

  auto update_kappa_hat =
      [fe_test](const Eigen::VectorXd& x) -> Eigen::VectorXd {
    return -compNonlinearTerm(fe_test, x);
  };

```

```
27    auto update_kappa = [this](const Eigen::VectorXd& x) -> Eigen::VectorXd {
28      return phi_ - (A_ * x);
29    };
30
31    // Define some temporary helper variables
32    Eigen::VectorXd tmp(N);
33    tmp.setZero();
34    Eigen::VectorXd u(N);
35
36    // We now perform one step of the IMEXRKSSM
37    kappa_hat.col(0) = update_kappa_hat(y);
38
39    for (unsigned int i = 0; i < 2; ++i) {
40      for (unsigned int j = 0; j < i; ++j) {
41        tmp += a(i, j) * kappa.col(j) + a_hat(i + 1, j) * kappa_hat.col(j);
42      }
43      tmp += a_hat(i + 1, i) * kappa_hat.col(i);
44
45      tmp += a(i, i) * phi_;
46
47      u = solver_MplustauA_[i].solve(M_ * y + tau_ * tmp);
48      kappa.col(i) = update_kappa(u);
49      kappa_hat.col(i + 1) = update_kappa_hat(u);
50      tmp.setZero();
51    }
52
53    y += tau_ * solver_M_.solve(kappa * b + kappa_hat * b_hat);
54
55  }
```

Remark. Every timestep is very expensive, because a dense $N \times N$ LSE has to be solved several times. Every single one incurs asymptotic computational cost of $O(N^3)$ for $N \to \infty$.

HINT 1 for (9-16.a): More explicitly (9.16.1) can be written as

$$\int_0^1 \rho \frac{\partial u}{\partial t}(x,t)v(x)\,\mathrm{d}x + \int_0^1 \frac{\partial j}{\partial x}(x,t)v(x)\,\mathrm{d}x = 0\,,$$

$$\int_0^1 \sigma \frac{\partial j}{\partial t}(x,t)q(x)\,\mathrm{d}x - \int_0^1 \kappa u(x,t)\frac{\partial q}{\partial x}(x)\,\mathrm{d}x + \int_0^1 j(x,t)q(x) + \mu \frac{\partial j}{\partial x}(x,t)\frac{dq}{dx}(x)\,\mathrm{d}x = 0$$

for all $v \in L^2(]0,1[)$, $q \in H_0^1(]0,1[)$.

SOLUTION of (9-16.a):

The first equation of (9.16.1) is

$$\int_0^1 \rho \frac{\partial u}{\partial t}(x,t)v(x)\,\mathrm{d}x + \int_0^1 \frac{\partial j}{\partial x}(x,t)v(x)\,\mathrm{d}x = 0 \quad \forall v \in L^2(]0,1[)\ . \tag{9.16.2}$$

There are no derivatives on the test function v. So the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.1.13] immediately yields

$$\rho \frac{\partial u}{\partial t}(x,t) + \frac{\partial j}{\partial x}(x,t) = 0 \quad \text{in} \quad]0,1[\times]0,T[\ . \tag{9.16.3}$$

Plugging in the definitions of the bilinear forms, the second equation of (9.16.1) reads

$$\int_0^1 \sigma \frac{\partial j}{\partial t}(x,t)q(x)\,\mathrm{d}x - \int_0^1 \kappa u(x,t)\frac{\partial q}{\partial x}(x)\,\mathrm{d}x + \int_0^1 j(x,t)q(x) + \mu \frac{\partial j}{\partial x}(x,t)\frac{dq}{dx}(x)\,\mathrm{d}x = 0 \tag{9.16.4}$$

for all $q \in H_0^1(]0,1[)$.

There are two terms featuring the derivative of the test function q. We apply the integration by parts formula

$$\int_a^b f(\xi)v'(\xi)\,\mathrm{d}\xi = -\int_a^b f'(\xi)v(\xi)\,\mathrm{d}\xi + f(b)v(b) - f(a)v(a) \quad \forall f,v \in C^1_{\mathrm{pw}}([a,b])$$

[Lecture → (1.5.1.8)]

and remove the derivative $\frac{d}{dx}$ from q, *cf.* [Lecture → § 1.5.1.9]. Note that both j and q belong to $H_0^1(]0,1[)$, which means $q(0) = q(1) = 0$ and $j(0) = j(1) = 0$. As a consequence, *no boundary terms* are introduced and we obtain (note the change of sign)

$$\int_0^1 \sigma \frac{\partial j}{\partial t}(x,t)q(x) + \kappa \frac{\partial u}{\partial x}(x,t)q(x) + j(x)q(x) - \mu \frac{\partial^2 j}{\partial x^2}(x,t)q(x)\,\mathrm{d}x = 0\ .$$

Based on the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.1.13] we extract the PDE

$$\sigma \frac{\partial j}{\partial t}(x,t) + \kappa \frac{\partial u}{\partial x}(x,t) + j(x,t) - \mu \frac{\partial^2 j}{\partial x^2}(x,t) = 0 \quad \text{in} \quad]0,1[\times]0,T[\ . \tag{9.16.5}$$

Both in (9.16.3) and (9.16.5) "(x,t)" can also be dropped.

SOLUTION of (9-16.b):

We use the chain rule under the integral to differentiate the energy w.r.t. time:

$$\frac{dE}{dt}(t) = \int_0^1 \rho \frac{\partial u}{\partial t}(x,t)u(x,t)\,\mathrm{d}x + \int_0^1 \frac{\sigma}{\kappa}\frac{\partial j}{\partial t}(x,t)j(x,t)\,\mathrm{d}x\,. \tag{9.16.7}$$

We use the first variational equation of (9.16.1) with $v = u$ to recast the first term:

$$\int_0^1 \rho \frac{\partial u}{\partial t}(x,t)u(x,t)\,\mathrm{d}x = -\int_0^1 \frac{\partial j}{\partial x}(x,t)u(x,t)\,\mathrm{d}x\,.$$

As regards the second term in (9.16.7) we rely on the second variational equation from (9.16.1) with $q = j$:

$$\int_0^1 \frac{\sigma}{\kappa}\frac{\partial j}{\partial t}(x,t)j(x,t)\,\mathrm{d}x = \int_0^1 \frac{\partial j}{\partial x}(x,t)u(x,t)\,\mathrm{d}x - \int_0^1 \frac{1}{\kappa}j(x,t)^2 + \frac{\mu}{\kappa}\left(\frac{\partial j}{\partial x}(x,t)\right)^2\,\mathrm{d}x\,.$$

Adding both expressions we observe a cancellation of the highlighted terms and get

$$\frac{dE}{dt}(t) = -\int_0^1 \frac{1}{\kappa}j(x,t)^2 + \frac{\mu}{\kappa}\left(\frac{\partial j}{\partial x}(x,t)\right)^2\,\mathrm{d}x \leq 0\,.$$

This means that $t \mapsto E(t)$ is decreasing: $E(t) \leq E(0)$ for all $t \in [0,T]$.

SOLUTION of (9-16.c):

The element matrices are obtained by restricting integration to K and then plugging the **local shape functions** into the restricted/local bilinear form.

- For $\mathcal{S}_0^{-1}(\mathcal{M})$ there is only one local shape function on K: $\beta_K^1(x) \equiv 1$.
- For $\mathcal{S}_1^0(\mathcal{M})$ we have two local shape functions

$$b_k^1(x) = 1 - \frac{x}{h} \quad , \quad b_k^2(x) = \frac{x}{h} \quad \blacktriangleright \quad \frac{db_K^1}{dx}(x) = -\frac{1}{h} \quad , \quad \frac{db_K^2}{dx}(x) = \frac{1}{h} \, . \tag{9.16.8}$$

$$\mathbf{M}_K = \left[\rho \int_0^h \beta_K^1(x)\beta_K^1(x)\,\mathrm{d}x \right] = [\rho h] \in \mathbb{R}^{1,1} \, , \tag{9.16.9}$$

$$\mathbf{N}_K = \left[\int_0^h \sigma\, b_K^j(x) b_K^i(x)\,\mathrm{d}x \right]_{i,j=1}^2 = \begin{bmatrix} \int_0^h \sigma\,(1-\frac{x}{h})^2\,\mathrm{d}x & \int_0^h \sigma\,\frac{x}{h}(1-\frac{x}{h})\,\mathrm{d}x \\ \int_0^h \sigma\,\frac{x}{h}(1-\frac{x}{h})\,\mathrm{d}x & \int_0^h \sigma\,(\frac{x}{h})^2\,\mathrm{d}x \end{bmatrix} \tag{9.16.10}$$

$$= \begin{bmatrix} \frac{\sigma h}{3} & \frac{\sigma h}{6} \\ \frac{\sigma h}{6} & \frac{\sigma h}{3} \end{bmatrix} \in \mathbb{R}^{2,2} \, ,$$

$$\mathbf{B}_K = \left[\int_0^h \frac{b_K^j}{dx}(x)\beta_K^1(x)\,\mathrm{d}x \right]_{j=1}^2 = \begin{bmatrix} -1 & 1 \end{bmatrix} \in \mathbb{R}^{1,2} \, , \tag{9.16.11}$$

$$\mathbf{D}_K = \left[\int_0^h b_K^j(x) b_K^i(x) + \mu \frac{b_K^j}{dx}(x) \frac{b_K^i}{dx}(x)\,\mathrm{d}x \right]_{i,j=1}^2 = \begin{bmatrix} \frac{h}{3} + \frac{\mu}{h} & \frac{h}{6} - \frac{\mu}{h} \\ \frac{h}{6} - \frac{\mu}{h} & \frac{h}{3} + \frac{\mu}{h} \end{bmatrix} \in \mathbb{R}^{2,2} \, , \tag{9.16.12}$$

$$\mathbf{C}_K = \left[\int_0^h \kappa \frac{b_K^i}{dx}(x)\beta_K^1(x)\,\mathrm{d}x \right]_{i=1}^2 = \begin{bmatrix} -\kappa \\ \kappa \end{bmatrix} \in \mathbb{R}^{2,1} \, . \tag{9.16.13}$$

SOLUTION of (9-16.d):

To begin with note that

- $\dim \mathcal{S}_0^{-1}(\mathcal{M}) = M$, equal to the number of mesh cells,
- $\dim \mathcal{S}_{1,0}^0(\mathcal{M}) = M-1$, equal to the number of interior nodes.

❶ $\mathbf{B}$: The bilinear form b is restricted to $\mathcal{S}_{1,0}^0(\mathcal{M}) \times \mathcal{S}_0^{-1}(\mathcal{M})$, that is, trial space is $\mathcal{S}_{1,0}^0(\mathcal{M})$, test space $\mathcal{S}_0^{-1}(\mathcal{M})$. Thus, the rows of $\mathbf{B}$ correspond to basis functions of $\mathcal{S}_0^{-1}(\mathcal{M})$, the columns to tent basis functions of $\mathcal{S}_{1,0}^0(\mathcal{M})$. Note that

$$\operatorname{meas}\Big(\operatorname{supp}(\beta_h^i) \cap \operatorname{supp}(b_h^j)\Big) > 0 \quad \Longleftrightarrow \quad i \in \{j, j+1\}\,.$$

As a consequence $\mathbf{B}$ will be a *bi-diagonal* $M \times (M-1)$-matrix with entries

$$(\mathbf{B})_{i,j} = \begin{cases} \int_0^1 \beta_h^{j+1}(x)\frac{db_h^j}{dx}(x)\,\mathrm{d}x = -1 & \text{for} \quad i = j+1\,, \\ \int_0^1 \beta_h^{j}(x)\frac{db_h^j}{dx}(x)\,\mathrm{d}x = 1 & \text{for} \quad i = j\,, \\ 0 & \text{else,} \end{cases} \qquad \begin{array}{l} i \in \{1,\ldots,M\}\,, \\ j \in \{1,\ldots,M-1\}\,, \end{array}$$

where the numerical values of the entries can be deduced from (9.16.11). Hence, we obtain the Galerkin matrix

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & \ldots & & & & & \ldots & 0 \\ -1 & 1 & 0 & & & & & & \vdots \\ 0 & -1 & 1 & 0 & & & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & & & & \\ & & & & & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & & & & 0 & -1 & 1 & 0 \\ \vdots & & & & & & & 0 & -1 & 1 \\ 0 & \ldots & & & & & & \ldots & 0 & -1 \end{bmatrix} \in \mathbb{R}^{M,M-1}\,. \tag{9.16.14}$$

❷ $\mathbf{N}$: $\mathbf{N}$ is the σ-scaled mass matrix for the finite-element space $\mathcal{S}_{1,0}^0(\mathcal{M})$. We can perform "assembly on

paper" based on (9.16.10) and get the *tridiagonal* matrix

$$\mathbf{N} = \frac{\sigma h}{6} \begin{bmatrix} 4 & 1 & 0 & \dots & & & & & \dots & 0 \\ 1 & 4 & 1 & 0 & & & & & & \vdots \\ 0 & 1 & 4 & 1 & 0 & & & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & & & \\ & & & & & & & & & \\ & & & & & & & & & \\ & & & & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & & & & 0 & 1 & 4 & 1 & 0 \\ \vdots & & & & & & & 0 & 1 & 4 & 1 \\ 0 & \dots & & & & & & \dots & 0 & 1 & 4 \end{bmatrix} \in \mathbb{R}^{M-1,M-1} . \tag{9.16.15}$$

The computation of this matrix can be done along the lines of [Lecture → Section 2.3.3].

HINT 1 for (9-16.e): The following code demonstrates the initialization of a *tridiagonal* **Eigen::SparseMat** object from a vector of triplets. Then a linear system with that sparse system matrix is solved by direct sparse elimination using EIGEN's built-in functions.

C++ code 9.16.17: Triplet initialization of a sparse matrix and use of sparse direct solver

```
2  void demo_EigenTripletInit(unsigned int n) {
3    const double diag = 2.0;
4    const double offd = -1.0;
5    // Initialize tridiagonal sparse matrix via COO (Triplet) format
6    std::vector<Eigen::Triplet<double>> X_COO;
7    X_COO.emplace_back(0, 0, diag);
8    X_COO.emplace_back(0, 1, offd);
9    for (int j = 1; j < n - 1; ++j) {
10     X_COO.emplace_back(j, j - 1, offd);
11     X_COO.emplace_back(j, j, diag);
12     X_COO.emplace_back(j, j + 1, offd);
13   }
14   X_COO.emplace_back(n - 1, n - 1, diag);
15   X_COO.emplace_back(n - 1, n - 2, offd);
16   // Build sparse matrix
17   Eigen::SparseMatrix<double> X(n, n);
18   X.setFromTriplets(X_COO.begin(), X_COO.end());
19   // Solve linear system
20   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
21   solver.compute(X);
22   if (solver.info() != Eigen::Success) {
23     throw std::runtime_error("Could not factorize X");
24   }
25   std::cout << "x = "
26             << solver.solve(Eigen::VectorXd::Constant(n, 1.0)).transpose()
27             << std::endl;
28 }
```

SOLUTION of (9-16.e):

The spatial finite element Galerkin discretization of (9.16.1) yields

$$\begin{bmatrix} \mathbf{M} & \mathbf{O} \\ \mathbf{O} & \mathbf{N} \end{bmatrix} \begin{bmatrix} \dot{\vec{\mu}} \\ \dot{\vec{\zeta}} \end{bmatrix} + \begin{bmatrix} \mathbf{O} & \mathbf{B} \\ -\mathbf{C} & \mathbf{D} \end{bmatrix} \begin{bmatrix} \vec{\mu}(t) \\ \vec{\zeta}(t) \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix},$$

where the Galerkin matrices $\mathbf{M}, \mathbf{N}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ arise from the finite-element Galerkin discretization of the bilinear forms $\mathsf{m}, \mathsf{n}, \mathsf{b}, \mathsf{c}, \mathsf{d}$. Rearranging gives us

$$\underbrace{\begin{bmatrix} \mathbf{M} & \mathbf{O} \\ \mathbf{O} & \mathbf{N} \end{bmatrix}}_{=\widehat{\mathbf{M}}} \frac{d}{dt} \begin{bmatrix} \vec{\mu}(t) \\ \vec{\zeta}(t) \end{bmatrix} = \underbrace{\begin{bmatrix} \mathbf{O} & -\mathbf{B} \\ \mathbf{C} & -\mathbf{D} \end{bmatrix}}_{=\widehat{\mathbf{A}}} \begin{bmatrix} \vec{\mu}(t) \\ \vec{\zeta}(t) \end{bmatrix}. \tag{9.16.18}$$

Owing to our choice of basis functions, all the occuring matrices are banded matrices and their entries can be deduced from those of the element matrices computed in Sub-problem (9-16.c) by cell oriented assembly (“in one’s head/on paper”):

$$\begin{aligned} \mathbf{M} &= \rho \operatorname{diag}(h, \ldots, h) \in \mathbb{R}^{M,M}, \\ \mathbf{N} &= \sigma \operatorname{tridiag}(\frac{h}{6}, \frac{2h}{3}, \frac{h}{6}) \in \mathbb{R}^{M-1,M-1}, \\ \mathbf{B} &= \operatorname{bidiag}(-1, 1) \in \mathbb{R}^{M,M-1}, \\ \mathbf{C} &= \operatorname{bidiag}(-\kappa, \kappa) \in \mathbb{R}^{M-1,1}, \\ \mathbf{D} &= \operatorname{tridiag}(\frac{h}{6} - \frac{\mu}{h}, \frac{2h}{3} + \frac{2\mu}{h}, \frac{h}{6} - \frac{\mu}{h}) \in \mathbb{R}^{M-1,M-1}. \end{aligned}$$

The matrices $\mathbf{N}$ and $\mathbf{B}$ have already been computed in Sub-problem (9-16.d), see (9.16.10) and (9.16.11).

Care has to be taken to get the index offsets right:

- The matrix $\mathbf{N}$ has to be placed into $\widehat{\mathbf{M}}$ from index (M, M) (C++ indexing).
- The matrix $\mathbf{B}$ is inserted into $\widehat{\mathbf{A}}$ from index $(0, M)$ (C++ indexing).
- The matrix $\mathbf{C}$ is put into $\widehat{\mathbf{A}}$ from index $(M, 0)$ (C++ indexing)
- The matrix $\mathbf{D}$ has to be placed into $\widehat{\mathbf{A}}$ from index (M, M) (C++ indexing).

C++ code 9.16.19: Implementation of `generateMOLMatrices()`

```
std::pair<std::vector<Eigen::Triplet<double>>,
          std::vector<Eigen::Triplet<double>>>
generateMOLMatrices(unsigned int M, double rho, double sigma, double mu,
                    double kappa) {
  const unsigned int N = 2 * M - 1;  // Total number of d.o.f.s
  const double h = 1.0 / M;          // Meshwidth
  // Triplet vectors to be returned
  std::vector<Eigen::Triplet<double>> Mh_COO;
  std::vector<Eigen::Triplet<double>> Ah_COO;
// I. Matrix M̂
// Initialize diagonal M-part of M̂
  for (int j = 0; j < M; ++j) {
    Mh_COO.emplace_back(j, j, rho * h);
```

```
  }
// Initialize tri-diagonal N-part of M̂
  const double fac = sigma * h / 6.0;
  Mh_COO.emplace_back(M, M, 4.0 * fac);
  Mh_COO.emplace_back(M, M + 1, fac);
  for (int j = 1; j < M - 2; ++j) {
    Mh_COO.emplace_back(j + M, j + M, 4 * fac);
    Mh_COO.emplace_back(j + M, j + M - 1, fac);
    Mh_COO.emplace_back(j + M, j + M + 1, fac);
  }
  Mh_COO.emplace_back(N - 1, N - 1, 4 * fac);
  Mh_COO.emplace_back(N - 1, N - 2, fac);
// II. Matrix Â
// Initialize bi-diagonal −B-part and C-part of Â, see
// (9.16.14)
  Ah_COO.emplace_back(0, M, -1.0);
  Ah_COO.emplace_back(M, 0, kappa);
  for (int j = 1; j < M - 1; ++j) {
    Ah_COO.emplace_back(j, M + j, -1.0);
    Ah_COO.emplace_back(j, M + j - 1, 1.0);
    Ah_COO.emplace_back(M + j, j, kappa);
    Ah_COO.emplace_back(M + j - 1, j, -kappa);
  }
  Ah_COO.emplace_back(M - 1, N - 1, 1.0);
  Ah_COO.emplace_back(N - 1, M - 1, -kappa);
// Initialize tri-diagonal −D-part of Â, see (9.16.12)
  const double diag =
      -2.0 * (h / 3.0 + mu / h);
  const double offd =
      -h / 6.0 + mu / h;
  Ah_COO.emplace_back(M, M, diag);
  Ah_COO.emplace_back(M, M + 1, offd);
  for (int j = 1; j < M - 2; ++j) {
    Ah_COO.emplace_back(M + j, M + j, diag);
    Ah_COO.emplace_back(M + j, M + j - 1, offd);
    Ah_COO.emplace_back(M + j, M + j + 1, offd);
  }
  Ah_COO.emplace_back(N - 1, N - 1, diag);
  Ah_COO.emplace_back(N - 1, N - 2, offd);

  return {Mh_COO, Ah_COO};
}
```

HINT 1 for (9-16.f): Of course, you can call the function `generateMOLMatrices()`. ⌟

SOLUTION of (9-16.f):

Implicit Euler timestepping for (9.16.16) leads to the recursion

$$\begin{aligned} \widehat{\mathbf{M}}(\vec{\boldsymbol{\nu}}^{(k)} - \vec{\boldsymbol{\nu}}^{(k-1)}) &= \tau\widehat{\mathbf{A}}\boldsymbol{\nu}^{\vec{(k)}} \\ \Updownarrow \qquad & \quad , \quad k = 1,\ldots,L \\ \vec{\boldsymbol{\nu}}^{(k)} &= (\widehat{\mathbf{M}} - \tau\widehat{\mathbf{A}})^{-1}\widehat{\mathbf{M}}\vec{\boldsymbol{\nu}}^{(k-1)} \end{aligned} \tag{9.16.20}$$

Note that a linear system of equations with system matrix $\widehat{\mathbf{M}} - \tau\widehat{\mathbf{A}}$ has to be solved in every timestep. For the sake of efficiency an LU-factorization of that matrix should be computed before entering the main timestepping loop.

C++ code 9.16.21: Function `timestepping_GKHeat()`

```
template <typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
std::pair<Eigen::VectorXd, Eigen::VectorXd> timestepping_GKHeat(
    double T, unsigned int L, unsigned int M, const Eigen::VectorXd &mu0,
    const Eigen::VectorXd &zeta0, double rho, double sigma, double mu,
    double kappa, RECORDER &&rec = [](const Eigen::VectorXd & /*nu*/) {}) {
  const double tau = T / L;           // size of timestep
  const unsigned int N = 2 * M - 1;  // Total number of FE d.o.f.s

  assert((void("Size mismatch for mu0"), mu0.size() == M));
  assert((void("Size mismatch for zeta0"), zeta0.size() == M - 1));
  // D.o.f.  vector for timestepping
  Eigen::VectorXd nu(N);
  nu << mu0, zeta0;
  (void)rec(nu);

  // Obtain triplet vectors for MOL ODE matrices
  auto [Mh_COO, Ah_COO] = generateMOLMatrices(M, rho, sigma, mu, kappa);
  Eigen::SparseMatrix<double> Mh(N, N);
  Eigen::SparseMatrix<double> Xh(N, N);
  // Fill sparse matrix M̂
  Mh.setFromTriplets(Mh_COO.begin(), Mh_COO.end());
  // Build matrix X̂ := M̂ − τÂ
  std::vector<Eigen::Triplet<double>> Xh_COO = Mh_COO;
  for (Eigen::Triplet<double> &t : Ah_COO) {
    Xh_COO.emplace_back(t.row(), t.col(), -tau * t.value());
  }
  // Build sparse matrix in CRS format
  Xh.setFromTriplets(Xh_COO.begin(), Xh_COO.end());
  // LU factorization of X̂
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
  solver.compute(Xh);
  if (solver.info() != Eigen::Success) {
    throw std::runtime_error("Could not factorize Xh");
  }
  // Main timestepping loop
  for (int k = 0; k < L; ++k) {
    // Implement (9.16.20)
    nu = solver.solve(Mh * nu);
    (void)rec(nu);
  }
  return {nu.head(M), nu.tail(M - 1)};
```

```
43 }
```

HINT 1 for (9-16.g): Remember that inside a lambda function you can access and even modify local variables of the containing function, if you put them into the comma-separated list of captured variables inside the `[ ]` brackets. If read access is sufficient, just add the plain variable name. If you want to overwrite the variable's value give the prefix `&`. ⌟

SOLUTION of (9-16.g):

For $u_h \in \mathcal{S}_0^{-1}(\mathcal{M})$ and $j_h \in \mathcal{S}_{1,0}^0(\mathcal{M})$ with associated basis expansion coefficient vectors $\vec{\mu} \in \mathbb{R}^M$ and $\vec{\zeta} \in \vec{\mathbb{R}}^{M-1}$ we have

$$\int_0^1 u_h(x)^2 \,\mathrm{d}x = h \,\|\vec{\mu}\|_2^2 ,$$

$$\int_0^1 j_h(x)^2 \,\mathrm{d}x = \frac{h}{6} \vec{\zeta}^\top \begin{bmatrix} 4 & 1 & 0 & \dots & & & & & \dots & 0 \\ 1 & 4 & 1 & 0 & & & & & & \vdots \\ 0 & 1 & 4 & 1 & 0 & & & & & \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & & & \\ & & & & & & & & & \\ & & & & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ & & & & & & 0 & 1 & 4 & 1 & 0 \\ \vdots & & & & & & & 0 & 1 & 4 & 1 \\ 0 & \dots & & & & & & \dots & 0 & 1 & 4 \end{bmatrix} \vec{\zeta} .$$

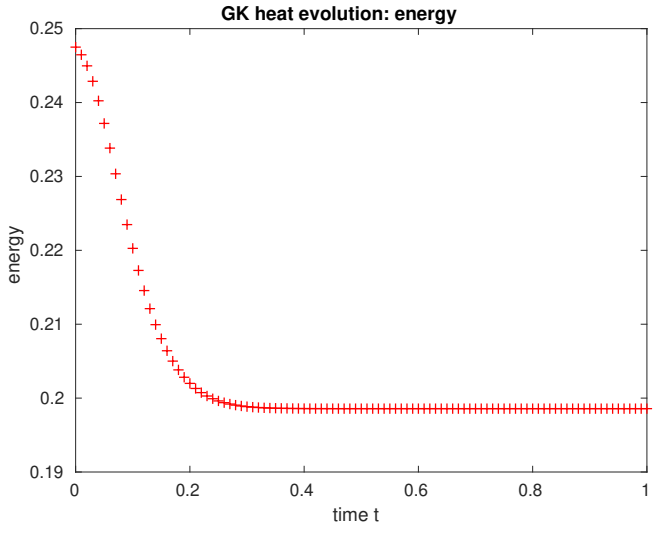
The matrix is just the $\mathcal{S}_{1,0}^0(\mathcal{M})$ mass matrix that we already computed in Sub-problem (9-16.d), see (9.16.15). From this formula we get with $\zeta_j := \left(\vec{\zeta}\right)_j$

$$\int_0^1 j_h(x)^2 \,\mathrm{d}x = \frac{h}{6}\left(4\sum_{j=1}^{M-1} \zeta_j^2 + 2\sum_{j=1}^{M-2} \zeta_j \zeta_{j+1}\right) .$$

The sums can be implemented based on the Euclidean inner product of vectors.

C++ code 9.16.22: Implementation of `track_GKHeatEnergy()`

```
2  std::vector<double> track_GKHeatEnergy(double T, unsigned int L, unsigned int M,
3                                         const Eigen::VectorXd &mu0,
4                                         const Eigen::VectorXd &zeta0, double rho,
5                                         double sigma, double mu, double kappa) {
6    std::vector<double> energies;
7    auto rec = [&energies, M, rho, sigma,
8                kappa](const Eigen::VectorXd &nu) -> void {
9      const double h = 1.0 / M;
10     auto mu = nu.head(M);
11     auto zeta = nu.tail(M - 1);
12     const double E_mu = 0.5 * h * rho * mu.squaredNorm();
13     const double E_zeta =
14         0.5 * sigma / kappa * h / 6.0 *
15         (4.0 * zeta.dot(zeta) + 2.0 * zeta.head(M - 2).dot(zeta.tail(M - 2)));
16     energies.push_back(E_mu + E_zeta);
17   };
18   (void)timestepping_GKHeat(T, L, M, mu0, zeta0, rho, sigma, mu, kappa, rec);
19   return energies;
20 }
```



◁ energies produced by fully discrete evolution for the GK heat equation with $T = 1$, $\kappa = 1$, $\mu = 0.1$, $\rho = 1.0$, $\sigma = 0.2$, $M = 100$, $L = 100$,

$$\vec{\boldsymbol{\mu}}^{(0)} = [\sin(\pi * j/(M-1))]_{j=0}^{M-1} ,$$
$$\vec{\boldsymbol{\zeta}}^{(0)} = \mathbf{0} .$$

After some time the decay of the energy stagnates. This is possible, because $u(x,t) \equiv \text{const}$ and $j(x,t) = 0$ is a non-zero stationary solution of the Guyer-Krumhansl heat equation (9.16.1).

SOLUTION of (9-17.a):

Refer to the general discussion of [Lecture → Section 9.2.4]. Concretely we have

$$\mathbf{M} = \left[\int_{\partial\Omega} b_h^j(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \right]_{i,j=1}^{N}, \quad \mathbf{A} = \left[\int_{\Omega} \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \right]_{i,j=1}^{N}.$$

SOLUTION of (9-17.b):

We consider the semi-discrete evolution in the finite-element space $V_{0,h} := \mathcal{S}_2^0(\mathcal{M})$

$$u_h(t) \in V_{0,h}: \quad \frac{d}{dt}\mathsf{m}(u_h(t), v_h) + \mathsf{a}(u_h, v_h) = 0 \quad \forall v_h \in V_{0,h} \,, \tag{9.17.4}$$
$$\mathsf{m}(u,v) := \int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \quad , \quad \mathsf{a}(u,v) := \int_{\Omega} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,.$$

Set

$$c := \left\{ \boldsymbol{y} \in \Omega \mapsto \frac{1}{|\partial\Omega|} \int_{\partial\Omega} u_{0,h}(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \right\} \tag{9.17.5}$$

for the constant function whose average over $\partial\Omega$ is the same as that of the initial data $u_{0,h}$ of the semi-discrete evolution.

Note that $\mathsf{a}(c, v_h) = 0$ for all $v_h \in V_{0,h}$, which implies that

$$\frac{d}{dt}\mathsf{m}(u_h(t), c) = 0 \quad \Rightarrow \quad t \mapsto \int_{\partial\Omega} u_h(\boldsymbol{x}, t) \,\mathrm{d}S(\boldsymbol{x}) \equiv \text{const} \,; \tag{9.17.6}$$

the average of $u_h(t)$ over $\partial\Omega$ does not change with time. In particular, this implies that $u_h(t) \not\to 0$ for $t \to \infty$, if $c \neq 0$. So the answer to (C) must be NO.

Now set $\overline{u}_h(t) := u_h(t) - c$ and observe that $t \mapsto \overline{u}_h(t)$ solves

$$\frac{d}{dt}\mathsf{m}(\overline{u}_h(t), v_h) + \mathsf{a}(\overline{u}_h(t), v_h) = 0 \,, \quad \overline{u}_h(0) = u_{0,h} - c \,. \tag{9.17.7}$$

Define

$$\overline{V}_{0,h} := \left\{ v_h \in V_{0,h} : \int_{\partial\Omega} v_h(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = 0 \right\} . \tag{9.17.8}$$

Thanks to (9.17.6) we know that $\overline{u}_h(t) \in \overline{V}_{0,h}$ for all times t. Since $\overline{V}_{0,h} \subset V_{0,h}$, $t \mapsto \overline{u}_h(t)$ solves the following semi-discrete parabolic evolution problem in variational form:

$$t \mapsto \overline{w}_h(t): \quad \frac{d}{dt}\mathsf{m}(\overline{w}_h(t), \overline{v}_h) + \mathsf{a}(\overline{w}_h(t), v_h) = 0 \quad \forall v_h \in \overline{V}_{0,h} \,. \tag{9.17.9}$$

Recall the generalization of the Poincaré-Friedrichs inequality:

Theorem [Lecture → Thm. 1.8.0.26]. Poincaré-Friedrichs-type estimate

Let $\Omega \subset \mathbb{R}^d$ a bounded and connected Lipschitz domain and $F : H^1(\Omega) \to W$ a bounded/continuous linear operator, where W is some normed vector space.
If $F(\{\boldsymbol{x} \mapsto 1\}) \neq \mathbf{0}$, then

$$\exists C > 0: \quad \|u\|_{L^2(\Omega)} \leq C\big(\|\mathbf{grad}\, u\|_{L^2(\Omega)} + \|F(u)\|_W\big) \quad \forall u \in H^1(\Omega) \,.$$

Apply this with $F(v) := \int_{\partial\Omega} v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})$ and conclude that

$$\exists C > 0: \quad \|\overline{v}_h\|_{L^2(\Omega)} \le C\|\mathbf{grad}\,\overline{v}_h\|_{L^2(\Omega)} \quad \forall v_h \in \overline{V}_{0.h}\,, \tag{9.17.10}$$

Then by the multiplicative trace inequality

Theorem [Lecture → Thm. 1.9.0.10]. Multiplicative trace inequality

$$\exists C = C(\Omega) > 0: \quad \|u\|^2_{L^2(\partial\Omega)} \le C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega)\,.$$

for any $\overline{v}_h \in \overline{V}_{0,h}$ we obtain the estimate

$$\mathsf{m}(\overline{v}_h,\overline{v}_h) = \left\|v_h^2\right\|^2_{L^2(\partial\Omega)} \le C_1\|\overline{v}_h\|_{L^2(\Omega)}\|\overline{v}_h\|_{H^1(\Omega)} \le C_2\|\mathbf{grad}\,\overline{v}_h\|^2_{L^2(\Omega)} = C_2\mathsf{a}(v_h,v_h)\,, \tag{9.17.11}$$

with some constants $C_1, C_2 > 0$ independent of $\overline{v}_h$.

The key idea is to recognize that (9.17.11) can play the role of [Lecture → (9.2.3.6)] and that this makes it possible to apply a variant of [Lecture → Lemma 9.2.3.8] to (9.17.9), because both m and a are symmetric and positive semi-definite. Following the proof of [Lecture → Lemma 9.2.3.8] we find the exponential decay laws

$$\begin{aligned} \mathsf{m}(\overline{u}_h(t),\overline{u}_h(t)) &\le e^{-\sigma t}\mathsf{m}(\overline{u}_h(0),\overline{u}_h(0)) \quad \forall t \ge 0\,,\\ \mathsf{a}(\overline{u}_h(t),\overline{u}_h(t)) &\le e^{-\sigma t}\mathsf{a}(\overline{u}_h(0),\overline{u}_h(0)) \quad \forall t \ge 0\,, \end{aligned} \tag{9.17.12}$$

with $\sigma = \frac{2}{C_2} > 0$. Hence $\overline{u}_h(t) \to 0$ for $t \to \infty$. As $\mathsf{a}(u_h(t),u_h(t)) = \mathsf{a}(\overline{u}_h(t),\overline{u}_h(t))$, we also have $\mathsf{a}(u_h(t),u_h(t)) \to 0$ for $t \to \infty$.

Now we split the basis of $\mathcal{S}_2^0(\mathcal{M})$/degrees of freedom into two groups: those asscoiated with mesh entities on $\partial\Omega$ and those "located" in the interior of Ω. This induces a splitting of the matrices $\mathbf{M}$ and $\mathbf{A}$ as we have already seen in, for instance, [Lecture → (2.7.6.9)]:

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_{\partial\partial} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} \end{bmatrix} \quad , \quad \mathbf{A} = \begin{bmatrix} \mathbf{A}_{\partial\partial} & \mathbf{A}_{\partial 0} \\ \mathbf{A}_{\partial 0}^\top & \mathbf{A}_{00} \end{bmatrix}\,. \tag{9.17.13}$$

Also the vector $\vec{\boldsymbol{\mu}}(t) \in \mathbb{R}^N$ can be partitioned into a "boundary part" $\vec{\boldsymbol{\mu}}_\partial(t)$ and an interior part $\vec{\boldsymbol{\mu}}_0(t)$. With respect to these splittings the MOL-ODE (9.17.3) reads

$$\mathbf{M}_{\partial\partial}\frac{d\vec{\boldsymbol{\mu}}_\partial}{dt}(t) = -\mathbf{A}_{\partial\partial}\vec{\boldsymbol{\mu}}_\partial(t) - \mathbf{A}_{\partial 0}\vec{\boldsymbol{\mu}}_0(t)\,, \tag{9.17.14a}$$

$$0 = \mathbf{A}_{\partial 0}^\top\vec{\boldsymbol{\mu}}_\partial(t) + \mathbf{A}_{00}\vec{\boldsymbol{\mu}}_0(t)\,. \tag{9.17.14b}$$

This is a linear **differential algebraic equation (DAE)**, a differential equation (9.17.14a) coupled with a linear equation (9.17.14b). Since $\mathbf{A}_{00}$ is regular (the Galerkin matrix for a pure Dirichlet problem for $-\Delta$), we can eliminate $\vec{\boldsymbol{\mu}}_0$ and get

$$\mathbf{M}_{\partial\partial}\frac{d\vec{\boldsymbol{\mu}}_\partial}{dt}(t) = -\mathbf{S}\vec{\boldsymbol{\mu}}_\partial(t) \quad \text{with } \textbf{Schur complement} \quad \mathbf{S} := \mathbf{A}_{\partial\partial} - \mathbf{A}_{\partial 0}\mathbf{A}_{00}^{-1}\mathbf{A}_{\partial 0}^\top\,. \tag{9.17.15}$$

Since $\mathcal{N}(\mathbf{A}) = \mathrm{Span}\{\mathbf{1}\}$ we also have $\mathcal{N}(\mathbf{S}) = \mathrm{Span}\{\mathbf{1}\}$. In addition $\mathbf{S}$ is positive semi-definite, $\mathbf{M}_{\partial\partial}$ is s.p.d., and on the orthogonal complement of $\mathbf{1}$ we have

$$\exists\gamma > 0: : \quad \gamma\,\vec{\boldsymbol{\eta}}_\partial^\top\mathbf{M}_{\partial\partial}\vec{\boldsymbol{\eta}}_\partial \le \vec{\boldsymbol{\eta}}_\partial^\top\mathbf{S}\vec{\boldsymbol{\eta}} \quad \forall\vec{\boldsymbol{\eta}}_\partial : \ \mathbf{1}^\top\vec{\boldsymbol{\eta}}_\partial = 0\,. \tag{9.17.16}$$

Based on similar argument as in the proof of [Lecture → Lemma 9.2.3.8] we conclude that

$$\vec{\mu}^*_\partial(t) := \vec{\mu}_\partial(t) - \mathbf{1}^\top \vec{\mu}_\partial(1)\vec{\mu}_\partial(t)$$

satisfies

$$\big(\vec{\mu}^*_\partial(t)\big)^\top \mathbf{M}_{\partial\partial}\vec{\mu}^*_\partial(t) \le e^{-\gamma t}\big(\vec{\mu}^*_\partial(0)\big)^\top \mathbf{M}_{\partial\partial}\vec{\mu}^*_\partial(0) \quad , \quad \big(\vec{\mu}^*_\partial(t)\big)^\top \mathbf{S}\vec{\mu}^*_\partial(t) \le e^{-\gamma t}\big(\vec{\mu}^*_\partial(0)\big)^\top \mathbf{S}\vec{\mu}^*_\partial(0)\,. \tag{9.17.17}$$

By virtue of the ODE (9.17.15) this implies

$$\frac{d\vec{\mu}_\partial}{dt}(t) = -\mathbf{M}^{-1}_{\partial\partial}\mathbf{S}\vec{\mu}_\partial(t) \to 0 \quad \text{for} \quad t\to\infty\,, \tag{9.17.18}$$

$$\Rightarrow \qquad \frac{d\vec{\mu}}{dt}(t) = \begin{bmatrix} \vec{\mu}_{\partial\partial} \\ -\mathbf{A}^{-1}_{00}\mathbf{A}^\top_{\partial 0} \end{bmatrix} \vec{\mu}_\partial(t) \to 0 \quad \text{for} \quad t\to\infty\,. \tag{9.17.19}$$

Thus, the right answer to (C) is “YES”.

Finally, neither $\mathbf{M}$ nor $\mathbf{A}$ are positive semi-definite, because both bilinear forms m and a have a nontrivial kernel:

$$\mathcal{N}(\mathbf{A}) = \{\mathbf{1}\} \quad , \quad \mathcal{N}(\mathbf{B}) = \Big\{\vec{\xi}\in\mathbb{R}^N : \ (\xi)_i = 0 \text{ if node } i \text{ is on } \partial\Omega\Big\}\,.$$

(A)	Is the matrix $\mathbf{M}$ is symmetric, positive definite?	NO
(B)	Is the matrix $\mathbf{A}$ is symmetric, positive definite?	NO
(C)	Does $\vec{\mu}(t)$ converge to $\mathbf{0}$ for $t\to\infty$?	NO
(D)	Does $\frac{d\vec{\mu}}{dt}(t)$ converge to $\mathbf{0}$ for $t\to\infty$?	YES
(E)	Does $(\mu(t))^\top \mathbf{A}\vec{\mu}(t)$ converge to $\mathbf{0}$ for $t\to\infty$?	YES

HINT 1 for (9-17.c):

Definition [Lecture → Def. 7.3.3.1]. General Runge-Kutta single step method

For $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, a single step of size $\tau > 0$ of an **s-stage Runge-Kutta single step method** (RK-SSM) for the IVP $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$, is defined by

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i\tau, \mathbf{y}_0 + \tau\sum_{j=1}^{s} a_{ij}\mathbf{k}_j)\,, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + \tau\sum_{i=1}^{s} b_i\mathbf{k}_i\,.$$

As before, the vectors $\mathbf{k}_i \in \mathbb{R}^N$ are called **increments**.

Shorthand notation for an s-stage Runge-Kutta methods

Butcher scheme ▷ $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array}$.

$\mathfrak{A} = \left[a_{ij}\right]_{i,j=1}^{s}$ can be a general $s \times s$-matrix.

⌟

SOLUTION of (9-17.c):

We simply recall the arguments of [Lecture → § 9.2.7.6] and [Lecture → (9.2.7.7)], which give us for the concrete case of the SDIRK SSM the following increment equations:

$$\begin{aligned}\mathbf{M}\vec{\kappa}_1 + \tau\gamma\mathbf{A}\vec{\kappa}_1 &= -\mathbf{A}\vec{\mu}^{(k)} \,,\\ \mathbf{M}\vec{\kappa}_2 + \tau(1-2\gamma)\mathbf{A}\vec{\kappa}_1 + \tau\gamma\mathbf{A}\vec{\kappa}_2 &= -\mathbf{A}\vec{\mu}^{(k)} \,.\end{aligned}$$

This is equivalent to

$$\begin{bmatrix} \mathbf{M}+\tau\gamma\mathbf{A} & \mathbf{O} \\ \tau(1-2\gamma)\mathbf{A} & \mathbf{M}+\tau\gamma\mathbf{A} \end{bmatrix} \begin{bmatrix} \vec{\kappa}_1 \\ \vec{\kappa}_2 \end{bmatrix} = \begin{bmatrix} -\mathbf{A}\vec{\mu}^{(k)} \\ -\mathbf{A}\vec{\mu}^{(k)} \end{bmatrix} . \tag{9.17.22}$$

SOLUTION of (9-17.d):

Obviously, both the spatial and the temporal error have to be reducced by a factor of 100 alike, because they *add up* to the total error.

❶ Spatial error: Uniform regular refinement involves halving of $h_{\mathcal{M}}$. By (9.17.23) $h_{\mathcal{M}} \to h_{\mathcal{M}}/2$ leads to a drop of the spatial error by a factor of 8. Since $8^2 < 100$, but $8^3 > 100$, we need three steps of mesh refinement.

❷ Temporal error: Denote by $q \in]0,1[$ the reduction of the timestep. By (9.17.23) it has to satisfy

$$q^2 = 0.01 \quad \Leftrightarrow \quad q = 0.1 \quad \Leftrightarrow \quad \text{reduction factor for } \tau = 10 \ .$$

SOLUTION of (9-18.a):

We follow the usual approach first seen in [Lecture → Ex. 1.5.3.11].

❶ First we test with $v \in C_0^\infty(\Omega)$ and apply Green's formula

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ and functions $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ holds

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \tag{9.18.2}$$

with $\mathbf{j} := \mathbf{grad}\, u$ to the term $\int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ noting that no boundary terms pop up, since $v|_{\partial\Omega} = 0$. This yields

$$\int_\Omega \left(\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) - \Delta u(\boldsymbol{x}) \right) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega)\,, \tag{9.18.3}$$

and by the fundamental lemma of the calculus of variations

Lemma [Lecture → Lemma 1.5.3.4]. Fundamental lemma of calculus of variations in higher dimensions

If $f \in L^2(\Omega)$ satisfies

$$\int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega)\,,$$

then $f \equiv 0$ can be concluded.

we conclude the PDE, the plain and simple linear wave equation:

$$\frac{\partial^2 u}{\partial t^2} - \Delta u = 0 \quad \text{in} \quad \Omega \times]0,T[\,. \tag{9.18.4}$$

❷ Next we test with general $v \in C^\infty(\overline{\Omega})$, again applying Green's formula to the term $\int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}$ which yields

$$\int_\Omega \left(\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) - \Delta u(\boldsymbol{x},t) \right) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) + \int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t) v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in C^\infty(\overline{\Omega})\,. \tag{9.18.5}$$

The integrand of $\int_\Omega \ldots \mathrm{d}\boldsymbol{x}$ vanishes thanks to (9.18.4), leaving us with

$$\int_{\partial\Omega} \left(\mathbf{grad}\, u(\boldsymbol{x},t) \cdot \boldsymbol{n}(\boldsymbol{x}) + \frac{\partial u}{\partial t}(\boldsymbol{x},t) \right) v(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in C^\infty(\overline{\Omega})\,. \tag{9.18.6}$$

Appealing to a "fundamental lemma of calculus of variations on $\partial\Omega$" we conclude the boundary condition

$$\mathbf{grad}\, u \cdot \boldsymbol{n} + \frac{\partial u}{\partial t} = 0 \quad \text{on} \quad \partial\Omega \times]0,T[\,, \tag{9.18.7}$$

where $\boldsymbol{n} : \partial\Omega \to \mathbb{R}^2$ denotes the exterior unit normal vector field on $\partial\Omega$.

❸ The initial condition can just be copied from (9.18.1b).

SOLUTION of (9-18.b):

To prove $\mathcal{E}(t) \leq \mathcal{E}(0)$ we show that $\dfrac{d\mathcal{E}}{dt}(t) \leq 0$. By the product rule for symmetric bilinear expressions

$$\begin{aligned}\frac{d\mathcal{E}}{dt}(t) &= \int_\Omega \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\frac{\partial u}{\partial t}(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\frac{\partial}{\partial t}\mathbf{grad}\, u(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x}\\ &= -\int_{\partial\Omega}\left|\frac{\partial u}{\partial t}(\boldsymbol{x},t)\right|^2 \mathrm{d}\boldsymbol{x} \leq 0\ ,\end{aligned}$$

from the the variational equation (9.18.1a) with $v(\boldsymbol{x}) := \frac{\partial u}{\partial t}(\boldsymbol{x},t)$.

SOLUTION of (9-18.c):

Following the considerations of [Lecture → Section 9.2.4] and [Lecture → Section 9.3.3] we obtain the formulas

$$(\mathbf{M})_{i,j} = \left(\int_\Omega b_h^j(\boldsymbol{x}) b_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right)_{i,j=1}^N . \quad i,j \in \{1,\ldots,N\} , \tag{9.18.10a}$$

$$(\mathbf{B})_{i,j} = \left(\int_{\partial\Omega} b_h^j(\boldsymbol{x}) b_h^i(\boldsymbol{x}) \,\mathrm{d}S(\boldsymbol{x}) \right)_{i,j=1}^N . \quad i,j \in \{1,\ldots,N\} , \tag{9.18.10b}$$

$$(\mathbf{A})_{i,j} = \left(\int_\Omega \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right)_{i,j=1}^N . \quad i,j \in \{1,\ldots,N\} . \tag{9.18.10c}$$

This reveals that

- $\mathbf{M}$ is the standard mass matrix [Lecture → Rem. 2.2.2.5] induced by the $L^2(\Omega)$ inner product,
- $\mathbf{B}$ is a "boundary mass matrix", *cf.* [Lecture → Ex. 2.7.4.41],
- $\mathbf{A}$ is a finite-element Galerkin matrix for $-\Delta$.

HINT 1 for (9-18.d): Of course you may employ the LEHRFEM++ **ENTITY_MATRIX_PROVIDER** classes **lf::uscalfe::ReactionDiffusionElementMatrixProvider** and **lf::fe::MassEdgeMatrixProvider**.

⌟

SOLUTION of (9-18.d):

As regards $\mathbf{A}$ and $\mathbf{M}$, we can take the cue from [Lecture → Ex. 2.8.3.29] and use **lf::uscalfe::ReactionDiff** noting that the coefficient functions α and γ are constant now: $\alpha \equiv 1$, $\gamma \equiv 0$.

For the assembly of $\mathbf{B}$ refer to [Lecture → Ex. 2.7.4.41].

C++ code 9.18.11: Implementation of `computeGalerkinMatrices()`

```
2  std::tuple<lf::assemble::COOMatrix<double>, lf::assemble::COOMatrix<double>,
3             lf::assemble::COOMatrix<double>>
4  computeGalerkinMatrices(
5      std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fe_space_p) {
6    // Pointer to current mesh
7    std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
8    // Obtain local->global index mapping for current finite element space
9    const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
10   // Dimension of finite element space
11   const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
12   // Instantiating N_dofs x N_dofs zero matrices in triplet format
13   lf::assemble::COOMatrix<double> M_COO(N_dofs, N_dofs);
14   lf::assemble::COOMatrix<double> B_COO(N_dofs, N_dofs);
15   lf::assemble::COOMatrix<double> A_COO(N_dofs, N_dofs);
16   /* Creating dummy coefficient-functions as Lehrfem++ mesh functions */
17   // These coefficient-functions are needed in the class templates
18   // ReactionDiffusionElementMatrixProvider and MassEdgeMatrixProvider
19   auto zero_mf = lf::mesh::utils::MeshFunctionConstant(0.0);
20   auto one_mf = lf::mesh::utils::MeshFunctionConstant(1.0);
21   /* Initialization of local matrices builders */
22   // Initialize objects taking care of local computations for volume integrals
23   lf::uscalfe::ReactionDiffusionElementMatrixProvider M_locmat_builder(
24       fe_space_p, zero_mf, one_mf);
25   lf::uscalfe::ReactionDiffusionElementMatrixProvider A_locmat_builder(
26       fe_space_p, one_mf, zero_mf);
27   // Initialize objects taking care of local computations for boundary integrals
28   // Creating a predicate that will guarantee that the computations for the
29   // boundary mass matrix are carried only on the edges of the mesh
30   lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
31       lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
32   lf::uscalfe::MassEdgeMatrixProvider B_locmat_builder(fe_space_p, one_mf,
33                                                        bd_flags);
34   /* Assembling the Galerkin matrices */
35   // Information about the mesh and the local-to-global map is passed through
36   // a Dofhandler object, argument 'dofh'. This function call adds triplets to
37   // the internal COO-format representation of the sparse matrix.
38   // Invoke assembly on cells (co-dimension = 0 as first argument)
39   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, M_locmat_builder, M_COO);
40   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, A_locmat_builder, A_COO);
41   // Invoke assembly on edges (co-dimension = 1 as first argument)
42   lf::assemble::AssembleMatrixLocally(1, dofh, dofh, B_locmat_builder, B_COO);
43   return {M_COO, B_COO, A_COO};
44 }
```

SOLUTION of (9-18.e):

Let's examined a shortened table with M doubling from row to row $\triangleright$

M	$\lvert y(1) - y_M \rvert$
10	0.0011660874580030
20	0.0002911403222123
40	0.0000727612849977
80	0.0000181888348186
160	0.0000045471158128
320	0.0000011367731537
640	0.0000002841929364

Evidently, the error at final time drops by a factor ~ 4 as we double M (or halve the timestep size τ).

We conjecture algebraic covnergence $O(\tau^2)$ as $\tau \to 0$, which indicates that the timestepping scheme (9.18.12), (9.18.13) provides a method of order 2 .

SOLUTION of (9-18.f):

The following equivalent form of (9.18.12) results from simple linear-algebra manipulations:

$$\vec{\boldsymbol{\mu}}^{(j)} = \vec{\boldsymbol{\mu}}^{(j-1)} + \tau\vec{\boldsymbol{v}}^{j-\frac{1}{2}}\ , \quad j = 1,\ldots,M\ , \tag{9.18.16}$$

$$\vec{\boldsymbol{v}}^{(j+\frac{1}{2})} = (\mathbf{M} + \tfrac{1}{2}\tau\mathbf{B})^{-1}(-\tau\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} + (\mathbf{M} - \tfrac{1}{2}\tau\mathbf{B})\vec{\boldsymbol{v}}^{(j-\frac{1}{2})})\ , \quad j = 1,\ldots,M\ . \tag{9.18.17}$$

From (9.18.13) we get $\vec{\boldsymbol{v}}^{(-\frac{1}{2})} = 2\vec{\boldsymbol{v}}_0 - \vec{\boldsymbol{v}}^{(\frac{1}{2})}$. We plug this into the second equation of (9.18.12) for $j = 0$ and get

$$\vec{\boldsymbol{v}}^{(\frac{1}{2})} = \vec{\boldsymbol{v}}_0 - \tfrac{1}{2}\tau\mathbf{M}^{-1}(\mathbf{A}\vec{\boldsymbol{\mu}}_0 + \mathbf{B}\vec{\boldsymbol{v}}_0)\ . \tag{9.18.18}$$

HINT 1 for (9-18.g): The function `computeGalerkinMatrices()` as introduced in Sub-problem (9-1
can be used. ⌟

SOLUTION of (9-18.g):

Inspecting (9.18.16) and (9.18.17) we notice that the vector $\vec{\mu}^{(j-1)}$ will never be used again, once $\vec{\mu}^{(j)}$ has been computed, and that we can discard $\vec{v}^{(j-\frac{1}{2})}$ as soon as we have used it to get $\vec{v}^{(j+\frac{1}{2})}$. This makes it possible to overwrite the state vectors, because we can rewrite (9.18.16) and (9.18.17) as

$$\vec{\mu} \leftarrow \vec{\mu} + \tau\vec{v}\,, \quad j = 1,\ldots,M\,, \tag{9.18.19}$$

$$\vec{v} \leftarrow (\mathbf{M} + \tfrac{1}{2}\tau\mathbf{B})^{-1}(-\tau\mathbf{A}\vec{\mu} + (\mathbf{M} - \tfrac{1}{2}\tau\mathbf{B})\vec{v})\,, \quad j = 1,\ldots,M\,, \tag{9.18.20}$$

after the initialization $\vec{\mu} := \vec{\mu}_0$, $\vec{v} := \vec{v}^{(\frac{1}{2})}$. The latter vector is available from (9.18.18).

C++ code 9.18.21: Implementation of `timestepDissipativeWaveEquation()`

```
Eigen::VectorXd timestepDissipativeWaveEquation(
    std::shared_ptr<const lf::uscalfe::FeSpaceLagrangeO2<double>> fe_space_p,
    double T, unsigned int M, Eigen::VectorXd mu0, Eigen::VectorXd nu0) {
  // Pointer and reference to current mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
  const lf::mesh::Mesh &mesh{*mesh_p};
  // Obtain local->global index mapping
  // for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
  // Dimension of finite element space = number of nodes of the mesh
  const lf::base::size_type N_dofs(dofh.NumDofs());
  LF_VERIFY_MSG(mu0.size() == N_dofs, "Wrong length of mu0");
  LF_VERIFY_MSG(nu0.size() == N_dofs, "Wrong length of nu0");
  // The solution vector at time T
  Eigen::VectorXd mu = mu0;

  // Obtain matrices in triplet format
  const auto [M_COO, B_COO, A_COO] = computeGalerkinMatrices(fe_space_p);
  LF_VERIFY_MSG((M_COO.cols() == N_dofs) && (M_COO.rows() == N_dofs),
                "Wrong size of M");
  LF_VERIFY_MSG((B_COO.cols() == N_dofs) && (B_COO.rows() == N_dofs),
                "Wrong size of B");
  LF_VERIFY_MSG((A_COO.cols() == N_dofs) && (A_COO.rows() == N_dofs),
                "Wrong size of A");
  // For convenience convert in CRS format in order to be able to exploit
  // Eigen's linear-algebra operations.
  const Eigen::SparseMatrix<double> M_crs{M_COO.makeSparse()};
  const Eigen::SparseMatrix<double> B_crs{B_COO.makeSparse()};
  const Eigen::SparseMatrix<double> A_crs{A_COO.makeSparse()};
  // Build sparse matrices M+0.5*tau*B and M-0.5*tau*B
  // Timestep size
  const double tau = T / M;
  const std::vector<Eigen::Triplet<double>> &B_trp = B_COO.triplets();
  std::vector<Eigen::Triplet<double>> MBp_trp = M_COO.triplets();
  std::vector<Eigen::Triplet<double>> MBm_trp = M_COO.triplets();
  for (auto &triplet : B_trp) {
    MBp_trp.emplace_back(triplet.row(), triplet.col(),
                         0.5 * tau * triplet.value());
    MBm_trp.emplace_back(triplet.row(), triplet.col(),
                         -0.5 * tau * triplet.value());
  }
  Eigen::SparseMatrix<double> MBp(N_dofs, N_dofs);
```

```
44    MBp.setFromTriplets(MBp_trp.begin(), MBp_trp.end());
45    Eigen::SparseMatrix<double> MBm(N_dofs, N_dofs);
46    MBm.setFromTriplets(MBm_trp.begin(), MBm_trp.end());
47    // We have to solve a linear system with the matrix M+0.5*tau*B
48    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_MBp;
49    // LU decomposition, important: outside main timestepping loop
50    solver_MBp.compute(MBp);
51    LF_VERIFY_MSG(solver_MBp.info() == Eigen::Success,
52                  "LU decomposition of M+0.5*tau*B failed");
53    // The auxiliary vector in the timestepping loop
54    Eigen::VectorXd nu(N_dofs);
55    // Special initial step
56    {
57      Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_M;
58      solver_M.compute(M_crs);  // LU decomposition
59      LF_VERIFY_MSG(solver_M.info() == Eigen::Success,
60                    "LU decomposition of M failed");
61      nu = nu0 - 0.5 * tau * solver_M.solve(A_crs * mu0 + B_crs * nu0);
62      LF_VERIFY_MSG(solver_M.info() == Eigen::Success, "Solving LSE failed");
63    }
64    // Main timestepping loop
65    for (int j = 1; j <= M; ++j) {
66      mu += tau * nu;
67      nu = solver_MBp.solve(MBm * nu - tau * A_crs * mu);
68    }
69    return mu;
70  }
```

SOLUTION of (9-18.h):

For evolution problems discretized by the method of lines we expect [Lecture → § 9.3.5.9]

total discretization error	=	spatial error	+	temporal error

provided that the timestep is small enough such that timestepping is stable.

- Spatial error: The finite-element Galerkin error is determined by the best approximation error in $\mathcal{S}_2^0(\mathcal{M}_\ell)$. According to [Lecture → Thm. 3.3.5.6] we have

$$\inf_{v_h \in \mathcal{S}_2^0(\mathcal{M}_\ell)} |u(\cdot,t) - v_h|_{H^1(\Omega)} = O(h_\ell^2) \quad \text{for} \quad \ell \to \infty \,,$$

  where h_ℓ is the meshwidth of $\mathcal{M}_\ell$, for which we know $h_\ell \approx 2^{-\ell}$.

- In Sub-problem (9-18.e) we found that the dissipative leapfrog timestepping scheme is a method of order 2. This implies that the timestepping error can be expected to behave like $O(M_\ell^{-2})$ for $\ell \to \infty$.

Adding both error contributions add expressing h_ℓ and M_ℓ in terms of ℓ, we arrive a the following conjectured asymptotic behavior of the error norm:

$$\left| u(\cdot,T) - u_\ell^{(M_\ell)} \right|_{H^1(\Omega)} = O(h_\ell^2 + M_\ell^{-2}) = O(4^{-\ell}) \quad \text{for} \quad \ell \to \infty \,.$$

SOLUTION of (9-19.a):

We harness the techniques of [Lecture → Section 1.5.3], in particular Green's first formula [Lecture → Thm. 1.5.2.7], which is applied to the second integral to lift the derivatives off the test function and onto the trial function. Since functions in the test space do not vanish in $\partial\Omega$ we follow the two-step approach of [Lecture → Ex. 1.5.3.11].

❶ Test with $v \in C_0^\infty(\Omega)$, followed by integration by parts,

$$\int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = -\int_\Omega \Delta u(\boldsymbol{x},t)\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega)\ .$$

The time-dependent term and boundary terms vanish, since $v|_{\partial\Omega} = 0$. By the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4] we conclude that

$$\Delta_{\boldsymbol{x}} u(\boldsymbol{x},t) = 0 \quad \text{in} \quad \Omega\ . \tag{9.19.3}$$

❷ Now we test with $v \in C^\infty(\overline{\Omega})$, carry out integration by parts, and use (9.19.3) to get rid of volume terms:

$$\begin{aligned}&\int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\\ &= \int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) - \int_\Omega \underbrace{\Delta u(\boldsymbol{x},t)}_{=0}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0\end{aligned} \tag{9.19.4}$$

for all $v \in C^\infty(\overline{\Omega})$,

$$\int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x}) + \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in C^\infty(\overline{\Omega})\ .$$

Invoking the fundamental lemma of the calculus of variations on the boundary $\partial\Omega$ we end up with the **boundary conditions**

$$\frac{\partial u}{\partial t}(\boldsymbol{x},t) + \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \text{for} \quad \boldsymbol{x} \in \partial\Omega\ .$$

SOLUTION of (9-19.b):

Refer to the general discussion of [Lecture → Section 9.2.4]. Concretely we have

$$\mathbf{M} = \left[\int_{\partial\Omega} b_h^j(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \right]_{i,j=1}^{N}, \quad \mathbf{A} = \left[\int_{\Omega} \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \right]_{i,j=1}^{N}.$$

SOLUTION of (9-19.c):

We first note that $\mathbf{M}$ and $\mathbf{A}$ are obviously *symmetric*. We recall from linear algebra the following results:

Lemma 9.19.6. Kernel and Image of symmetric matrices

If $\mathbf{X} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$, is symmetric, $\mathbf{X}^\top = \mathbf{X}$, then

(i) $\mathcal{N}(\mathbf{X}) = \left\{ \vec{\boldsymbol{v}} \in \mathbb{R}^N : \vec{\boldsymbol{v}}^\top \mathbf{X} \vec{\boldsymbol{v}} = 0 \right\}$, *and*

(ii) $\mathcal{R}(\mathbf{X}) = \mathcal{N}(\mathbf{X})^\perp := \left\{ \vec{\boldsymbol{v}} \in \mathbb{R}^N : \vec{\boldsymbol{v}}^\top \vec{\boldsymbol{\zeta}} = 0 \ \forall \vec{\boldsymbol{\zeta}} \in \mathcal{N}(\mathbf{X}) \right\}$.

❶ kernel and range of $\mathbf{M}$:

Let us partition the set $\mathfrak{B}_h \subset \mathcal{S}_2^0(\mathcal{M})$ of global shape functions of $\mathcal{S}_2^0(\mathcal{M})$ into

(i) into the set $\mathfrak{B}_\partial$ of shape functions associated with geometric entities on $\partial\Omega$, and

(ii) the set $\mathfrak{B}_0$ of global shape functions belonging to interior mesh entities.

Sorting the basis functions according to this partitioning of $\mathfrak{B}_h$ we obtain a block-partitioning of $\mathbf{M}$:

$$\begin{bmatrix} \mathbf{M}_{\partial\partial} & \mathbf{M}_{0\partial} \\ \mathbf{M}_{\partial 0} & \mathbf{M}_{00} \end{bmatrix} = \begin{bmatrix} \mathbf{M}_{\partial\partial} & \mathbf{O} \\ \mathbf{O} & \mathbf{O} \end{bmatrix}, \tag{9.19.7}$$

where $\mathbf{M}_{\partial\partial}$ is symmetric and positive definite (s.p.d.). From this block partitioning we immediately conclude that

$$\mathcal{N}(\mathbf{M}) = \operatorname{Span}\{\boldsymbol{e}_i : b_h^i \in \mathfrak{B}_0\} \quad , \quad \mathcal{R}(\mathbf{M}) = \operatorname{Span}\{\boldsymbol{e}_i : b_h^i \in \mathfrak{B}_\partial\} = \mathcal{N}(\mathbf{M})^\perp ,$$

where $\boldsymbol{e}_i$ is the i-th Cartesian coordinate vector of $\mathbb{R}^N$. We can write this more explicitly

$$\mathcal{N}(\mathbf{M}) = \operatorname{Span}\left\{ \boldsymbol{e}_i : \ b_h^i \text{ belongs to a mesh entity in the interior of } \Omega \right\} ,$$

$$\mathcal{R}(\mathbf{M}) = \operatorname{Span}\left\{ \boldsymbol{e}_i : \ b_h^i \text{ belongs to a mesh entity on } \partial\Omega \right\} .$$

❷ kernel and range of $\mathbf{A}$:

Since $\mathbf{A}$ is the $\mathcal{S}_2^0(\mathcal{M})$-Galerkin matrix for the bilinear form $(u, v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \, \mathrm{d}\boldsymbol{x}$ and the constant function $\{\boldsymbol{x} \mapsto 1\} \in \mathcal{S}_2^0(\mathcal{M})$ has the basis expansion coefficient vector $\mathbf{1} \in \mathbb{R}^N$, we conclude $\mathbf{A1} = \mathbf{0} \Leftrightarrow \mathbf{1} \in \mathcal{N}(\mathbf{A})$ ($\mathbf{1}$ is the vector with entries all $= 1$).

The bilinear form is s.p.d. on the space of functions in $H^1(\Omega)$ with vanishing average. Hence, no non-constant function can be in its nullspace, and no vector that is not a multiple of $\mathbf{1}$ can be in the kernel of $\mathbf{A}$:

$$\blacktriangleright \quad \mathcal{N}(\mathbf{A}) = \operatorname{Span}\{\mathbf{1}\} .$$

It is a fact taught in linear algebra that for every symmetric matrix $\mathbf{S} \in \mathbb{R}^{N,N}$

$$\mathcal{N}(\mathbf{S}) = \mathcal{R}(\mathbf{S})^\perp , \quad \text{where } \perp \text{ designates an (Euclidean) } \textbf{orthogonal complement} .$$

From this we conclude

$$\mathcal{R}(\mathbf{A}) = \left\{ \vec{v} \in \mathbb{R}^N : \mathbf{1}^\top \vec{v} = \sum_{i=1}^{N} (\vec{v})_i = 0 \right\} .$$

SOLUTION of (9-19.d):

Multiply

$$\mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \mathbf{0} \tag{9.19.9}$$

from the left with $\mathbf{1}^\top$ and recall that $\mathbf{A1} = \mathbf{0}$, which, thanks to the symmetry of $\mathbf{A}$, implies $\mathbf{1}^\top\mathbf{A} = \mathbf{0}^\top$.

We get

$$\frac{d}{dt}\mathbf{1}^\top\mathbf{M}\vec{\boldsymbol{\mu}}(t) = 0 \quad \Leftrightarrow \quad \mathbf{1}^\top\mathbf{M}\vec{\boldsymbol{\mu}}(t) = \text{const}\,.$$

Hence, a possible choice for L is

$$L(\vec{\boldsymbol{\nu}}) = \xi\,\mathbf{1}^\top\mathbf{M}\vec{\boldsymbol{\nu}}\,, \quad \vec{\boldsymbol{\nu}} \in \mathbb{R}^N \quad \text{for any} \quad \xi \in \mathbb{R}\,. \tag{9.19.10}$$

Remark. Is this the only solution? Let the linear functional $L : \mathbb{R}^N \to \mathbb{R}$, $L(\vec{\boldsymbol{\nu}}) := \vec{\boldsymbol{\lambda}}^\top\vec{\boldsymbol{\nu}}$, $\vec{\boldsymbol{\lambda}} \in \mathbb{R}^N$, satisfy (9.19.8), which means

$$0 = \frac{d}{dt}L(\vec{\boldsymbol{\mu}}(t)) = \vec{\boldsymbol{\lambda}}^\top\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) \quad \forall t \in [0,T]\,. \tag{9.19.11}$$

This is a consequence of the chain rule. Now, for the sake of simplicity, *assume* that $\mathbf{M}$ is invertible, which implies $\frac{d}{dt}\vec{\boldsymbol{\mu}}(t) = -\mathbf{M}^{-1}\mathbf{A}\vec{\boldsymbol{\mu}}(t)$. We can convert (9.19.11) into

$$0 = \vec{\boldsymbol{\lambda}}^\top\mathbf{M}^{-1}\mathbf{A}\vec{\boldsymbol{\mu}}(t) \quad \forall t \in [0,T]\,, \tag{9.19.12}$$

which is equivalent to

$$\begin{aligned} 0 = \vec{\boldsymbol{\lambda}}^\top\mathbf{M}^{-1}\mathbf{A}\vec{\boldsymbol{\nu}} \quad \forall\vec{\boldsymbol{\nu}} \in \mathbb{R}^N \quad &\Leftrightarrow \quad \vec{\boldsymbol{\lambda}} \in \mathcal{N}(\mathbf{A}\mathbf{M}^{-1}) \\ &\Leftrightarrow \quad \mathbf{M}^{-1}\vec{\boldsymbol{\lambda}} \in \mathcal{N}(\mathbf{A}) = \text{Span}\{\mathbf{1}\} \quad \Leftrightarrow \quad \vec{\boldsymbol{\lambda}} \in \text{Span}\{\mathbf{M1}\}\,. \end{aligned}$$

Thus (9.19.10) is recovered, which is the only possible solution, as can be seen by a more refined analysis based on the decomposition (9.19.7).

HINT 1 for (9-19.e): Rely on the **ENTIY_MATRIX_PROVIDER** class **lf::fe::MassEdgeMatrixProvider** and refer to [Lecture → Code 2.7.4.46]. ⌟

HINT 2 for (9-19.e): Looking at the function `buildA()` also implemented in `irkdegenerateevl.cc` may give you some ideas. ⌟

SOLUTION of (9-19.e):

The key LEHRFEM++-built-in tool is **lf::fe::MassEdgeMatrixProvider**. It is important to perform assembly over edges = mesh entities of co-dimension 1 and to restrict assembly to edges on the boundary. This can be done based on a flag array filled by calling `flagEntitiesOnBoundary()`.

C++ code 9.19.13: Implementation of `buildM()`

```
2  lf::assemble::COOMatrix<double> buildM(
3      std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>> fes_p) {
4    // Extract mesh
5    const std::shared_ptr<const lf::mesh::Mesh> mesh_p{fes_p->Mesh()};
6    // Fetch DofHandler
7    const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
8    const size_t N = dofh.NumDofs();
9    // Sparse matrix in triplet format
10   lf::assemble::COOMatrix<double> M_coo(N, N);
11   // Flag edges on the boundary
12   lf::mesh::utils::CodimMeshDataSet<bool> bd_flags{
13       lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
14   // Set up ENTITY_MATRIX_PROVIDER
15   lf::mesh::utils::MeshFunctionGlobal mf_one{
16       [](Eigen::Vector2d /*x*/) -> double { return 1.0; }};
17   lf::uscalfe::MassEdgeMatrixProvider<double, decltype(mf_one),
18                                       decltype(bd_flags)>
19       memp(fes_p, mf_one, bd_flags);
20   // Assembly of matrix M
21   lf::assemble::AssembleMatrixLocally(1, dofh, dofh, memp, M_coo);
22   return M_coo;
23 }
```

SOLUTION of (9-19.f):

The code implements the increment form of the IRK-SSM recursion for (9.19.5),

$$\vec{\kappa}_i \in \mathbb{R}^N: \quad \mathbf{M}\vec{\kappa}_i + \sum_{m=1}^{s} \tau a_{im}\mathbf{A}\vec{\kappa}_m = \vec{\varphi}(t_j + c_i\tau) - \mathbf{A}\vec{\mu}^{(j)} \;, \quad i = 1,\ldots,s \;, \qquad \text{[Lecture → (9.2.7.7)]}$$

$$\vec{\mu}^{(j+1)} = \vec{\mu}^{(j)} + \tau \sum_{m=1}^{s} \vec{\kappa}_m b_m \;. \qquad \text{[Lecture → (9.2.7.8)]}$$

As elaborated in [Lecture → § 9.2.7.6] the increments $\vec{\kappa}_i$, $i = 1,\ldots,s$, can be obtained by solving the following linear system of equations

$$\text{[Lecture → (9.2.7.7)]} \quad \Leftrightarrow \quad (\mathbf{I}_s \otimes \mathbf{M} + \tau\mathfrak{A} \otimes \mathbf{A}) \begin{bmatrix} \vec{\kappa}_1 \\ \vdots \\ \vec{\kappa}_s \end{bmatrix} = - \begin{bmatrix} \mathbf{A}\vec{\mu}^{(j)} \\ \vdots \\ \mathbf{A}\vec{\mu}^{(j)} \end{bmatrix} . \qquad \text{[Lecture → (9.2.7.9)]}$$

Written in more explicit fashion the system matrix of [Lecture → (9.2.7.9)] has the $s \times s$ block structure

$$\begin{bmatrix} \mathbf{M}+\tau a_{11}\mathbf{A} & \tau a_{12}\mathbf{A} & \ldots & \ldots & \ldots & \tau a_{1s}\mathbf{A} \\ \tau a_{21}\mathbf{A} & \mathbf{M}+\tau a_{22}\mathbf{A} & \tau a_{23}\mathbf{A} & \ldots & \ldots & \tau a_{2s}\mathbf{A} \\ \vdots & & \ddots & & & \vdots \\ \vdots & & & \ddots & & \vdots \\ \vdots & & & & \ddots & \vdots \\ \tau a_{s1}\mathbf{A} & \ldots & & & \ldots & \mathbf{M}+\tau a_{ss}\mathbf{A} \end{bmatrix} \in \mathbb{R}^{Ns,Ns} .$$

This matrix is first set up in triplet/COO format, then converted into CCS format, followed by the computation of its sparse LU-decomposition using EIGEN's built-in sparse Gaussian elimination. Doing this outside the timestepping loop is crucial for efficiency.

In addition we exploit that $b_i = (\mathfrak{A})_{s,i}$, the necessary condition for **L-stability** [Lecture → Def. 9.2.7.46] of an IRK-SSM. The final update [Lecture → (9.2.7.8)] is realized as

$$\vec{\mu}^{(k+1)} = \vec{\mu}^{(j)} + \tau \begin{bmatrix} \vec{\kappa}_1 & \vec{\kappa}_2 & \ldots & \vec{\kappa}_s \end{bmatrix} (\mathfrak{A})_{s,:} .$$

We use EIGEN's **Eigen::Map** helper class to reshape the long increment vector into an $N \times s$-matrix.

C++ code 9.19.16: Code of `timesteppingIRKMOLODE()`

```
2  template <typename MATRIX,
3            typename RECORDER = std::function<void(const Eigen::VectorXd &)>>
4  Eigen::VectorXd timesteppingIRKMOLODE(
5      const lf::assemble::COOMatrix<double> &M,
6      const lf::assemble::COOMatrix<double> &A, const MATRIX &Ark,
7      const Eigen::VectorXd &mu0, unsigned int no_ts, double T,
8      RECORDER &&rec = [](const Eigen::VectorXd & /*mu_vec*/) -> void {}) {
9    const Eigen::Index s = Ark.cols();
10   const Eigen::Index N = A.cols();
11   LF_ASSERT_MSG(s == Ark.rows(), "Butcher matrix must be square");
12   LF_ASSERT_MSG(N == A.rows(), "Matrix A must be square");
13   LF_ASSERT_MSG((N == M.rows()) and (N == M.cols()),
```

```
                "Matrix M same size as A!");
  LF_ASSERT_MSG(mu0.size() == N, "Illegal size of mu0!");
  const double tau = T / no_ts;
  // Initialization of system matrix for comnined increment equations
  lf::assemble::COOMatrix<double> ISM(N * s, N * s);
  const lf::assemble::COOMatrix<double>::TripletVec Mtv{M.triplets()};
  const lf::assemble::COOMatrix<double>::TripletVec Atv{A.triplets()};
  for (int i = 0; i < s; ++i) {
    for (int j = 0; j < s; ++j) {
      if (i == j) {
        for (const Eigen::Triplet<double> &M_trp : Mtv) {
          ISM.AddToEntry(M_trp.row() + i * N, M_trp.col() + j * N,
                         M_trp.value());
        }
      }
      for (const Eigen::Triplet<double> &A_trp : Atv) {
        ISM.AddToEntry(A_trp.row() + i * N, A_trp.col() + j * N,
                       tau * Ark(i, j) * A_trp.value());
      }
    }
  }
  // Advance LU decomposition of system matrix for increment equations
  Eigen::SparseMatrix<double> ISM_crs{ISM.makeSparse()};
  Eigen::SparseLU<Eigen::SparseMatrix<double>> ISLU;
  ISLU.compute(ISM_crs);
  LF_VERIFY_MSG(ISLU.info() == Eigen::Success, "LU decomposition failed");
  // Main timestepping loop
  Eigen::VectorXd mu{mu0};
  rec(mu);
  Eigen::VectorXd incs(N * s);
  Eigen::VectorXd Amu(N);
  for (int k = 0; k < no_ts; ++k) {
    // Solve increment system
    Amu = -A.MatVecMult(1.0, mu);
    incs = ISLU.solve(Amu.replicate(s, 1));
    LF_VERIFY_MSG(ISLU.info() == Eigen::Success,
                  "Solving increment system failed");
    mu += tau * Eigen::Map<Eigen::MatrixXd>(incs.data(), N, s) *
          Ark.row(s - 1).transpose();
    rec(mu);
  }
  return mu;
}
```

SOLUTION of (9-19.g):

From [Lecture → Thm. 9.2.8.5] we learn, sharpness of the asymptotic estimate taken for granted,

$$\left(\tau \sum_{j=1}^{M} \left|u(\tau j) - u_h^{(j)}\right|^2_{H^1(\Omega)}\right)^{\frac{1}{2}} \approx C\left(h_{\mathcal{M}}^2 + \tau^3\right) \quad \text{for small} \quad h_{\mathcal{M}}, \tau > 0\,, \tag{9.19.18}$$

where $\tau > 0$ is the timestep size, and $h_{\mathcal{M}} > 0$ the meshwidth. The constant $C > 0$ is not known, but neither depends on $h_{\mathcal{M}}$ nor τ.

As is obvious from (9.19.18), both the spatial and the temporal error have to be reducced by a factor of 100 alike, because they *add up* to the total error bound.

❶ Spatial error: Uniform regular refinement involves halving of $h_{\mathcal{M}}$. By (9.19.18) $h_{\mathcal{M}} \to h_{\mathcal{M}}/2$ leads to a drop of the spatial error by a factor of 4. Since $4^4 > 100$, but $4^3 < 100$, we need four steps of mesh refinement.

❷ Temporal error: Denote by $q \in]0,1[$ the reduction of the timestep. By (9.19.18) it has to satisfy

$$q^3 = 0.01 \quad \Leftrightarrow \quad q \approx 0.2154 \quad \Leftrightarrow \quad \text{reduction factor for } \tau \approx 4.64\,.$$

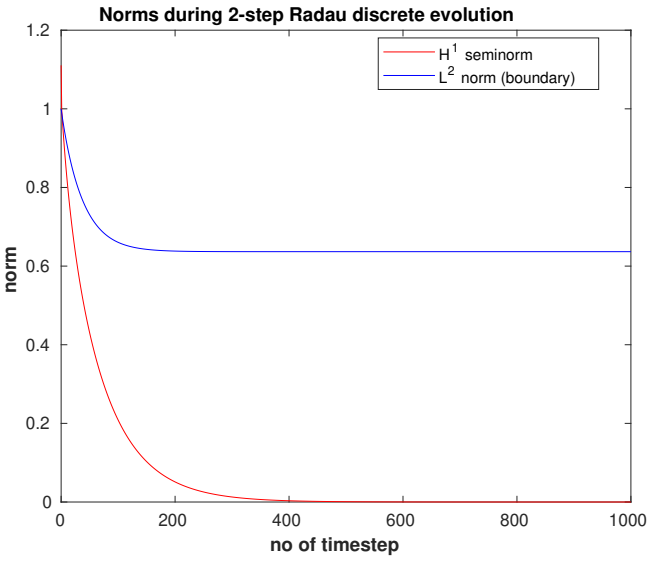
SOLUTION of (9-19.h):

The key insight is that we can directly compute the norms using the Galerkin matrices,

$$\left|u_h^{(j)}\right|_{H^1(\Omega)}^2 = \left(\vec{\mu}^{(j)}\right)^\top \mathbf{A}\vec{\mu}^{(j)} ,$$
$$\left\|u_h^{(j)}\right\|_{L^2(\partial\Omega)}^2 = \left(\vec{\mu}^{(j)}\right)^\top \mathbf{M}\vec{\mu}^{(j)} .$$

C++ code 9.19.19: Code of `tabulateSolNorms()`

```
template <typename U0FUNCTOR>
std::vector<std::pair<double, double>> tabulateSolNorms(
    std::shared_ptr<const lf::uscalfe::UniformScalarFESpace<double>> fes_p,
    U0FUNCTOR &&u0, unsigned int no_ts, double T, std::ostream &o = std::cout) {
  // Build MOL ODE matrices
  lf::assemble::COOMatrix<double> M{IRKDegenerateEvl::buildM(fes_p)};
  lf::assemble::COOMatrix<double> A{IRKDegenerateEvl::buildA(fes_p)};
  // Butcher matrix for 2-stage Radau method
  const Eigen::Matrix<double, 2, 2> Ark =
      (Eigen::Matrix<double, 2, 2>() << 5.0 / 12.0, -1.0 / 12.0, 3.0 / 4.0,
       1.0 / 4.0)
          .finished();
  // Coefficient vector of for initial data
  const Eigen::VectorXd mu0 =
      lf::fe::NodalProjection(*fes_p, lf::mesh::utils::MeshFunctionGlobal(u0));
  // Vector for storing the two desired norms
  // First:  L2-norm on the boundary, Second:  H1 seminorm on the domain
  std::vector<std::pair<double, double>> it_norms{};
  // "Recorder" lambda function
  auto rec = [&A, &M, &it_norms](const Eigen::VectorXd &mu_vec) -> void {
    it_norms.emplace_back(std::sqrt(mu_vec.dot(M.MatVecMult(1.0, mu_vec))),
                          std::sqrt(mu_vec.dot(A.MatVecMult(1.0, mu_vec))));
  };
  (void)timesteppingIRKMOLODE(M, A, Ark, mu0, no_ts, T, rec);
  // Print table
  o << std::setw(16) << "no. t.s." << std::setw(16) << "H1 seminorm"
    << std::setw(16) << "L2 norm bd\n";
  for (int j = 0; j < it_norms.size(); ++j) {
    auto [l2b, h1s] = it_norms[j];
    o << j << std::setw(16) << h1s << std::setw(16) << l2b << " ; ... \n";
  }
  return it_norms;
}
```



- 2-stage Radau timestepping
- Initial function

$$u_0(\boldsymbol{x}) = \cos(\tfrac{1}{2}\pi x_1)\cos(\tfrac{1}{2}\pi x_2)$$

- $\Omega =]0,1[^2$, simple triangular mesh
- Finite element space $\mathcal{S}_2^0(\mathcal{M})$
- Final time $T = 10$, 1000 timesteps

◁ norms of $u_h^{(j)}$, $j = 0, \ldots, 1000$

Both exponentially tend to a limit for $j \to \infty$. This limit is non-zero for $\left\|u_h^{(j)}\right\|_{L^2(\partial\Omega)}$, zero for $\left|u_h^{(j)}\right|_{H^1(\Omega)}$.

HINT 1 for (9-2.a): Combine the considerations leading to [Lecture → (9.2.2.4)] with the approach explained in [Lecture → Ex. 1.8.0.6]. ⌟

SOLUTION of (9-2.a):

In order to derive the spatial Formulation of (9.2.1), we multiply with a test function v (does not depend on time!), integrate over Ω, and then integrate by parts based on Green's first formula.

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \qquad [\text{Lecture} \to (12.2.3.2)]$$

A boundary term emerges, which can be recast using the spatial boundary condition.

$$\int_\Omega \dot{u}\, v \,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} v \mathbf{grad}\, u \cdot \boldsymbol{n} \,\mathrm{d}\boldsymbol{x} = 0$$

$$\Longrightarrow \quad \underbrace{\int_\Omega \dot{u}\, v \,\mathrm{d}\boldsymbol{x}}_{m(\dot{u},v)} + \underbrace{\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} cu\, v \,\mathrm{d}\boldsymbol{x}}_{a(u,v)} = \underbrace{0}_{\ell(v)}\,. \tag{9.2.3}$$

Thus, the relevant bilinear forms read

$$\begin{aligned} \mathsf{a}(u,v) &:= \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} cu\, v \,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1(\Omega)\,, \\ \mathsf{m}(u,v) &:= \int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}\,, \qquad\qquad\qquad u,v \in H^1(\Omega)\,. \end{aligned} \tag{9.2.4}$$

For any $t \in [0,T]$ the function spaces are

$$u(t,\cdot) \in H^1(\Omega) =: V \quad , \quad v \in H^1(\Omega) = V\,.$$

For further information refer to [Lecture → Ex. 1.8.0.6].

HINT 1 for (9-2.j): From [Lecture → Def. 7.3.3.1] and [Lecture → (7.3.3.3)] it should be clear how to obtain the linear systems of equations [Lecture → (9.2.7.7)], [Lecture → (9.2.7.8)] for the Runge-Kutta increments. ⌟

SOLUTION of (9-2.j):

Recall from [Lecture → (9.2.7.7)] the linear system of equations that we want to solve:

$$\vec{\boldsymbol{\kappa}}_i \in \mathbb{R}^N : \quad \mathbf{M}\vec{\boldsymbol{\kappa}}_i + \tau a_{i1}\mathbf{A}\vec{\boldsymbol{\kappa}}_1 + \tau a_{i2}\mathbf{A}\vec{\boldsymbol{\kappa}}_2 = \vec{\varphi}(t_j + c_i\tau) - \mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} , \quad i = 1, \ldots, 2 ,$$

where the entries a_{ij} correspond to the entries of the Butcher matrix $\mathfrak{A}$ from (9.2.11).

Thus, formula [Lecture → (9.2.7.7)] gives us two linear systems of equations, which we have to solve for the unknown vectors $\vec{\boldsymbol{\kappa}}_1 \in \mathbb{R}^N$ and $\vec{\boldsymbol{\kappa}}_2 \in \mathbb{R}^N$, $N := \dim \mathcal{S}_1^0(\mathcal{M})$:

$$\begin{aligned} \mathbf{M}\vec{\boldsymbol{\kappa}}_1 + \tau\zeta\mathbf{A}\vec{\boldsymbol{\kappa}}_1 &= -\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} , \\ \mathbf{M}\vec{\boldsymbol{\kappa}}_2 + \tau(1-\zeta)\mathbf{A}\vec{\boldsymbol{\kappa}}_1 + \tau\zeta\mathbf{A}\vec{\boldsymbol{\kappa}}_2 &= -\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} . \end{aligned} \tag{9.2.24}$$

Remark. It is possible to rewrite this system of equations using the **Kronecker product** of matrices, see [Lecture → § 9.2.7.6]:

$$\left(\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \mathbf{M} + \tau \begin{bmatrix} \zeta & 0 \\ 1-\zeta & \zeta \end{bmatrix} \otimes \mathbf{A} \right) \begin{bmatrix} \vec{\boldsymbol{\kappa}}_1 \\ \vec{\boldsymbol{\kappa}}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} \\ \mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} \end{bmatrix} . \tag{9.2.25}$$

SOLUTION of (9-2.k):

There are two options:

(I) We can loop over the cells of the mesh using the formula

$$\int_\Omega v_h(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \sum_{K\in\mathcal{M}} \frac{|K|}{\sharp\mathcal{V}(K)} \sum_{\ell=1}^{\sharp\mathcal{V}(K)} v_h(\boldsymbol{a}_K^\ell)\,, \tag{9.2.26}$$

where $\boldsymbol{a}_K^1,\ldots,\boldsymbol{a}_K^{\sharp\mathcal{V}(K)}$ are the vertices of K. Thanks to the cardinal basis property of the "tent function" basis $\{b_h^\ell\}_{\ell=1}^N$ of $\mathcal{S}_1^0(\mathcal{M})$ the values $v_h(\boldsymbol{a}_K^\ell)$ are directly available as entries of $\vec{\boldsymbol{\eta}}$. Their indices can be queried through the **lf::assemble::DofHandler** object.

(II) Let $\{b_h^\ell\}_{\ell=1}^N$ be the "tent function" nodal basis of $\mathcal{S}_1^0(\mathcal{M})$. Using the partition of unity of b_h^j, we can write $1 = \sum_{k=1}^N b_h^k(\mathbf{x})$.

Now, as $v_h(\boldsymbol{x}) = \sum_{k=1}^N (\vec{\boldsymbol{\eta}})_k b_h^k(\boldsymbol{x})$, we combine these two expressions and rewrite

$$\begin{aligned}\int_\Omega v_h\,\mathrm{d}\boldsymbol{x} &= \int_\Omega \left(\sum_{j=1}^N b_h^j(\boldsymbol{x})\right)\left(\sum_{k=1}^N (\vec{\boldsymbol{\eta}})_k b_h^k(\boldsymbol{x})\right)\mathrm{d}\boldsymbol{x}\\ &= \int_\Omega \sum_{j=1}^N\sum_{k=1}^N b_h^j(\boldsymbol{x})(\vec{\boldsymbol{\eta}})_k b_h^k(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \sum_{j=1}^N\sum_{k=1}^N \int_\Omega b_h^j(\boldsymbol{x}) b_h^k(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,(\vec{\boldsymbol{\eta}})_k = \vec{\mathbf{1}}^T\mathbf{M}\vec{\boldsymbol{\mu}}\,.\end{aligned}$$

where $\mathbf{M}$ is the Galerkin mass matrix. We can also pursue this approach in the implementation of `thermalEnergy()`, though the need to recompute $\mathbf{M}$ for every function call is an issue.

C++ code 9.2.27: Sub-problem (9-2.k), function `thermalEnergy()` ⇝ GitLab

```
2  double thermalEnergy(const lf::assemble::DofHandler &dofh,
3                       const Eigen::VectorXd &temperature_vec) {
4    double thermal_energy = 0.0;
5    auto mesh_p = dofh.Mesh();  // pointer to mesh
6
7    // The thermal energy of the system is defined as the integral of the
8    // temperature solution over the domain. It is computed for linear lagrangian
9    // finite elements using the trapezoidal rule by summing up the contribution
10   // of that quadrature rule over each triangle
11   double thermal_energy_loc;
12   for (const lf::mesh::Entity *tria : mesh_p->Entities(0)) {
13     thermal_energy_loc = 0.0;
14     // Compute the area of the triangle
15     const double area = lf::geometry::Volume(*(tria->Geometry()));
16     // Obtain the global indices of the nodal degrees of freedom
17     for (const lf::assemble::gdof_idx_t &g_idx : dofh.GlobalDofIndices(*tria)) {
18       thermal_energy_loc += temperature_vec[g_idx];
19     }
20     // Sum local contribution of quadrature rule
21     thermal_energy += thermal_energy_loc * area / 3.0;
22   }
23   return thermal_energy;
24 }  // thermalEnergy
```

HINT 1 for (9-2.l): Pay attention to efficiency: for instance, the LU-factorization of the coefficient matrix of the linear increment equations can be done in advance before carrying out any timestep. ⌟

SOLUTION of (9-2.l):

In each timestep we have to determine the increments $\vec{\kappa}_1$ and $\vec{\kappa}_2$ from

$$\begin{aligned}\big(\mathbf{M}+\tau\zeta\mathbf{A}\big)\vec{\kappa}_1 &= -\mathbf{A}\vec{\mu}^{(j)}\,,\\ \big(\mathbf{M}+\tau\zeta\mathbf{A}\big)\vec{\kappa}_2+\tau(1-\zeta)\mathbf{A}\vec{\kappa}_1 &= -\mathbf{A}\vec{\mu}^{(j)}\,.\end{aligned} \tag{9.2.24}$$

This is two $N\times N$ linear system of equations with the *same* coefficient matrix $\mathbf{M}+\tau\zeta\mathbf{A}$. This is the benefit of a RK-SSM with lower-triangular Butcher matrix.

We form this matrix once from $\mathbf{A}$ and $\mathbf{M}$ relying on EIGEN's allegedly efficient sparse matrix arithmetic. We also factorize this sparse matrix once by calling the `compute()` method, see ➙ documentation. The only function call we make twice in every timestep is that of the member function `solve()`, which carries out inexpensive forward and backward substitutions.

Then [Lecture → (9.2.7.8)] gives us an update formula for a single step of the SDIRK-2 SSM:

$$\vec{\mu}^{(j+1)} = \vec{\mu}^{(j)} + \tau\vec{\kappa}_1(1-\zeta) + \tau\vec{\kappa}_2\zeta. \tag{9.2.28}$$

Here, the $\vec{\mu}^{(j)}\in\mathbb{R}^N$, $N := \dim\mathcal{S}_1^0(\mathcal{M})$, are basis expansion coefficient vectors.

Our implementation makes use of an auxliary class **SDIRK2Timestepper**, whose constructor precomputes the factorized matrices. It features a member function `discreteEvolutionOperator()` that carries out a single timestep.

C++ code 9.2.29: Sub-problem (9-2.l), Auxliary class SDIRK2Timestepper ➙ GitLab

```
2  class SDIRK2Timestepper {
3   public:
4    // Disabled constructor
5    SDIRK2Timestepper() = delete;
6    SDIRK2Timestepper(const SDIRK2Timestepper &) = delete;
7    SDIRK2Timestepper(SDIRK2Timestepper &&) = delete;
8    SDIRK2Timestepper &operator=(const SDIRK2Timestepper &) = delete;
9    SDIRK2Timestepper &operator=(const SDIRK2Timestepper &&) = delete;
10   // Main constructor; precomputations are done here
11   explicit SDIRK2Timestepper(const lf::assemble::DofHandler &dofh, double tau,
12                              double cool_coeff);
13   // Destructor
14   virtual ~SDIRK2Timestepper() = default;
15
16   /* Class member functions */
17   // Discrete evolution operator for SDIRK-2
18   Eigen::VectorXd discreteEvolutionOperator(const Eigen::VectorXd &mu) const;
19
20  private:
21   double tau_;  // step size (in time)
22   // Sparses matrices holding Galerkin matrices
23   Eigen::SparseMatrix<double> A_;  // Element matrix (Laplace + bdy mass)
24   double lambda_;                  // coefficient for SDIRK-2 Butcher tableau
25   // For fixed step-size in time, the linear system of equations implicitely
26   // defining the Runge-Kutta increments are independent of time. We can thus
27   // precompute the LU decompositions for more efficiency. SDIRK-2 is diagonally
28   // implicit: precomputing the LU decomposition for the two stages
```

```
29    // independently makes for an easy implementation.
30    Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
31  };
```

C++ code 9.2.30: Sub-problem (9-2.l), member function `discreteEvolutionOperator()` of SDIRK2Timestepper ↬ GitLab

```
2   template <typename FUNCTOR>
3   Eigen::Matrix<double, 2, 2> LinearMassEdgeMatrixProvider<FUNCTOR>::Eval(
4       const lf::mesh::Entity &edge) {
5     Eigen::Matrix<double, 2, 2> elBdyEdgeMat;
6     // Throw error in case not edge entity
7     LF_VERIFY_MSG(edge.RefEl() == lf::base::RefEl::kSegment(),
8                   "Unsupported entity type " << edge.RefEl());
9     if (predicate_(edge)) {
10      // Obtain endpoints of the edge
11      auto endpoints = lf::geometry::Corners(*(edge.Geometry()));
12      LF_ASSERT_MSG(endpoints.cols() == 2, "Wrong no endpoints in " << edge);
13      // Compute length of the edge
14      double edge_length = (endpoints.col(1) - endpoints.col(0)).norm();
15      // Compute the local edge mass element matrix
16      // clang-format off
17      elBdyEdgeMat << 1.0 / 3.0, 1.0 / 6.0,
18                      1.0 / 6.0, 1.0 / 3.0;
19      // clang-format on
20      elBdyEdgeMat *= edge_length * cool_coeff_;
21
22      return elBdyEdgeMat;  // return the local edge mass element matrix
23    }
24    return (Eigen::Matrix<double, 2, 2>::Zero());
25  }  // LinearMassEdgeMatrixProvider<FUNCTOR>::Eval
```

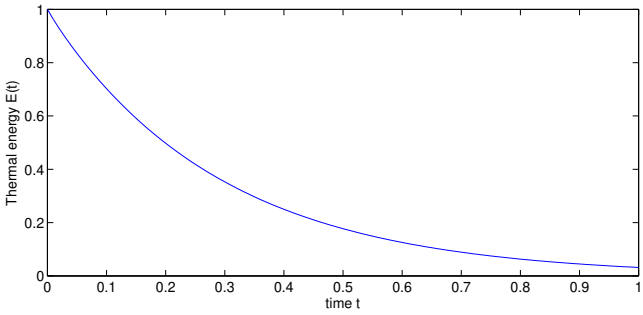
Of course, your implementation need not rely on such an auxiliary class.

C++ code 9.2.31: Sub-problem (9-2.l), function `solveTemperatureEvolution()` ↬ GitLab

```
2   std::pair<Eigen::VectorXd, Eigen::VectorXd> solveTemperatureEvolution(
3       const lf::assemble::DofHandler &dofh, unsigned int m, double cool_coeff,
4       Eigen::VectorXd initial_temperature_vec) {
5     std::pair<Eigen::VectorXd, Eigen::VectorXd> solution_pair;
6     double tau = 1.0 / m;                                  // step size
7     const lf::uscalfe::size_type N_dofs(dofh.NumDofs());  // dim. of FE space
8     std::cout << "\n>>> SolveTemperatureEvolution: m = " << m
9               << ", N = " << N_dofs << ", cool_coeff= " << cool_coeff << "\n"
10              << std::endl;
11
12    // Initialization for the energies of the solution at times t = 0, tau, 2*tau,
13    // etc... This information is not required to solve the parabolic evolution
14    Eigen::VectorXd energies(m + 1);
15
16    // Precomputing the required data for SDIRK-2
17    SDIRK2Timestepper SDIRK2_stepper(dofh, tau, cool_coeff);
18
19    std::cout << "\n>> Iterating the action of discreteEvolutionOperator"
```

```
20              << std::endl;
21   // Starting the evolution using the initial conditions
22   Eigen::VectorXd discrete_solution_cur =
23       SDIRK2_stepper.discreteEvolutionOperator(initial_temperature_vec);
24   energies[0] = thermalEnergy(dofh, initial_temperature_vec);
25   energies[1] = thermalEnergy(dofh, discrete_solution_cur);
26   // Evolving the parabolic temperature system
27   // While less elegant, we use a current and next step solution vector in
28   // the iteration to stay away from potential harming aliasing effects of
29   // putting an Eigen::Vector on both sides of an assignment statement.
30   Eigen::VectorXd discrete_solution_next;
31   for (int i = 1; i < m; i++) {
32     discrete_solution_next =
33         SDIRK2_stepper.discreteEvolutionOperator(discrete_solution_cur);
34     discrete_solution_cur = discrete_solution_next;
35     energies[i + 1] = thermalEnergy(dofh, discrete_solution_cur);
36   }
37   solution_pair = std::make_pair(discrete_solution_cur, energies);
38   return solution_pair;
39 }  // solveTemperatureEvolution
```

SOLUTION of (9-2.m):



◁ Plot of $E_h(t_k)$ versus t_k, $k = 1, \ldots, 100$

As predicted by the analysis of Sub-problem (9-2.b) we witness an exponential decay of the total heat stored in Ω.

C++ code 9.2.33: Part of `main()`-function reading the mesh and running the numerical experiment. ⇝ GitLab

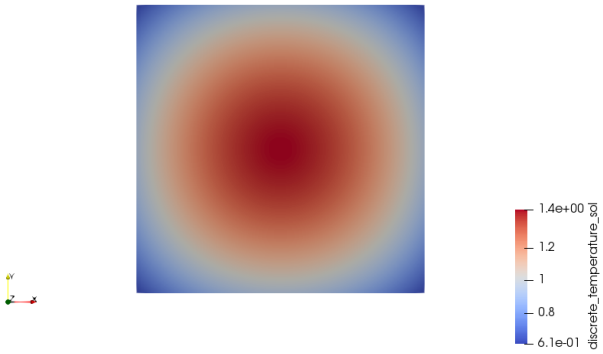
```
2
3   // Load mesh into a Lehrfem++ object
4   auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
5   const lf::io::GmshReader reader(std::move(mesh_factory),
6                                   "meshes/square64_bnd.msh");
7   auto mesh_p = reader.mesh();  // type shared_ptr< const lf::mesh::Mesh>
8
9   // Finite element space
10  auto fe_space =
11      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
12  // Obtain local->global index mapping for current finite element space
13  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
14  // Dimension of finite element space
15  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
16
17  // Building initial condition vector
18  Eigen::VectorXd initial_temperature_vec(N_dofs);
19  for (int idx = 0; idx < N_dofs; idx++) {
20    // Obtain coordinates of vertex at global index idx
21    auto coords = lf::geometry::Corners(*(dofh.Entity(idx).Geometry()));
22    LF_ASSERT_MSG(coords.cols() == 1, "Wrong no of coords in vertex");
23    initial_temperature_vec(idx) = 5.0;
24  }
25
26  // SDIRK-2 evolution of parabolic problem
27  unsigned int m = 100;
28  std::pair<Eigen::VectorXd, Eigen::VectorXd> solution_pair =
29      solveTemperatureEvolution(dofh, m, 1.0, initial_temperature_vec);
30  Eigen::VectorXd discrete_temperature_sol = solution_pair.first;
31  LF_ASSERT_MSG(
32      discrete_temperature_sol.size() == N_dofs,
33      "Size of discrete solution and dimension of FE space mismatch.");
34  Eigen::VectorXd energies = solution_pair.second;
35  LF_ASSERT_MSG(energies.size() == m + 1, "Wrong number of energie values.");
36  // Define output file format for the energies
37  const static Eigen::IOFormat CSVFormat(Eigen::StreamPrecision,
38                                         Eigen::DontAlignCols, ", ", "\n");
39
40  // Corresponding time grid for plotting
41  Eigen::VectorXd time = Eigen::VectorXd::LinSpaced(m + 1, 0.0, 1.0);
42
```

```
43    // Write .csv file of energy vs. time
44    std::ofstream file;
45    file.open("energies.csv");
46    file << time.transpose().format(CSVFormat) << std::endl;
47    file << energies.transpose().format(CSVFormat) << std::endl;
48    file.close();
49    std::cout << "Generated energies.csv" << std::endl;
50
51    // Plot from .csv file using python
52    systemcall::execute(
53        "python3 ms_scripts/plot_energies.py energies.csv "
54        "energies.eps");
```

HINT 1 for (9-2.o): The `writePointData()` member function of **lf::io::VtkWriter** needs the values stored in a **lf::mesh::utils::CodimMeshDataSet** object. Thus you have to use the **lf::assemble::DofHand** object to transfer the values from the basis expansion coefficient vector to a **lf::mesh::utils::CodimMeshDa** object. ⌟

SOLUTION of (9-2.o):

The field $t = 1$ is displayed in Fig. 120.



◁ Finite element/SDIRK-2 solution at $T = 1$ starting from $u_0 \equiv 5.0$.
Visualization by PARAVIEW, reading the file `discrete_temperature_sol.vtk`

The temperature drops close to the boundary due to heat drained away by the convective cooling boundary conditions.

C++ code 9.2.34: Code snippet from `main()` dedicated to VTK output ↪ GitLab

```
2  // Output results for the temperature function to vtk file
3  lf::io::VtkWriter vtk_writer(mesh_p, "discrete_temperature_sol.vtk");
4  // Write nodal data taking the values of the discrete solution at the vertices
5  auto nodal_data = lf::mesh::utils::make_CodimMeshDataSet<double>(mesh_p, 2);
6  for (int global_idx = 0; global_idx < N_dofs; global_idx++) {
7    nodal_data->operator()(dofh.Entity(global_idx)) =
8        discrete_temperature_sol[global_idx];
9  };
10 vtk_writer.WritePointData("discrete_temperature_sol", *nodal_data);
```

HINT 1 for (9-2.b):

- Appeal to the heat conduction background to justify the assumption that $u(\boldsymbol{x},t) \geq 0$ for all $(\boldsymbol{x},t)$, which is a rigorous mathematical result, the maximum principle for parabolic evolution problems .
- Use the test function $v \equiv 1$ in the variational formulation.

⌟

SOLUTION of (9-2.b):

The quantity u specifies the temperature distribution. We track the temporal evolution of the diffusion of the temperature. These two physical effects can never lead to negative values in the temperature distribution unless u_0 is negative somewhere.

The boundary condition models convective cooling ($\rightarrow$ [Lecture $\rightarrow$ Ex. 1.7.0.5]) where our domain Ω is embeded in a fluid at bulk temperature 0. So, if $u \geq 0$ on the boundary, then the 2D body will be cooled. The convective cooling dissapears as $u \to 0$ (it would even warm the 2D body for negative temperature $u < 0$). Thus $u(\mathbf{x},t) \geq 0$ and $E(t) := \int_\Omega u(\mathbf{x},t)\ \mathrm{d}\mathbf{x} = \int_\Omega |u(\mathbf{x},t)|\ \mathrm{d}\mathbf{x} \geq 0$.

We do a similar computation as in [Lecture $\rightarrow$ Lemma 9.2.3.8], but we don't need to use the function w that is introduced there, we can do the calculation directly relying on the spatial variational formulation (9.2.3) with $\ell \equiv 0$ in step $(\star)$:

$$\frac{d}{dt}E(t) = \frac{d}{dt}\int_\Omega u(\mathbf{x},t)\ \mathrm{d}\mathbf{x} = \frac{d}{dt}m(u,1) = \mathsf{m}(\dot{u},1) \overset{(\star)}{=} -\mathsf{a}(u,1) = -c\underbrace{\int_{\partial\Omega} u\,\mathrm{d}\mathbf{x}}_{\geq 0} \leq 0\ .$$

In the last step we used the positivity of u, $u \geq 0$, which we assumed. $\frac{d}{dt}E(t) = 0$ occurs only for $u \equiv 0$.

HINT 1 for (9-2.c): The following lemma for $d = 2$ could be useful.

Lemma [Lecture → Lemma 2.7.5.5]. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1, \ldots, \lambda_{d+1}$ and exponents $\alpha_j \in \mathbb{N}$, $j = 1, \ldots, d+1$,

$$\int_K \lambda_1^{\alpha_1} \cdot \cdots \cdot \lambda_{d+1}^{\alpha_{d+1}} \, \mathrm{d}\boldsymbol{x} = d!|K| \, \frac{\alpha_1!\alpha_2! \cdot \cdots \cdot \alpha_{d+1}!}{(\alpha_1 + \alpha_2 + \cdots + \alpha_{d+1} + d)!} \quad \forall \boldsymbol{\alpha} \in \mathbb{N}_0^{d+1} \, .$$

[Lecture → (2.7.5.6)]

SOLUTION of (9-2.c):

Recall that the local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ on a triangle are the barycentric coordinate functions λ_ℓ, $\ell = 1,2,3$. Thus the entries of the element mass matrix for $\widehat{K}$ are given by

$$\mathbf{M}_{\widehat{K}} = \left[\int_{\widehat{K}} \lambda_j(\boldsymbol{x})\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^{3} . \tag{9.2.6}$$

Use [Lecture → Lemma 2.7.5.5] for $d = 2$ to compute the values, which yields

$$\mathbf{M}_{\widehat{K}} = \frac{1}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix} . \tag{9.2.7}$$

Note that this element matrix does not depend on the shape of K, but only on its area.

HINT 1 for (9-2.d): See [Lecture → Def. 2.7.4.5] for the definition of element matrices, and [Lecture → Section 2.4.5] for concrete formulas. ⌟

SOLUTION of (9-2.d):

As said before, the local shape functions are the barycentric coordinate functions. On the reference triangle $\widehat{K}$ they read

$$\lambda_1(\boldsymbol{x}) = 1 - x_1 - x_2 \quad , \quad \lambda_2(\boldsymbol{x}) = x_1 \quad , \quad \lambda_3(\boldsymbol{x}) = x_2 \, ,$$

with constant gradients

$$\mathbf{grad}\, \lambda_1 = \begin{bmatrix} -1 \\ -1 \end{bmatrix} \quad , \quad \mathbf{grad}\, \lambda_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \quad , \quad \mathbf{grad}\, \lambda_3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} .$$

Moreover, the local computations for the volume contribution were explained in detail in [Lecture → Section 2.4.5.1], see [Lecture → (2.4.5.8)], and yield the following element stiffness matrix

$$\mathbf{A}_{\widehat{K}} = \frac{1}{2} \begin{bmatrix} 2 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix} + \frac{\sqrt{2}}{6} c \begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} .$$

The computations of the second part of the matrix is explained in [Lecture → Ex. 2.7.4.41] and its non-zero entries are given in [Lecture → (2.7.4.44)]. Note that here we consider the boundary contribution with respect to the full set of reference shape functions for $\widehat{K}$.

SOLUTION of (9-2.e):

The variational formulation (9.2.3) involves a boundary term of exactly the same form as that in [Lecture $\rightarrow$ (2.7.4.42)].

SOLUTION of (9-2.f):

Of course, we use LEHRFEM++'s assembly function `AssembleMatrixLocally()` to generate the Galerkin matrix. We call it twice, once for cells (co-dimension 0) and once for edges (co-dimension 1), where in the latter case the method `isActive())` selects boundary edges only.

C++ code 9.2.8: Sub-problem (9-2.f): function `assembleGalerkinMatrices()` ⇝ GitLab

```
2  std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>>
3  assembleGalerkinMatrices(const lf::assemble::DofHandler &dofh,
4                           double cool_coeff) {
5    std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>>
6        sparse_pair;
7    std::cout << ">> Constructing SDIRK2Timestepper " << std::endl;
8    auto mesh_p = dofh.Mesh();  // pointer to current mesh
9    // Instantiating Galerkin matrices to be pre-computed
10   // Dimension of finite element space
11   const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
12
13   // Obtain an array of boolean flags for the edges of the mesh: 'true'
14   // indicates that the edge lies on the boundary. This predicate will
15   // guarantee that the computations are carried only on the boundary edges
16   auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
17   // Creating predicate that will guarantee that the computations are carried
18   // only on the edges of the mesh using the boundary flags
19   auto edges_predicate = [&bd_flags](const lf::mesh::Entity &edge) -> bool {
20     return bd_flags(edge);
21   };
22   // Matrices in triplet format holding Galerkin matrices, zero initially.
23   lf::assemble::COOMatrix<double> A_COO(N_dofs, N_dofs);
24   lf::assemble::COOMatrix<double> M_COO(N_dofs, N_dofs);
25
26   std::cout << "> Initializing the Galerking local matrices builders"
27             << std::endl;
28   // Initialize classes containing the information required for the local
29   // computations of the Galerkin matrices. Simple implementations of
30   // LinFEMassMatrixProvider adapted to this particular problem was written to
31   // spare some of the overhead calculations involved in the use of the more
32   // general LehrFEM++ matrices providers.
33   lf::uscalfe::LinearFELaplaceElementMatrix elLapMat_builder;
34   LinearMassEdgeMatrixProvider<decltype(edges_predicate)> el_MassEdge_builder(
35       edges_predicate, cool_coeff);
36   LinFEMassMatrixProvider elMassMat_builder;
37
38   std::cout << "> Assembling Galerking matrices in COO format" << std::endl;
39   // Compute the Galerkin matrices
40   // Invoke assembly on cells and edges (co-dimension = 0 and 1 respectively as
41   // first argument). Information about the mesh and the local-to-global map is
42   // passed through a Dofhandler object, argument 'dofh'. This function call
43   // adds triplets to the internal COO-format representation of sparse matrices
44   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elLapMat_builder, A_COO);
45   lf::assemble::AssembleMatrixLocally(1, dofh, dofh, el_MassEdge_builder,
46                                       A_COO);
47   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elMassMat_builder, M_COO);
48
49   std::cout << "> Converting to triplets to sparse matrices" << std::endl;
```

```
50    // Assembling the private Galerkin matrices
51    Eigen::SparseMatrix<double> A = A_COO.makeSparse();
52    Eigen::SparseMatrix<double> M = M_COO.makeSparse();
53    sparse_pair = std::make_pair(A, M);
54    return sparse_pair;
55  }
```

C++ code 9.2.9: Sub-problem (9-2.f): class LinFEMassMatrixProvider ⇝ GitLab

```
2   Eigen::Matrix<double, 3, 3> LinFEMassMatrixProvider::Eval(
3       const lf::mesh::Entity &tria) {
4     Eigen::Matrix<double, 3, 3> elMat;
5     // Throw error in case no triangular cell
6     LF_VERIFY_MSG(tria.RefEl() == lf::base::RefEl::kTria(),
7                   "Unsupported cell type " << tria.RefEl());
8     // Compute the area of the triangle cell
9     const double area = lf::geometry::Volume(*(tria.Geometry()));
10    // Compute the mass element matrix over the cell
11    // clang-format off
12    elMat << 2.0, 1.0, 1.0,
13             1.0, 2.0, 1.0,
14             1.0, 1.0, 2.0;
15    // clang-format on
16    elMat *= area / 12.0;
17
18    return elMat;  // return the local mass element matrix
19  }  // LinFEMassMatrixProvider::Eval
```

Note that the following **ENTITY_MATRIX_PROVIDER** is meant `SEGMENT`-type entities and the finite-element space $\mathcal{S}_1^0(\mathcal{M})$. Since there are only two local shape functions associated with an edge, `Eval()` has to return a 2×2-matrix, see [Lecture → Ex. 2.7.4.41] for more explanations.

C++ code 9.2.10: Sub-problem (9-2.f): class LinearMassEdgeMatrixProvider ⇝ GitLab

```
2   template <typename FUNCTOR>
3   Eigen::Matrix<double, 2, 2> LinearMassEdgeMatrixProvider<FUNCTOR>::Eval(
4       const lf::mesh::Entity &edge) {
5     Eigen::Matrix<double, 2, 2> elBdyEdgeMat;
6     // Throw error in case not edge entity
7     LF_VERIFY_MSG(edge.RefEl() == lf::base::RefEl::kSegment(),
8                   "Unsupported entity type " << edge.RefEl());
9     if (predicate_(edge)) {
10      // Obtain endpoints of the edge
11      auto endpoints = lf::geometry::Corners(*(edge.Geometry()));
12      LF_ASSERT_MSG(endpoints.cols() == 2, "Wrong no endpoints in " << edge);
13      // Compute length of the edge
14      double edge_length = (endpoints.col(1) - endpoints.col(0)).norm();
15      // Compute the local edge mass element matrix
16      // clang-format off
17      elBdyEdgeMat << 1.0 / 3.0, 1.0 / 6.0,
18                      1.0 / 6.0, 1.0 / 3.0;
19      // clang-format on
20      elBdyEdgeMat *= edge_length * cool_coeff_;
21
22      return elBdyEdgeMat;  // return the local edge mass element matrix
```

```
23      }
24      return (Eigen::Matrix<double, 2, 2>::Zero());
25  }  // LinearMassEdgeMatrixProvider<FUNCTOR>::Eval
```

SOLUTION of (9-2.g):

Here $\dot{y} = \lambda y =: f(y)$. According to the inpretation of the Butcher scheme [Lecture → (7.3.3.3)] given in [Lecture → Def. 7.3.3.1] the increment equations for the SDIRK-2 single step method are

$$\begin{aligned} \kappa_1 &= \lambda y + \lambda\tau\zeta\kappa_1 \,, \\ \kappa_2 &= \lambda y + \lambda\tau(1-\zeta)\kappa_1 - \lambda\tau\zeta\kappa_2 \,. \end{aligned}$$

Solving this linear system, we get

$$\kappa_1 = \frac{\lambda y}{1-\lambda\tau\zeta}, \qquad \kappa_2 = \frac{\lambda y(1-2\lambda\tau\zeta+\lambda\tau)}{(1-\lambda\tau\zeta)^2} \tag{9.2.12}$$

Finally, updating the solution, we get

$$\begin{aligned} y^{(j+1)} &= y^{(j)} + \tau(1-\zeta)\kappa_1 + \tau\zeta\kappa_2 && (9.2.13) \\ &= y^{(j)}\left(1 + \frac{\lambda\tau(1-\lambda\tau\zeta^2)}{(1-\lambda\tau\zeta)^2}\right) . && (9.2.14) \end{aligned}$$

Remark. In fact we have found the discrete evolution operator for the SDIRK-2 RK-SSM applied to $\dot{y} = \lambda y$. Note that, as is always the case when applying RK-SSMs to $\dot{y} = \lambda y$, the discrete evolution operator amounts to multiplication with a number. This number is $S(\tau\lambda)$, where S is the **stability function** of the method, see [Lecture → § 9.2.7.44]/. Thus, for SDIRK-2 we have

$$S(z) = 1 + \frac{z(1-z\zeta^2)}{(1-z\zeta)^2} \,, \quad z \in \mathbb{C} \setminus \{\zeta^{-1}\} \,. \tag{9.2.15}$$

We are coming back to this in Sub-problem (9-2.i).

SOLUTION of (9-2.h):

Since we expect **algebraic convergence** , we compute the discretization error for a geometrically increasing number of timesteps, see for more details.

You only have to implement (9.2.13). In the code we double the number of timesteps in each step.

C++ code 9.2.17: Function `sdirk2SteppingLinScalODE()` ↪ GitLab

```
std::vector<double> sdirk2SteppingLinScalODE(unsigned int m) {
  std::vector<double> sol_vec;
  double step_size = 2.0 / m;  // "tau" in this example
  // Initial conditions
  sol_vec.push_back(1.0);
// Discrete evolution operator. For the SDIRK-2 method applied to the
// scalar linear ODE (d/dt)y = -y, this turns out to be a scalar valued
// function depending on the step_size. Since we take equidistant step sizes
// in this example, the action of the evolution operator is simply
// multiplication by a constant double
  double lambda = 1.0 - 0.5 * sqrt(2.0);
  double evolution_op = 1.0 - step_size *
                                (1.0 + step_size * std::pow(lambda, 2)) /
                                std::pow(1.0 + step_size * lambda, 2);
  // Compute discrete evolution by applying the evolution operator at each step
  for (int i = 1; i < m + 1; i++) {
    sol_vec.push_back(evolution_op * sol_vec.at(i - 1));
  }
  return sol_vec;
}
```

C++ code 9.2.18: Convergence test for SDIRK-2 in the scalar case ↪ GitLab

```
  double diff;  // temporary variable used to compute error at various nodes
  double max_norm_errors[nIter];  // errors vector for all approx. sols
  std::vector<double> approx_sol_vec;
  for (int k = 0; k < nIter; k++) {
    m = 10 * std::pow(2, k);

    // Creating exact solution vector. This vector is created by evaluating the
    // exact solution using the analytic formula y(t) = exp(-t) at the
    // equidistant nodes of the time steps.
    double step_size = 2.0 / m;  // "tau" in this example
    std::vector<double> exact_sol_vec;
    for (int i = 0; i < m + 1; i++) {
      exact_sol_vec.push_back(std::exp(-i * step_size));
    }
    // Computing approximate solution
    approx_sol_vec = sdirk2SteppingLinScalODE(m);
    // Computing the error in the maximum norm
    for (int i = 0; i < m + 1; i++) {
      max_norm_errors[k] = 0;
      diff = std::abs(approx_sol_vec.at(i) - exact_sol_vec.at(i));
      max_norm_errors[k] =
          (diff > max_norm_errors[k]) ? diff : max_norm_errors[k];
    }
```

```
  }
  // Computing rates of convergence
  double rates[nIter - 1];
  double avg_rate = 0.0;
  for (int k = 0; k < nIter - 1; k++) {
    rates[k] = log2(max_norm_errors[k] / max_norm_errors[k + 1]);
    avg_rate += rates[k];
  }
  avg_rate = avg_rate / (nIter - 1);
```

♯timesteps	error	estimated rate
10	0.000446558	
20	0.000110501	2.01478
40	2.74922e-05	2.00697
80	6.85696e-06	2.00338
160	1.71226e-06	2.00167
320	4.2782e-07	2.00083
640	1.06924e-07	2.00041
1280	2.67273e-08	2.00021
2560	6.68136e-09	2.0001
5120	1.67028e-09	2.00005

Algebraic convergence [Lecture → Def. 3.2.2.1] in the number of timesteps is evident, because using twice as many timestep roughly reduces the error by a factor of 4. Thus, the observed rate is 2 as expected. See Code 9.2.17 and Code 9.2.18 for the details of this numerical experiment.

Formula for the estimated rate α, *cf.* [Lecture → § 3.2.2.5]:

$$\alpha \approx -\frac{\log \epsilon_\ell - \log \epsilon_{\ell-1}}{\log m_\ell - \log m_{\ell-1}} \,.$$

HINT 1 for (9-2.i): Of course, the recursion found in Sub-problem (9-2.g) can be used or you may rely on [Lecture → (9.2.7.45b)].

Then the problem boils down to discussing the behavior of a rational function on the negative real axis. ⌟

SOLUTION of (9-2.i):

I. We use the recursion found in Sub-problem (9-2.g) to get

$$y^j = \left(1 - \frac{\lambda\tau(1+\lambda\tau\zeta^2)}{(1+\lambda\tau\zeta)^2}\right)^j y^0, \tag{9.2.19}$$

i.e. the method has the **stability function** [Lecture → § 9.2.7.44]

$$S(\tau\lambda) := \left(1 - \frac{\lambda\tau(1+\lambda\tau\zeta^2)}{(1+\lambda\tau\zeta)^2}\right) \quad \Leftrightarrow \quad S(z) = 1 + \frac{z(1-z\zeta^2)}{(1-z\zeta)^2}\,. \tag{9.2.20}$$

II. For a RK-SSM described by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ the stability function is given by the formula

$$S(z) = 1 + z\mathbf{b}^\top(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1} = \frac{\det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^\top)}{\det(\mathbf{I} - z\mathfrak{A})}\,, \quad \mathbf{1} = [1,\ldots,1]^\top \in \mathbb{R}^s\,.$$

[Lecture → (9.2.7.45b)]

For the concrete case of the SDIRK-2 method with

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \hat{=} \quad \begin{array}{c|cc} \zeta & \zeta & 0 \\ 1 & 1-\zeta & \zeta \\ \hline & 1-\zeta & \zeta \end{array}\,, \quad \zeta := 1 - \tfrac{1}{2}\sqrt{2}\,, \tag{9.2.11}$$

this means

$$S(z) = \frac{\det\begin{bmatrix} 1 - z\zeta + z(1-\zeta) & (1-\zeta)z \\ -z(1-\zeta) + \zeta z & 1 - z\zeta + z\zeta \end{bmatrix}}{\det\begin{bmatrix} 1-z\zeta & 0 \\ -z(1-\zeta) & 1-z\zeta \end{bmatrix}}\,.$$

We recall the definition of L-stability:

Definition . L-stable Runge-Kutta method →

A Runge-Kutta method is L-stable/asymptotically stable, if its stability function $z \mapsto S(z)$ satisfies

$$(i) \qquad \operatorname{Re} z < 0 \;\Rightarrow\; |S(z)| < 1\,, \tag{9.2.21}$$

$$(ii) \qquad \lim_{\operatorname{Re} z \to -\infty} S(z) = 0\,. \tag{9.2.22}$$

❶ The requirment (9.2.22) is straightforward from the formula (9.2.20) for S.

❷ Now $|S(z)| < 1$ if and only if

$$-1 < 1 + \frac{z(1-z\zeta^2)}{(1-z\zeta)^2} < 1. \tag{9.2.23}$$

As we assume $z < 0$, it is clear that $\frac{z(1-z\zeta^2)}{(1-z\zeta)^2} < 0$, so the upper bound holds. Proving the lower bound is equivalent to showing

$$\begin{aligned} & 2 + \frac{z(1-z\zeta^2)}{(1-z\zeta)^2} > 0 \\ \Leftrightarrow \quad & 2(1-2z\zeta+z^2\zeta^2) + z - z^2\zeta^2 > 0 \\ \Leftrightarrow \quad & 2 - 4z\zeta + z^2\zeta^2 + z > 0\,. \end{aligned}$$

As $-4z\zeta + z = -z(4\zeta - 1) > 0$, since $\zeta = 1 - \frac{\sqrt{2}}{2}$, the last assertion follows and the lower bound holds.

SOLUTION of (9-20.a):

We harness the techniques of [Lecture → Section 1.5.3], in particular Green's first formula [Lecture → Thm. 1.5.2.7], which is applied to the second integral to move the derivatives off the test function and onto the trial function. Since functions in the test space do not vanish in $\partial\Omega$ we follow the two-step approach of [Lecture → Ex. 1.5.3.11].

❶ Test with $v \in C_0^\infty(\Omega)$, followed by integration by parts,

$$\int_\Omega \sigma(\boldsymbol{x})\frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega \left(\sigma(\boldsymbol{x})\frac{\partial u}{\partial t}(\boldsymbol{x},t) - \Delta u(\boldsymbol{x},t)\right) v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega)\,.$$

The boundary terms vanish, since $v|_{\partial\Omega} = 0$. By the fundamental lemma of the calculus of variations [Lecture → Lemma 1.5.3.4] we conclude that

$$\sigma(\boldsymbol{x})\frac{\partial u}{\partial t}(\boldsymbol{x},t) - \Delta_{\boldsymbol{x}} u(\boldsymbol{x},t) = 0 \quad \text{in} \quad \Omega\,. \tag{9.20.3}$$

❷ Now we test with $v \in C^\infty(\overline{\Omega})$, carry out integration by parts, and use (9.20.3) to get rid of volume terms:

$$\int_\Omega \sigma(\boldsymbol{x})\frac{\partial u}{\partial t}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega \underbrace{\left(\sigma(\boldsymbol{x})\frac{\partial u}{\partial t}(\boldsymbol{x},t) - \Delta u(\boldsymbol{x},t)\right)}_{=0} v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0 \tag{9.20.4}$$

for all $v \in C^\infty(\overline{\Omega})$,

$$\int_{\partial\Omega} \mathbf{grad}\, u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in C^\infty(\overline{\Omega})\,.$$

Invoking the fundamental lemma of the calculus of variations on the boundary $\partial\Omega$ we end up with the homogeneous **Neumann boundary conditions**

$$\mathbf{grad}\, u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \text{for} \quad \boldsymbol{x} \in \partial\Omega\,.$$

SOLUTION of (9-20.b):

Refer to the general discussion of [Lecture → Section 9.2.4]. Concretely we have

$$\mathbf{M} = \left[\int_\Omega \sigma(\boldsymbol{x}) b_h^j(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x})\right]_{i,j=1}^N, \quad \mathbf{A} = \left[\int_\Omega \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\right]_{i,j=1}^N .$$

Both matrices are obviously symmetric, $\mathbf{M}$ is positive definite.

HINT 1 for (9-20.c): It is advisable to write an s-stage Runge-Kutta single step method according to [Lecture → Def. 7.3.3.1] and defined by the **Butcher scheme** [Lecture → (7.3.3.3)]

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & \ddots & & a_{2s} \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & \dots & & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} , \quad \mathbf{c}, \mathbf{b} \in \mathbb{R}^s, \quad \mathfrak{A} \in \mathbb{R}^{s,s} ,$$

in **stage form** [Lecture → Rem. 7.3.3.6]: When applied to the autonomous ODE $\frac{d\vec{\mu}}{dt} = \mathbf{f}(\vec{\mu})$ with state space $\mathbb{R}^N$, a single step $\vec{\mu}^{(k-1)} \to \vec{\mu}^{(k)}$ of size $\tau > 0$ can be realized as

$$\begin{aligned} &\vec{\gamma}_i \in \mathbb{R}^N: \quad \vec{\gamma}_i = \vec{\mu}^{(k-1)} + \tau \sum_{j=1}^{s} a_{i,j} \mathbf{f}(\vec{\gamma}_j) , \quad i = 1, \dots, s , \\ &\vec{\mu}^{(k)} := \vec{\mu}^{(k-1)} + \tau \sum_{i=1}^{s} b_i \mathbf{f}(\vec{\gamma}_i) . \end{aligned} \tag{9.20.7}$$

⌟

SOLUTION of (9-20.c):

First, write the stage form (9.20.7) for $s = 2$ and the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \hat{=} \quad \begin{array}{c|cc} \zeta & \zeta & 0 \\ 1 & 1-\zeta & \zeta \\ \hline & 1-\zeta & \zeta \end{array} \quad , \quad \zeta := 1 - \tfrac{1}{2}\sqrt{2} > 0 \, . \tag{9.20.6}$$

For the ODE $\frac{d\vec{\mu}}{dt} = \mathbf{f}(\vec{\mu})$ we arrive at

$$\begin{aligned} \vec{\gamma}_1 \in \mathbb{R}^N: \quad & \vec{\gamma}_1 = \vec{\mu}^{(k-1)} + \tau\zeta\mathbf{f}(\vec{\gamma}_1) \, , \\ \vec{\gamma}_2 \in \mathbb{R}^N: \quad & \vec{\gamma}_2 = \vec{\mu}^{(k-1)} + \tau(1-\zeta)\mathbf{f}(\vec{\gamma}_1) + \tau\zeta\mathbf{f}(\vec{\gamma}_2) \, , \\ & \vec{\mu}^{(k)} := \vec{\mu}^{(k-1)} + \tau(1-\zeta)\mathbf{f}(\vec{\gamma}_1) + \tau\zeta\mathbf{f}(\vec{\gamma}_2) \, . \end{aligned} \tag{9.20.8}$$

We notice that $\vec{\mu}^{(k)} = \vec{\gamma}_2$! This is typical of L-stable RK-SSMs for which the last row of the Butcher matrix $\mathfrak{A}$ agrees with the row vector $[b_1, \ldots, b_s]$, recall [Lecture → Rem. 7.3.4.18].

Now we rewrite the formulas (9.20.8) for $\mathbf{f}(\vec{\mu}) := -\mathbf{M}^{-1}\mathbf{A}\vec{\mu}$, that is, we apply the 2-stage RK-SSM to the MOL-ODE (9.20.5):

$$\begin{aligned} \vec{\gamma}_1 \in \mathbb{R}^N: \quad & \vec{\gamma}_1 = \vec{\mu}^{(k-1)} - \tau\zeta\mathbf{M}^{-1}\mathbf{A}\vec{\gamma}_1 \, , \\ & \vec{\mu}^{(k)} = \vec{\mu}^{(k-1)} - \tau(1-\zeta)\mathbf{M}^{-1}\mathbf{A}\vec{\gamma}_1 - \tau\zeta\mathbf{M}^{-1}\mathbf{A}\vec{\mu}^{(k)} \, . \end{aligned} \tag{9.20.9}$$

We multiply both equations with $\mathbf{M}$ and rearrange:

$$\begin{aligned} \vec{\gamma}_1 \in \mathbb{R}^N: \quad & (\mathbf{M} + \tau\zeta\mathbf{A})\vec{\gamma}_1 = \mathbf{M}\vec{\mu}^{(k-1)} \, , \\ & (\mathbf{M} + \tau\zeta\mathbf{A})\vec{\mu}^{(k)} = \mathbf{M}\vec{\mu}^{(k-1)} - \tau(1-\zeta)\mathbf{A}\vec{\gamma}_1 \, . \end{aligned} \tag{9.20.10}$$

Renaming $\vec{\gamma} \leftarrow \vec{\gamma}_1$ this is equivalent to the linear system of equations

$$\begin{bmatrix} \mathbf{M} + \tau\zeta\mathbf{A} & \mathbf{O} \\ \tau(1-\zeta)\mathbf{A} & \mathbf{M} + \tau\zeta\mathbf{A} \end{bmatrix} \begin{bmatrix} \vec{\gamma} \\ \vec{\mu}^{(k)} \end{bmatrix} = \begin{bmatrix} \mathbf{M} \\ \mathbf{M} \end{bmatrix} \vec{\mu}^{(k-1)} \, . \tag{9.20.11}$$

SOLUTION of (9-20.d):

Of course, we appeal to

Theorem [Lecture → Thm. 9.2.8.5]. Convergence of solutions of fully discrete parabolic evolution problems

Assume that

- *the solution of the parabolic IBVP [Lecture → (9.2.1.6)]–[Lecture → (9.2.1.9)] is "sufficiently smooth" (both in space and time),*
- *its spatial Galerkin finite element discretization relies on degree p Lagrangian finite elements (→ [Lecture → Section 2.6]) on uniformly shape-regular families of meshes,*
- *timestepping is based on an L(π)-stable single step method of order q with uniform timestep $\tau > 0$.*

Then we can expect an asymptotic behavior of the total discretization error according to

$$\left(\tau \sum_{j=1}^{M} \left| u(\tau j) - u_h^{(j)} \right|_{H^1(\Omega)}^2 \right)^{\frac{1}{2}} \leq C\left(h_{\mathcal{M}}^p + \tau^q\right) , \tag{9.20.12}$$

where $C > 0$ must not depend on the meshwidth $h_{\mathcal{M}}$ and τ.

This result will be applied with

- $p = 2$ (quadratic Lagrangian finite elements), and
- $q = 2$ (second-order SDIRK-2 timestepping).

So we immediately get

$$\left(\tau \sum_{j=1}^{M} \left| u(\tau j) - u_h^{(j)} \right|_{H^1(\Omega)}^2 \right)^{\frac{1}{2}} \leq C\left(h_{\mathcal{M}}^2 + \tau^2\right) . \tag{9.20.13}$$

We can expect asymptotic algebraic convergence with rate 2 in the case of regular uniform mesh refinement when $h_{\mathcal{M}} \to 0$ and $\tau \to 0$.

SOLUTION of (9-21.a):

We face **essential boundary conditions** directly built into the trial and test space $H_0^1(\Omega)$,

$$u = 0 \quad \text{on} \quad \partial\Omega \times]0,T[\ . \tag{9.21.4}$$

To tease out the governing PDE, we remove all derivative from the test function v by means of integration by parts using [Lecture $\rightarrow$ Thm. 1.5.2.7]. This converts the first equation of (9.21.1) into

$$\int_\Omega \left(\sigma(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t) - \operatorname{div} \frac{1}{\mu(\boldsymbol{x})} \mathbf{grad}\, u(\boldsymbol{x},t) + U(t)\sigma(\boldsymbol{x}) \right) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0$$

for all $v \in H_0^1(\Omega)$. Boundary terms can be dropped since the test function vanishes on $\partial\Omega$. Then, the fundamental lemma of the calculus of variations [Lecture $\rightarrow$ Lemma 1.5.3.4] gives the "PDE"

$$\sigma(\boldsymbol{x}) \frac{\partial u}{\partial t}(\boldsymbol{x},t) - \operatorname{div}\big(\frac{1}{\mu(\boldsymbol{x})} \mathbf{grad}\, u(\boldsymbol{x},t) \big) + U(t)\sigma(\boldsymbol{x}) 1_{\Omega_C}(\boldsymbol{x}) = 0 \quad \text{in} \quad \Omega \ . \tag{9.21.5}$$

Here 1_{Ω_C} designates the **characteristic function** of the "wire domain" Ω_C,

$$1_{\Omega_C}(\boldsymbol{x}) = \begin{cases} 1 & \text{for} \quad \boldsymbol{x} \in \Omega_C \ , \\ 0 & \text{elsewhere.} \end{cases} \tag{9.21.6}$$

Yet, 1_{Ω_C} can also be dropped, because the information is already contained in the definition of σ.

The second equation of (9.21.1) can simply be copied:

$$\int_\Omega \sigma(\boldsymbol{x}) u(\boldsymbol{x},t) \,\mathrm{d}\boldsymbol{x} = I(t) \quad \text{for all } 0 \le t \le T \ . \tag{9.21.7}$$

SOLUTION of (9-21.b):

We match the different parts of (9.21.1) and of (9.21.8).

- $\mathbf{A}$ is the $\mathcal{S}^0_{2,0}(\mathcal{M})$ Galerkin matrix for the bilinear form $(u,v) \mapsto \int_\Omega \mu(\boldsymbol{x})^{-1} \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$.

$$\blacktriangleright \quad \mathbf{A} = \left[\int_\Omega \mu(\boldsymbol{x})^{-1} \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right]_{i,j=1}^N \in \mathbb{R}^{N,N} . \tag{9.21.9}$$

- $\mathbf{M}$ is the $\mathcal{S}^0_{2,0}(\mathcal{M})$ Galerkin matrix for the bilinear form $(u,v) \mapsto \int_\Omega \sigma(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$.

$$\blacktriangleright \quad \mathbf{M} = \left[\int_\Omega \sigma(\boldsymbol{x}) b_h^j(\boldsymbol{x}) b_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right]_{i,j=1}^N \in \mathbb{R}^{N,N} . \tag{9.21.10}$$

- $\mathbf{q}$ is the vector representing the linear form $v \mapsto \int_\Omega \sigma(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$ in $\mathcal{S}^0_{2,0}(\mathcal{M})$.

$$\blacktriangleright \quad \mathbf{q} = \left[\int_\Omega \sigma(\boldsymbol{x}) b_h^i(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \right]_{i=1}^N \in \mathbb{R}^N . \tag{9.21.11}$$

- We directly read off: $\psi(t) = I(t)$!
- All other vectors and scalars vanish,

$$\mathbf{d} = 0 \quad , \quad \alpha = 0 \quad , \quad \beta = 0 \quad , \quad \vec{\boldsymbol{\varphi}}(t) \equiv \mathbf{0} .$$

  In fact, this could have been concluded immediately from inspecting Code 9.21.22.

HINT 1 for (9-21.c): Note that at the point where you should insert your code the variables `At` and `Mt` contain the *full* Galerkin matrices; boundary conditions have not yet been taken into account. ⌟

HINT 2 for (9-21.c): The entries of the vector $\mathbf{q}$ can be obtained by left-multiplying a suitable vector with `Mt` using the method `MatVecMult()`. ⌟

HINT 3 for (9-21.c): Entries of an **lf::assemble::COOMatrix** object can set by the method `AddToEntry` ⌟

SOLUTION of (9-21.c):

The task is to initialize the vector $\mathbf{q} \in \mathbb{R}^N$ and insert it and its transpose into the bottom row and right column of $\widetilde{\mathbf{A}}$.

Exploit the **partition of unity property** of the global shape functions $\widehat{b}_h^i$ of the *full* Lagrangian finite element spapce $\mathcal{S}_2^0(\mathcal{M})$ (no boundary conditions imposed!)

$$\sum_{j=1}^{\widehat{N}} \widehat{b}_h^j(\boldsymbol{x}) = 1 \quad \forall \boldsymbol{x} \in \Omega\,, \quad \widehat{N} := \dim \mathcal{S}_2^0(\mathcal{M})\,. \tag{9.21.14}$$

First, we also compute the counterpart $\widehat{\mathbf{q}}$ of $\mathbf{q}$ defined on the *full* Lagrangian finite element space $\mathcal{S}_2^0(\mathcal{M})$. The actual $\mathbf{q}$ will automatically emerge from the elimination of the degrees of freedom on the boundary as done in the last part of the function by calling `FIxFlaggedSolutionComponents()`.

We have found

$$\widehat{\mathbf{q}} = \left[\int_\Omega \sigma(\boldsymbol{x})\widehat{b}_h^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i=1}^{\widehat{N}} \overset{(9.21.14)}{=} \left[\sum_{j=1}^{\widehat{N}} \int_\Omega \sigma(\boldsymbol{x})\widehat{b}_h^i(\boldsymbol{x})\widehat{b}_h^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i=1}^{\widehat{N}}$$

$$= \left[\int_\Omega \sigma(\boldsymbol{x})\widehat{b}_h^i(\boldsymbol{x})\widehat{b}_h^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right]_{i,j=1}^{\widehat{N}} \begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix} \overset{(9.21.10)}{=} \widehat{\mathbf{M}} \begin{bmatrix}1\\ \vdots \\ 1\end{bmatrix},$$

where $\widehat{\mathbf{M}}$ is the *full* $\mathcal{S}_2^0(\mathcal{M})$ Galerkin matrix for the bilinear form $(u,v) \mapsto \int_\Omega \sigma(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, that is, global shape functions on $\partial\Omega$ are included.

C++ code 9.21.15: Function `buildExtMOLMatrices()`: initialization of extended Galerkin matrices

```
2  std::pair<lf::assemble::COOMatrix<double>, lf::assemble::COOMatrix<double>>
3  buildExtMOLMatrices(std::shared_ptr<const lf::fe::ScalarFESpace<double>> fes_p,
4                      lf::mesh::utils::CodimMeshDataSet<bool> Oc_flags,
5                      double sigma_c, double mu_c) {
6    // Obtain handle to dof handler and mesh
7    const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
8    const std::shared_ptr<const lf::mesh::Mesh> mesh_p = dofh.Mesh();
9    // Dimension of finite element space
10   const lf::base::size_type N(dofh.NumDofs());
11   // Matrix in triplet format holding Galerkin matrix, zero initially.
12   lf::assemble::COOMatrix<double> Mt(N + 1, N + 1);
13   lf::assemble::COOMatrix<double> At(N + 1, N + 1);
14   // Mesh function providing conductivity
15   MeshFunctionPWConst mf_sigma(Oc_flags, sigma_c, 0.0);
16   // Mesh function providing permeability mu
17   LF_ASSERT_MSG(mu_c > 0, "mu must be positive");
18   MeshFunctionPWConst mf_mu(Oc_flags, 1.0 / mu_c, 1.0);
19   // ELEMENT_MATRIX_PROVIDERs:  LehrFEM utility classes
20   lf::fe::MassElementMatrixProvider emp_M(fes_p, mf_sigma);
21   lf::fe::DiffusionElementMatrixProvider emp_A(fes_p, mf_mu);
22   // Invoke assembly on cells (co-dimension = 0)
23   lf::assemble::AssembleMatrixLocally(0, dofh, dofh, emp_M, Mt);
```

```
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, emp_A, At);

  // Fill last row and column of At; exploit partition of unity property of
  // global shape functions:  the vector c is the product of M and a vector of
  // all ones.
  Eigen::VectorXd vec_one = Eigen::VectorXd::Constant(N + 1, 1.0);
  vec_one[N] = 0.0;
  const Eigen::VectorXd c = Mt.MatVecMult(1.0, vec_one);
  for (int j = 0; j < N; ++j) {
    At.AddToEntry(N, j, c[j]);
    At.AddToEntry(j, N, c[j]);
  }
  // Suppress basis functions on the boundary
  // Flag any entity located on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p)};
  // Flag vector for d.o.f.  on the boundary
  std::vector<std::pair<bool, double>> ess_dof_select(N + 1, {false, 0.0});
  // Run through all d.o.f.s and check whether they are associated with a mesh
  // entity on the boundary.  Then mark those d.o.f.s in the flag array
  for (lf::assemble::gdof_idx_t j = 0; j < N; ++j) {
    if (bd_flags(dofh.Entity(j))) {
      ess_dof_select[j].first = true;
      ess_dof_select[j].second = 0.0;
    }
  }
  // Here we have to use FIxFlaggedSolutionComponents, because we have to
  // decouple d.o.f.s on the boundary completely from the other d.o.f.s.
  // FixFlaggedSolutionCompAlt would not achieve this
  Eigen::VectorXd dummy{Eigen::VectorXd::Zero(N + 1)};
  // Modify the matrices as in [Lecture → (2.7.6.15)]
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&ess_dof_select](lf::assemble::glb_idx_t dof_idx)
          -> std::pair<bool, double> { return ess_dof_select[dof_idx]; },
      At, dummy);
  lf::assemble::FixFlaggedSolutionComponents<double>(
      [&ess_dof_select](lf::assemble::glb_idx_t dof_idx)
          -> std::pair<bool, double> { return ess_dof_select[dof_idx]; },
      Mt, dummy);
  return {Mt, At};
}
```

HINT 1 for (9-21.d): It is advisable to write an s-stage Runge-Kutta single step method according to [Lecture → Def. 7.3.3.1] and defined by the **Butcher scheme** [Lecture → (7.3.3.3)]

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & \ddots & & a_{2s} \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & \dots & & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \quad , \quad \mathbf{c}, \mathbf{b} \in \mathbb{R}^s, \quad \mathfrak{A} \in \mathbb{R}^{s,s} ,$$

in **stage form** [Lecture → Rem. 7.3.3.6]: When applied to the non-autonomous ODE $\frac{d\vec{\mu}}{dt} = \mathbf{f}(t, \vec{\mu})$ with state space $\mathbb{R}^{N+1}$, a single step $\vec{\mu}^{(k-1)} \to \vec{\mu}^{(k)}$ of size $\tau > 0$ can be realized as

$$\begin{aligned} \vec{\gamma}_i \in \mathbb{R}^{N+1}: \quad \vec{\gamma}_i &= \vec{\mu}^{(k-1)} + \tau \sum_{j=1}^{s} a_{i,j} \mathbf{f}(t_{k-1} + c_j \tau, \vec{\gamma}_j) , \quad i = 1, \dots, s , \\ \vec{\mu}^{(k)} &:= \vec{\mu}^{(k-1)} + \tau \sum_{i=1}^{s} b_i \mathbf{f}(t_{k-1} + c_i \tau, \vec{\gamma}_i) . \end{aligned} \tag{9.21.17}$$

⌟

HINT 2 for (9-21.d): For the derivations you may (temporarily) assume that $\widetilde{\mathbf{M}}$ is invertible. No inverses of $\widetilde{\mathbf{M}}$ must occur in the final expressions, of course. ⌟

SOLUTION of (9-21.d):

First, write the stage form (9.21.17) for $s = 2$ and the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \widehat{=} \quad \begin{array}{c|cc} \zeta & \zeta & 0 \\ 1 & 1-\zeta & \zeta \\ \hline & 1-\zeta & \zeta \end{array} \quad , \quad \zeta := 1 - \tfrac{1}{2}\sqrt{2} > 0 \, . \tag{9.21.16}$$

For the ODE $\frac{d\vec{\mu}}{dt} = \mathbf{f}(t, \vec{\mu})$ we arrive at

$$\begin{aligned} \vec{\gamma}_1 \in \mathbb{R}^N\text{:} \quad & \vec{\gamma}_1 = \vec{\mu}^{(k-1)} + \tau\zeta \mathbf{f}(t_{k-1} + \zeta\tau, \vec{\gamma}_1) \, , \\ \vec{\gamma}_2 \in \mathbb{R}^N\text{:} \quad & \vec{\gamma}_2 = \vec{\mu}^{(k-1)} + \tau(1-\zeta)\mathbf{f}(t_{k-1} + 1\cdot\tau, \vec{\gamma}_1) + \tau\zeta \mathbf{f}(t_{k-1} + 1\cdot\tau, \vec{\gamma}_2) \, , \\ & \vec{\mu}^{(k)} := \vec{\mu}^{(k-1)} + \tau(1-\zeta)\mathbf{f}(t_{k-1} + \zeta\tau, \vec{\gamma}_1) + \tau\zeta \mathbf{f}(t_{k-1} + 1\cdot\tau, \vec{\gamma}_2) \, . \end{aligned} \tag{9.21.18}$$

We notice that $\vec{\mu}^{(k)} = \vec{\gamma}_2$! This is typical of L-stable RK-SSMs for which the last row of the Butcher matrix $\mathfrak{A}$ agrees with the row vector $[b_1, \ldots, b_s]$, recall [Lecture → Rem. 7.3.4.18].

Now we rewrite the formulas (9.21.18) for $\mathbf{f}(\vec{\mu}) := -\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}}\vec{\mu} + \widetilde{\mathbf{M}}^{-1}\vec{\rho}(t)$, that is, we apply the 2-stage RK-SSM to the MOL-ODE (9.21.12):

$$\begin{aligned} \vec{\gamma}_1 \in \mathbb{R}^{N+1}\text{:} \quad \vec{\gamma}_1 &= \vec{\mu}^{(k-1)} - \tau\zeta\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}}\vec{\gamma}_1 + \tau\zeta\widetilde{\mathbf{M}}^{-1}\vec{\rho}(t_{k-1} + \zeta\tau) \, , \\ \vec{\mu}^{(k)} &= \vec{\mu}^{(k-1)} - \tau(1-\zeta)\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}}\vec{\gamma}_1 \quad + \tau(1-\zeta)\widetilde{\mathbf{M}}^{-1}\vec{\rho}(t_{k-1} + \tau) - \\ &\qquad \tau\zeta\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}}\vec{\mu}^{(k)} \qquad + \tau\zeta\widetilde{\mathbf{M}}^{-1}\vec{\rho}(t_{k-1} + \tau) \, . \\ &= \vec{\mu}^{(k-1)} - \tau(1-\zeta)\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}}\vec{\gamma}_1 - \tau\zeta\widetilde{\mathbf{M}}^{-1}\widetilde{\mathbf{A}}\vec{\mu}^{(k)} + \tau\widetilde{\mathbf{M}}^{-1}\vec{\rho}(t_k) \end{aligned} \tag{9.21.19}$$

We multiply both equations with $\widetilde{\mathbf{M}}$ and rearrange:

$$\begin{aligned} \vec{\gamma}_1 \in \mathbb{R}^{N+1}\text{:} \quad (\widetilde{\mathbf{M}} + \tau\zeta\widetilde{\mathbf{A}})\vec{\gamma}_1 &= \widetilde{\mathbf{M}}\vec{\mu}^{(k-1)} \qquad + \tau\zeta\vec{\rho}(t_{k-1} + \zeta\tau) \, , \\ (\widetilde{\mathbf{M}} + \tau\zeta\widetilde{\mathbf{A}})\vec{\mu}^{(k)} &= \widetilde{\mathbf{M}}\vec{\mu}^{(k-1)} - \tau(1-\zeta)\widetilde{\mathbf{A}}\vec{\gamma}_1 \quad + \tau\vec{\rho}(t_k) \, . \end{aligned} \tag{9.21.20}$$

This is equivalent to the linear system of equations

$$\begin{bmatrix} \widetilde{\mathbf{M}} + \tau\zeta\widetilde{\mathbf{A}} & \mathbf{O} \\ \tau(1-\zeta)\widetilde{\mathbf{A}} & \widetilde{\mathbf{M}} + \tau\zeta\widetilde{\mathbf{A}} \end{bmatrix} \begin{bmatrix} \vec{\gamma}_1 \\ \vec{\mu}^{(k)} \end{bmatrix} = \begin{bmatrix} \widetilde{\mathbf{M}} \\ \widetilde{\mathbf{M}} \end{bmatrix} \vec{\mu}^{(k-1)} + \tau \begin{bmatrix} \zeta\vec{\rho}(t_{k-1} + \zeta\tau) \\ \vec{\rho}(t_k) \end{bmatrix} . \tag{9.21.21}$$

HINT 1 for (9-21.e): LEHRFEM++'s **lf::assemble::COOMatrix** class offers a member function `MatVecM`
for computing a matrix×vector product. ⌟

SOLUTION of (9-21.e):

❶ Direct implementation of (9.21.20), solution of a block-triangular linear system of equations by block forward substitution. First solve for $\vec{\gamma}_1$ and then for $\vec{\mu}^{(k)}$.

C++ code 9.21.22: Function `sdirMagDiffWire()`: SDIRK-2 timestepping, stage-based implementation

```
2  template <typename SCALARFUNCTOR, typename RECORDER = std::function<
3                                          void(double, const Eigen::VectorXd &)>>
4  void sdirkMagDiffWire(
5      std::shared_ptr<const lf::fe::ScalarFESpace<double>> fes_p,
6      lf::mesh::utils::CodimMeshDataSet<bool> Oc_flags, double sigma_c,
7      double mu_c, SCALARFUNCTOR l_source, unsigned int M, double T_final,
8      Eigen::VectorXd &mu_vec,
9      RECORDER rec = [](double, const Eigen::VectorXd &) -> void {}) {
10   std::cout << "sdirMagDiffWire_org\n";
11   // Obtain extended Galerkin matrices as lf::asseemble::COOMatrix
12   auto [Mt, At] = buildExtMOLMatrices(fes_p, Oc_flags, sigma_c, mu_c);
13   const size_t N = At.cols() - 1;
14   LF_ASSERT_MSG(At.rows() == N + 1, "A must be square");
15   LF_ASSERT_MSG(Mt.rows() == N + 1, "M wrong row number");
16   LF_ASSERT_MSG(Mt.cols() == N + 1, "M wrong col number");
17   LF_ASSERT_MSG(mu_vec.size() == N + 1, "State vector must have N+1 componnts");
18   // We have built a sparse matrix containing the diagonal block(s) of
       the
19   // SDIRK2 linear system, because this has to be supplied to Eigen's
       built-in
20   // sparse elimination solver.
21   auto &At_triplet_vec{At.triplets()};
22   auto &Mt_triplet_vec{Mt.triplets()};
23   // Form weighted sum of sparse matrices in triplet format by merging
       triplet
24   // vectors
25   const double zeta = 1.0 - 0.5 * std::sqrt(2.0);  // SDIRK parameter
26   const double tau = T_final / M;
27   std::vector<Eigen::Triplet<double>> Dt_triplet_vec{};
28   for (const auto &triplet : Mt_triplet_vec) {
29     Dt_triplet_vec.push_back(triplet);
30   }
31   for (const auto &triplet : At_triplet_vec) {
32     Dt_triplet_vec.emplace_back(triplet.row(), triplet.col(),
33                                 zeta * tau * triplet.value());
34   }
35   // Convert to CRS format
36   Eigen::SparseMatrix<double> D(N + 1, N + 1);
37   D.setFromTriplets(Dt_triplet_vec.begin(), Dt_triplet_vec.end());
38   D.makeCompressed();
39   // For the sake of efficiency:  LU-decomposition outside the actual
40   // timestepping loop
41   Eigen::SparseLU<Eigen::SparseMatrix<double>> DLU;
42   DLU.compute(D);
43   LF_VERIFY_MSG(DLU.info() == Eigen::Success, "LU decomposition failed");
44
45 // Main timestepping loop
46 // A few auxliary vectors
47   // Right hand side vector
```

```
  Eigen::VectorXd rhs = Eigen::VectorXd::Zero(N + 1);
  // Vector holding M * current state
  Eigen::VectorXd Mmu(N + 1);
  // Temporary vector
  Eigen::VectorXd tmp(N + 1);
  // Current time t_{k-1}
  double t = 0.0;
  for (int k = 1; k <= M; ++k, t += tau) {
    rhs = Mt.MatVecMult(1.0, mu_vec);
    const double rhsN = rhs[N];
    rhs[N] += zeta * tau * l_source(t + zeta * tau);
    tmp = DLU.solve(rhs);
    LF_VERIFY_MSG(DLU.info() == Eigen::Success, "Solving LSE failed");
    rhs[N] = rhsN + tau * l_source(t + tau);
    At.MatVecMult(-tau * (1 - zeta), tmp, rhs);
    mu_vec = DLU.solve(rhs);
    rec(t, mu_vec);
  }
}
```

❷ As an alternative also the formulas of [Lecture → § 9.2.7.6] can be used. For a general RK-SSM they read

$$\vec{\kappa}_i \in \mathbb{R}^{N+1}: \quad \widetilde{\mathbf{M}}\vec{\kappa}_i + \sum_{j=1}^{s} \tau a_{i,j} \widetilde{\mathbf{A}}\vec{\kappa}_j = \vec{\rho}(t_{k-1} + c_i\tau) - \widetilde{\mathbf{A}}\vec{\mu}^{(k-1)}\,, \quad i = 1,\ldots,s\,, \quad \text{[Lecture → (9.2.7.7)]}$$

$$\vec{\mu}^{(k)} = \vec{\mu}^{(k-1)} + \tau \sum_{i=1}^{s} \vec{\kappa}_i b_i\,.$$

Specialized to the SDIRK-2 method this means:

$$\begin{aligned}
\widetilde{\mathbf{M}}\vec{\kappa}_1 + \tau\zeta\widetilde{\mathbf{A}}\vec{\kappa}_1 &= \vec{\rho}(t_{k-1} + \zeta\tau) - \widetilde{\mathbf{A}}\vec{\mu}^{(k-1)}\,,\\
\widetilde{\mathbf{M}}\vec{\kappa}_2 + \tau(1-\zeta)\widetilde{\mathbf{A}}\vec{\kappa}_1 + \tau\zeta\widetilde{\mathbf{A}}\vec{\kappa}_2 &= \vec{\rho}(t_{k-1} + \tau) - \widetilde{\mathbf{A}}\vec{\mu}^{(k-1)}\,,\\
\vec{\mu}^{(k)} &= \vec{\mu}^{(k-1)} + \tau(1-\zeta)\vec{\kappa}_1 + \tau\zeta\vec{\kappa}_2\,.
\end{aligned}$$

Close to implementation is the equivalent formulation

$$\begin{aligned}
(\widetilde{\mathbf{M}} + \tau\zeta\widetilde{\mathbf{A}})\vec{\kappa}_1 &= \vec{\rho}(t_{k-1} + \zeta\tau) - \widetilde{\mathbf{A}}\vec{\mu}^{(k-1)}\,, \qquad (9.21.23)\\
(\widetilde{\mathbf{M}} + \tau\zeta\widetilde{\mathbf{A}})\vec{\kappa}_2 &= \vec{\rho}(t_{k-1} + \tau) - \tau(1-\zeta)\widetilde{\mathbf{A}}\vec{\kappa}_1 - \widetilde{\mathbf{A}}\vec{\mu}^{(k-1)}\,,\\
\vec{\mu}^{(k)} &= \vec{\mu}^{(k-1)} + \tau(1-\zeta)\vec{\kappa}_1 + \tau\zeta\vec{\kappa}_2\,. \qquad (9.21.24)
\end{aligned}$$

C++ code 9.21.25: Function `sdirMagDiffWire_alt()`: SDIRK-2 timestepping based on increments

```
template <typename SCALARFUNCTOR, typename RECORDER = std::function<
                                    void(double, const Eigen::VectorXd &)>>
void sdirkMagDiffWire_alt(
    std::shared_ptr<const lf::fe::ScalarFESpace<double>> fes_p,
    lf::mesh::utils::CodimMeshDataSet<bool> Oc_flags, double sigma_c,
    double mu_c, SCALARFUNCTOR l_source, unsigned int M, double T_final,
    Eigen::VectorXd &mu_vec,
```

```
    RECORDER rec = [](double, const Eigen::VectorXd &) -> void {}) {
  std::cout << "sdirMagDiffWire_alt\n";
  // Obtain extended Galerkin matrices as lf::asseemble::COOMatrix
  auto [Mt, At] = buildExtMOLMatrices(fes_p, Oc_flags, sigma_c, mu_c);
  const size_t N = At.cols() - 1;
  LF_ASSERT_MSG(At.rows() == N + 1, "A must be square");
  LF_ASSERT_MSG(Mt.rows() == N + 1, "M wrong row number");
  LF_ASSERT_MSG(Mt.cols() == N + 1, "M wrong col number");
  LF_ASSERT_MSG(mu_vec.size() == N + 1, "State vector must have N+1 componnts");
  // We have built a sparse matrix containing the diagonal block(s) of the
  // SDIRK2 linear system, because this has to be supplied to Eigen's built-in
  // sparse elimination solver.
  auto &At_triplet_vec{At.triplets()};
  auto &Mt_triplet_vec{Mt.triplets()};
  // Form weighted sum of sparse matrices in triplet format by merging triplet
  // vectors
  const double zeta = 1.0 - 0.5 * std::sqrt(2.0);  // SDIRK parameter
  const double tau = T_final / M;
  std::vector<Eigen::Triplet<double>> Dt_triplet_vec{};
  for (const auto &triplet : Mt_triplet_vec) {
    Dt_triplet_vec.push_back(triplet);
  }
  for (const auto &triplet : At_triplet_vec) {
    Dt_triplet_vec.emplace_back(triplet.row(), triplet.col(),
                                zeta * tau * triplet.value());
  }
  // Convert to CRS format
  Eigen::SparseMatrix<double> D(N + 1, N + 1);
  D.setFromTriplets(Dt_triplet_vec.begin(), Dt_triplet_vec.end());
  D.makeCompressed();
  // For the sake of efficiency:  LU-decomposition outside the actual
  // timestepping loop
  Eigen::SparseLU<Eigen::SparseMatrix<double>> DLU;
  DLU.compute(D);
  LF_VERIFY_MSG(DLU.info() == Eigen::Success, "LU decomposition failed");

  // Main timestepping loop, increment-based implementation
  // Current time t_{k-1}
  double t = 0.0;
  for (int k = 1; k <= M; ++k, t += tau) {
    const Eigen::VectorXd Amu = At.MatVecMult(1.0, mu_vec);
    Eigen::VectorXd rho_vec = Eigen::VectorXd::Zero(N + 1);
    rho_vec[N] = l_source(t + zeta * tau);
    const Eigen::VectorXd kappa1 = DLU.solve(rho_vec - Amu);
    LF_VERIFY_MSG(DLU.info() == Eigen::Success, "Solving LSE failed");
    rho_vec[N] = l_source(t + tau);
    const Eigen::VectorXd kappa2 = DLU.solve(
        rho_vec - tau * (1 - zeta) * At.MatVecMult(1.0, kappa1) - Amu);
    LF_VERIFY_MSG(DLU.info() == Eigen::Success, "Solving LSE failed");
    mu_vec += tau * (1 - zeta) * kappa1 + tau * zeta * kappa2;
    rec(t, mu_vec);
  }
}
```

SOLUTION of (9-3.a):

$$\vec{\boldsymbol{\mu}}^{(j)} = (\tau\mathbf{A} + \mathbf{M})^{-1}(\mathbf{M}\vec{\boldsymbol{\mu}}^{(j-1)}) \quad \text{or equivalently} \quad \mathbf{M}(\vec{\boldsymbol{\mu}}^{(j)} - \vec{\boldsymbol{\mu}}^{(j-1)}) = -\tau\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} \ .$$

The formulas are already given in [Lecture → Ex. 9.2.7.1]. Refer to this part of the lecture notes for further information.

SOLUTION of (9-3.c):

We repeat the considerations of [Lecture → § 9.2.7.35], which is just a special case of [Lecture → § 9.2.7.38].

We multiply both sides of the equation

$$\mathbf{M}(\vec{\boldsymbol{\mu}}^{(j)} - \vec{\boldsymbol{\mu}}^{(j-1)}) = -\tau\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)}$$

by $\mathbf{T}^\top$. This results in

$$\mathbf{T}^\top\mathbf{M}(\vec{\boldsymbol{\mu}}^{(j)} - \vec{\boldsymbol{\mu}}^{(j-1)}) = -\mathbf{T}^\top\tau\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)},$$

which we can rewrite as

$$\mathbf{T}^\top\mathbf{M}(\vec{\boldsymbol{\mu}}^{(j)} - \vec{\boldsymbol{\mu}}^{(j-1)}) = -\tau\mathbf{T}^\top\mathbf{A}\mathbf{T}\mathbf{T}^{-1}\vec{\boldsymbol{\mu}}^{(j)}$$

Therefore, denoting $\vec{\boldsymbol{\eta}}^{(j)} = \mathbf{T}^\top\mathbf{M}\vec{\boldsymbol{\mu}}^{(j)} = \mathbf{T}^{-1}\vec{\boldsymbol{\mu}}^{(j)}$ this becomes

$$\vec{\boldsymbol{\eta}}^{(j)} - \vec{\boldsymbol{\eta}}^{(j-1)} = -\tau\mathbf{T}^\top\mathbf{A}\mathbf{T}\vec{\boldsymbol{\eta}}^{(j)}\ .$$

Finally, observe $\mathbf{T}^{-1}\mathbf{M}^{-1}\mathbf{T}^{-\top} = \mathbf{I}$ and obtain

$$\vec{\boldsymbol{\eta}}^{(j)} - \vec{\boldsymbol{\eta}}^{(j-1)} = -\tau\mathbf{T}^\top\mathbf{A}\mathbf{T}\vec{\boldsymbol{\eta}}^{(j)} = -\tau\mathbf{D}\vec{\boldsymbol{\eta}}^{(j)}\ ,$$

which amounts to applying the implicit Euler method to the decoupled linear scalar equations $\dot{\eta}_i = -\lambda_i\eta_i$.

SOLUTION of (9-3.d):

First, since $\vec{\eta} = \mathbf{T}^\top \mathbf{M} \vec{\mu}$, we have $\vec{\mu} = \mathbf{M}^{-1} (\mathbf{T})^{-\top} \vec{\eta}$, so

$$\begin{aligned} \|\vec{\mu}\|_{\mathbf{M}} &= (\vec{\mu}^\top \mathbf{M} \vec{\mu})^{1/2} = \left((\mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta})^\top \mathbf{M} (\mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta}) \right)^{1/2} \\ &= (\vec{\eta}^\top \mathbf{T}^{-1} (\mathbf{M}^{-1})^\top \mathbf{M} \mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta})^{1/2} \\ &= (\vec{\eta}^\top \mathbf{T}^{-1} \mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta})^{1/2} = \|\vec{\eta}\| \ . \end{aligned}$$

Here, we used $\mathbf{T}^{-1} \mathbf{M}^{-1} \mathbf{T}^{-\top} = \mathbf{I}$ from (9.3.4). Next,

$$\begin{aligned} \|\vec{\mu}\|_{\mathbf{A}} &= (\vec{\mu}^\top \mathbf{A} \vec{\mu})^{1/2} = \left((\mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta})^\top \mathbf{A} (\mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta}) \right)^{1/2} \\ &= (\vec{\eta}^\top \mathbf{T}^{-1} (\mathbf{M}^{-1})^\top \mathbf{A} \mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta})^{1/2} \\ &= (\vec{\eta}^\top \mathbf{T}^{-1} \mathbf{M}^{-1} \mathbf{A} \mathbf{T} \mathbf{T}^{-1} \mathbf{M}^{-1} \mathbf{T}^{-\top} \vec{\eta})^{1/2} = (\vec{\eta}^\top \mathbf{D} \vec{\eta})^{1/2} \ . \end{aligned}$$

Notice that, again, we used $\mathbf{T}^{-1} \mathbf{M}^{-1} \mathbf{T}^{-\top} = (\mathbf{T}^\top \mathbf{M} \mathbf{T})^{-1} = \mathbf{I}$ and $\mathbf{A}\mathbf{T} = \mathbf{M}\mathbf{T}\mathbf{D}$.

SOLUTION of (9-3.e):

We get

$$\mathsf{m}(u_h, u_h) = \mathsf{m}\left(\sum_{i=1}^{N} \mu_i b_h^i, \sum_{j=1}^{N} \mu_j b_h^j\right) = \sum_{i=1}^{N}\sum_{j=1}^{N} \mu_i \mu_j \mathsf{m}(b_h^i, b_h^j) = \|\vec{\boldsymbol{\mu}}\|_{\mathbf{M}}^2$$

$$\mathsf{a}(u_h, u_h) = \mathsf{a}\left(\sum_{i=1}^{N} \mu_i b_h^i, \sum_{j=1}^{N} \mu_j b_h^j\right) = \sum_{i=1}^{N}\sum_{j=1}^{N} \mu_i \mu_j \mathsf{a}(b_h^i, b_h^j) = \|\vec{\boldsymbol{\mu}}\|_{\mathbf{A}}^2$$

It follows that (9.3.2) can be rewritten as

$$\exists \gamma > 0: \qquad \vec{\boldsymbol{\mu}}^\top \mathbf{A} \vec{\boldsymbol{\mu}} \geq \gamma \vec{\boldsymbol{\mu}}^\top \mathbf{M} \vec{\boldsymbol{\mu}}, \quad \forall \vec{\boldsymbol{\mu}} \in \mathbb{R}^N. \tag{9.3.6}$$

Using the transformation from Sub-problem (9-3.c): $\vec{\boldsymbol{\eta}} = \mathbf{T}^\top \mathbf{M} \vec{\boldsymbol{\mu}} = \mathbf{T}^{-1} \vec{\boldsymbol{\mu}}$, we have that

$$\vec{\boldsymbol{\mu}}^\top \mathbf{A} \vec{\boldsymbol{\mu}} \geq \gamma \vec{\boldsymbol{\mu}}^\top \mathbf{M} \vec{\boldsymbol{\mu}} \quad \forall \vec{\boldsymbol{\mu}} \in \mathbb{R}^N \iff \vec{\boldsymbol{\eta}}^\top \mathbf{D} \vec{\boldsymbol{\eta}} \geq \gamma \vec{\boldsymbol{\eta}}^\top \vec{\boldsymbol{\eta}} \quad \forall \vec{\boldsymbol{\eta}} \in \mathbb{R}^N\,, \tag{9.3.7}$$

which, since it has to be satisfied *for all vectors*, implies

$$\lambda_i \boldsymbol{\eta}_i^2 \geq \gamma \boldsymbol{\eta}_i^2\,, \quad \forall i \in \{1, \ldots, N\}\,.$$

From here it follows that

$$\gamma \leq \min\{\lambda_1, ..., \lambda_h\}\,.$$

HINT 1 for (9-3.f): You may use the Cauchy-Schwarz inequality

$$\vec{\xi}^{\top}\mathbf{M}\vec{\zeta} \leq \left\|\vec{\xi}\right\|_{\mathbf{M}}\left\|\vec{\zeta}\right\|_{\mathbf{M}}, \tag{9.3.9}$$

and the implicit Euler recursion formula from Sub-problem (9-3.a). ⌟

HINT 2 for (9-3.f): Another way to tackle the problem relies on the diagonalization technique explored in the previous sub-problems: rephrase (9.3.8) in terms of $\vec{\eta}^{(k)}$ and look at the implicit Euler method for these transformed coefficient vectors, see Sub-problem (9-3.c). ⌟

SOLUTION of (9-3.f):

I. (Direct estimate)

Start from the recursion of the implicit Euler method with timestep $\tau > 0$:

$$\mathbf{M}(\vec{\mu}^{(j)} - \vec{\mu}^{(j-1)}) = -\tau \mathbf{A}\vec{\mu}^{(j)} \,. \tag{9.3.10}$$

On both sides form the Euclidean inner product with $\vec{\mu}^{(j)}$ and use the definitions (9.3.5):

$$\blacktriangleright \quad \left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}}^2 - \left(\vec{\mu}^{(j)}\right)^\top \mathbf{M}\vec{\mu}^{(j-1)} = -\tau \left\|\vec{\mu^{(j)}}\right\|_{\mathbf{A}}^2 \tag{9.3.11}$$

$$\blacktriangleright \quad \left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}}^2 = \left(\vec{\mu}^{(j)}\right)^\top \mathbf{M}\vec{\mu}^{(j-1)} - \tau \left\|\vec{\mu^{(j)}}\right\|_{\mathbf{A}}^2 \,. \tag{9.3.12}$$

From (9.3.2) we infer

$$\exists \gamma > 0 : \qquad \vec{\mu}^\top \mathbf{A}\vec{\mu} \geq \gamma \vec{\mu}^\top \mathbf{M}\vec{\mu}, \quad \forall \vec{\mu} \in \mathbb{R}^N \,. \tag{9.3.6}$$

Together with the Cauchy-Schwarz inequality (9.3.9) this permits us to estimate the right-hand side of (9.3.12) from above:

$$\blacktriangleright \quad \left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}}^2 \leq \left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}} \left\|\vec{\mu}^{(j-1)}\right\|_{\mathbf{M}} - \tau\gamma \left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}}^2 \,.$$

Finally cancel $\left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}}$ on both sides to obtain the estimate (9.3.8).

II. (Diagonalization technique)

We already know that

$$\left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}} \leq \frac{1}{1+\gamma\tau} \left\|\vec{\mu}^{(j-1)}\right\|_{\mathbf{M}} \Leftrightarrow \left\|\vec{\eta}^{(j)}\right\| \leq \frac{1}{1+\gamma\tau} \left\|\vec{\eta}^{(j-1)}\right\| \,.$$

$$\vec{\eta}^{(j)} - \vec{\eta}^{(j-1)} = -\tau \mathbf{D}\vec{\eta}^{(j)} \Leftrightarrow \vec{\eta}^{(j)} = (1+\tau\mathbf{D})^{-1}\vec{\eta}^{(j-1)} \,.$$

$$\left\|\vec{\eta}^{(j)}\right\| = \left\|(1+\tau\mathbf{D})^{-1}\vec{\eta}^{(j-1)}\right\| \leq \frac{1}{1+\tau\gamma} \left\|\vec{\eta}^{(j-1)}\right\| \,,$$

because all diagonal entries of the diagonal matrix $\mathbf{D}$ are bounded from below by γ, see (9.3.7).

HINT 1 for (9-3.g): Here it is recommended to use diagonalization and the result of Sub-problem (9-3.d). ⌟

SOLUTION of (9-3.g):

$$
\begin{aligned}
\left\|\vec{\boldsymbol{\mu}}^{(j)}\right\|_{\mathbf{A}} \leq \frac{1}{1+\gamma\tau}\left\|\vec{\boldsymbol{\mu}}^{(j-1)}\right\|_{\mathbf{A}} \quad &\Longleftrightarrow \quad \left((\vec{\boldsymbol{\eta}}^{(j)})^{\top} D\vec{\boldsymbol{\eta}}^{(j)}\right)^{1/2} \leq \frac{1}{1+\gamma\tau}\left((\vec{\boldsymbol{\eta}}^{(j-1)})^{\top} D\vec{\boldsymbol{\eta}}^{(j-1)}\right)^{1/2} \\
&\Longleftrightarrow \quad \left\|\sqrt{D}\boldsymbol{\eta}^{(j)}\right\| \leq \frac{1}{1+\gamma\tau}\left\|\sqrt{D}\boldsymbol{\eta}^{(j-1)}\right\| \\
&\Longleftrightarrow \quad \sqrt{\lambda_i}\left\|\boldsymbol{\eta}_i^{(j)}\right\| \leq \frac{1}{1+\gamma\tau}\sqrt{\lambda_i}\left\|\boldsymbol{\eta}_i^{(j-1)}\right\|, \quad \forall i \in \{1,\ldots,N\} \\
&\Longleftrightarrow \quad \left\|\boldsymbol{\eta}_i^{(j)}\right\| \leq \frac{1}{1+\gamma\tau}\left\|\boldsymbol{\eta}_i^{(j-1)}\right\|, \quad \forall i \in \{1,\ldots,N\}\,,
\end{aligned}
$$

which follows directly from Sub-problem (9-3.f).

HINT 1 for (9-3.h): Bound $\left\|\vec{\mu}^{(j)}\right\|_{\mathbf{M}}$ and $\left\|\vec{\mu}^{(j)}\right\|_{\mathbf{A}}$ in terms of $\left\|\vec{\mu}^{(0)}\right\|_{\mathbf{M}}$ and $\left\|\vec{\mu}^{(0)}\right\|_{\mathbf{A}}$, respectively. Then, for fixed t consider $\tau \to 0$ and $j \approx t/\tau \to \infty$. Remember the limit

$$e^t = \lim_{n\to\infty} (1 + \tfrac{t}{n})^n \,, \quad t \in \mathbb{R} \,.$$

SOLUTION of (9-3.h):

[Lecture → Lemma 9.2.3.8] tells us that

$$\|u(t)\|_{L^2(\Omega)} \leq e^{-\gamma t}\|u_0\|_{L^2(\Omega)} \quad , \qquad |u(t)|_{H^1(\Omega)} \leq e^{-\gamma t}|u_0|_{H^1(\Omega)} \, .$$

Writing the approximate solution $u_h(t) = \sum_{i=1}^{N} \mu_i b_h^i$ we get

$$\|u(t)\|_{L^2(\Omega)} = \|\vec{\boldsymbol{\mu}}(t)\|_{\mathbf{M}} \quad , \qquad |u_h(t)|_{H^1(\Omega)} = \|\vec{\boldsymbol{\mu}}(t)\|_{\mathbf{A}} \, .$$

From the previous point we have that

$$\left\|\vec{\boldsymbol{\mu}}^{(j)}\right\|_{\mathbf{M}} \leq \frac{1}{1+\gamma\tau}\left\|\vec{\boldsymbol{\mu}}^{(j-1)}\right\|_{\mathbf{M}} = \left(\frac{1}{1+\gamma\tau}\right)^j \left\|\vec{\boldsymbol{\mu}}^{(0)}\right\|_{\mathbf{M}} \, .$$

We now consider that the interval $]0,t[=]0,t_j[$ has been discretized into n intervals, with $\tau = t/n$ it follows that

$$\|\vec{\boldsymbol{\mu}}(t)\|_{\mathbf{M}} \leq \left(\frac{1}{1+\gamma t/n}\right)^n \|\vec{\boldsymbol{\mu}}(0)\|_{\mathbf{M}} = \left(1 - \frac{\gamma t}{n+\gamma t}\right)^n \|\vec{\boldsymbol{\mu}}(0)\|_{\mathbf{M}} \, ,$$

which becomes for $n \to \infty$

$$\|\vec{\boldsymbol{\mu}}(t)\|_{\mathbf{M}} \leq e^{-\gamma t}\|\vec{\boldsymbol{\mu}}(0)\|_{\mathbf{M}} \, .$$

In the same way,

$$\|\vec{\boldsymbol{\mu}}(t)\|_{\mathbf{A}} \leq e^{-\gamma t}\|\vec{\boldsymbol{\mu}}(0)\|_{\mathbf{A}} \, .$$

It follows that the relations from subproblems (9-3.f) and (9-3.g) converge to the ones from [Lecture → Lemma 9.2.3.8] in the limit $n \to \infty$.

HINT 1 for (9-4.a): Connect (9.4.2a) with a non-homogeneous Neumann boundary condition and remember how these are treated in a variational context, see [Lecture → Ex. 1.8.0.6]. A term with first-order derivatives in time will emerge. ⌟

SOLUTION of (9-4.a):

The derivation of a spatial variational formulation from the PDE (9.4.1) follows the procedure detailed in [Lecture → Section 1.8]. We first multiply (9.4.1) with a test function $v :]0,1[\to \mathbb{R}$ that vanishes in $x = 1$, because in that point we already know the solution ("do not test where the solution is known"). Then we perform integration by parts using the formula [Lecture → (1.5.1.11)].

$$\int_0^1 \frac{\partial^2 u}{\partial t^2}(x,t)\, v(x)\,\mathrm{d}x + c^2 \int_0^1 \frac{\partial u}{\partial x}(x,t)\,\frac{\partial v}{\partial x}(x)\,\mathrm{d}x + c^2 \frac{\partial u}{\partial x}(0,t)\, v(0) = 0 \quad \forall v\ ,$$

where the boundary term at $x = 1$ vanishes, because $v(1) = 0$.

Then we recast the boundary term at $x = 0$ using the absorbing boundary condition (9.4.2a) and obtain.

$$\int_0^1 \frac{\partial^2 u}{\partial t^2}(x,t)\, v(x)\,\mathrm{d}x + c^2 \int_0^1 \frac{\partial u}{\partial x}(x,t)\,\frac{\partial v}{\partial x}(x)\,\mathrm{d}x + c \frac{\partial u}{\partial t}(0,t)\, v(0) = 0 \quad \forall v\ .$$

The trial and test spaces at time t are

$$\begin{aligned} u(t) \in \widehat{V}(t) &:= \{v \in H^1(]0,1[) : v(1) = g(t)\}\ , \quad t \in]0,T[\ , \\ v \in V_0 &:= \{v \in H^1(]0,1[) : v(1) = 0\}\ , \quad t \in]0,T[\ . \end{aligned}$$

Note that the trial space $\widehat{V}(t)$ is a *time-dependent affine space* parallel to V_0, whereas the test space V_0 is a vector space that does not depend on time.

In an abstract fashion, analoguous to [Lecture → (9.3.1.8)], this variational problem can be rewritten as: seek $u : [0,T] \to \widehat{V}(t)$

$$\mathsf{m}(\ddot{u}, v) + \mathsf{b}(\dot{u}, v) + \mathsf{a}(u, v) = 0 \quad \forall v \in V_0\ , \tag{9.4.3}$$

where

$$\begin{aligned} \mathsf{m}(w,v) &:= \int_0^1 w(x)v(x)\,\mathrm{d}x\ , \\ \mathsf{b}(w,v) &:= c\, w(0)v(0)\ , \qquad w, v \in H^1(]0,1[)\ . \\ \mathsf{a}(w,v) &:= c^2 \int_0^1 \frac{dw}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x\ , \end{aligned} \tag{9.4.4}$$

Here the dots designate temporal derivatives.

HINT 1 for (9-4.b): The mass matrix will be diagonal because the use of this particular quadrature formula causes “mass lumping” [Lecture → Rem. 9.3.4.18]. The stiffness matrix has already been computed in [Lecture → Section 2.3]. ⌟

HINT 2 for (9-4.b): To take into account the non-zero time-dependent Dirichlet boundary conditions at $x = 1$, rely on the offset function technique according to [Lecture → Rem. 2.3.3.15]. ⌟

HINT 3 for (9-4.b): Consult [Lecture → § 2.3.3.1] and [Lecture → Rem. 2.3.3.3]. ⌟

SOLUTION of (9-4.b):

To take into account non-homogeneous Dirichlet boundary conditions we rely on the offset-function technique: We set

$$u_h(x,t) = \sum_{\ell=1}^{N} \mu_\ell(t) b_h^\ell(x) + g(t) b_h^{N+1}(x) \in \widehat{V}_h(t) \,,$$

with a time-dependent vector $t \mapsto \vec{\boldsymbol{\mu}}(t) = [\mu_\ell(t)]_{\ell=1}^N \in \mathbb{R}^N$ of basis expansion coefficients. We insert this expression into (9.4.3) choosing the test function from $V_{0,h}$:

$$\sum_{\ell=1}^{N} \Big\{ \ddot{\mu}_\ell(t) \mathsf{m}_h(b_h^\ell, b_h^i) + \dot{\mu}_\ell(t) \mathsf{b}(b_h^\ell, b_h^i) + \mu_\ell(t) \mathsf{a}(b_h^\ell, b_h^i) \Big\} +$$
$$\ddot{g}(t) \mathsf{m}_h(b_h^{N+1}, b_h^i) + \dot{g}(t) \mathsf{b}(b_h^{N+1}, b_h^i) + g(t) \mathsf{a}(b_h^{N+1}, b_h^i) = 0 \,, \quad i = 1, \ldots, N \,.$$

The notation m_h stresses that the $L^2(]0,1[)$-inner product $\mathsf{m}(\cdot,\cdot)$ is evaluated only approximately using numerical quadrature by means of the 1D trapezoidal rule.

Next, we can introduce and evaluate the Galerkin matrices, following the consideration of [Lecture → § 2.3.3.1],

$$\mathbf{M} := \left[\mathsf{m}_h(b_h^\ell, b_h^i) \right]_{i,\ell=1}^N = h \begin{bmatrix} \frac{1}{2} & & & & \\ & 1 & & & \\ & & \ddots & & \\ & & & 1 & \\ & & & & 1 \end{bmatrix} \in \mathbb{R}^{N,N} \,,$$

$$\mathbf{A} := \left[\mathsf{a}(b_h^\ell, b_h^i) \right]_{i,\ell=1}^N = \frac{c^2}{h} \begin{bmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix} \in \mathbb{R}^{N,N} \,,$$

$$\mathbf{B} := \left[\mathsf{b}(b_h^\ell, b_h^i) \right]_{i,\ell=1}^N = \begin{bmatrix} c & & & & \\ & 0 & & & \\ & & \ddots & & \\ & & & 0 & \\ & & & & 0 \end{bmatrix} \in \mathbb{R}^{N,N} \,,$$

and the time-dependent vector

$$\vec{\boldsymbol{\varphi}}(t) = - \left[\ddot{g}(t) \underbrace{\mathsf{m}_h(b_h^{N+1}, b_h^i)}_{=0} + \dot{g}(t) \underbrace{\mathsf{b}(b_h^{N+1}, b_h^i)}_{=0} + g(t) \mathsf{a}(b_h^{N+1}, b_h^i) \right]_{i=1}^N = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ \frac{c^2}{h} g(t) \end{bmatrix} \in \mathbb{R}^N \,,$$

because $\mathsf{m}_h(b_h^{N+1}, b_h^i) = \mathsf{b}(b_h^{N+1}, b_h^i) = 0$ for all $i = 1, \ldots, N$ and

$$\mathsf{a}(b_h^{N+1}, b_h^i) = \begin{cases} -\frac{c^2}{h} & \text{, if } i = N \,, \\ 0 & \text{, if } i \in \{1, \ldots, N-1\} \,. \end{cases}$$

Thus, we have found all constituent parts of the method-of-lines ODE.

Note that $V_{0,h} \neq \mathcal{S}^0_{1,0}(\mathcal{M})$, because one basis function is dropped. Therefore the matrix $\mathbf{A}$ obtained above is different from that given in [Lecture $\rightarrow$ (2.3.3.2)], which is the Galerkin matrix for the pure Dirichlet problem and an equidistant mesh. (9.4.7).

HINT 1 for (9-4.c): Remember that triplet-based initialization of sparse matrices of type **Eigen::SparseMa** in EIGEN can be skipped, if the maximal number of non-zero entries per row and column is known. This knowledge can be told the code via the `Eigen::SparseMatrix<T>::reserve()` member function, see EIGEN documentation. The following code illustrates its use.

C++11 code 9.4.10: `reserve()`-based initializatin of a sparse matrix in EIGEN ↪ GitLab

```
2  const int n = 20, m = 10;
3  // Set up zero sparse matrix with row major storage format
4  // This format is essential for being able to set the maximal
5  // number of non-zero entries per row.
6  Eigen::SparseMatrix<int, Eigen::RowMajor> X(n, m);
7  // Reserve space for at most nnz_row non-zero entries per row
8  const std::size_t nnz_row = 3;
9  X.reserve(Eigen::VectorXi::Constant(n, nnz_row));
10 // Initialize nnz_row entries per row
11 for (int row_idx = 0; row_idx < n; ++row_idx) {
12   for (int k = 0; k < nnz_row; ++k) {
13     X.coeffRef(row_idx, (row_idx * k) % m) += 1;
14   }
15 }
```

SOLUTION of (9-4.c):

Owing to (9.4.9) we can compute $\vec{\nu}^{(1/2)}$ from the initial values $\vec{\nu}_0$ and $\vec{\mu}_0$ and, thus, kickstart the leapfrog iteration:

$$2\mathbf{M}\vec{\nu}^{(1/2)} = -\tau\mathbf{A}\vec{\mu}_0 - \tau\mathbf{B}\vec{\nu}_0 + 2\mathbf{M}\vec{\nu}_0 + \tau\vec{\varphi}(0)\,. \tag{9.4.11}$$

$$\vec{\mu}^{(j+1)} := \vec{\mu}^{(j)} + \tau\vec{\nu}^{(j+1/2)}\,, \quad j \in \mathbb{N}_0\,, \tag{9.4.12}$$

$$(\mathbf{M} + \tfrac{1}{2}\tau\mathbf{B})\vec{\nu}^{(j+1/2)} := -\tau\mathbf{A}\vec{\mu}^{(j)} + (\mathbf{M} - \tfrac{1}{2}\tau\mathbf{B})\vec{\nu}^{(j-1/2)} + \tau\vec{\varphi}(t_j)\,, \quad j \in \mathbb{N}\,. \tag{9.4.13}$$

Note that for the special initial values from (9.4.2) we have $\vec{\nu}_0 = \vec{\mu}_0 = \vec{\varphi}(0) = \mathbf{0}$, which means $\vec{\nu}^{(1/2)} = \mathbf{0}$. This is why a special starting step can be skipped.

The following codes demonstrate only one possible implementation. It relies on three auxiliary functions that provide the $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrices for the bilinear forms a, b, and m_h, that is, the *full* Galerkin matrices over $\mathrm{Span}\{b_h^1, \ldots, b_h^N, b_h^{N+1}\}$. These are returned in CRS format as **Eigen::SparseMatrix** objects.

C++ code 9.4.14: Sub-problem (9-4.c): function `getA_full()` ↪ GitLab

```
2  Eigen::SparseMatrix<double> getA_full(unsigned int N, double c, double h) {
3    std::vector<Eigen::Triplet<double>> triplets;
4    triplets.reserve(3 * (N + 1) - 2);  // that many triplets needed
5    const double scale = c * c / h;
6    // store first row separately
7    triplets.push_back(Eigen::Triplet<double>(0, 0, scale));
8    triplets.push_back(Eigen::Triplet<double>(0, 1, -scale));
9    // loop over interior rows
10   for (unsigned i = 1; i < N; ++i) {
11     triplets.push_back(Eigen::Triplet<double>(i, i - 1, -scale));
12     triplets.push_back(Eigen::Triplet<double>(i, i, 2.0 * scale));
13     triplets.push_back(Eigen::Triplet<double>(i, i + 1, -scale));
14   }
15   // store last row separately
16   triplets.push_back(Eigen::Triplet<double>(N, N - 1, -scale));
17   triplets.push_back(Eigen::Triplet<double>(N, N, scale));
18   // Creat (N+1)x(N+1) sparse matrix in CRS format
19   Eigen::SparseMatrix<double> A(N + 1, N + 1);
20   A.setFromTriplets(triplets.begin(), triplets.end());
21   return A;
22 }
```

C++ code 9.4.15: Sub-problem (9-4.c): function `getB_full()` ↪ GitLab

```
2  Eigen::SparseMatrix<double> getB_full(unsigned int N, double c) {
3    Eigen::SparseMatrix<double> B(N + 1, N + 1);
4    // Just a single non-zero entry; we can afford to sete it directly
5    B.coeffRef(0, 0) = c;
6    return B;
7  }
```

C++ code 9.4.16: Sub-problem (9-4.c): function `getM_full()` ⇝ GitLab

```
Eigen::SparseMatrix<double> getM_full(unsigned int N, double h) {
  Eigen::SparseMatrix<double> M(N + 1, N + 1);
  M.setIdentity();  // Supposed to be efficient
  M *= h;
  // Modify two entries; efficiency does not matter much
  M.coeffRef(0, 0) = h / 2;
  M.coeffRef(N, N) = h / 2;
  return M;
}
```

The following code probably sacrifices a lot of efficiency, by relying on block operations applied to matrices in CRS format in order to extract sub-matrices, see comments in the code.

C++ code 9.4.17: Sub-problem (9-4.c): function `waveLeapfrogABC()` ⇝ GitLab

```
Eigen::MatrixXd waveLeapfrogABC(double c, double T, unsigned int N,
                                unsigned int m) {
  // N is also the number of cells of the mesh
  double h = 1.0 / N;
  // Obtain Galerkin matrices for truncated finite element space
  // Note that the functions get*_full return the matrices for the full finite
  // element space including the tent function located at x=1. Removing the last
  // row and column of that matrix amounts to dropping that basis function.
  // However, the efficiency of this block() operation in the case of sparse
  // matrices is in doubt, in particular, since the result is assigned to
  // another sparse matrix, which foils Eigen's expression template
  // optimization. The use of "auto" would be highly advisable here!
  Eigen::SparseMatrix<double> A = getA_full(N, c, h).block(0, 0, N, N);
  Eigen::SparseMatrix<double> B = getB_full(N, c).block(0, 0, N, N);
  Eigen::SparseMatrix<double> M = getM_full(N, h).block(0, 0, N, N);
  // Matrix for returning solution
  Eigen::MatrixXd R(m + 1, N + 1);
  Eigen::VectorXd mu = Eigen::VectorXd::Zero(N);    // = mu^(0)
  Eigen::VectorXd nu = Eigen::VectorXd::Zero(N);    // = nu^(-1/2)
  Eigen::VectorXd phi = Eigen::VectorXd::Zero(N);   // = phi(t_0)
  // Universally zero initial conditions make it possible to skip
  // the special initial step usually required for leapfrog.
  double tau = T / m;  // Timestep size
  // The diagonal matrix to be "inverted" in each timestep
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver(M + 0.5 * tau * B);
  for (int j = 0; j < m; ++j) {
    R.row(j).head(N) = mu.transpose();
    phi(N - 1) = c * c / h * g(j * tau);
    // Maybe, this can be done more efficiently by extracting the diagonal,
    // converting it into an Eigen::Array object and then perform componentwise
    // division. However, a really smart sparse elimination solver should be
    // able to detect a diagonal coefficient matrices and optimize the
    // elimination accordingly.
    nu = solver.solve(-tau * A * mu + (M - 0.5 * tau * B) * nu + tau * phi);
    mu = mu + tau * nu;
  }
  R.row(m).head(N) = mu.transpose();
  // The value at x=1 has to be incorporated into the output
  for (int i = 0; i < m + 1; ++i) {
    R(i, N) = g(i * tau);
```

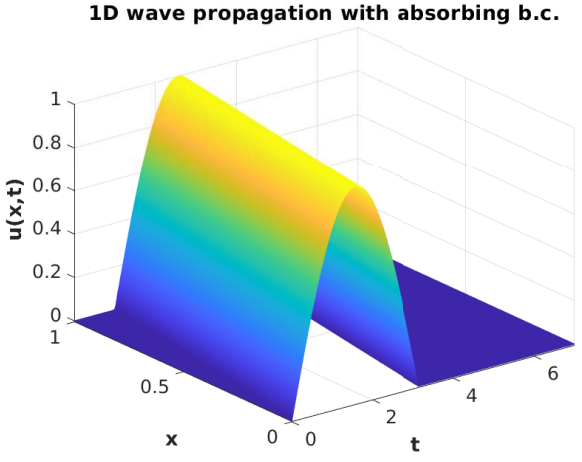
```
42      }
43      return R;
44  }
```

HINT 1 for (9-4.d): EIGEN provides the member function `Eigen::MatrixXd::format()`, see ➙ documentation, for the formatted output of matrices. Call this member function for the output matrix of `waveLeapfrogABC()` from Sub-problem (9-4.c). ⌟

SOLUTION of (9-4.d):

C++ code 9.4.18: Sub-problem (9-4.d): snippet in `main()` ↪ GitLab

```
2   double c = 1.0;
3   double T = 7.0;
4   unsigned int N = 100;
5   unsigned int m = 2000;
6   Eigen::MatrixXd R = waveLeapfrogABC(c, T, N, m);
7   Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(m + 1, 0.0, T);
8
9   // print the data, e.g. to a .csv file, in a suitable way
10  std::ofstream solution_file;
11
12  solution_file.open("solution.csv");
13  Eigen::MatrixXd tR(m + 1, N + 2);
14  tR.col(0) = t;
15  tR.block(0, 1, m + 1, N + 1) = R;
16  solution_file << tR.format(CSVFormat) << std::endl;
17  solution_file.close();
18  std::cout << "Generated solution.csv" << std::endl;
19  systemcall::execute(
20      "python3 ms_scripts/viswave.py solution.csv solution.eps");
```



◁ 3D plot of traveling wave excited by boundary condition at $x = 1$ and swallowed by the absorbing boundary condition at $x = 0$.

HINT 1 for (9-4.e): It is tempting to use the *truncated* Galerkin matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{M}$ from (9.4.7) for the computations. However, note that

$$E_{\mathrm{pot}}^{(j)} \neq \frac{1}{2}(\vec{\boldsymbol{\mu}}^{(j)})^{\top}\mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} \quad , \quad E_{\mathrm{kin}}^{(j)} \neq \frac{1}{2}(\vec{\boldsymbol{\nu}}^{(j+\frac{1}{2})})^{\top}\mathbf{M}\vec{\boldsymbol{\nu}}^{(j+\frac{1}{2})} \quad \mathbf{!} \tag{9.4.19}$$

These formulas miss the contribution of b_h^{N+1}. If you want to use formulas like (9.4.20), you have to rely on the Galerkin matrices for the full finite-element space $\mathcal{S}_1^0(\mathcal{M})$. Denoting them by $\widetilde{\mathbf{A}}$, $\widetilde{\mathbf{B}}$, and $\widetilde{\mathbf{M}}$ we have

$$E_{\mathrm{pot}}^{(j)} = \frac{1}{2}(\vec{\boldsymbol{\mu}}^{(j)})^{\top}\widetilde{\mathbf{A}}\vec{\boldsymbol{\mu}}^{(j)} \quad , \quad E_{\mathrm{kin}}^{(j)} = \frac{1}{2}(\vec{\boldsymbol{\nu}}^{(j+\frac{1}{2})})^{\top}\widetilde{\mathbf{M}}\vec{\boldsymbol{\nu}}^{(j+\frac{1}{2})} \, . \tag{9.4.20}$$

⌟

SOLUTION of (9-4.e):

The full Galerkin matrices $\widetilde{\mathbf{A}}$ and $\widetilde{\mathbf{M}}$ mentioned in the hint can be computed with the function `getA_full(` and `getM_full` listed in Code 9.4.14 and Code 9.4.16.

Be aware that the function `waveLeapfrogABC()` returns only the vectors $\vec{\mu}^{(j)}$, $j = 0, \ldots, m$. By (9.4.12), from them we can recover

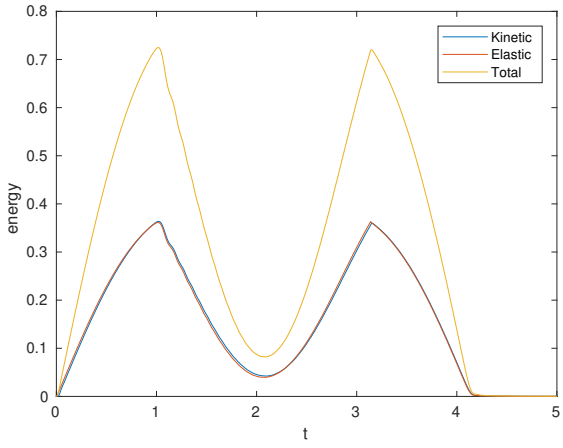
$$\vec{v}^{(j+1/2)} = \frac{\vec{\mu}^{(j+1)} - \vec{\mu}^{(j)}}{\tau}, \quad j = 0, \ldots, m-1 .$$

C++ code 9.4.21: Sub-problem (9-4.e): function `computeEnergies()` ⇝ GitLab

```
2  std::pair<Eigen::VectorXd, Eigen::VectorXd> computeEnergies(
3      const Eigen::MatrixXd &full_solution, double c, double tau) {
4    int m = full_solution.rows() - 1;
5    int N = full_solution.cols() - 1;
6    double h = 1.0 / N;
7
8    Eigen::SparseMatrix<double> A = getA_full(N, c, h);
9    Eigen::SparseMatrix<double> M = getM_full(N, h);
10
11   Eigen::VectorXd E_pot(m + 1);
12   Eigen::VectorXd E_kin(m);
13
14   Eigen::VectorXd mu0;
15   Eigen::VectorXd mu1 = full_solution.row(0).transpose();
16   for (int j = 0; j < m; ++j) {
17     mu0.swap(mu1);
18     E_pot(j) = 0.5 * mu0.dot(A * mu0);
19     mu1 = full_solution.row(j + 1).transpose();
20
21     Eigen::VectorXd nu = (mu1 - mu0) / tau;
22     E_kin(j) = 0.5 * nu.dot(M * nu);
23   }
24   E_pot(m) = 0.5 * mu1.dot(A * mu1);
25
26   return std::make_pair(E_pot, E_kin);
27 }
```

HINT 1 for (9-4.f): Take into account that $E_{\text{kin}}^{(j)}$ actually approximates the kinetic energy at time $t = (j + \frac{1}{2})$ $\tau > 0$ the uniform timestep size. ⌟

SOLUTION of (9-4.f):



◁ Energy plot for Sub-problem (9-4.f).

After the excitation through the boundary condition at $x = 1$ stops at $t = \pi$, the wave energy decays quickly.

C++ code 9.4.22: Sub-problem (9-4.f): snippet in `main()` ⇝ GitLab

```
2   std::ofstream energies_file;
3   energies_file.open("energies.csv");
4   energies_file << t.transpose().format(CSVFormat) << std::endl
5                 << E_pot.transpose().format(CSVFormat) << std::endl
6                 << E_kin.transpose().format(CSVFormat) << std::endl;
7   energies_file.close();
8   std::cout << "Generated energies.csv" << std::endl;
9   systemcall::execute(
10      "python3 ms_scripts/visenergies.py energies.csv "
11      "energies.eps");
```

SOLUTION of (9-4.g):

Note that (9.4.2) is a wave propagation problem with a source term, which will inject or extract energy. Therefore we cannot expect conservation of the total energy.

After $t = \pi$ the source is turned off, but, nevertheless, the energy decays quickly. The reason is that the absorbing boundary condition at $t = 0$ lets all the energy escape from the $]0, 1[$ until nothing is left: the wave gets swallowed/absorbed at $x = 0$.

HINT 1 for (9-4.h): Differentiate $E(t)$, and use the "product rule"

$$\frac{\partial(\vec{\mu}^{\top}\mathbf{S}\vec{\mu})}{\partial t} = 2\dot{\vec{\mu}}^{\top}\mathbf{S}\vec{\mu} = 2\vec{\mu}^{\top}\mathbf{S}\dot{\vec{\mu}}. \tag{9.4.24}$$

for any symmetric matrix $\mathbf{S}$, see [Lecture $\rightarrow$ Rem. 9.2.3.7]. Of course, you will also have to use (9.4.7). Remember that $\mathbf{B}$ is symmetric positive semi-definite. ⌟

SOLUTION of (9-4.h):

By the product rule (9.4.24) we find

$$\frac{dE}{dt}(t) = \dot{\vec{\mu}}(t)^\top \mathbf{M}\ddot{\vec{\mu}}(t) + \dot{\vec{\mu}}(t)^\top A\vec{\mu}(t) \ .$$

Let us now left-multiply (9.4.7) by $\dot{\vec{\mu}}$. Assuming $\boldsymbol{\varphi} = 0$, we obtain

$$\dot{\vec{\mu}}(t)^\top \mathbf{M}\ddot{\vec{\mu}}(t) + \dot{\vec{\mu}}(t)^\top A\vec{\mu}(t) + \dot{\vec{\mu}}(t)^\top \mathbf{B}\dot{\vec{\mu}}(t) = 0 \ .$$

We can then write

$$\frac{dE}{dt}(t) = -\dot{\vec{\mu}}(t)^\top \mathbf{B}\dot{\vec{\mu}}(t) \ .$$

Since $\dot{\vec{\mu}}^\top \mathbf{B}\dot{\vec{\mu}} \geq 0$, we conclude.

HINT 1 for (9-5.a): The statement of a variational formulation is complete only with suitable trial and test spaces. ⌟

SOLUTION of (9-5.a):

We follow the recipes from [Lecture → Section 1.8], in particular [Lecture → Ex. 1.8.0.10]. We multiply the PDE $\frac{\partial^2 u}{\partial t^2} - \Delta u + cu = 0$ with a test function $v \in H^1(\Omega)$ and integrate by parts via Green's formula. We obtain the following spatial variational formulation of (9.5.1): we seek a time-dependent function-space-valued function $t \in]0,T[\to u(t) \in H^1(\Omega)$ (**trial space**) such that

$$\begin{cases} \int_\Omega \left(\frac{\partial^2 u}{\partial t^2} v + \textbf{grad}\, u \cdot \textbf{grad}\, v + cuv \right) \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H^1(\Omega) \quad (\textbf{test space}) \,, \\ u(0) = u_0 \in H^1(\Omega) \,. \end{cases} \tag{9.5.2}$$

Boundary terms introduced by integration by parts can all be dropped since the evolution problem features homogeneous spatial Neumann boundary conditions, see [Lecture → Ex. 1.8.0.10].

HINT 1 for (9-5.j): You may also contemplate adding some more data members to the class in order to facilitate the evaluations of `computeEnergies()` ⌟

SOLUTION of (9-5.j):

It is a good idea to add the sparse matrix $\mathbf{M}$ to the class data, beside the matrix $\mathbf{A}$.

C++11 code 9.5.19: Member function `computeEnergies()` of SympTimestepWaveEq ↪ GitLab

```
2 template <typename FUNCTION>
3 double SympTimestepWaveEq<FUNCTION>::computeEnergies(
4     const Eigen::VectorXd &p, const Eigen::VectorXd &q) const {
5   double energy;
6   energy = 0.5 * (p.dot(M_ * p) + q.dot(A_ * q));
7   return energy;
8 } // SympTimestepWaveEq<FUNCTION>::computeEnergies
```

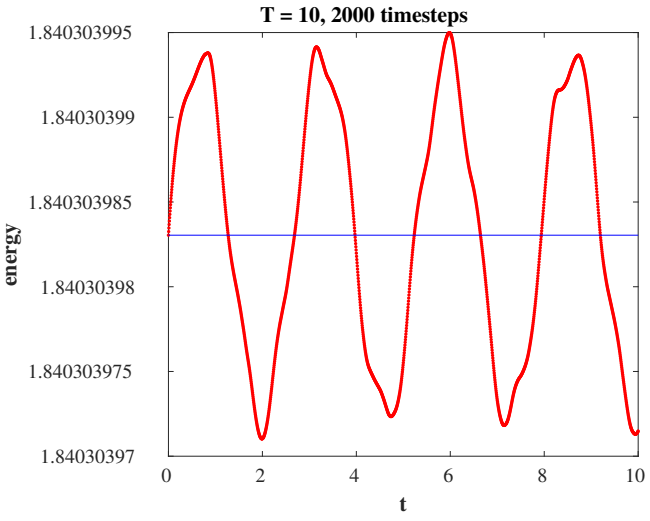
SOLUTION of (9-5.k):

Note that the following implementation of `solvewave()` throws an exception whenever the total energy surpasses its initial value by more than a factor of ten. This is used as an indicator for blow-up.

C++11 code 9.5.20: Implementation of function `solvewave()` ↪ GitLab

```
2  template <typename FUNCTION>
3  std::pair<Eigen::VectorXd, Eigen::VectorXd> solvewave(
4      std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fes_p,
5      FUNCTION c, const Eigen::VectorXd &u0_vec, const Eigen::VectorXd &v0_vec,
6      double T, unsigned int m) {
7    std::pair<Eigen::VectorXd, Eigen::VectorXd> solution_pair;
8    double tau = T / m;  // time step
9    std::cout << "Solving with uniform step size tau = " << tau << std::endl;
10   progress_bar progress{std::clog, 70u, "Timestepping"};
11   double progress_pourcentage;
12   // Obtain local->global index mapping for current finite element space
13   const lf::assemble::DofHandler &dofh{fes_p->LocGlobMap()};
14   const lf::uscalfe::size_type N_dofs(dofh.NumDofs());  // dim.  of FE space size
15   LF_VERIFY_MSG(u0_vec.size() == N_dofs, "Wrong size of initial conditions u0");
16   LF_VERIFY_MSG(v0_vec.size() == N_dofs, "Wrong size of initial conditions");
17
18   // Precomputing the required data for symplectic time stepping
19   SympTimestepWaveEq<decltype(c)> timestepper(fes_p, c);
20
21   /* Starting the evolution using the initial conditions */
22   Eigen::VectorXd q = u0_vec;
23   Eigen::VectorXd p = v0_vec;
24   // Initialization to store energy at different times
25   Eigen::VectorXd energies(m + 1);
26
27   /* Iterating symplectic stepping */
28   for (int i = 0; i < m; i++) {
29     energies[i] = timestepper.computeEnergies(p, q);
30     timestepper.compTimestep(tau, p, q);
31     // Throw an exception in case of severe increase of the total
32     // energy, which is a conserved quantity for the exact evolution.
33     if (energies[i] > 10.0 * energies[0]) {
34       throw "ENERGY BLOW UP";
35     }
36     progress_pourcentage = ((double)i) / m * 100.0;
37     progress.write(progress_pourcentage / 100.0);
38   }
39
40   energies[m] = timestepper.computeEnergies(p, q);
41   solution_pair = std::make_pair(q, energies);
42   return solution_pair;
43 }
```

SOLUTION of (9-5.l):



◁ Behavior of total energy in the course of timestepping.

We observe tiny oscillations around the exact value of the total energy. This is the typical behavior of so-called **symplectic** timestepping schemes for wave equations.

C++11 code 9.5.22: Implementation of function `wavePropSimulation()` ↪ GitLab

```
2  void wavePropSimulation(unsigned int m) {
3    double T = 10.0;  // final time
4
5    // Load mesh into a Lehrfem++ object
6    auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
7    const lf::io::GmshReader reader(std::move(mesh_factory), "meshes/hex4.msh");
8    auto mesh_p = reader.mesh();
9
10   // Finite element space
11   auto fe_space =
12       std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
13   // Obtain local->global index mapping for current finite element space
14   const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
15   // Dimension of finite element space
16   const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
17
18   // Building initial condition vector
19   Eigen::VectorXd u0_vec(N_dofs);
20   Eigen::VectorXd v0_vec(N_dofs);
21   u0_vec.setOnes();
22   v0_vec.setZero();
23
24   // Creating coefficient function
25   auto c = [](Eigen::Vector2d x) -> double { return 1.0 + x.dot(x); };
26
27   // Solve the initial value problem
28   std::pair<Eigen::VectorXd, Eigen::VectorXd> solution_pair =
29       solvewave(fe_space, c, u0_vec, v0_vec, T, m);
30   Eigen::VectorXd discrete_wave_sol = solution_pair.first;
31   LF_ASSERT_MSG(
32       discrete_wave_sol.size() == N_dofs,
33       "Size of discrete solution and dimension of FE space mismatch.");
34   Eigen::VectorXd energies = solution_pair.second;
35   LF_ASSERT_MSG(energies.size() == m + 1, "Wrong number of energie values.");
36
37   // Define output file format for the energies
```

```
38    const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
39                                           Eigen::DontAlignCols, ", ", "\n");
40    std::string errors_file_name = "waves_energies.csv";
41    std::ofstream file(errors_file_name.c_str());
42    if (file.is_open()) {
43      file << energies.format(CSVFormat);
44    }
45
46    std::cout << ">>The energies were written to:" << std::endl;
47    std::cout << "\t waves_energies.csv" << std::endl;
48    /* SAM_LISTING_BEGIN_2 */
49    // Output results for the wave function to vtk file
50    lf::io::VtkWriter vtk_writer(mesh_p, "discrete_wave_sol.vtk");
51    // Write nodal data taking the values of the discrete solution at the
          vertices
52    auto nodal_data = lf::mesh::utils::make_CodimMeshDataSet<double>(mesh_p, 2);
53    for (int global_idx = 0; global_idx < N_dofs; global_idx++) {
54      nodal_data->operator()(dofh.Entity(global_idx)) =
55          discrete_wave_sol[global_idx];
56    };
57    vtk_writer.WritePointData("discrete_wave_sol", *nodal_data);
58    /* SAM_LISTING_END_2 */
59    std::cout << ">>The output discrete_wave_sol was written to:" << std::endl;
60    std::cout << "\t discrete_wave_sol.vtk" << std::endl;
61  }
```

HINT 1 for (9-5.m): Monitor the “conserved energy” and conclude a blow-up, once it rises above 10 times its initial value. Then do “bisection” in order to find a threshold for the timestep. ⌟

SOLUTION of (9-5.m):

Note that `solvewave()` will throw an **exception** ➢ CppRef in case it detects a blow-up.

C++11 code 9.5.23: Sub-problem (9-5.m): function `testStab()` ⇢ GitLab

```
2  double testStab() {
3    double maxUniformTimestep;
4    std::cout << "\n********** STABILITY EXPERIMENT "
5                 "**********"
6              << std::endl;
7    unsigned int m_upper = 1100;
8    unsigned int m_lower = 1000;
9    unsigned int m = m_lower + (m_upper - m_lower) / 2;
10   while (m_upper - m_lower > 1) {
11     std::cout << "\nTesting symplectic method with m = " << m << std::endl;
12     // Catch exception thrown in case of blow-up
13     try {
14       wavePropSimulation(m);
15     } catch (const char *msg) {
16       // Blow-up detected!
17       std::cout << "Energy blows up!" << std::endl;
18       m_lower = m;
19       m = m_lower + (m_upper - m_lower) / 2;
20       continue;
21     }
22     // No blow-up
23     std::cout << "Energy is conserved" << std::endl;
24     m_upper = m;
25     m = m_lower + (m_upper - m_lower) / 2;
26   }
27   maxUniformTimestep = 10.0 / m_upper;
28   return maxUniformTimestep;
29 }
```

The maximum uniform timestep size detected by the code is roughly 0.00957854.

HINT 1 for (9-5.b): You can find the relevant expressions in [Lecture → § 9.3.2.13].

SOLUTION of (9-5.b):

We learn from [Lecture → § 9.3.2.13] that the total energy is an invariant of the evolution described by wave equations. Solving this sub-problem just boils down to applying [Lecture → Thm. 9.3.2.16].

By (9.5.2), we define the bilinear forms

$$\mathsf{m}(u,v) := \int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,,$$
$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x}) + c(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

$$E(u) := \tfrac{1}{2}\mathsf{m}\left(\tfrac{\partial u}{\partial t},\tfrac{\partial u}{\partial t}\right) + \tfrac{1}{2}\mathsf{a}(u,u)\,. \tag{9.5.3}$$

Now we must prove that $t \to E(\{x \to u(x,t)\}) = \mathsf{const}$, if u solves (9.5.1). This is immediate. In fact, if we take as test functions $\dot{u} \in H^1(\Omega)$, then (9.5.2) can be read as $\mathsf{m}(\ddot{u},\dot{u}) + \mathsf{a}(\dot{u},u) = 0$. Since

$$\frac{dE}{dt}(t) = \mathsf{m}(\ddot{u},\dot{u}) + \mathsf{a}(\dot{u},u)\,,$$

then $\dfrac{dE}{dt} = 0$ and this implies $t \to E(\{x \to u(x,t)\}) = \mathsf{const}$.

Note that merely the statement of E is requested.

SOLUTION of (9-5.c):

Following the method-of-lines policy, (9.5.1a) is discretized as follows: we seek $u_h \in \mathcal{S}_1^0(\mathcal{M}))$ such that

$$\int_\Omega \left(\frac{\partial^2 u_h}{\partial t^2} v_h + \textbf{grad}\, u_h \cdot \textbf{grad}\, v_h + c\, u_h\, v_h \right) \mathrm{d}\boldsymbol{x} = 0 \quad \forall v_h \in \mathcal{S}_1^0(\mathcal{M})) \,.$$

Using the standard nodal basis $\{b_h^l\}_{l=1}^N$ of $\mathcal{S}_1^0(\mathcal{M}))$, which is *independent of time*, we have

$$u_h(t) = \sum_{i=1}^N \mu_i(t)\, b_h^i \,.$$

This leads to the following initial value problem for a second-order ordinary differential equation:

$$\mathbf{M}\left(\frac{d^2}{dt^2} \vec{\boldsymbol{\mu}}(t) \right) + \mathbf{A}(\vec{\boldsymbol{\mu}}(t)) = 0 \,,$$

where

$$(\mathbf{A})_{i,j} = \int_\Omega \textbf{grad}\, b_h^i \cdot \textbf{grad}\, b_h^j + c\, b_h^i\, b_h^j \,\mathrm{d}\boldsymbol{x} \quad , \quad i,j \in \{1,\ldots,N\} \,,$$

and

$$(\mathbf{M})_{i,j} = \int_\Omega b_h^i\, b_h^j \,\mathrm{d}\boldsymbol{x} \;; \quad i,j \in \{1,\ldots,N\} \,.$$

SOLUTION of (9-5.d):

In this case we have

$$\mathbf{f}(\mathbf{q},t) = q \quad , \quad \mathbf{g}(\mathbf{p}) = -p \quad \Rightarrow \quad f(q) := q \quad , \quad g(p) := -p \, .$$

Also note that the timestepping scheme (9.5.6) is *explicit* for this ODE. The formulas (9.5.6) simplify, because $\mathbf{f}$ does not depend on time t.

C++ code 9.5.9: Sub-problem (9-5.d): Implementation of `sympTimestep` ➔ GitLab

```
2  void sympTimestep(double tau, Eigen::Vector2d &pq_j) {
3    // Coefficients of the method
4    Eigen::VectorXd a(3);
5    a << 2. / 3., -2. / 3., 1.;
6    Eigen::VectorXd b(3);
7    b << 7. / 24., 3. / 4., -1. / 24;
8
9    // one step method
10   for (int i = 0; i < 3; ++i) {
11     pq_j(0) += tau * b(i) * pq_j(1);
12     pq_j(1) -= tau * a(i) * pq_j(0);
13   }
14 }
```

HINT 1 for (9-5.e): The exact solution of the initial value problem for (9.5.8) is $p(t) = \sin(t)$ and $q(t) = \cos(t)$. ⌟

SOLUTION of (9-5.e):

C++ code 9.5.10: Sub-problem (9-5.e): Tabulation of error ⇢ GitLab

```
void sympTimesteppingODETest() {
  int nIter = 9;   // total number of iterations
  unsigned int m;  // number of equidistant steps

  // Evaluating the error at the final step between the approx solutions as
  // given by the symplectic method and the exact solution computed from
  // the anlytic formula
  Eigen::Vector2d approx_sol;
  double errors[nIter];  // errors vector for all approx.  sols
  for (int k = 0; k < nIter; k++) {
    m = 10 * std::pow(2, k);
    // Computing approximate solution
    approx_sol = sympTimesteppingHarmonicOscillatorODE(m);
    // Computing the error in the maximum norm
    errors[k] = std::abs(sin(2.0 * M_PI) - approx_sol[0]) +
                std::abs(cos(2.0 * M_PI) - approx_sol[1]);
  }
  // Computing rates of convergence
  double rates[nIter - 1];
  double avg_rate = 0.0;

  for (int k = 0; k < nIter - 1; k++) {
    rates[k] = log2(errors[k] / errors[k + 1]);
    avg_rate += rates[k];
  }
  avg_rate = avg_rate / (nIter - 1);

  // Printing results
  std::cout << "\n" << std::endl;
  std::cout << "*********************************************************"
            << std::endl;
  std::cout << "     Convergence of Symplectic Time Stepping Method      "
            << std::endl;
  std::cout << "*********************************************************"
            << std::endl;
  std::cout << "---------------------- RESULTS -------------------------"
            << std::endl;
  std::cout << "Iteration"
            << "\t| Nsteps"
            << "\t| error"
            << "\t\t| rates" << std::endl;
  std::cout << "-------------------------------------------------------------"
            << std::endl;
  for (int k = 0; k < nIter; k++) {
    std::cout << k << "\t"
              << "\t|" << 10 * std::pow(2, k) << "\t\t|" << errors[k];
    if (k > 0) {
      std::cout << "\t|" << rates[k - 1];
    }
    std::cout << "\n";
  }
  std::cout << "-------------------------------------------------------------"
```

```
54                 << std::endl;
55   std::cout << "Average rate of convergence: " << avg_rate << "\n" << std::endl;
56 }
```

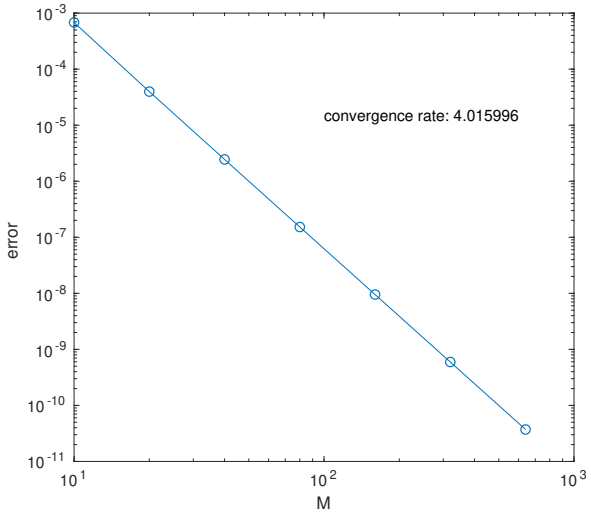


Table of error:

M	$\mathrm{err}(M)$
10	0.000680607
$10 \cdot 2^1$	3.9793e-05
$10 \cdot 2^2$	2.4477e-06
$10 \cdot 2^3$	1.52386e-07
$10 \cdot 2^4$	9.51496e-09
$10 \cdot 2^5$	5.94542e-10
$10 \cdot 2^6$	3.71564e-11

◁ Log-log plot of `err` vs. M. The error curve is almost a line and from its slope we can read off a convergence rate of ~ 4.

A doubling of M leads to a decrease of the error by roughly a factor of 16: $\mathrm{err}(M) = O(M^{-4})$, algebraic convergence with rate 4. As the algebraic rate of convergence of a single-step method is tightly linked to its order, we conjecture that the timestepping scheme from (9.5.6) is order fourth order .

HINT 1 for (9-5.f): The problem is related to rewriting the wave equation as a first-order system as presented in [Lecture → § 9.3.1.16]. ⌟

SOLUTION of (9-5.f):

The following rephrasing is straightforward:

$$\mathbf{M}\left(\frac{d^2}{dt^2}\vec{\boldsymbol{\mu}}(t)\right) + \mathbf{A}(\vec{\boldsymbol{\mu}}(t)) = 0 \quad \blacktriangleright \quad \frac{d^2}{dt^2}\vec{\boldsymbol{\mu}} = -\mathbf{M}^{-1}\mathbf{A}\vec{\boldsymbol{\mu}}\ .$$

We introduce the functions (notations are meant to be consistent with (9.5.5))

$$\mathbf{p}(t) := \frac{d\vec{\boldsymbol{\mu}}}{dt}(t) \quad , \quad \mathbf{q}(t) := \vec{\boldsymbol{\mu}}(t)\ .$$

Then, we have $\dot{\mathbf{q}} = \mathbf{p}$ and $\dot{\mathbf{p}} = \frac{d^2}{dt^2}\vec{\boldsymbol{\mu}} = -\mathbf{M}^{-1}\mathbf{A}\mathbf{q}$. Finally, we can write

$$\frac{d}{dt}\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix} = \begin{bmatrix}\mathbf{f}(\mathbf{q},t)\\ \mathbf{g}(\mathbf{p})\end{bmatrix},$$

where

$$\mathbf{f}(\mathbf{q},t) = -\mathbf{M}^{-1}\mathbf{A}\mathbf{q} \quad , \quad \mathbf{g}(\mathbf{p}) = \mathbf{p} \quad \Leftrightarrow \quad \frac{d}{dt}\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix} = \begin{bmatrix}-\mathbf{M}^{-1}\mathbf{A}\mathbf{q}\\ \mathbf{p}\end{bmatrix}\ . \tag{9.5.11}$$

We point out that this is not the only way to rewrite the semi-discrete wave equation in the form (9.5.5).

SOLUTION of (9-5.g):

In order to be able to use **lf::uscalfe::ReactionDiffusionElementMatrixProvider** we have to wrap the coefficient functions into objects of type **MeshFunction**.

C++11 code 9.5.13: Implementation of `assembleGalerkinMatrix()` ↪ GitLab

```
2  template <typename FUNC_ALPHA, typename FUNC_GAMMA, typename FUNC_BETA>
3  Eigen::SparseMatrix<double> assembleGalerkinMatrix(
4      std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space_p,
5      FUNC_ALPHA &&alpha, FUNC_GAMMA &&gamma, FUNC_BETA &&beta) {
6    Eigen::SparseMatrix<double> galMat;
7
8    /* Creating coefficient-functions as Lehrfem++ mesh functions */
9    // Coefficient-functions used in the class template
10   // ReactionDiffusionElementMatrixProvider<SCALAR,DIFF_COEFF,REACTION_COEFF>
11   auto alpha_mf = lf::mesh::utils::MeshFunctionGlobal(alpha);
12   auto gamma_mf = lf::mesh::utils::MeshFunctionGlobal(gamma);
13   // Coefficient-function used in the class template
14   // MassEdgeMatrixProvider< SCALAR, COEFF, EDGESELECTOR >
15   auto beta_mf = lf::mesh::utils::MeshFunctionGlobal(beta);
16
17   /* Retrieving FE data */
18   // pointer to current mesh
19   std::shared_ptr<const lf::mesh::Mesh> mesh_p = fe_space_p->Mesh();
20   // Obtain local->global index mapping for current finite element space
21   const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
22   // Obtain an array of boolean flags for the edges of the mesh, 'true'
23   // indicates that the edge lies on the boundary
24   // Dimension of finite element space
25   const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
26   auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
27   // Creating a predicate that will guarantee that the computations for the
28   // boundary Mass matrix part of the full Galerkin matrix are carried only on
29   // the edges of the mesh using the boundary flags
30   auto edges_predicate = [&bd_flags](const lf::mesh::Entity &edge) -> bool {
31     return bd_flags(edge);
32   };
33
34   /* Instantiating FE Galerkin matrix */
35   // Matrix in triplet format holding Galerkin matrix, zero initially.
36   lf::assemble::COOMatrix<double> galMat_COO(N_dofs, N_dofs);
37
38   /* Initialization of local matrices builders */
39   // Initialize objects taking care of local computations for volume integrals
40   lf::uscalfe::ReactionDiffusionElementMatrixProvider<
41       double, decltype(alpha_mf), decltype(gamma_mf)>
42       elmat_builder(fe_space_p, alpha_mf, gamma_mf);
43   // Initialize objects taking care of local computations for boundary integrals
44   lf::uscalfe::MassEdgeMatrixProvider<double, decltype(beta_mf),
45                                       decltype(edges_predicate)>
46       edgemat_builder(fe_space_p, beta_mf, edges_predicate);
47
48   /* Assembling the Galerkin matrix */
49   // Information about the mesh and the local-to-global map is passed through
50   // a Dofhandler object, argument 'dofh'. This function call adds triplets to
```

```
51    // the internal COO-format representation of the sparse matrix A.
52    // Invoke assembly on cells (co-dimension = 0 as first argument)
53    lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, galMat_COO);
54    // Invoke assembly on edges by specifying co-dimension = 1
55    lf::assemble::AssembleMatrixLocally(1, dofh, dofh, edgemat_builder,
56                                        galMat_COO);
57    /* Convert COO matrix A into sparse matrix format using Eigen's */
58    galMat = galMat_COO.makeSparse();
59    return galMat;
60  }
```

HINT 1 for (9-5.h): EIGEN allows to split matrix factorization and elimination steps through calls to the member functions `compute()` and `solve()` of direct sparse elimination solver classes, see ➙ documentation.

For the s.p.d. matrix $\mathbf{M}$ you can use the Cholesky decomposition offered by EIGEN, check the documentation of the class **Eigen::SimplicialLDLT**. ⌟

SOLUTION of (9-5.h):

We invoke `assembleGalerkinMatrix()` with suitable arguments to build the Galerkin matrices:

$$\begin{array}{l} \mathbf{A} : \alpha \equiv 1 \quad \gamma \equiv 0 \quad \beta \equiv 0 \\ \mathbf{M} : \alpha \equiv 0 \quad \gamma \equiv 1 \quad \beta \equiv 0 \end{array}$$

The Cholesky factorization of the s.p.d. matrix $\mathbf{M}$ is left to an EIGEN built-in class, see ➙ documentation. For the sake of efficiency this factorization of the mass matrix has to be computed in the constructor to be reused in every timestep.

C++ code 9.5.14: Sub-problem (9-5.h): Implementation of constructor of class SympTimestepWaveEq ➙ GitLab

```
2  template <typename FUNCTION>
3  class SympTimestepWaveEq {
4   public:
5    /* Constructor */
6    SympTimestepWaveEq(
7        std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space_p,
8        FUNCTION c) {
9      /* Creating the Galerkin Matrices for the wave equation*/
10     // Assembling the element Galerkin matrix for the volume integrals
11     auto one_coeff = [](Eigen::Vector2d x) -> double { return 1.0; };
12     auto zero_coeff = [](Eigen::Vector2d x) -> double { return 0.0; };
13     A_ = assembleGalerkinMatrix(fe_space_p, one_coeff, c, zero_coeff);
14     // Assembling the Galerkin boundary mass matrix for the boundary integral
15     // This requires a new call to assembleGalerkinMatrix using new coefficient
16     // funtions so that only the mass integrals remains
17     M_ = assembleGalerkinMatrix(fe_space_p, zero_coeff, one_coeff, zero_coeff);
18     /* Precompute sparse Cholesky decomposition for the boundary mass matrix */
19     solver_M_.compute(M_);
20     LF_VERIFY_MSG(solver_M_.info() == Eigen::Success,
21                   "Cholesky LDLT decomposition for sparse matrix M_ failed");
22   }
23   /* Public member functions */
24   void compTimestep(double tau, Eigen::VectorXd &p, Eigen::VectorXd &q) const;
25   double computeEnergies(const Eigen::VectorXd &p,
26                          const Eigen::VectorXd &q) const;
27
28  private:
29   Eigen::SparseMatrix<double> A_;  // Galerkin matrix for volume integrals
30   Eigen::SparseMatrix<double> M_;  // Galerkin Matrix for boundary integral
31   Eigen::SimplicialLDLT<Eigen::SparseMatrix<double>> solver_M_;
32 };  // class SympTimestepWaveEq
```

C++ code 9.5.15: Sub-problem (9-5.h): Implementation of timestepping method for class SympTimestepWaveEq ➙ GitLab

```
2  template <typename FUNCTION>
3  void SympTimestepWaveEq<FUNCTION>::compTimestep(double tau, Eigen::VectorXd &p,
4                                                  Eigen::VectorXd &q) const {
5    // Coefficients of the method
6    Eigen::VectorXd a(3);
7    a << 2.0 / 3.0, -2.0 / 3.0, 1.0;
```

```
8     Eigen::VectorXd b(3);
9     b << 7.0 / 24.0, 3.0 / 4.0, -1.0 / 24.0;
10    // Solving for f(q,t) using precomputed Galerkin matrices and the
11    // stored factorization of the mass matrix
12    Eigen::VectorXd fq_in = -solver_M_.solve(A_ * q);
13    // one step method
14    for (int i = 0; i < 3; ++i) {
15      p += tau * b(i) * fq_in;
16      q += tau * a(i) * p;
17      fq_in = -solver_M_.solve(A_ * q);
18    }
19  }  // SympTimestepWaveEq<FUNCTION>::compTimestep
```

SOLUTION of (9-5.i):

We start from (9.5.3), which in the spatially semi-discrete setting becomes

$$E(u_h)(t) := \frac{1}{2}\mathsf{m}(\dot{u}_h(t), \dot{u}_h(t)) + \frac{1}{2}\mathsf{a}(u_h(t), u_h(t)) \,. \tag{9.5.16}$$

This can be expressed by means of Galerkin matrices and the basis expansion coefficient vector $\vec{\boldsymbol{\mu}}(t)$ for $u_h(t)$:

$$E(\vec{\boldsymbol{\mu}})(t) = \frac{1}{2}\frac{d\vec{\boldsymbol{\mu}}}{dt}^{\top}(t)\mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}^{\top}(t) + \frac{1}{2}\vec{\boldsymbol{\mu}}(t)^{\top}\mathbf{A}\vec{\boldsymbol{\mu}}(t) \,. \tag{9.5.17}$$

After the substitution $\mathbf{p} := \frac{d\vec{\boldsymbol{\mu}}}{dt}$, $\mathbf{q} := \vec{\boldsymbol{\mu}}$, we end up with

$$E(\mathbf{p},\mathbf{q}) = \frac{1}{2}\mathbf{p}(t)^{\top}\mathbf{M}\mathbf{p}(t) + \frac{1}{2}\mathbf{q}(t)^{\top}\mathbf{A}\mathbf{q}(t) \,. \tag{9.5.18}$$

Using (9.5.11) and the symmetry of $\mathbf{A}$ it is immediate that $t \mapsto E(\mathbf{p}(t),\mathbf{q}(t))$ is constant, if $t \mapsto \mathbf{p}(t)$ and $t \mapsto \mathbf{q}(t)$ solve (9.5.11).

SOLUTION of (9-6.a):

We can rewrite (9.6.2) as an abstract second-order ODE in function space:

$$u : [0,T] \to H^1(\Omega): \quad \mathsf{m}(\frac{d^2u}{dt^2}, v) + \mathsf{a}(u,v) = 0 \quad \forall v \in H^1(\Omega) \; . \tag{9.6.3}$$

with

$$\mathsf{m}(w,v) := \int_{\partial\Omega} w(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) \; , \tag{9.6.4}$$

$$\mathsf{a}(w,v) := \int_{\Omega} (1 + \|\boldsymbol{x}\|^2) \, \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \; . \tag{9.6.5}$$

Then we take the cue from [Lecture → § 9.3.2.13] and [Lecture → Thm. 9.3.2.16], which leads us to set

$$E(v,w) = \tfrac{1}{2}\mathsf{m}(w,w) + \tfrac{1}{2}\mathsf{a}(v,v) \; . \tag{9.6.6}$$

Of course, this expression can be multiplied with an arbitrary factor and will still yield a valid E-functional.

The proof that $t \to E(u(t), \frac{du}{dt}(t))$ is constant is accomplished by simple differentiation and the use of the generalized product rule for bilinear mappings, *cf.* [Lecture → Rem. 9.2.3.7].

Remark. From the variational formulation (9.6.2) we learn that

$$\frac{d}{dt} \int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t) \, \mathrm{d}S(\boldsymbol{x}) = \frac{d^2}{dt^2} \int_{\partial\Omega} u(\boldsymbol{x},t) \, \mathrm{d}S(\boldsymbol{x}) = 0 \; ,$$

Therefore we may add the term $\int_{\partial\Omega} w(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x})$ to the expression for the conserved quantity E.

HINT 1 for (9-6.j): The Crank-Nicolson scheme has order 2. ⌟

SOLUTION of (9-6.j):

The "metatheorem" [Lecture → Thm. 9.3.5.10] gives us the asymptotic estimate

$$\left(\tau \sum_{j=1}^{M} \left\| u - u_h^{(j)} \right\|_{H^1(\Omega)}^2 \right)^{\frac{1}{2}} \leq C(h_{\mathcal{M}}^p + \tau^2) \,,$$

with $p = 1$, which we *assume to be sharp*. Thus halving the error $\epsilon(\mathcal{M})$ requires

- reducing the timestep by a factor of $\sqrt{2}$, taking $\sqrt{2}$ more timesteps,
- halving the meshwidth h, which leads to a fourfold increase in the number of unknowns: $N \to 4N$

Observe that forward and backward elimination are the most expensive operations to be done in every timestep. Hence the computational effort increases by a factor of

$$\max\{4^2, \sqrt{2} \cdot 4^{\frac{3}{2}}\} = \boxed{16} \,.$$

The first factor is due to the LU/Cholesky factorization that has to be done once before entering the timestepping loop. The second factor reflects the increased cost of a single timestep and the larger number of timesteps required.

Surprisingly, it is the initial matrix factorization that causes the bulk of the extra work required to reduce the error. If you did not resort to advance factorization, the picture would be different.

SOLUTION of (9-6.b):

We face a *linear* evolution problem. Therefore it is sufficient to show that $u_0 \equiv v_0 \equiv 0$ implies $u \equiv 0$.

Remark. This is a fundamental argument for linear equations $\mathsf{L}x = y$, L a linear operator between vector spaces, no matter whether set in finite or infinite dimensions. Solutions are unique, *if and only if* $\mathsf{L}x = 0$ has only the trivial solution $x = 0$.

In Sub-problem (9-6.a) we found

$$\mathsf{m}(\frac{du}{dt}(t), \frac{du}{dt}(t)) + \mathsf{a}(u(t), u(t)) = \mathsf{m}(v_0, v_0) + \mathsf{a}(u_0, u_0) \quad \forall t \in [0, T] \,. \tag{9.6.7}$$

If $u_0 \equiv v_0 \equiv 0$ this implies

$$\left\| \frac{du}{dt}(t) \right\|_{L^2(\partial\Omega)} = 0 \quad , \quad \int_\Omega (1 + \|\boldsymbol{x}\|^2) \|\mathbf{grad}\, u(t)\|^2 \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall t \in [0, T] \,.$$

We infer $\left.\frac{du}{dt}\right|_{\partial\Omega} = 0$, which means that $u(t)|_{\partial\Omega}$ does not depend on time. Since $\|u(0)\|_{L^2(\partial\Omega)} = 0$, we conclude that $u|_{\partial\Omega} = 0$ for all times. We also know that $\mathbf{grad}\, u(t) \equiv 0 \Leftrightarrow u(t) \equiv \mathrm{const}$ for all t and, thus, $u(t) \equiv 0$ for all t.

Remark. For the present wave propagation problem the situation is more complicated than for hyperbolic evolution problems of the form

$$u \in V(t): \quad \mathsf{m}(\ddot{u}, v) + \mathsf{a}(u, v) = 0 \quad \forall v \in V_0 \,, \qquad \text{([Lecture → (9.3.1.8)])}$$

with *symmetric, positive definite (s.p.d.)* bilinear forms $\mathsf{m}(\cdot,\cdot)$ and $\mathsf{a}(\cdot,\cdot)$. In this case, the energy conservation principle permits us to conclude:

$$u(0) = \dot{u}(0) = 0 \quad \Rightarrow \quad \mathsf{m}(\dot{u}(t), \dot{u}(t)) + \mathsf{a}(u(t), u(t)) = 0 \quad \forall t \quad \overset{\text{a s.p.d.}}{\Rightarrow} \quad u(t) = 0 \quad \forall t \,.$$

SOLUTION of (9-6.c):

We follow the approach of [Lecture → Ex. 1.5.3.11]. First we test with $v \in C_0^\infty(\Omega)$ and apply Green's formula

$$\int_\Omega (1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div}\big((1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u\big)\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C_0^\infty(\Omega)$$

and we conclude

$$-\operatorname{div}\big((1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x})\big) = 0 \quad \text{on} \quad \Omega\times[0,T]\,.$$

Then we test with general $v \in C^\infty(\overline{\Omega})$, incur a boundary term and use the PDE just established:

$$\begin{aligned} 0 &= \frac{d^2}{dt^2}\int_{\partial\Omega} u(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) + \int_\Omega (1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x},t)\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ &= \int_{\partial\Omega}\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) - \int_\Omega \underbrace{\operatorname{div}\big((1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x},t)\big)}_{=0}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \\ &\quad \int_{\partial\Omega}(1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}).v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \\ &= \int_{\partial\Omega}\left(\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) + (1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\right)v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,. \end{aligned}$$

This gives the boundary condition

$$\frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) + (1+\|\boldsymbol{x}\|^2)\,\mathbf{grad}\,u(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \text{on} \quad \partial\Omega\times[0,T]\,. \tag{9.6.8}$$

SOLUTION of (9-6.d):

Using the abstract notation (9.6.3), the formulas are the same as those in [Lecture → Section 9.2.4]:

$$
\begin{aligned}
(\mathbf{M})_{i,j} &= \mathsf{m}(b_h^j, b_h^i) = \int_{\partial\Omega} b_h^j(\boldsymbol{x})\, b_h^i(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \;, \\
(\mathbf{A})_{i,j} &= \mathsf{a}(b_h^j, b_h^i) = \int_{\Omega} (1 + \|\boldsymbol{x}\|^2)\, \mathbf{grad}\, b_h^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \;,
\end{aligned}
\qquad 1 \le i, j \le N \;.
$$

SOLUTION of (9-6.e):

The matrix $\mathbf{M}$ is merely symmetric and positive semi-definite, but *not invertible*, because

$$v_h \in \mathcal{S}_1^0(\mathcal{M}) \cap H_0^1(\Omega) \quad \Rightarrow \quad \mathsf{m}(v_h, v_h) = 0 \, .$$

This means that any coefficient vector whose entries corresponding to basis functions associated with nodes on $\partial\Omega$ all vanish belongs to the kernel of $\mathbf{M}$.

However, the leapfrog timestepping scheme as given in [Lecture $\rightarrow$ (9.3.4.17)] can find $\vec{\boldsymbol{v}}^{(j+\frac{1}{2})}$ only if linear systems of equations with coefficient matrix $\mathbf{M}$ have unique solutions.

HINT 1 for (9-6.f): You may use LEHRFEM++'s local assembly facilities together with the auxiliary class **lf::fe::MassEdgeMatrixProvider** that fits the concept of an **ENTITY_MATRIX_PROVIDER**. ⌟

HINT 2 for (9-6.f): The assembly of Galerkin matrices belonging to an $L^2(\partial\Omega)$ inner product is explained in [Lecture $\rightarrow$ Ex. 2.7.4.41].

SOLUTION of (9-6.f):

An implementation using **lf::fe::MassEdgeMatrixProvider**.

C++ code 9.6.10: Implementation of assembly for matrix $\mathbf{M}$ ↬ GitLab

```
lf::assemble::COOMatrix<double> buildM(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p) {
  // I. TOOLS AND DATA
  // Pointer to current fe_space and mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p(fe_space_p->Mesh());
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());

  // II : ASSEMBLY
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> M(N_dofs, N_dofs);
  // Obtain an array of boolean flags for the edges of the mesh, 'true'
  // indicates that the edge lies on the boundary
  auto bd_flags{lf::mesh::utils::flagEntitiesOnBoundary(mesh_p, 1)};
  // Creating a predicate that will guarantee that the computations are carried
  // only on the edges of the mesh using the boundary flags
  // Actually a redundant step, because 'bdflags' is a predicate already.
  auto edges_predicate = [&bd_flags](const lf::mesh::Entity &edge) -> bool {
    return bd_flags(edge);
  };
  // Coefficient function used in the class template MassEdgeMatrixProvider
  auto eta = lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> double { return 1.0; });
  lf::uscalfe::MassEdgeMatrixProvider<double, decltype(eta),
                                      decltype(edges_predicate)>
      edgemat_builder(fe_space_p, eta, edges_predicate);
  // Invoke assembly on edges by specifying co-dimension = 1
  lf::assemble::AssembleMatrixLocally(1, dofh, dofh, edgemat_builder, M);
  return M;
};
```

Alternatively, one could write a simple **ENTITY_MATRIX_PROVIDER** class, whose `Eval()` function returns the 2×2 mass matrix for an edge and whose `isActive()` function selects edges on the boundary. This has been discussed in [Lecture → Ex. 2.7.4.41].

SOLUTION of (9-6.g):

We have to wrap the diffusion coefficient $\boldsymbol{x} \mapsto (1 + \|\boldsymbol{x}\|^2)$ into an **MeshFunction** object to pass it to the **ENTITY_MATRIX_PROVIDER** class.

C++ code 9.6.11: Implementation of assembly for matrix $\mathbf{M}$ ⇝ GitLab

```
lf::assemble::COOMatrix<double> buildA(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p) {
  // I. TOOLS AND DATA
  // Pointer to current fe_space and mesh
  std::shared_ptr<const lf::mesh::Mesh> mesh_p(fe_space_p->Mesh());
  // Obtain local->global index mapping for current finite element space
  const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
  // Dimension of finite element space
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());

  // II : ASSEMBLY
  // Matrix in triplet format holding Galerkin matrix, zero initially.
  lf::assemble::COOMatrix<double> A(N_dofs, N_dofs);
  // Coefficient functions used in class ReactionDiffusionElementMatrixProvider
  auto alpha = lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> double { return 1.0 + x.dot(x); });
  auto gamma = lf::mesh::utils::MeshFunctionGlobal(
      [](Eigen::Vector2d x) -> double { return 0.0; });
  // Initialize element matrix builder
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double, decltype(alpha),
                                                      decltype(gamma)>
      elmat_builder(fe_space_p, alpha, gamma);
  // Invoke assembly on cells (co-dimension = 0 as first argument)
  // Information about the mesh and the local-to-global map is passed through
  // a Dofhandler object, argument 'dofh'. This function call adds triplets to
  // the internal COO-format representation of the sparse matrix A.
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elmat_builder, A);
  return A;
};
```

SOLUTION of (9-6.h):

Note that `lf::uscalfe::NodalProjection()` wants the information about the function to be interpolated to be passed through functors compatible with **MeshFunction**.

C++ code 9.6.12: Implementation of `interpolateInitialData()` ↪ GitLab

```
2  template <typename FUNCTOR_U, typename FUNCTOR_V>
3  std::pair<Eigen::VectorXd, Eigen::VectorXd> interpolateInitialData(
4      std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space_p,
5      FUNCTOR_U &&u0, FUNCTOR_V &&v0) {
6    Eigen::VectorXd dof_vector_u0, dof_vector_v0;
7
8    // Generate Lehrfem++ mesh functions out of the functors
9    auto mf_u0 = lf::mesh::utils::MeshFunctionGlobal(
10       [&u0](Eigen::Vector2d x) -> double { return u0(x); });
11   auto mf_v0 = lf::mesh::utils::MeshFunctionGlobal(
12       [&v0](Eigen::Vector2d x) -> double { return v0(x); });
13
14   dof_vector_u0 = lf::fe::NodalProjection(*fe_space_p, mf_u0);
15   dof_vector_v0 = lf::fe::NodalProjection(*fe_space_p, mf_v0);
16
17   std::pair<Eigen::VectorXd, Eigen::VectorXd> initialData =
18       std::make_pair(dof_vector_u0, dof_vector_v0);
19   return initialData;
20 }
```

SOLUTION of (9-6.i):

There are two options:

❶ You may implement the first-order-system form of Crank-Nicolson timestepping as given in [Lecture → (9.3.4.11)]:

$$\begin{aligned} \vec{\mu}^{(j)} - \vec{\mu}^{(j-1)} &= \tfrac{1}{2}\tau(\vec{v}^{(j-1)} + \vec{v}^{(j)}) \\ \mathbf{M}(\vec{v}^{(j)} - \vec{v}^{(j-1)}) &= -\tfrac{1}{2}\tau\mathbf{A}(\vec{\mu}^{(j-1)} + \vec{\mu}^{(j)}) \,. \end{aligned} \tag{9.6.13}$$

It is convenient to eliminate the unknown $\vec{\mu}^{(j)}$ in order to avoid building a $2N \times 2N$ system matrix.

$$\mathbf{M}(\vec{v}^{(j)} - \vec{v}^{(j-1)}) = -\tfrac{1}{2}\tau\mathbf{A}(2\vec{\mu}^{(j-1)} + \tfrac{1}{2}\tau(\vec{v}^{(j-1)} + \vec{v}^{(j)})) \tag{9.6.14}$$

$$\Updownarrow$$

$$(\mathbf{M} + \tfrac{1}{4}\tau^2\mathbf{A})\vec{v}^{(j)} = (\mathbf{M} - \tfrac{1}{4}\tau^2\mathbf{A})\vec{v}^{(j-1)} - \tau\mathbf{A}\mu^{(j-1)} \,. \tag{9.6.15}$$

Thus, in every timestep a sparse $N \times N$ linear system of equations has to be solved. For the sake of efficiency, the LU/Cholesky factorization of the system matrix should be computed beforehand.

❷ Crank-Nicolson timestepping can also be implemented in "second-order form" [Lecture → (9.3.4.12)]

$$\begin{aligned} \mathbf{M}\frac{\vec{\mu}^{(j+1)} - 2\vec{\mu}^{(j)} + \vec{\mu}^{(j-1)}}{\tau^2} = &-\tfrac{1}{2}\mathbf{A}(\vec{\mu}^{(j-1)} + \vec{\mu}^{(j+1)}) + \\ &\tfrac{1}{2}(\vec{\varphi}(t - \tfrac{1}{2}\tau) + \vec{\varphi}(t + \tfrac{1}{2}\tau)) \,, \quad j = 1, 2, \ldots \end{aligned} \,. \tag{9.6.16}$$

In this case a special initial step has to be supplied.

C++ code 9.6.17: Implementation of `solveBoundaryWave()` ⇝ GitLab

```
2  template <typename FUNCTOR_U, typename FUNCTOR_V>
3  Eigen::VectorXd solveBoundaryWave(
4      const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
5      FUNCTOR_U &&u0, FUNCTOR_V &&v0, double T, unsigned int N) {
6    Eigen::VectorXd bdyWaveSol;
7
8    double step_size = T / N;
9    // Obtain initial data
10   std::pair<Eigen::VectorXd, Eigen::VectorXd> initialData =
11       interpolateInitialData<std::function<double(Eigen::Vector2d)>,
12                              std::function<double(Eigen::Vector2d)>>(
13           fe_space_p, std::forward<FUNCTOR_U>(u0), std::forward<FUNCTOR_V>(v0));
14   // Obtain Galerkin matrices
15   lf::assemble::COOMatrix<double> M = buildM(fe_space_p);
16   lf::assemble::COOMatrix<double> A = buildA(fe_space_p);
17   // Convert COO matrix M and A into CRS format using Eigen's internal
18   // conversion routines.
19   Eigen::SparseMatrix<double> M_sparse = M.makeSparse();
20   Eigen::SparseMatrix<double> A_sparse = A.makeSparse();
21   // Compute LU decomposition of coefficient matrix of LSE to be solved in every
22   // step of Crank-Nicolson timestepping
23   Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
24   solver.compute(M_sparse + 0.25 * (step_size * step_size) * A_sparse);
```

```
25    LF_VERIFY_MSG(solver.info() == Eigen::Success,
26                  "LU decomposition of M failed");
27
28    // Crank-Nicolson timestepping, first-order-system version
29    Eigen::VectorXd u_cur = initialData.first;
30    Eigen::VectorXd v_cur = initialData.second;
31    Eigen::VectorXd u_next, v_next;
32    for (int i = 1; i < N + 1; i++) {
33      // step foward
34      v_next = solver.solve(M_sparse * v_cur -
35                            0.25 * (step_size * step_size) * A_sparse * v_cur -
36                            step_size * A_sparse * u_cur);
37      u_next = u_cur + 0.5 * step_size * (v_cur + v_next);
38      // update
39      v_cur = v_next;
40      u_cur = u_next;
41    }
42    bdyWaveSol = u_cur;
43    return bdyWaveSol;
44  };
```

SOLUTION of (9-7.a):

Following the standard approach from [Lecture → Section 1.8], treating time as a mere parameter, we multiply (9.7.1a) with a test function $v : \Omega \to \mathbb{R}$ and integrate over Ω, which yields

$$\int_\Omega \rho(\boldsymbol{x}) \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\, v(\boldsymbol{x}) - \Delta u(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^\infty(\overline{\Omega}) \,.$$

Then we apply Green's formula from [Lecture → Thm. 1.5.2.7] and get

$$\int_\Omega \rho(\boldsymbol{x}) \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\, v(\boldsymbol{x}) + \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \mathbf{grad}\, u \cdot \boldsymbol{n}\, v\, \mathrm{d}S = 0 \,.$$

Finally, we use (9.7.1b) to replace $\mathbf{grad}\, u(\boldsymbol{x},t) \cdot \boldsymbol{n}(\boldsymbol{x})$ with $-\frac{\partial u}{\partial t}(\boldsymbol{x},t)$ and get

$$\int_\Omega \rho(\boldsymbol{x}) \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\, v(\boldsymbol{x}) + \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}S = 0 \,.$$

The right Sobolev space for the test function v is $H^1(\Omega)$. Hence, the final spatial variational form is: seek $u : [0,T] \to H^1(\Omega)$, $u(0) = u_0$, such that

$$\int_\Omega \rho(\boldsymbol{x}) \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\, v(\boldsymbol{x}) + \mathbf{grad}\, u(\boldsymbol{x},t) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}S = 0 \quad \forall v \in H^1(\Omega) \,.$$

SOLUTION of (9-7.j):

Obviously, doubling the number of timesteps roughly leads to a reduction of the (norm of) the discretization error by a factor of 4. Thus, we observe *algebraic convergence* with rate 2:

$$\mathrm{err}(M) = O(M^{-2}) \quad \text{for} \quad M \to \infty \,.$$

Thus, we conclude that the timestepping scheme probably is of order 2, because usually the order of a single-step timestepping method directly translates into algebraic convergence with a rate equal to the order, see .

Remark. The timestepping scheme from (9.7.5) is a blend of

- the Crank-Nicolson timestepping method for the wave equation as introduced in [Lecture → § 9.3.4.7]
- the implicit midpoint method [Lecture → (9.2.6.7)], which is known to be of order 2, see [Lecture → Ex. 9.2.6.8].

HINT 1 for (9-7.k): The imperative of efficient implementation demands that you rely on EIGEN's sparse direct elimination solvers and reuse LU-decomposition(s). ⌟

SOLUTION of (9-7.k):

C++ code 9.7.15: Class definition of WaveABC2DTimestepper ⇝ GitLab

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
class WaveABC2DTimestepper {
 public:
  // Main constructor; precomputations are done here
  WaveABC2DTimestepper(
      const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
      FUNC_RHO rho, unsigned int M, double T);

  // Public member functions
  Eigen::VectorXd solveWaveABC2D(FUNC_MU0 mu0, FUNC_NU0 nu0);
  double energies();

 private:
  double T_;          // final time
  unsigned int M_;    // nb of steps
  double step_size_;  // time inverval
  std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> fe_space_p_;
  bool timestepping_performed_;  // bool to assert that energies are computed
  lf::uscalfe::size_type N_dofs_;  // nb of degrees of freedom
                                   // only after timestepping
  // Precomputed objects
  std::vector<Eigen::Triplet<double>> A_triplets_vec_;  // stiffness matrix
  std::vector<Eigen::Triplet<double>> M_triplets_vec_;  // mass matrix
  Eigen::SparseMatrix<double> R_;                       // rhs evaluation matrix
  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver_;  // linear solver
  Eigen::VectorXd
      full_sol_;  // Vector for discrete solution and discrete velocity
};  // class WaveABC2DTimestepper
```

Of course, implementation has to be based on the recursion (9.7.6). However, it can be done in two different ways:

(I) You can directly build the $2N \times 2N$ coefficient matrix of the linear system of equations (9.7.6), compute its LU decomposition, and reuse that in every timestep. This entails modifying the row and column indices of some of the triplets.

(II) You can make use of (9.7.7) to solve (9.7.6). In this case you need the LU-decomposition of the $N \times N$ sparse matrix $\mathbf{M} + \frac{1}{2}\mathbf{B} + \frac{1}{4}\tau^2\mathbf{A}$ that should be precomputed.

For both approaches the method `MatVecMult()` of **lf::assmble::COOMatrix** may be used.

C++ code 9.7.16: Implementation of the constructor of WaveABC2DTimestepper ⇝ GitLab

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
WaveABC2DTimestepper<FUNC_RHO, FUNC_MU0, FUNC_NU0>::WaveABC2DTimestepper(
    const std::shared_ptr<lf::uscalfe::FeSpaceLagrangeO1<double>> &fe_space_p,
    FUNC_RHO rho, unsigned int M, double T)

    : fe_space_p_(fe_space_p),
      M_(M),
      T_(T),
```

```
    step_size_(T / M),
    timestepping_performed_(false) {
/* Creating coefficient-functions as Lehrfem++ mesh functions */
// Coefficient-functions used in the class template
// ReactionDiffusionElementMatrixProvider and MassEdgeMatrixProvider
auto rho_mf = lf::mesh::utils::MeshFunctionGlobal(rho);
auto zero_mf = lf::mesh::utils::MeshFunctionGlobal(
    [](Eigen::Vector2d) -> double { return 0.0; });
auto one_mf = lf::mesh::utils::MeshFunctionGlobal(
    [](Eigen::Vector2d) -> double { return 1.0; });

// On construction no timestepping was yet performed

std::cout << "Assembling Galerkin matrices..." << std::endl;
lf::assemble::COOMatrix<double> A_COO = computeGalerkinMat(
    fe_space_p, one_mf, zero_mf, zero_mf);  // stiffness matrix
lf::assemble::COOMatrix<double> M_COO =
    computeGalerkinMat(fe_space_p, zero_mf, rho_mf, zero_mf);  // Mass matrix
lf::assemble::COOMatrix<double> B_COO =
    computeGalerkinMat(fe_space_p, zero_mf, zero_mf,
                       one_mf);  // Boundary mass matrix

const lf::assemble::DofHandler &dofh{fe_space_p->LocGlobMap()};
N_dofs_ = dofh.NumDofs();
std::cout << "Number of degrees of freedom : " << N_dofs_ << std::endl;

std::cout << "Assembling the evolution matrix..." << std::endl;
/* Assemble the full linear system matrix of the stepping method */
/*              _                                  _
//       L  =  |  M+(1/2)*tau*B    (1/2)*tau*A  |
//             |_  -(1/2)*tau*I          I     _|
//                                                  */
lf::assemble::COOMatrix<double> L_COO(2 * N_dofs_, 2 * N_dofs_);
A_triplets_vec_ = A_COO.triplets();
M_triplets_vec_ = M_COO.triplets();
const std::vector<Eigen::Triplet<double>> B_triplets_vec = B_COO.triplets();
// Inserting M in L
for (auto &triplet : M_triplets_vec_) {
  L_COO.AddToEntry(triplet.row(), triplet.col(), triplet.value());
}
// Inserting B in L
for (auto &triplet : B_triplets_vec) {
  L_COO.AddToEntry(triplet.row(), triplet.col(),
                   0.5 * step_size_ * triplet.value());
}
// Inserting A in L
for (auto &triplet : A_triplets_vec_) {
  L_COO.AddToEntry(triplet.row(), triplet.col() + N_dofs_,
                   0.5 * step_size_ * triplet.value());
}
// Inserting I in L
for (int i = 0; i < N_dofs_; i++) {
  L_COO.AddToEntry(i + N_dofs_, i, -0.5 * step_size_);
  L_COO.AddToEntry(i + N_dofs_, i + N_dofs_, 1.0);
}
Eigen::SparseMatrix<double> L = L_COO.makeSparse();

```

```
  std::cout << "Computing the solver..." << std::endl;
  solver_.compute(L);
  if (solver_.info() != Eigen::Success) {
    throw std::runtime_error("Could not decompose the matrix!");
  }

  std::cout << "Assembling the RHS evaluation matrix..." << std::endl;
  /* Assemble the full rhs matrix of the stepping method */
  /*                _                                  _
  //       R  =  |  M-(1/2)*tau*B    -(1/2)*tau*A  |
  //             |_  (1/2)*tau*I          I        _|
  //                                                    */
  lf::assemble::COOMatrix<double> R_COO(2 * N_dofs_, 2 * N_dofs_);
  // Inserting M in R
  for (auto &triplet : M_triplets_vec_) {
    R_COO.AddToEntry(triplet.row(), triplet.col(), triplet.value());
  }
  // Inserting B in R
  for (auto &triplet : B_triplets_vec) {
    R_COO.AddToEntry(triplet.row(), triplet.col(),
                     -0.5 * step_size_ * triplet.value());
  }
  // Inserting A in R
  for (auto &triplet : A_triplets_vec_) {
    R_COO.AddToEntry(triplet.row(), triplet.col() + N_dofs_,
                     -0.5 * step_size_ * triplet.value());
  }
  // Inserting I in R
  for (int i = 0; i < N_dofs_; i++) {
    R_COO.AddToEntry(i + N_dofs_, i, 0.5 * step_size_);
    R_COO.AddToEntry(i + N_dofs_, i + N_dofs_, 1.0);
  }
  R_ = R_COO.makeSparse();
}
```

C++ code 9.7.17: Implementation of the method `solveWaveABC2D()` of WaveABC2DTimestepper ⇝ GitLab

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
Eigen::VectorXd WaveABC2DTimestepper<FUNC_RHO, FUNC_MU0,
                                     FUNC_NU0>::solveWaveABC2D(FUNC_MU0 mu0,
                                                               FUNC_NU0 nu0) {
  std::cout << "\nSolving variational problem of WaveABC2D." << std::endl;
  Eigen::VectorXd sol;

  // Initial conditions
  auto mf_mu0 = lf::mesh::utils::MeshFunctionGlobal(mu0);
  auto mf_nu0 = lf::mesh::utils::MeshFunctionGlobal(nu0);
  Eigen::VectorXd nu0_nodal = lf::fe::NodalProjection(*fe_space_p_, mf_nu0);
  Eigen::VectorXd mu0_nodal = lf::fe::NodalProjection(*fe_space_p_, mf_mu0);

  // Setup loop and tools
  Eigen::VectorXd cur_step_vec(2 * N_dofs_);
  Eigen::VectorXd next_step_vec(2 * N_dofs_);
  cur_step_vec.head(N_dofs_) = nu0_nodal;
  cur_step_vec.tail(N_dofs_) = mu0_nodal;
```

```
20    std::cout << "Performing discrete evolution..." << std::endl;
21    progress_bar progress{std::clog, 55u, "Timestepping"};
22    double progress_pourcentage;
23
24    // Performing timesteps
25    for (int i = 1; i < M_; i++) {
26      next_step_vec = solver_.solve(R_ * cur_step_vec);
27      cur_step_vec = next_step_vec;
28
29      // Display progress
30      progress_pourcentage = ((double)i + 1.0) / M_ * 100.0;
31      progress.write(progress_pourcentage / 100.0);
32    }
33
34    full_sol_ = cur_step_vec;
35    sol = full_sol_.tail(N_dofs_);
36    // Full solution is computed
37    timestepping_performed_ = true;
38
39    return sol;
40  } // solveWaveABC2D
```

SOLUTION of (9-7.l):

C++ code 9.7.18: Implementation of the method `energies()` of WaveABC2DTimestepper ↪ GitLab

```
template <typename FUNC_RHO, typename FUNC_MU0, typename FUNC_NU0>
double WaveABC2DTimestepper<FUNC_RHO, FUNC_MU0, FUNC_NU0>::energies() {
  double energy;
  Eigen::SparseMatrix<double> A_sps(N_dofs_, N_dofs_);
  A_sps.setFromTriplets(A_triplets_vec_.begin(), A_triplets_vec_.end());
  Eigen::SparseMatrix<double> M_sps(N_dofs_, N_dofs_);
  M_sps.setFromTriplets(M_triplets_vec_.begin(), M_triplets_vec_.end());

  double tmp1, tmp2;
  if (timestepping_performed_) {
    tmp1 =
        full_sol_.tail(N_dofs_).transpose() * A_sps * full_sol_.tail(N_dofs_);
    tmp2 =
        full_sol_.head(N_dofs_).transpose() * M_sps * full_sol_.head(N_dofs_);
    energy = tmp1 + tmp2;
  } else {
    energy = 0.0;
    std::cout << "You have not computed the solution and its velocity yet!"
              << std::endl;
  }
  return energy;
}
```

SOLUTION of (9-7.b):

In an abstract fashion, analoguous to [Lecture → (9.3.1.8)], this variational problem can be rewritten as: seek $u : [0,T] \to V(t)$

$$\mathsf{m}(\ddot{u}, v) + \mathsf{b}(\dot{u}, v) + \mathsf{a}(u, v) = 0 \quad \forall v \in V_0 \,, \tag{9.7.3}$$

where

$$\begin{aligned} \mathsf{m}(u, v) &:= \int_\Omega \rho(\boldsymbol{x}) u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,, \\ \mathsf{b}(u, v) &:= \int_{\partial\Omega} u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \,, \\ \mathsf{a}(u, v) &:= \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,. \end{aligned} \tag{9.7.4}$$

The matrices in the MOL-ODE (9.7.2) arise from inserting the basis functions of the trial and test space $\mathcal{S}_1^0(\mathcal{M})$ into the bilinear form defined in (9.7.3), see [Lecture → Section 9.3.3]. This yields

$$\begin{aligned} \mathbf{M} &= \left[\mathsf{m}(b_N^i, b_N^j)\right]_{i,j=1}^N \in \mathbb{R}^{N,N} && \text{(mass matrix)} \,, \\ \mathbf{A} &= \left[\mathsf{a}(b_N^i, b_N^j)\right]_{i,j=1}^N \in \mathbb{R}^{N,N} && \text{(stiffness matrix)} \,, \\ \mathbf{B} &= \left[\mathsf{b}(b_N^i, b_N^j)\right]_{i,j=1}^N \in \mathbb{R}^{N,N} && \text{“boundary mass matrix”} \,. \end{aligned}$$

SOLUTION of (9-7.c):

From (9.7.5), after multiplying both equations with τ, we read off

$$\begin{bmatrix} \mathbf{M} + \frac{1}{2}\tau\mathbf{B} & \frac{1}{2}\tau\mathbf{A} \\ -\frac{1}{2}\tau\mathbf{I} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\nu}^{(k+1)} \\ \vec{\mu}^{(k+1)} \end{bmatrix} = \begin{bmatrix} \mathbf{M}\vec{\nu}^{(k)} - \frac{1}{2}\tau\mathbf{B}\vec{\nu}^{(k)} - \frac{1}{2}\tau\mathbf{A}\vec{\mu}^{(k)} \\ \vec{\mu}^{(k)} + \frac{1}{2}\tau\vec{\nu}^{(k)} \end{bmatrix} . \tag{9.7.6}$$

$$= \begin{bmatrix} \mathbf{M} - \frac{1}{2}\tau\mathbf{B} & -\frac{1}{2}\tau\mathbf{A} \\ \frac{1}{2}\tau\mathbf{I} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\nu}^{(k)} \\ \vec{\mu}^{(k)} \end{bmatrix}$$

There are various ways to write this $2n \times 2n$ linear system of equations; permutations of rows and columns are possible as well as row transformations.

Remark. In (9.7.6) it is easy to eliminate $\vec{\mu}^{(k+1)}$. Abbreviating the right-hand side vector of (9.7.6) we obtain by block-Gaussian elimination

$$\begin{bmatrix} \mathbf{M} + \frac{1}{2}\tau\mathbf{B} & \frac{1}{2}\tau\mathbf{A} \\ -\frac{1}{2}\tau\mathbf{I} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\nu}^{(k+1)} \\ \vec{\mu}^{(k+1)} \end{bmatrix} = \begin{bmatrix} \vec{\beta}_1 \\ \vec{\beta}_2 \end{bmatrix}$$

$$\blacktriangleright \quad (\mathbf{M} + \tfrac{1}{2}\tau\mathbf{B} + \tfrac{1}{2}\tau^2\mathbf{A})\vec{\nu}^{(k+1)} = \vec{\beta}_1 - \tfrac{1}{2}\tau\mathbf{A}\vec{\beta}_2 . \tag{9.7.7}$$

At any step, $\vec{\mu}^{(k+1)}$ can instantly be recovered by the formula

$$\vec{\mu}^{(k+1)} = \vec{\beta}_2 + \tfrac{1}{2}\tau\vec{\nu}^{(k+1)} .$$

HINT 1 for (9-7.d): Exploit that the matrix $\mathbf{M}$ is positive semi-definite, and that the matrices $\mathbf{A}$ and $\mathbf{B}$ are symmetric and positive semi-definite. ⌟

HINT 2 for (9-7.d): One way to show that a matrix is regular, it to subject it to row transformations, as it is done in the course of Gaussian elimination. Here we face a block-structured matrix and, therefore, you can resort to block-row transformations. ⌟

SOLUTION of (9-7.d):

Implicitly using block-row transformations, we perform a block-LU factorization of the system matrix from (9.7.6):

$$\begin{bmatrix} \mathbf{M} + \frac{1}{2}\tau\mathbf{B} & \frac{1}{2}\tau\mathbf{A} \\ -\frac{1}{2}\tau\mathbf{I} & \mathbf{I} \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \frac{1}{2}\tau\mathbf{A} \\ \mathbf{O} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{M} + \frac{1}{2}\tau\mathbf{B} + \frac{1}{4}\tau^2\mathbf{A} & \mathbf{O} \\ -\frac{1}{2}\tau\mathbf{I} & \mathbf{I} \end{bmatrix} := \mathbf{L}\mathbf{U} \,. \tag{9.7.8}$$

The matrix $\mathbf{M}$ is positive semi-definite, and that the matrices $\mathbf{A}$ and $\mathbf{B}$ are symmetric and positive semi-definite. Hence, their weighted sum $\mathbf{M} + \frac{1}{2}\tau\mathbf{B} + \frac{1}{4}\tau^2\mathbf{A}$ is symmetric and positive semi-definite.

In case $\rho \equiv 0$, the matrix $\mathbf{M} = \mathbf{O}$ will be singular. Yet, the matrix sum $\frac{1}{2}\tau\mathbf{B} + \frac{1}{4}\tau^2\mathbf{A}$ is positive definite, because it arises from the Galerkin discretization of the bilinear form

$$c(u,v) := \tfrac{1}{2}\tau \int\limits_{\partial\Omega} u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) + \tfrac{1}{4}\tau^2 \int\limits_{\Omega} \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,.$$

This bilinear form is clearly symmetric and positive semi-definite. Yet, it is even positive definite, because

$$c(u,u) = 0 \quad \Rightarrow \quad \mathbf{grad}\, u \equiv 0 \quad \text{and} \quad u|_{\partial\Omega} \equiv 0 \quad \Rightarrow \quad u \equiv 0 \,.$$

As a consequence, $\frac{1}{2}\tau\mathbf{B} + \frac{1}{4}\tau^2\mathbf{A}$ will be invertible.

This ensures that the factor matrix $\mathbf{U}$ in (9.7.8) is invertible. Invertibility of $\mathbf{L}$ is immediate.

SOLUTION of (9-7.e):

The second line of (9.7.5) is a discretization of

$$\dot{\vec{\mu}} = \vec{v} \quad \leftrightarrow \quad \frac{\vec{\mu}^{(k+1)} - \vec{\mu}^{(k)}}{\tau} = \frac{1}{2}\left(\vec{v}^{(k+1)} + \vec{v}^{(k)}\right) .$$

Hence, we can read $\vec{v}^{(k)}$ as

> the basis expansion coefficient vector of a finite-element approximation of the function $x \mapsto \frac{\partial u}{\partial t}(x, t_{k+}$ in $\mathcal{S}_1^0(\mathcal{M})$.

SOLUTION of (9-7.f):

We follow the manipulations in [Lecture → § 9.3.4.7]. First insert the second equation of (9.7.5) into the first:

$$\mathbf{M}\,\frac{\vec{\boldsymbol{v}}^{(k+1)}-\vec{\boldsymbol{v}}^{(k)}}{\tau} = -\tfrac{1}{2}\mathbf{B}\frac{\vec{\boldsymbol{\mu}}^{(k+1)}-\vec{\boldsymbol{\mu}}^{(k)}}{\tau} - \mathbf{A}\left(\vec{\boldsymbol{\mu}}^{(k+1)}+\vec{\boldsymbol{\mu}}^{(k)}\right) .$$

Add these equations for two consecutive timesteps:

$$\mathbf{M}\,\frac{\vec{\boldsymbol{v}}^{(k+1)}-\vec{\boldsymbol{v}}^{(k-1)}}{\tau} = -\tfrac{1}{2}\mathbf{B}\frac{\vec{\boldsymbol{\mu}}^{(k+1)}-\vec{\boldsymbol{\mu}}^{(k-1)}}{\tau} - \mathbf{A}\left(\vec{\boldsymbol{\mu}}^{(k+1)}+2\vec{\boldsymbol{\mu}}^{(k)}+\vec{\boldsymbol{\mu}}^{(k-1)}+\right) . \tag{9.7.9}$$

Then subtract the two versions of the second equations for consecutive timesteps:

$$\mathbf{M}\frac{\vec{\boldsymbol{\mu}}^{(k+1)}-2\vec{\boldsymbol{\mu}}^{(k)}+\vec{\boldsymbol{\mu}}^{(k-1}}{\tau} = \tfrac{1}{2}(\vec{\boldsymbol{v}}^{(k+1)}-\vec{\boldsymbol{v}}^{(k-1)}) ,$$

and plug this into (9.7.9):

$$\frac{\vec{\boldsymbol{\mu}}^{(k+1)}-2\vec{\boldsymbol{\mu}}^{(k)}+\vec{\boldsymbol{\mu}}^{(k-1}}{\tau^2} + \mathbf{B}\frac{\vec{\boldsymbol{\mu}}^{(k+1)}-\vec{\boldsymbol{\mu}}^{(k-1)}}{2\tau} + \tfrac{1}{4}\mathbf{A}\left(\vec{\boldsymbol{\mu}}^{k+1}+2\vec{\boldsymbol{\mu}}^{(k)}+\vec{\boldsymbol{\mu}}^{(k-1)}\right) = 0 . \tag{9.7.10}$$

From this, after multiplying with τ^2, we conclude

$$\vec{\boldsymbol{\mu}}^{(k+1)} = (\mathbf{M}+\tfrac{1}{2}\tau\mathbf{B}+\tfrac{1}{4}\tau^2\mathbf{A})^{-1}\Big((2\mathbf{M}+\tfrac{1}{2}\tau^2\mathbf{A})\vec{\boldsymbol{\mu}}^{(k)} - (\mathbf{M}-\tfrac{1}{2}\tau\mathbf{B}+\tfrac{1}{4}\tau^2\mathbf{A})\vec{\boldsymbol{\mu}}^{(k-1)}\Big) .$$

SOLUTION of (9-7.g):

- We choose $\vec{\boldsymbol{\mu}}(0)$ as the basis expansion coefficient vector of a projection of u_0 onto $\mathcal{S}_1^0(\mathcal{M})$. For instance, one may use the nodal interpolant

$$\sum_{j=1}^{N} (\vec{\boldsymbol{\mu}}(0))_j b_N^j = \mathsf{I}_1 u_0 \,,$$

  if u_0 is continuous.

- Since $\dot{u}(0) \equiv 0$ and $\vec{\boldsymbol{v}}(0)$ approximates this function, we can simply set $\vec{\boldsymbol{v}}(0) = \mathbf{0}$.

SOLUTION of (9-7.h):

We repeat the arguments from [Lecture → § 9.3.4.7]. Multiply the second equation of (9.7.5) with $\left(\vec{\mu}^{(k+1)} + \vec{\mu}^{(k)}\right)^{\top}\mathbf{A}$ from left,

$$\left(\vec{\mu}^{(k+1)} + \vec{\mu}^{(k)}\right)^{\top}\mathbf{A}\left(\vec{\mu}^{(k+1)} - \vec{\mu}^{(k)}\right) = \tfrac{1}{2}\tau\left(\vec{\mu}^{(k+1)} + \vec{\mu}^{(k)}\right)^{\top}\mathbf{A}\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right) ,$$

and the first equation with $\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right)^{\top}$ from left:

$$\begin{aligned}\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right)^{\top}\mathbf{M}\,\frac{\vec{\nu}^{(k+1)} - \vec{\nu}^{(k)}}{\tau} \\ = -\tfrac{1}{2}\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right)^{\top}\mathbf{B}\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right) - \tfrac{1}{2}\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right)^{\top}\mathbf{A}\left(\vec{\mu}^{(k+1)} + \vec{\mu}^{(k)}\right) .\end{aligned}$$

Make use of the "$a^2 - b^2 = (a+b)(a-b)$" identity for inner products, e.g.,

$$\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right)^{\top}\mathbf{M}\left(\vec{\nu}^{(k+1)} - \vec{\nu}^{(k)}\right) = \left(\vec{\nu}^{(k+1)}\right)^{\top}\mathbf{M}\vec{\nu}^{(k+1)} - \left(\vec{\nu}^{(k)}\right)^{\top}\mathbf{M}\vec{\nu}^{(k)} .$$

This reveals:

$$E^{(k+1)} - E^{(k)} = -\tfrac{1}{2}\tau\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right)^{\top}\mathbf{B}\left(\vec{\nu}^{(k+1)} + \vec{\nu}^{(k)}\right) \le 0 ,$$

because the Galerkin matrix $\mathbf{B}$ is *positive semi-definite*, which is clear from the underlying bilinear form $b(\cdot,\cdot)$ as given in (9.7.4).

SOLUTION of (9-7.i):

We face the special situation of (9.7.2) with $N = 1$ and "1×1-matrices"

$$\mathbf{M} = \epsilon \quad , \quad \mathbf{A} = 1 \quad , \quad \mathbf{B} = 1 - \epsilon \, .$$

Thus, the linear systems of equations obtained in Sub-problem (9-7.c) becomes

$$\begin{bmatrix} \epsilon + \frac{1}{2}\tau(1-\epsilon) & \frac{1}{2}\tau \\ -\frac{1}{2}\tau & 1 \end{bmatrix} \begin{bmatrix} \nu^{(k+1)} \\ \mu^{(k+1)} \end{bmatrix} = \begin{bmatrix} \epsilon\nu^{(k)} - \frac{1}{2}\tau(1-\epsilon)\nu^{(k)} - \frac{1}{2}\tau\mu^{(k)} \\ \mu^{(k)} + \frac{1}{2}\tau\nu^{(k)} \, , \quad k \in \mathbb{N}_0 \end{bmatrix} .$$

This recursion can be implemented directly.

C++ code 9.7.13: Implementation of `scalarImplicitTimestepping()` ↬ GitLab

```
2  Eigen::VectorXd scalarImplicitTimestepping(double epsilon, unsigned int M) {
3    /* PROBLEM SETUP */
4    double step_size = 1.0 / M;  // time step "tau"
5    // Rows contain sequence of solutions states
6    Eigen::MatrixXd x(M + 1, 2);
7    // INITIAL DATA
8    x(0, 0) = 1.0;
9    x(0, 1) = 1.0;
10
11   Eigen::MatrixXd A(2, 2);
12   Eigen::VectorXd b(2);
13   // clang-format off
14     A << epsilon + 0.5*step_size*(1-epsilon), 0.5*step_size,
15          -0.5*step_size,                      1.0;
16   // clang-format on
17   // TIMESTEPPING
18   for (int k = 1; k < M + 1; k++) {
19     // Update right hand side
20     b(0) = (epsilon - 0.5 * step_size * (1 - epsilon)) * x(k - 1, 0) -
21            0.5 * step_size * x(k - 1, 1);
22     b(1) = x(k - 1, 1) + 0.5 * step_size * x(k - 1, 0);
23     // Solve the linear system
24     x.row(k) = A.fullPivLu().solve(b);
25   }
26   return x.col(1);
27 }  // scalarImplicitTimestepping
```

SOLUTION of (9-8.a):

We can write $u = v + \imath w$ with real valued functions $v, w : \Omega \to \mathbb{R}$ and convert (NLSE) into

$$\begin{aligned} &\imath \frac{\partial}{\partial t}(v + \imath w) = -\Delta(v + \imath w) + |v + \imath w|^2 (v + \imath w) \quad \text{in} \quad \Omega \times [0, T] \,, \\ &-\frac{\partial w}{\partial t} + \imath \frac{\partial v}{\partial t} \Delta v + \imath \Delta w + (v^2 + w^2)(v + \imath w) \quad \text{in} \quad \Omega \times [0, T] \,. \end{aligned}$$

Separating real and imaginary parts we arrive at

$$\begin{aligned} &\text{Im}: \quad \dot{v} = -\Delta w + (v^2 + w^2) w \,, \\ &\text{Re}: \quad \dot{w} = \Delta v - (v^2 + w^2) v \end{aligned} \qquad \text{in} \quad \Omega \times [0, T] \,. \tag{9.8.1}$$

In block-operator notation we can write

$$\frac{\partial}{\partial t} \begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} 0 & \Delta \\ -\Delta & 0 \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} + \begin{bmatrix} (v^2 + w^2) & 0 \\ 0 & (v^2 + w^2) \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \quad \text{in} \quad \Omega \times [0, T] \,. \tag{9.8.2}$$

SOLUTION of (9-8.j):

We apply the implicit midpoint rule with timestep $\tau/2$ to $\mathbf{M}\dot{\vec{\mu}}(t) = \mathbf{A}\vec{\mu}(t)$,

$$\mathbf{M}\,\frac{\vec{\mu}^{(k+1/2)} - \vec{\mu}^{(k)}}{\tau/2} = \frac{1}{2}\mathbf{A}\left(\vec{\mu}^{(k+1/2)} + \vec{\mu}^{(k)}\right) .$$

and obtain

$$\vec{\mu}^{(k+1/2)} = \left(\mathbf{M} - \frac{\tau}{4}\mathbf{A}\right)^{-1}\left(\mathbf{M} + \frac{\tau}{4}\mathbf{A}\right)\vec{\mu}^{(k)} . \tag{9.8.27}$$

Obviously the matrices do not depend on j, which has to be exploited in an efficient implementation.

HINT 1 for (9-8.k): The discrete evolution operator $\vec{\boldsymbol{\mu}} \in \mathbb{C}^N \mapsto \boldsymbol{\Psi}_{\mathrm{IMP}}^{0,\tau} \vec{\boldsymbol{\mu}}$ involves the solution of a sparse linear system of equations, whose LU-factorization has to be *precomputed* in the constructor for the sake of efficiency. ⌟

SOLUTION of (9-8.k):

The constructor and evaluation operator combined realize (9.8.27).

C++ code 9.8.28: Implementation of class KineticPropagator ⇝ GitLab

```
2  KineticPropagator::KineticPropagator(const SparseMatrixXd &A,
3                                       const SparseMatrixXcd &M, double tau) {
4    // Defeats the rationale of expression templates, but acceptable here,
       because
5    // executed only once in the constructor.
6    B_plus_ = M + 0.5 * tau * A.cast<std::complex<double>>();
7    SparseMatrixXcd B_minus = M - 0.5 * tau * A.cast<std::complex<double>>();
8    // This is the expensive step:  LU-factorization of a big sparse
       matrix.
9    // Precomputation is essential
10   solver_.compute(B_minus);
11 }
12
13 Eigen::VectorXcd KineticPropagator::operator()(
14     const Eigen::VectorXcd &mu) const {
15   // Cheap elimination steps operating on the LU-factors.  Effort is
       almost O(N)
16   // thanks to sophisticated fill-in avoiding techniques employed by the
       sparse
17   // solvers.
18   return solver_.solve(B_plus_ * mu);
19 }
```

SOLUTION of (9-8.l):

C++ code 9.8.29: Definition of class SplitStepPropagator ⇝ GitLab

```
2  class SplitStepPropagator : public Propagator {
3    using SparseMatrixXcd = Eigen::SparseMatrix<std::complex<double>>;
4    using SparseMatrixXd = Eigen::SparseMatrix<double>;
5
6   public:
7    // @brief Forwards the arguments to the constructors of the underlying
8    // propagators KineticPropagator (semi-step) and InteractionPropagator
9    // (full-step).
10   // @param A stiffness matrix of shape N × N
11   // @param M complex mass matrix of shape N × N
12   // @param tau size of the timestep to perform by Strang splitting
13   //
14   SplitStepPropagator(const SparseMatrixXd &A, const SparseMatrixXcd &M,
15                       double tau);
16   //* @brief Performs the propagation according Strang splitting between
       the
17   //* kinetic (semi-step) and interaction (full-step) propagator.
18   //* @param mu vector of length N containing nodal values
19   //* before the timestep
20   //* @return vector of length N containg the nodal values
21   //* after the timestep
22   //*
23   Eigen::VectorXcd operator()(const Eigen::VectorXcd &mu) const override;
24
25  private:
26   // Kinetic propagator for semi step Ψ^{0,τ/2}
27   KineticPropagator kineticPropagator_;
28   // Interaction propagator for full step
29   InteractionPropagator interactionPropagator_;
30 };
```

In the constructor we set up objects of type **InteractionPropagator** (for timestep τ) and **KineticPropagator** (for timestep $\tau/2$). In the evaluation operator we call **operator()** for both, twice for the **KineticPropagator** object.

C++ code 9.8.30: Implementation of class SplitStepPropagator ⇝ GitLab

```
2  SplitStepPropagator::SplitStepPropagator(const SparseMatrixXd &A,
3                                           const SparseMatrixXcd &M, double tau)
4      : kineticPropagator_(A, M, 0.5 * tau), interactionPropagator_(tau) {}
5
6  Eigen::VectorXcd SplitStepPropagator::operator()(
7      const Eigen::VectorXcd &mu) const {
8    Eigen::VectorXcd nu(mu.size());
9    nu = kineticPropagator_(mu);
10   nu = interactionPropagator_(nu);
11   nu = kineticPropagator_(nu);
12   return nu;
13 }
```

SOLUTION of (9-8.m):

C++ code 9.8.32: Solver for nonlinear Schrödinger equation ⇢ GitLab

```
double Norm(const Eigen::VectorXcd &mu, const Eigen::SparseMatrix<double> &D) {
  return std::sqrt(mu.dot(D * mu).real());
}

double KineticEnergy(const Eigen::VectorXcd &mu,
                     const Eigen::SparseMatrix<double> &A) {
  return 0.5 * mu.dot(A * mu).real();
}

double InteractionEnergy(const Eigen::VectorXcd &mu,
                         const Eigen::SparseMatrix<double> &D) {
  Eigen::VectorXd mu_abs2 = mu.cwiseAbs2();
  return 0.25 * mu_abs2.dot(D * mu_abs2);
}
```

SOLUTION of (9-8.n):

C++ code 9.8.33: Solver for nonlinear Schrödinger equation ⇢ GitLab

```
  // Load mesh and initalize FE space and DOF handler
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  const lf::io::GmshReader reader(std::move(mesh_factory),
                                  "meshes/square_64.msh");
  auto mesh_p = reader.mesh();
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());

  // Mass matrix
  lf::assemble::COOMatrix<double> D_COO(N_dofs, N_dofs);
  NonLinSchroedingerEquation::MassElementMatrixProvider mass_emp;
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, mass_emp, D_COO);
  Eigen::SparseMatrix<double> D = D_COO.makeSparse();
  Eigen::SparseMatrix<std::complex<double>> M = std::complex<double>(0, 1) * D;

  // Stiffness matrix
  lf::assemble::COOMatrix<double> A_COO(N_dofs, N_dofs);
  lf::uscalfe::LinearFELaplaceElementMatrix stiffness_emp;
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, stiffness_emp, A_COO);
  Eigen::SparseMatrix<double> A = A_COO.makeSparse();

  // Prepare timestepping
  int timesteps = 100;
  double T = 1.0;
  double tau = T / timesteps;

// Prepare inital data
  const double PI = 3.14159265358979323846;
  auto u0 = [PI](Eigen::Vector2d x) -> double {
    return 4.0 * std::cos(PI * x(0)) * std::cos(PI * x(1));
  };
  lf::mesh::utils::MeshFunctionGlobal mf_u0{u0};
  Eigen::VectorXcd mu = lf::fe::NodalProjection(*fe_space, mf_u0);

  // Prepare split-step propagator for full step τ
  NonLinSchroedingerEquation::SplitStepPropagator splitStepPropagator(A, M,
                                                                      tau);

  // Arrays for storing "energies" contributing to the Hamiltonian
  Eigen::VectorXd norm(timesteps + 1);
  Eigen::VectorXd E_kin(timesteps + 1);
  Eigen::VectorXd E_int(timesteps + 1);
  // Timestepping
  for (int j = 0; j < timesteps; ++j) {
    // Compute norm and energy along the solution
    norm(j) = NonLinSchroedingerEquation::Norm(mu, D);
    E_kin(j) = NonLinSchroedingerEquation::KineticEnergy(mu, A);
    E_int(j) = NonLinSchroedingerEquation::InteractionEnergy(mu, D);
    // Timestep tau according to Strang splitting
    mu = splitStepPropagator(mu);
```

```
54      }
55      norm(timesteps) = NonLinSchroedingerEquation::Norm(mu, D);
56      E_kin(timesteps) = NonLinSchroedingerEquation::KineticEnergy(mu, A);
57      E_int(timesteps) = NonLinSchroedingerEquation::InteractionEnergy(mu, D);
58
59      // Timegrid
60      Eigen::VectorXd t = Eigen::VectorXd::LinSpaced(timesteps + 1, 0.0, T);
61
62      // Nice output format
63      const static Eigen::IOFormat CSVFormat(Eigen::FullPrecision,
64                                             Eigen::DontAlignCols, ", ", "\n");
65
66      // Write norm to file
67      std::ofstream norm_csv;
68      norm_csv.open("norm.csv");
69      norm_csv << t.transpose().format(CSVFormat) << std::endl;
70      norm_csv << norm.transpose().format(CSVFormat) << std::endl;
71      norm_csv.close();
```

HINT 1 for (9-8.o): First show that the moduli of components of solutions of the ODE from (9.8.19) are constant. ⌟

HINT 2 for (9-8.o): Next show that an application of linear operator $\mathbf{\Psi}_{\mathrm{IMP}}^{0,\tau}$ does not change the "$\mathbf{D}$-norm" of the vector, when $\mathbf{M} = \imath\mathbf{D}$ with a symmetric, positive definite matrix $\mathbf{D} \in \mathbb{R}^{N,N}$. ⌟

SOLUTION of (9-8.o):

1. The solutions (9.8.25) of (9.8.19) clearly have constant length because $|\exp(\imath\gamma)| = 1$ for all $\gamma \in \mathbb{R}$.
2. As we saw in Sub-problem (9-8.j), we have the representation

$$\Psi_{\mathrm{IMP}}^{0,\tau}\vec{\mu} = \left(\mathbf{M} - \frac{\tau}{2}\mathbf{A}\right)^{-1}\left(\mathbf{M} + \frac{\tau}{2}\mathbf{A}\right)\vec{\mu}\,, \quad \vec{\mu} \in \mathbb{C}^N\,,$$

with $\mathbf{M} = \imath\mathbf{D}$, and $\mathbf{D}, \mathbf{A} \in \mathbb{R}^{N,N}$ symmetric and positive definite. We can rewrite

$$\vec{\xi} := \Psi_{\mathrm{IMP}}^{0,\tau}\vec{\mu} \quad \Leftrightarrow \quad \imath\mathbf{D}(\vec{\xi} - \vec{\mu}) = \frac{\tau}{2}\mathbf{A}(\vec{\xi} + \vec{\mu})\,.$$

We can left-multiply the last equation with $(\vec{\xi} + \vec{\mu})^{\mathrm{H}}$:

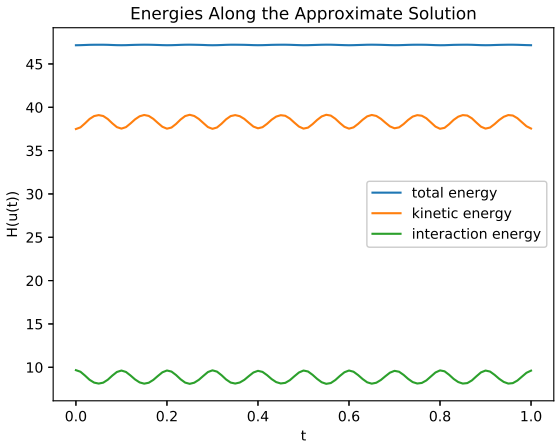
$$\blacktriangleright \quad \imath\Big(\underbrace{\vec{\xi}^{\mathrm{H}}\mathbf{D}\vec{\xi}}_{\in\mathbb{R}} - \underbrace{\vec{\mu}^{\mathrm{H}}\mathbf{D}\vec{\mu}}_{\in\mathbb{R}} + 2\,\mathrm{Im}\{\vec{\xi}^{\mathrm{H}}\mathbf{D}\vec{\mu}\}\imath\Big) = \underbrace{\frac{\tau}{2}\,(\vec{\xi} + \vec{\mu})^{\mathrm{H}}\mathbf{A}(\vec{\xi} + \vec{\mu})}_{\in\mathbb{R}}\,.$$

Take the imaginary part on both sides:

$$\blacktriangleright \quad \vec{\xi}^{\mathrm{H}}\mathbf{D}\vec{\xi} - \vec{\mu}^{\mathrm{H}}\mathbf{D}\vec{\mu} = 0\,, \quad \vec{\xi} := \Psi_{\mathrm{IMP}}^{0,\tau}\vec{\mu}\,, \quad \vec{\mu} \in \mathbb{C}^N\,.$$

SOLUTION of (9-8.p):

The plot for $H(u_h(t_j))$ is depicted in Fig. 125. As we expect from Sub-problem (9-8.o) the values $H(u_h(t_j))$ are independent of j.



◁ Hamiltonian $H(u_h(t_j))$, "kinetic energy", and "interaction energy" for timesteps $j = 0, \ldots, 100$.

HINT 1 for (9-8.b): The approach is the same as in [Lecture → § 9.2.2.2] and we can utterly "ignore" the fact that all functions involved are complex-valued.

Also note that the non-linear term does not require any special treatment. ⌟

SOLUTION of (9-8.b):

The variational formulation derived from (NLSE) by a straightforward application of Green's formula:

Theorem [Lecture → Thm. 1.5.2.7]. Green's first formula

For all vector fields $\mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d$ *and functions* $v \in C^1_{\mathrm{pw}}(\overline{\Omega})$ *holds*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v \,\mathrm{d}S\,. \tag{9.8.3}$$

$$u : [0,T] \to H^1(\Omega): \quad \int_\Omega \imath \dot{u}\, u' \,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, u' + |u|^2 u u' \,\mathrm{d}\boldsymbol{x} \quad \forall u' \in H^1(\Omega)\,, \tag{9.8.4}$$

where $H^1(\Omega)$ has to be understood as the Sobolev space of complex-valued functions. Boundary terms can be dropped thanks to the homogeneous Neumann boundary conditions, *cf.* [Lecture → Ex. 1.8.0.10].

Remark. To validate our result, we can also tackle the real reformulation (9.8.1), because we should end up with the same variational problem. We rely on the same Green's formula of [Lecture → Thm. 1.5.2.7] for both equations and get:

$$\begin{array}{l} v : [0,T] \to H^1(\Omega) \\ w : [0,T] \to H^1(\Omega) \end{array} : \quad \begin{array}{rcl} \int_\Omega \dot{v}\, w' \,\mathrm{d}\boldsymbol{x} & = & \int_\Omega \mathbf{grad}\, w \cdot \mathbf{grad}\, w' + (v^2 + w^2) w w' \,\mathrm{d}\boldsymbol{x}\,, \\ \int_\Omega \dot{w}\, v' \,\mathrm{d}\boldsymbol{x} & = & -\int_\Omega \mathbf{grad}\, v \cdot \mathbf{grad}\, v' + (v^2 + w^2) v v' \,\mathrm{d}\boldsymbol{x}\,, \end{array} \tag{9.8.5}$$

for all $w', v' \in H^1_0(\Omega)$. We have used the fact that

$$\mathbf{grad}\, u \cdot \boldsymbol{n} = 0 \quad \Leftrightarrow \quad \mathbf{grad}\, v \cdot \boldsymbol{n} = \mathbf{grad}\, w \cdot \boldsymbol{n} = 0\,,$$

which makes all boundary terms disappear.

Remark. A priori it is not clear, whether the non-linear terms in (9.8.4) and (9.8.5) are well-defined for $v, w \in H^1(\Omega)$. This can be confirmed using rather profound estimates for Sobolev norms of products of functions.

HINT 1 for (9-8.c): You can show this assertion based both on the complex-valued spatial variational formulation and an equivalent real-valued system. ⌟

HINT 2 for (9-8.c): If you have decided to rely on the complex-valued variational formulation, you can use the following version of the product rule:

$$\frac{d}{dt}\int_\Omega |u|^2\,\mathrm{d}\boldsymbol{x} = \frac{d}{dt}\int_\Omega u\overline{u}\,\mathrm{d}\boldsymbol{x} = \int_\Omega \dot{u}\overline{u} + u\dot{\overline{u}}\,\mathrm{d}\boldsymbol{x} = 2\,\mathrm{Re}\left\{\int_\Omega \dot{u}\overline{u}\,\mathrm{d}\boldsymbol{x}\right\}. \tag{9.8.7}$$

Here we designate complex conjugation $z \mapsto \overline{z}$ by an overbar. Note the real part! ⌟

SOLUTION of (9-8.c):

From (9.8.7) and (9.8.4) with $u' := \overline{u}$ we directly get ($\mathrm{Re}\{\imath z\} = -\operatorname{Im} z$)

$$\begin{aligned}\frac{d}{dt}\int_\Omega |u|^2 \,\mathrm{d}\boldsymbol{x} &= -2\operatorname{Im}\left\{\int_\Omega \imath \dot{u}\overline{u}\,\mathrm{d}\boldsymbol{x}\right\} - 2\operatorname{Im}\left\{\int_\Omega \mathbf{grad}\, u\cdot\mathbf{grad}\,\overline{u} + |u|^2 u\overline{u}\,\mathrm{d}\boldsymbol{x}\right\}\\ &= -2\operatorname{Im}\left\{\int_\Omega \|\mathbf{grad}\, u\|^2 + |u|^4\,\mathrm{d}\boldsymbol{x}\right\} = 0\,.\end{aligned}$$

Remark. Following Sub-problem (9-8.a) we can also split into real and imaginary part and then apply the product rule for the differentiation of bilinear expressions, see [Lecture → Rem. 9.2.3.7],

$$\frac{d}{dt}\int_\Omega |u|^2\,\mathrm{d}\boldsymbol{x} = \frac{d}{dt}\int_\Omega v^2 + w^2\,\mathrm{d}\boldsymbol{x} = 2\int_\Omega \dot{v}v\,\mathrm{d}\boldsymbol{x} + 2\int_\Omega \dot{w}w\,\mathrm{d}\boldsymbol{x}\,.$$

Since (9.8.5) holds $\forall w', v'$, by taking $w' = v$ and $v' = w$ and subtracting twice the second equation of (9.8.5) from twice the first equation of (9.8.5), we get $2\int_\Omega \dot{v}v\,\mathrm{d}\boldsymbol{x} + 2\int_\Omega \dot{w}w\,\mathrm{d}\boldsymbol{x} = 0$.

SOLUTION of (9-8.d):

We directly differentiate the Hamiltonian using the product rule (9.8.7) and the chain rule:

$$\begin{aligned}\frac{d}{dt}H(u) &= \frac{d}{dt}\int_\Omega \frac{1}{2}\,\mathbf{grad}\,u\cdot\mathbf{grad}\,\overline{u}+\frac{1}{4}(u\overline{u})^2\,\mathrm{d}\boldsymbol{x}\\ &= \int_\Omega \frac{1}{2}\frac{\partial}{\partial t}\big(\mathbf{grad}\,u\cdot\mathbf{grad}\,\overline{u}\big)+\tfrac{1}{2}|u|^2\frac{\partial}{\partial t}(u\overline{u})\,\mathrm{d}\boldsymbol{x}\\ &= \mathrm{Re}\left\{\int_\Omega \mathbf{grad}\,u\cdot\mathbf{grad}\,\frac{\partial\overline{u}}{\partial t}+|u|^2u\frac{\partial\overline{u}}{\partial t}\,\mathrm{d}\boldsymbol{x}\right\}\\ &\overset{(*)}{=} \mathrm{Re}\left\{\int_\Omega \boldsymbol{\imath}\frac{\partial u}{\partial t}\frac{\partial\overline{u}}{\partial t}\,\mathrm{d}\boldsymbol{x}\right\}=\mathrm{Re}\left\{\int_\Omega \boldsymbol{\imath}|\dot{u}|^2\,\mathrm{d}\boldsymbol{x}\right\}=0\,.\end{aligned}$$

In the final step $(*)$ we used the variational equation (9.8.4) with $u'=\dot{\overline{u}}$.

Remark. Of course, we can also rely on the splitting into real and imaginary part introduced in Sub-problem (9-8.a) and rewrite $H(u)$ using $u=v+\boldsymbol{\imath}w$:

$$H(u) := \frac{1}{2}\int_\Omega \|\mathbf{grad}\,v\|^2+\|\mathbf{grad}\,w\|^2\,\mathrm{d}\boldsymbol{x}+\frac{1}{4}\int_\Omega\big(|v|^2+|w|^2\big)^2\,\mathrm{d}\boldsymbol{x}\,.$$

Let us now take the time derivative of $H(u)$ using the chain rule for the non-linear terms:

$$\frac{d}{dt}H(u)=\int_\Omega \mathbf{grad}\,v\cdot\mathbf{grad}\,\dot{v}+\mathbf{grad}\,w\cdot\mathbf{grad}\,\dot{w}\,\mathrm{d}\boldsymbol{x}+\int_\Omega (v^2+w^2)(v\dot{v}+w\dot{w})\,\mathrm{d}\boldsymbol{x}\,. \tag{9.8.9}$$

Choose $w'=\dot{w}$ and $v'=\dot{w}$ as test functions in (9.8.5). Then sum the first and second equation of (9.8.5): The right hand side becomes the time derivative (9.8.9), the left hand side is 0.

SOLUTION of (9-8.e):

It is clear that the matrix $\mathbf{A}$ will arise from the Galerkin finite-element discretization of $-\Delta$ ("stiffness matrix"), whereas $\mathbf{M}$ is related to the Galerkin matrix for the inner product of $L^2(\Omega)$ ("mass matrix") multiplied with $\imath$. It is important to note that the tent basis functions b_h^ℓ are real-valued.

Thus, without using numerical quadrature, we would obtain the $N \times N$-matrices

$$\begin{aligned} (\mathbf{A})_{k,\ell} &= \int_\Omega \mathbf{grad}\, b_h^\ell(\boldsymbol{x}) \cdot \mathbf{grad}\, b_h^k(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \;, \\ (\mathbf{M})_{k,\ell} &= \imath \int_\Omega b_h^\ell(\boldsymbol{x})\, b_h^k(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \;, \end{aligned} \qquad k,l \in \{1,\ldots,N\} \;. \tag{9.8.11}$$

Since $\mathbf{grad}\, b_h^k \cdot \mathbf{grad}\, b_h^\ell$ is a piecewise constant function and the composite trapezoidal rule is exact for piecewise linear integrands, numerical quadrature does not affect $\mathbf{A}$.

The situation is different in the case of $\mathbf{M}$. Employing the mass lumping policy, we approximate

$$\int_\Omega \varphi(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \approx \sum_{K\in\mathcal{M}} \tfrac{1}{3}|K| \sum_{\ell=1}^{3} \varphi(\boldsymbol{a}_K^\ell) = \sum_{\boldsymbol{p}\in\mathcal{N}(\mathcal{M})} \tfrac{1}{3}\nu_{\boldsymbol{p}}\varphi(\boldsymbol{p}) \;, \tag{9.8.12}$$

where $\boldsymbol{a}_K^1, \boldsymbol{a}_K^2, \boldsymbol{a}_K^3$ are the vertices of the triangle K and $\nu_{\boldsymbol{p}}$ is the sum of the areas of the triangles abutting the node $\boldsymbol{p} \in \mathcal{N}(\mathcal{M})$. Thanks to the cardinal basis property of $\mathfrak{B}$ we get

$$\int_\Omega b_h^\ell(\boldsymbol{x})\, b_h^k(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \begin{cases} \frac{1}{3}\nu_{\boldsymbol{p}} & \text{for} \quad \ell = k \quad \leftrightarrow \text{node } \boldsymbol{p}, \\ 0 & \text{for} \quad k \neq \ell \;, \end{cases}$$

because the product $\boldsymbol{s} \mapsto b_h^\ell(\boldsymbol{x}) b_h^k(\boldsymbol{x})$ vanishes in all nodes of the mesh, unless $k = \ell$. This implies that $\mathbf{M}$ is a *diagonal* matrix

$$(\mathbf{M})_{k,\ell} = \begin{cases} \frac{1}{3}\imath\nu_{\boldsymbol{p}} & \text{for} \quad \ell = k \quad \leftrightarrow \text{node } \boldsymbol{p}, \\ 0 & \text{for} \quad k \neq \ell \;, \end{cases} \qquad k,\ell \in \{1,\ldots,N\} \;. \tag{9.8.13}$$

HINT 1 for (9-8.f): Obviously f arises from the non-linear term on the right-hand side of (NLSE). Exploit the cardinal basis property of the tent functions with respect to the nodes of the mesh. ⌟

SOLUTION of (9-8.f):

We elaborate the solution for the complex formulation. For (9.8.10) we write

$$\mathbf{M}\dot{\vec{\mu}} = \mathbf{A}\vec{\mu} + \mathbf{f}(\vec{\mu}) \ .$$

From the variational formulation

$$\int_\Omega \imath\dot{u}\, u' \,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, u' + |u|^2 u u' \,\mathrm{d}\boldsymbol{x} \quad \forall u' \in H^1(\Omega) \ , \tag{9.8.4}$$

and the fact that the component equations in (9.8.10) arise from testing with individual basis functions b_h^j, we deduce that the j-ht component of $\vec{\psi}(\vec{\mu}) \in \mathbb{C}^N$ is defined as

$$\psi_j(\vec{\mu}) := \int_\Omega \left(\left| \sum_{i=1}^N \mu_i b_h^i(\boldsymbol{x}) \right|^2 \sum_{i=1}^N \mu_i b_h^i(\boldsymbol{x}) \right) b_h^j(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \tag{9.8.14}$$

given the time-dependent basis expansion coefficient vector $\vec{\mu} = [\mu_1, \ldots, \mu_N]^\top \in \mathbb{C}^N$ and tent basis functions $b_h^i \in \mathcal{S}^0_{1,0}(\mathcal{M})$, $i = 1, \ldots, N$. Recall that a single basis function is associated to every node of the mesh.

Next, recall the composite trapezoidal rule (9.8.12) and the cardinal basis property of the tent function basis $\mathfrak{B}$ with respect to $\mathcal{N}(\mathcal{M})$. Thus (9.8.14) collapses to

$$\varphi_j(\vec{\mu}) = \tfrac{1}{3}\nu_{\boldsymbol{p}} |\mu_{\boldsymbol{p}}|^2 \mu_{\boldsymbol{p}} \ ,$$

with $\mu_{\boldsymbol{p}}$ that component of the basis expansion coefficient vector belonging to the node $\boldsymbol{p}$. This leads to

$$f_j(\xi) = \tfrac{1}{3}\nu_{\boldsymbol{p}} |\xi|^2 \xi \ , \quad \xi \in \mathbb{C} \ , \tag{9.8.15}$$

where $\boldsymbol{p}$ is the node with index j.

SOLUTION of (9-8.g):

From Sub-problem (9-8.e), (9.8.13) we know that $\mathbf{M} = \imath\mathbf{D}$ with a *symmetric positive definite* diagonal matrix $\mathbf{D} \in \mathbb{R}^{N,N}$. Combined with (9.8.15) we deduce the following structure of (9.8.10)

$$\imath\mathbf{D}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) = \mathbf{A}\vec{\boldsymbol{\mu}}(t) + \left[\alpha_j|\mu_j(t)|^2\mu_j(t)\right]_{j=1}^{N}, \tag{9.8.16}$$

with $\alpha_j > 0$. Left-multiply (9.8.16) with $\vec{\boldsymbol{\mu}}^{\mathrm{H}}$ and use that $\mathbf{D}$ is symmetric:

$$\begin{aligned}\imath\frac{d}{dt}\vec{\boldsymbol{\mu}}^{\mathrm{H}}(t)\mathbf{D}\vec{\boldsymbol{\mu}}(t) &= -2\,\mathrm{Im}\left\{\imath\vec{\boldsymbol{\mu}}^{\mathrm{H}}(t)\mathbf{D}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t)\right\}\\ &= -2\,\mathrm{Im}\underbrace{\left\{\vec{\boldsymbol{\mu}}^{\mathrm{H}}(t)\mathbf{A}\vec{\boldsymbol{\mu}}(t) + \sum_{k=1}^{N}|\mu_k(t)|^4\right\}}_{\in\mathbb{R}} = 0\,.\end{aligned}$$

HINT 1 for (9-8.h): Take into account that the matrix $\mathbf{M}$ is *diagonal* and the special form of $\vec{\boldsymbol{\mu}} \mapsto \mathbf{f}(\vec{\boldsymbol{\mu}})$, which implies that (9.8.20) boils down to N independent initial value problems in $\mathbb{C}$. ⌟

HINT 2 for (9-8.h):

It is advisable to view the initial value problem for each component as an initial value problem in $\mathbb{R}^2$ by splitting into real and imaginary part.

You have to find the solution for the initial value problem

$$\dot{y}_1 = (y_1^2 + y_2^2)y_2\,, \quad \dot{y}_2 = -(y_1^2 + y_2^2)y_1\,, \quad y_1(0) = u_1\,, \quad y_2(0) = u_2\,.$$

When visualized in the plane, this describes a rotation with angular speed $u_1^2 + u_2^2$. ⌟

SOLUTION of (9-8.h):

Using (9.8.13) and (9.8.15) and canceling the factor $\frac{1}{3}\nu_p$ the ODE from (9.8.20) amounts to N *identical* ODEs of the form

$$\imath\dot{\xi}_j(t) = |\xi_j(t)|^2\xi(t)\,, \quad \xi_j(t) \in \mathbb{C}\,. \tag{9.8.21}$$

Let us convert this into an ODE in $\mathbb{R}^2$ by splitting into real and imaginary part: $\xi_j(t) = y_1(t) + \imath y_2(t)$. We end up with

$$\begin{aligned} (9.8.21) \quad \Rightarrow \quad & \dot{y}_1 = (y_1^2 + y_2^2)y_2\,, \quad \dot{y}_2 = -(y_1^2 + y_2^2)y_1\,, \\ & y_1(0) = \operatorname{Re}\xi_0\,, \quad y_2(0) = \operatorname{Im}\xi_0\,. \end{aligned} \tag{9.8.22}$$

Writing this in more compact matrix-vector form is illuminating

$$(9.8.22) \quad \Leftrightarrow \quad \dot{\mathbf{y}} = \|\mathbf{y}\|^2 \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \mathbf{y}\,. \tag{9.8.23}$$

Recall that the ODE

$$\dot{\mathbf{w}} = \begin{bmatrix} 0 & \gamma \\ -\gamma & 0 \end{bmatrix} \mathbf{w}\,,$$

$\gamma \in \mathbb{R}$, describes a planar rotation,

$$\mathbf{w}(t) = \begin{bmatrix} \cos(\gamma t) & \sin(\gamma t) \\ -\sin(\gamma t) & \cos(\gamma t) \end{bmatrix} \mathbf{w}(0)$$

which leaves the length of the vector $\mathbf{w}(t)$ invariant. Hence, the factor $\|\mathbf{y}(t)\|^2$ in (9.8.23) is actually a constant and we find as solution of (9.8.23)

$$\mathbf{y}(t) = \begin{bmatrix} \cos(\gamma t) & \sin(\gamma t) \\ -\sin(\gamma t) & \cos(\gamma t) \end{bmatrix} \begin{bmatrix} \operatorname{Re}\xi_0 \\ \operatorname{Im}\xi_0 \end{bmatrix}\,, \quad \gamma := |\xi_0|^2\,. \tag{9.8.24}$$

A rotation in $\mathbb{R}^2$ corresponds to multiplication with $\exp(\imath\gamma)$, $\gamma \in \mathbb{R}$ the rotation angle (counterclockwise!), in $\mathbb{C}$. This permits us to rewrite (9.8.24) in terms of $\xi_j(t) \in \mathbb{C}$ as

$$\xi_j(t) = \exp(-\imath|\xi_0|^2 t)\xi_0\,, \quad \xi_0 := \left(\vec{\boldsymbol{\xi}}_0\right)_j\,, \quad t \in \mathbb{R}\,. \tag{9.8.25}$$

Remark. We see that (9.8.19) effects a phase shift.

SOLUTION of (9-8.i):

Once you have derived the formula (9.8.25), the implementation is straightforward.

C++ code 9.8.26: Implementation of class InteractionPropagator ⇢ GitLab

```
2  InteractionPropagator::InteractionPropagator(double tau) {
3    // We know that the discrete evolution operator for the non-linear
       part of the
4    // method-of-lines ODE boils down to componentwise multiplication of
       the
5    // vector with a single phase shift.  Thus, this phase shift can be
6    // precomputed, here by means of a lambda function
7    phase_multiplier_ = [tau](std::complex<double> z) {
8      const std::complex<double> i(0, 1);
9      return std::exp(-i * tau * std::norm(z)) * z;
10   };
11 }
12
13 Eigen::VectorXcd InteractionPropagator::operator()(
14     const Eigen::VectorXcd &mu) const {
15   // Eigen's way of applying a function to all components of a vector.
16   return mu.unaryExpr(phase_multiplier_);
17 }
```

SOLUTION of (9-9.a):

We follow the policy explained in [Lecture → Section 9.2.2]. Let $v \in C_c^\infty(\overline{\Omega})$ be a test function. Integrating over Ω yields

$$\begin{aligned}\int_\Omega \frac{\partial u}{\partial t}(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} &= \int_\Omega \mathrm{div}(\, c(\boldsymbol{x},t)\ \mathbf{grad}\ u(\boldsymbol{x},t))\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega \lambda\, (1-\frac{u(\boldsymbol{x},t)}{K(\boldsymbol{x},t)})\, u(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \\ &= -\int_\Omega c(\boldsymbol{x},t)\ \mathbf{grad}\ u(\boldsymbol{x},t)\cdot \mathbf{grad}\ v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} c(\boldsymbol{x},t)\ \mathbf{grad}\ u(\boldsymbol{x},t)\cdot \boldsymbol{n}(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{dS}(\boldsymbol{x}) \\ &\quad + \int_\Omega \lambda\, (1-\frac{u(\boldsymbol{x},t)}{K(\boldsymbol{x},t)})\, u(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \\ &= -\int_\Omega c(\boldsymbol{x},t)\ \mathbf{grad}\ u(t,\boldsymbol{x})\cdot \mathbf{grad}\ v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega \lambda\, (1-\frac{u(\boldsymbol{x},t)}{K(\boldsymbol{x},t)})\, u(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\end{aligned}$$

where we used Green's first formula [Lecture → Thm. 1.5.2.7] and the boundary conditions (9.9.2). Choosing suitable function spaces leads to the following variational formulation:

We seek $[0,T] \ni t \to u(t) \in H^1(\Omega)$ satisfying

$$\begin{aligned}&\int_\Omega \dot{u}(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_\Omega c(\boldsymbol{x},t)\ \mathbf{grad}\ u(\boldsymbol{x},t)\cdot \mathbf{grad}\ v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \\ &\quad - \int_\Omega \lambda\, (1-\frac{u(\boldsymbol{x},t)}{K(\boldsymbol{x},t)})\, u(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = 0, \quad \forall v \in H^1(\Omega)\ ,\end{aligned} \tag{9.9.5}$$

with $u(\boldsymbol{x},0) = u_0(\boldsymbol{x})$ for $\boldsymbol{x} \in \Omega$.

Let us introduce bilinear functionals $\mathsf{m},\mathsf{a} : H^1(\Omega)\times H^1(\Omega) \to \mathbb{R}$ and the linear form $\ell : H^1(\Omega) \to \mathbb{R}$ satisfying for $u, v \in H^1(\Omega)$,

$$\begin{aligned}\mathsf{m}(u,v) &:= \int_\Omega u(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\ , \\ \mathsf{a}(u,v) &:= \int_\Omega c(\boldsymbol{x},t)\ \mathbf{grad}\ u(\boldsymbol{x})\cdot \mathbf{grad}\ v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\ ,\ , \quad u,v \in H^1(\Omega)\ . \\ \ell(v) &:= 0\end{aligned} \tag{9.9.6}$$

Moreover, for the nonlinear term we introduce $r : H^1(\Omega)\times H^1(\Omega) \to \mathbb{R}$ defined as

$$r(u(t);v) := \int_\Omega \lambda\, (1-\frac{u(t,\boldsymbol{x})}{K(\boldsymbol{x},t)})\, u(\boldsymbol{x},t)\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\ . \tag{9.9.7}$$

With these notations, we arrive at the following formulation equivalent to (9.9.5): we seek $[0,T] \ni t \to u(t) \in H^1(\Omega)$ satisfying

$$\begin{cases}\mathsf{m}(\dot{u}(t),v) + \mathsf{a}(u(t),v) - r(u(t);v) = 0, & \forall v \in H^1(\Omega) \\ u(0) = u_0 & \in H^1(\Omega)\end{cases} \tag{9.9.8}$$

SOLUTION of (9-9.j):

C++ code 9.9.36: Member function `Evolution()` of StrangSplit StrangSplit ➙ GitLab

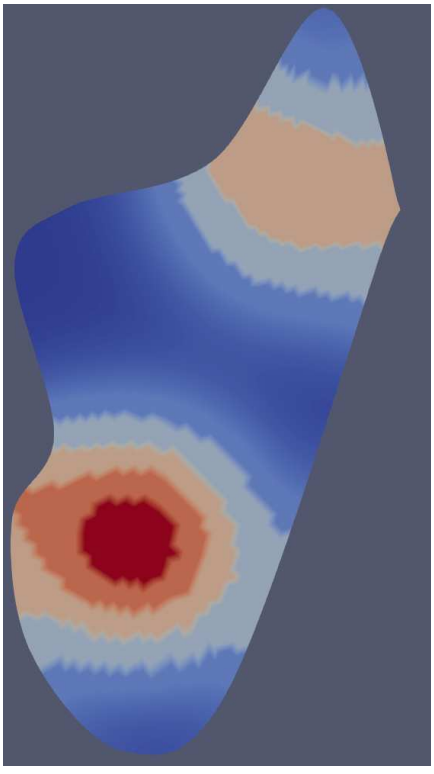
```
Eigen::VectorXd Evolution(const Eigen::VectorXd &cap,
                          const Eigen::VectorXd &mu) {
  // Obtain dofhandler
  const lf::assemble::DofHandler &dofh{fe_space_->LocGlobMap()};
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());

  Eigen::VectorXd sol(N_dofs);
  Eigen::VectorXd sol_cur(N_dofs);
  Eigen::VectorXd sol_next(N_dofs);

  Eigen::VectorXd ones = Eigen::VectorXd::Ones(N_dofs);

  // step size
  double tau = T_ / m_;

  /* Strang Splitting Method
   * First half time step [0, tau/2]: Diffusion
   */
  sol_cur = diffusionEvolutionOperator(tau / 2., mu);

  // Loop through time steps
  for (int i = 1; i < m_ - 1; i++) {
    // Reaction for next full time step
    sol_next = cap.cwiseQuotient(ones + (cap.cwiseQuotient(sol_cur) - ones) *
                                        std::exp(-lambda_ * tau));
    sol_cur = sol_next;
    // Diffusion for next full time step
    sol_next = diffusionEvolutionOperator(tau, sol_cur);
    sol_cur = sol_next;
  }

  // Last half time step: Reaction
  sol_next = cap.cwiseQuotient(ones + (cap.cwiseQuotient(sol_cur) - ones) *
                                      std::exp(-lambda_ * tau / 2.));
  sol_cur = sol_next;
  sol = sol_cur;

  return sol;
}
```

SOLUTION of (9-9.k):

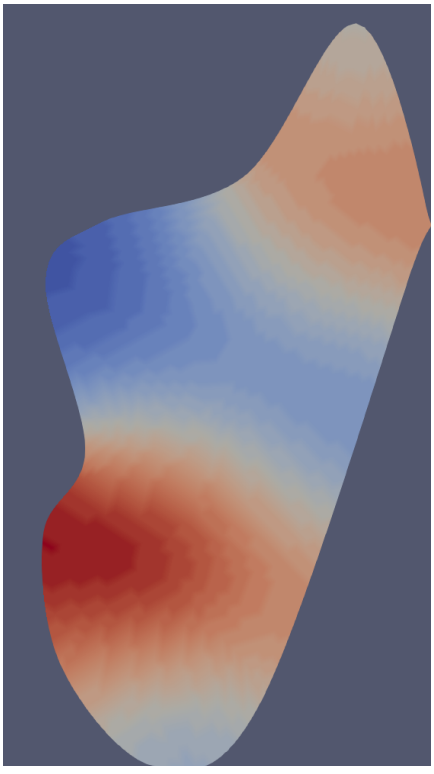
C++ code 9.9.37: Implementation of `modelproblem()` ⇢ GitLab

```
void modelproblem() {
  std::cout << "You are running the model problem, i.e. you solve the Fisher "
               "equation on the model domain. "
            << std::endl;
  // Obtain mesh
  auto mesh_factory = std::make_unique<lf::mesh::hybrid2d::MeshFactory>(2);
  const lf::io::GmshReader reader(std::move(mesh_factory), "meshes/island.msh");
  std::shared_ptr<const lf::mesh::Mesh> mesh_p = reader.mesh();
  // Finite Element Space
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh_p);
  // Dofhandler
  const lf::assemble::DofHandler &dofh{fe_space->LocGlobMap()};
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
  // Initial Population density
  Eigen::VectorXd u0(N_dofs);
  u0.setZero();
  u0(321) = 0.3;
  u0(567) = 0.3;

  // Diffusion Coefficient
  auto c = [](Eigen::Vector2d x) -> double { return 1.2; };
  double lambda = 2.1;  // Growth Factor
  // Carrying capacity
  Eigen::VectorXd K{0.8 * Eigen::VectorXd::Ones(N_dofs)};
  // Time Steps
  unsigned int m = 100;
  double T = 1.;

  // Compute the solution with method of lines and Strang splitting
  StrangSplit StrangSplitter(fe_space, T, m, lambda, c);
  // Five snapshots
  std::vector<Eigen::VectorXd> sol;
  std::cout << "Computing solution after 100 timesteps..." << std::endl;
  sol.push_back(StrangSplitter.Evolution(K, u0));
  // Uncomment the following calls to StrangSplitter.Evolution in order
  // to solve for a longer evolution.
  /*std::cout << "Computing solution after 200 timesteps..." << std::endl;
  sol.push_back(StrangSplitter.Evolution(K, sol[0]));
  std::cout << "Computing solution after 300 timesteps..." << std::endl;
  sol.push_back(StrangSplitter.Evolution(K, sol[1]));
  std::cout << "Computing solution after 400 timesteps..." << std::endl;
  sol.push_back(StrangSplitter.Evolution(K, sol[2]));
  std::cout << "Computing solution after 500 timesteps..." << std::endl;
  sol.push_back(StrangSplitter.Evolution(K, sol[3]));*/

  // Use VTK-Writer for Visualization of solution.
  std::cout << "Writting solution(s) in VTK format." << std::endl;
  std::cout << std::size(sol) << std::endl;
  for (int k = 0; k < std::size(sol); k++) {
    std::stringstream filename;
    filename << "model_problem_sol" << k + 1 << ".vtk";
```

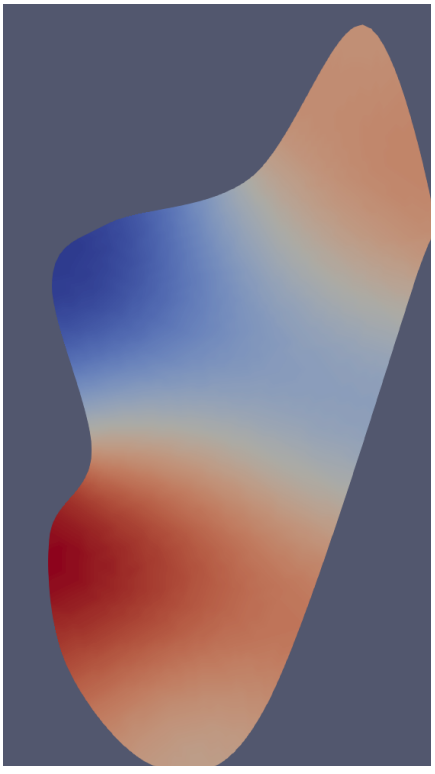
```
54      lf::io::VtkWriter vtk_writer(mesh_p, filename.str());
55      auto nodal_data = lf::mesh::utils::make_CodimMeshDataSet<double>(mesh_p, 2);
56      for (int global_idx = 0; global_idx < N_dofs; global_idx++) {
57        nodal_data->operator()(dofh.Entity(global_idx)) = sol[k][global_idx];
58      }
59      vtk_writer.WritePointData("sol", *nodal_data);
60    }
61    std::cout
62        << "Solution after i*100 timesteps written to model_problem_sol'i'.vtk"
63        << std::endl;
64  }
```



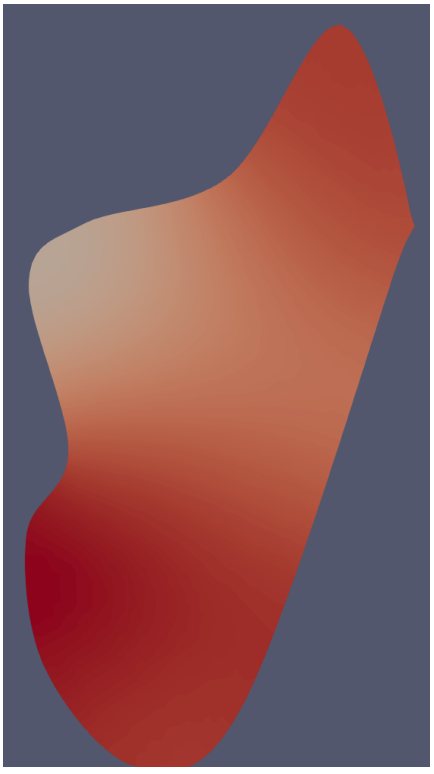
Solution after $M = 100$ time steps.



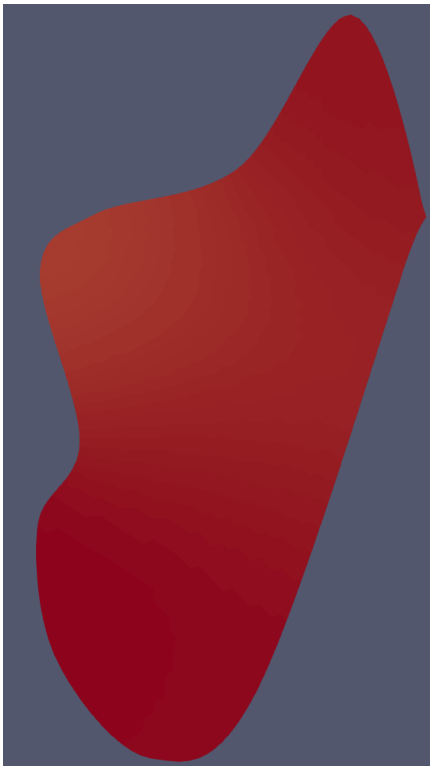
Solution after $M = 200$ time steps.



Solution after $M = 300$ time steps.



Solution after $M = 400$ time steps.



Solution after $M = 500$ time steps.

SOLUTION of (9-9.b):

The entries of $\mathbf{M}_{\hat{K}}$ and $\mathbf{A}_{\hat{K}}$ can be computed analytically, where we rely on [Lemma 2.7.5.5] in the lecture notes.

The local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ on a triangle are the barycentric coordinate functions $\lambda_l,\ l = 1,2,3$. Thus the entries of the element mass matrix on $\hat{K}$ are given by

$$\begin{aligned} \mathbf{M}_{\hat{K}} &:= [m(\lambda_j,\lambda_j)]_{i,j=1}^3 = [\int_\Omega \lambda_j(\boldsymbol{x},t)\lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,]_{i,j=1}^3 \\ \mathbf{A}_{\hat{K}} &:= [a(\lambda_j,\lambda_i)]_{i,j=1}^3 = [\int_\Omega c\ \mathbf{grad}\ \lambda_j(\boldsymbol{x},t)\cdot\mathbf{grad}\ \lambda_i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,]_{i,j=1}^3\,. \end{aligned} \tag{9.9.10}$$

Using [Lemma 2.7.5.5] for $d = 2$ yields

$$\mathbf{M}_{\hat{K}} = \frac{1}{24}\begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}, \tag{9.9.11}$$

as well as

$$\mathbf{A}_{\hat{K}} = \frac{c}{2}\begin{bmatrix} 2 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}. \tag{9.9.12}$$

SOLUTION of (9-9.c):

C++ code 9.9.13: Implementation of `assembleGalerkinMatrices` ⇢ GitLab

```cpp
template <typename DIFF_COEFF>
std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>>
assembleGalerkinMatrices(const lf::assemble::DofHandler &dofh, DIFF_COEFF &&c) {
  std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>> A_M;
  // Obtain mesh and finite element space
  std::shared_ptr<const lf::mesh::Mesh> mesh = dofh.Mesh();
  const lf::uscalfe::size_type N_dofs(dofh.NumDofs());
  auto fe_space =
      std::make_shared<lf::uscalfe::FeSpaceLagrangeO1<double>>(mesh);

  // Matrix format for assembly
  lf::assemble::COOMatrix<double> A_COO(N_dofs, N_dofs);
  lf::assemble::COOMatrix<double> M_COO(N_dofs, N_dofs);

  // Function handles for Stiffness and Mass matrix
  auto one = [](Eigen::Vector2d x) -> double { return 1.0; };
  auto zero = [](Eigen::Vector2d x) -> double { return 0.0; };

  // Mesh functions
  lf::mesh::utils::MeshFunctionGlobal mf_c{c};
  lf::mesh::utils::MeshFunctionGlobal mf_one{one};
  lf::mesh::utils::MeshFunctionGlobal mf_zero{zero};

  // Element matrix provider
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double, decltype(mf_c),
                                                      decltype(mf_zero)>
      elMat_Stiff(fe_space, mf_c, mf_zero);
  lf::uscalfe::ReactionDiffusionElementMatrixProvider<double, decltype(mf_zero),
                                                      decltype(mf_one)>
      elMat_Mass(fe_space, mf_zero, mf_one);

  // Assembly
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elMat_Stiff, A_COO);
  lf::assemble::AssembleMatrixLocally(0, dofh, dofh, elMat_Mass, M_COO);

  // Sparse matrix format
  Eigen::SparseMatrix<double> A = A_COO.makeSparse();
  Eigen::SparseMatrix<double> M = M_COO.makeSparse();

  A_M = std::make_pair(A, M);

  return A_M;
}
```

SOLUTION of (9-9.d):

Generically, the entries of $\vec{\boldsymbol{\rho}}$ can be computed as

$$(\vec{\boldsymbol{\rho}}(\vec{\boldsymbol{\mu}},t))_j = \int_\Omega \lambda\big(1-\frac{\sum_{\ell=1}^{N}\mu_\ell b_h^l(\boldsymbol{x})}{K(\boldsymbol{x},t)}\big)\,\big(\sum_{\ell=1}^{N}\mu_\ell b_h^\ell(\boldsymbol{x})\big)\,b_h^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,,\quad j=1,\ldots,N\,.$$

On the one hand, the tent-function basis is a cardinal basis with respect to the nodes of the mesh. On the other hand, the composite trapezoidal rule exclusively relies on evaluations in the nodes of the mesh. As a consequence, the j-th component of $\vec{\boldsymbol{\rho}}(\vec{\boldsymbol{\mu}},t)$ will depend only on the j-th component of $\vec{\boldsymbol{\mu}}$. More precisely, we obtain

$$(\vec{\boldsymbol{\rho}}(\vec{\boldsymbol{\mu}},t))_j = \omega_j\,\lambda\,\big(1-\frac{\mu_j}{K(\boldsymbol{p}_j,t)}\big)\mu_j\quad,\quad j=1,\ldots,N\,, \tag{9.9.15}$$

where $\boldsymbol{p}_j$ is the node $\in\mathcal{N}(\mathcal{M})$ carrying the tent function with index j, and ω_j the quadrature weight of the composite trapezoidal rule at that node:

$$\omega_j := \frac{1}{3}\sum_{\boldsymbol{p}\in\overline{K}}|K|\,.$$

SOLUTION of (9-9.e):

We specialize the considerations of [Lecture → § 9.2.7.6]: Following the Butcher Tableau, we compute $\vec{\mu}^{(k)}$ from $\vec{\mu}^{(k-1)}$ for $k \in \{1, ..., M\}$ by solving a linear system of equations [Lecture → (9.2.7.9)].

We seek $\vec{\kappa}_1, \vec{\kappa}_2 \in \mathbb{R}^N$ such that

$$\begin{aligned} \mathbf{M}\vec{\kappa}_1 + \tau\xi\mathbf{A}\vec{\kappa}_1 &= -\mathbf{A}\vec{\mu}^{(k-1)}, \\ \mathbf{M}\vec{\kappa}_2 + \tau(1-\xi)\mathbf{A}\vec{\kappa}_1 + \tau\xi\mathbf{A}\vec{\kappa}_2 &= -\mathbf{A}\vec{\mu}^{(k-1)}. \end{aligned} \tag{9.9.17}$$

Note that due to mass lumping, we obtain a diagonal matrix M when using the composite trapezoidal quadrature rule. Finally, we can recover the solution with

$$\vec{\mu}^{(k)} = \vec{\mu}^{(k-1)} + \tau(1-\xi)\vec{\kappa}_1 + \tau\xi\vec{\kappa}_2. \tag{9.9.18}$$

SOLUTION of (9-9.f):

We first divide (9.9.19) by $(1-\frac{u}{K})\,u$, where in the sequel we simply write K for $K(\boldsymbol{x},t)=K(\boldsymbol{x})$, $\forall(\boldsymbol{x},t)$ in $[T$ Ω. Then using differentials for the partial derivatives we obtain

$$\frac{1}{(1-\frac{u}{K})\,u}\frac{\mathrm{d}u}{\mathrm{d}t}=\lambda. \tag{9.9.20}$$

Equivalently

$$\frac{1-\frac{u}{K}+\frac{u}{K}}{(1-\frac{u}{K})\,u}\frac{\mathrm{d}u}{\mathrm{d}t}=(\frac{1}{u}+\frac{1}{K-u})\frac{\mathrm{d}u}{\mathrm{d}t}=\lambda. \tag{9.9.21}$$

Hence we obtain

$$(\frac{1}{u}+\frac{1}{K-u})\mathrm{d}u=\lambda\mathrm{d}t, \tag{9.9.22}$$

which yields after integrating

$$\ln(u)-\ln(K-u)=\lambda t+C_0, \tag{9.9.23}$$

with integration constant $C_0\in\mathbb{R}$. Thus we find

$$\frac{u}{K-u}=Ce^{\lambda t}, \tag{9.9.24}$$

with $C=e^{C_0}$. Finally, by properly rearranging we have

$$u=\frac{CKe^{\lambda t}}{1+Ce^{\lambda t}}. \tag{9.9.25}$$

We determine the constant C with the initial condition,

$$u=\frac{u_0Ke^{\lambda t}}{K+u_0(e^{\lambda t}-e^{\lambda T_0})}=\frac{K}{1+(Ku_0^{-1}-e^{\lambda T_0})e^{-\lambda t}}. \tag{9.9.26}$$

SOLUTION of (9-9.g):

Consider the following mappings for fixed $t \in [0,T]$:

$$\mathbf{\Phi}_f^t : \begin{cases} \mathbb{R}^N \to \mathbb{R}^N \\ \vec{\mu}_0 \to \vec{\mu}(t) \end{cases} \tag{9.9.27}$$

and

$$\mathbf{\Phi}_g^t : \begin{cases} \mathbb{R}^N \to \mathbb{R}^N \\ \vec{\mu}_0 \to \vec{\mu}(t) \end{cases} \tag{9.9.28}$$

corresponding to $\frac{d\vec{\mu}}{dt}(t) = f(t,\vec{\mu})$ and $\frac{d\vec{\mu}}{dt}(t) = g(t,\vec{\mu})$, respectively. These are well-defined mappings of the state space into itself, see . We can now define evolution operators for each part:

$$\mathbf{\Phi}_f : \begin{cases} [0,T] \times \mathbb{R}^N \to \mathbb{R}^N \\ (t,\vec{\mu}_0) \to \mathbf{\Phi}_f^t(\vec{\mu}_0) := \vec{\mu}(t) \end{cases} \tag{9.9.29}$$

$$\mathbf{\Phi}_g : \begin{cases} [0,T] \times \mathbb{R}^N \to \mathbb{R}^N \\ (t,\vec{\mu}_0) \to \mathbf{\Phi}_g^t(\vec{\mu}_0) := \vec{\mu}(t) \end{cases} \tag{9.9.30}$$

Then we find that $\frac{\partial \mathbf{\Phi}_f}{\partial t}(t,\vec{\mu}) = f(t,\mathbf{\Phi}_f^t(\vec{\mu}))$, as well as $\frac{\partial \mathbf{\Phi}_g}{\partial t}(t,\vec{\mu}) = g(t,\mathbf{\Phi}_g^t(\vec{\mu}))$. Discretizing the mappings (9.9.27) - (9.9.30) for our temporal mesh gives $\mathbf{\Psi}_f, \mathbf{\Psi}_g$.

In subtask Sub-problem (9-9.e), we have already derived the linear system of equations. We have found

$$\mathbf{\Psi}_f^\tau(\vec{\mu}^{(k-1)}) = \vec{\mu}^{(k)} = \vec{\mu}^{(k-1)} + \tau(1-\xi)\mathbf{k}_1 + \tau\xi\mathbf{k}_2. \tag{9.9.31}$$

For the reaction term, the exact solution is given in (9.9.26). For fixed $t \in [0,T]$, the mapping for the evolution operator reads

$$\text{For every component of } \vec{\mu}: \quad \mathbf{\Phi}_g^t(\vec{\mu}) = \frac{K}{1 + (K\vec{\mu}_0^{-1} - 1)e^{-\lambda t}}. \tag{9.9.32}$$

SOLUTION of (9-9.h):

The Galerkin matrices $\mathbf{A}$ and $\mathbf{M}$ must be kept as data members of the class **StrangSplit** and have to be built in the constructor for the sake of efficiency. Assembling them in every timestep is a waste of computational resources.

C++ code 9.9.33: Data members of StrangSplit. ➙ GitLab

```
2  private:
3   // Finite Element Space
4   const std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space_;
5   // Final Time
6   double T_;
7   // Number of Timesteps
8   unsigned int m_;
9   // Growth Factor
10  double lambda_;
11  // coefficient for SDIRK-2 Butcher Tableau
12  double xi_;
13  // Galerkin matrix corresponding to the negative Laplacian with Robin Boundary
14  // Conditions */
15  Eigen::SparseMatrix<double> A_;
16  // Galerkin matrix for the Mass Matrix
17  Eigen::SparseMatrix<double> M_;
18  // Precompute LU decomposition needed for time stepping
19  Eigen::SparseLU<Eigen::SparseMatrix<double>> solver;
```

C++ code 9.9.34: Implementation of the constructor of StrangSplit ➙ GitLab

```
2  template <typename DIFF_COEFF>
3  StrangSplit::StrangSplit(
4      const std::shared_ptr<lf::uscalfe::UniformScalarFESpace<double>> fe_space,
5      double T, unsigned m, double lambda, DIFF_COEFF &&c)
6      : fe_space_(fe_space),
7        T_(T),
8        m_(m),
9        lambda_(lambda),
10       xi_(1.0 - 0.5 * sqrt(2.0)) {
11   const lf::assemble::DofHandler &dofh{fe_space_->LocGlobMap()};
12   std::pair<Eigen::SparseMatrix<double>, Eigen::SparseMatrix<double>>
13       galerkinpair = assembleGalerkinMatrices(dofh, c);
14   A_ = galerkinpair.first;
15   M_ = galerkinpair.second;
16   // Butcher Tableau Coefficient for SDIRK-2
17
18 }
```

SOLUTION of (9-9.i):

C++ code 9.9.35: Member function `diffusionEvolutionOperator()` of StrangSplit ↪ GitLab

```
2   Eigen::VectorXd diffusionEvolutionOperator(double tau,
3                                              const Eigen::VectorXd &mu) {
4     Eigen::VectorXd evol_op;
5
6     // sparse LU decomposition: done in every timestep because timestep size may
7     // vary.
8     solver.compute(M_ + tau * xi_ * A_);
9     LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
10    Eigen::VectorXd rhs = -A_ * mu;
11
12    // First stage SDIRK-2
13    Eigen::VectorXd k1 = solver.solve(rhs);
14    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
15    // Second stage SDIRK-2
16    Eigen::VectorXd k2 = solver.solve(rhs - tau * (1 - xi_) * A_ * k1);
17    LF_VERIFY_MSG(solver.info() == Eigen::Success, "LU decomposition failed");
18    // Recover solution
19    evol_op = mu + tau * (1 - xi_) * k1 + tau * xi_ * k2;
20    return evol_op;
21  }
```