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

Numerical Methods for Partial Differential Equations

Prof. R. Hiptmair, SAM, ETH Zurich

(with contributions from Prof. Ch. Schwab, Prof. H. Harbrecht,
Dr. V. Gradinaru, Dr. A. Chernov, and Prof. P. Grohs)

Spring term 2016

URL: http://www.sam.math.ethz.ch/~hiptmair/tmp/NPDE/NPDE16.pdf

Always under construction!

SVN revision 87495

The online version will always be work in progress and subject to change.

(Nevertheless, structure and main contents can be expected to be stable)

Do not print!

Lecture homepages: D-MATH course page and entry in ETH Course Directory

Links to lecture recordings and tablet notes (operational from Feb 20, 2016)

Contents

Chapter 0

Introduction

Contents

0.1 Prerequisites

This course builds upon the ETH lecture 401-0663-00L Numerical Methods for CSE, see [8]. In particular, familiarity with the following topics from computational mathematics is taken for granted:

- Techniques for handling sparse matrices and sparse linear systems, see [8, Section 1.7].
- Numerical quadrature, concepts and methods as introduced in [8, Chapter 5].
- Numerical method for solving initial value problems for ordinary differential equations (numerical integration), in particular stiff initial value problems as discussed in [8, Chapter 12].

Lecture notes for the course "Numerical Methods for CSE are available" for download here.

Of course, a solid knowledge of calculus and linear algebra is also important. The following texts may be used for reference and self-study:

- Calculus: M. STRUWE, *Analysis für Informatiker*. Lecture notes, ETH Zürich, 2009, link.
- K. NIPP AND D. STOFFER, *Lineare Algebra*, vdf Hochschulverlag, Zürich, 5th ed., 2002.

0.2 Goals

This lecture is a core course for

- BSc in Computational Science and Engineering (RW/CSE),
- BSC in Computer Science with focus Computational Science.

Main *skills* to be acquired in this course:

✦ Ability to *implement* advanced numerical methods for the solution of partial differential equations in C++ efficiently (, based on numerical libraries, of course).

✦ Ability to *modify* and *adapt* numerical algorithms guided by awareness of their mathematical foundations

✦ Ability to *select* and *assess* numerical methods in light of the predictions of theory

✦ Ability to *identify features* of a PDE (= partial differential equation) based model that are relevant for the selection and performance of a numerical algorithm

✦ Ability to *understand research publications* on theoretical and practical aspects of numerical methods for partial differential equations.

Distinction from other courses

This course	$\neq$	Numerical analysis of PDE ($\rightarrow$ mathematics curriculum) (401-3651-00V *Numerical methods for elliptic and parabolic partial differential equations*,)
		Instruction on how to apply software packages

0.3 Course History

Precursor courses of the current lecture are the following:

- Summer semester 04, R. Hiptmair (for RW/CSE undergraduates)
- Winter semester 04/05, C. Schwab (for RW/CSE undergraduates)
- Winter semester 05/06, H. Harbrecht (for RW/CSE undergraduates)

- Winter semester 06/07, C. Schwab (for BSc RW/CSE)
- Autumn semester 07, A. Chernov (for BSc RW/CSE)
- Autumn semester 08, C. Schwab (for BSc RW/CSE)
- Autumn semester 09, V. Gradinaru (for BSc RW/CSE, Subversion Revision: 22844)
- Spring semester 10, R. Hiptmair (for BSc Computer Science)
- Autumn semester 10, R. Hiptmair (for BSc RW/CSE, Subversion Revision: 30025)
- Autumn semester 11, P. Grohs (for for BSc RW/CSE, Subversion Revision: 39100)
- Spring semester 12, R. Hiptmair (for RW/CSE & BSc Computer Science)
- Spring semester 13, R. Hiptmair (for BSc RW/CSE & BSc Computer Science)
- Spring semester 15, R. Hiptmair (for BSc RW/CSE & BSc Computer Science)
- Spring semester 16, R. Hiptmair (for BSc RW/CSE & BSc Computer Science)

Up to 2013, implementation of finite element method was discussed using a MATLAB library called "LehrFEM" [4]. From 2015 a C++ finite element programming environment based on the DUNE interface standard is used, first it was a code developed as part of the DUNE project, now the BETL code developed at the Seminar for Applied Mathematics of ETH Zürich.

0.4 Reading Instructions for Lecture Notes

These lecture notes have not been written as a self-contained textbook.
They are meant to be supplemented by explanations given in class.

Some pieces of advice:

- this material is dense and concise to complement explanations given in class
- this document is not meant for mere reading, but for working with,
- turn pages all the time and follow the numerous cross-references,
- study the relevant section of the course material when doing homework problems.
- these lecture notes come with review questions and quizzes for immediate testing of understanding after first thorough reading.

0.5 Characteristics of the Course

0.5.1 Level

- The course is difficult and demanding (*ie.* ETH level)
- Do **not** expect to understand everything in class. The average student will
  - understand about one third of the material when attending the lectures,

- understand another third when making a *serious effort* to solve the homework problems,
- hopefully understand the remaining third when studying for the examnination after the end of the course.

Perseverance will be rewarded!

0.5.2 Teaching Model

The bulk of the material will be presented in a traditional classroom setting using a tablet. The tablet notes will be made available as PDF documents shortly after each lecture. Most complex considerations will be elaborated in handwriting, while theorems, definitions, some long formulas and numerical results may just be pasted from this lecture document. Taking notes during class is not essential, but it may help some students to stay focused.

Some parts of the lecture will be covered using a flipped classroom model: students will be asked to prepare some topics based on this lecture document and, occasionally, short video tutorials. The topics will again be discussed in class, in the form of a questions and answers session, however.

0.5.3 Homework assignments

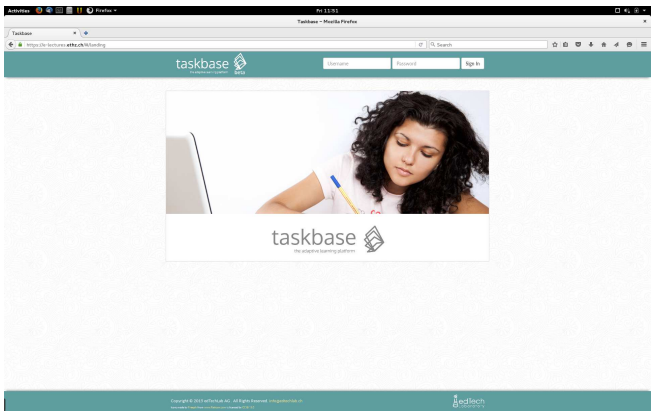
A problem sheet will be published every week comprising a number of homework problems, most of them involving both implementation and theoretical parts.

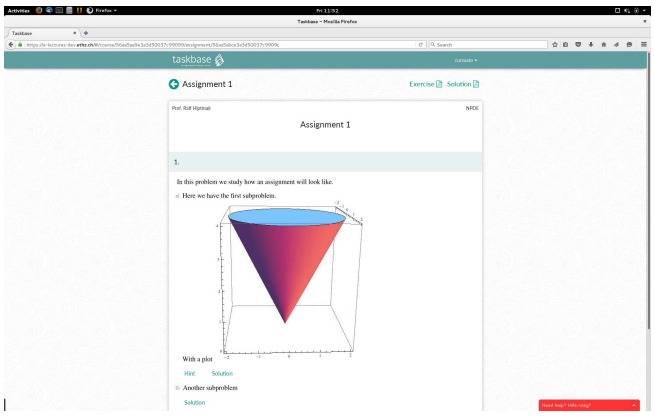
Some homework problems will be labelled core problems and we strongly recommend that an earnest attempt is made to solve them. We expect the average student to take *3-4* hours to solve the core problems completely. Of course, students are also encouraged to tackle the remaining non-core problems, time permitting.

The problems are published on the TASKBASE online platform, together with plenty of hints. A master solution will be made available shortly after a problem sheet has been released, but it is foolish to read the master solution parallel to working on a problem sheet, because *trying to find the solution on one's own is essential for developing problem solving skills*, though it may occasionally be frustrating.

Assignments will now be incorporated into TASKBASE "the adaptive learning platform". Students registered in the lecture can log in with their **nethz** username and password.

Taskbase login screen ▷





Once you logged in, you will arrive to a homepage including an icon for each one of your lectures using taskbase.

You can also explore the assignment online.

◁ online display of an assignment

Click on Exercise to download the assignment as a pdf file. Click on Solution to get it with solutions. You may click the Hint button to see the hints, if some are provided.

online display of hint for subproblem 1.a) ▷

You can click the Solution button to display the proposed It also allows you to give us feedback for each subproblem.

◁ online display of solution for subproblem 1.a)

Note: Homework problems can be *handed in*, if you want to receive feedback about your solution. Please write your name on your homework paper clearly and hand it to one of the assistants in the study center or deposit it in the tray in the aisle HG G 53-54.3 marked "Numerical Methods for Partial Differential Equations". One week later the corrections will usually be returned.

0.5.4 Study center

The tutorials for this course will be conducted in the format of *study groups* in the ETH "flexible lecture hall" (study center) HG E 41 ▷.

Several assistants will be present to explain and discuss homework problems both from the previous and the current problem sheet.

The study center session is also a good opportunity to do the homework in a group. In case you are stalled you may call an assistant and ask for advice.

Fig. 1

0.6 Practical Information

Course recordings: CMS link

Course: 401-0674-00L Numerical Methods for Partial Differential Equations

Lectures:	Mo	15-17	HG F 1
	Tu	15-17	HG F 1
Tutorials:	Mo	17-19	HG E 41

Lecturer: **Prof. Ralf Hiptmair**, office: HG G 58.2, e-mail: hiptmair@sam.math.ethz.ch

Assistants: **Roger Käppeli**, office: HG G 52.1,
e-mail: roger.kaeppeli@sam.math.ethz.ch
(Senior Scientist at Seminar of Applied Mathematics)

Kjetil Lye, office: HG G 56.1,
e-mail: kjetil.lye@sam.math.ethz.ch
(2nd year PhD student at Seminar of Applied Mathematics)

Laura Scarabosio, office: HG G 54.1,
e-mail: laura.scarabosio@sam.math.ethz.ch
(4th year PhD student at Seminar of Applied Mathematics)

Elke Spindler, office: HG G 53.1,
e-mail: elke.spindler@sam.math.ethz.ch
(4th year PhD student at Seminar of Applied Mathematics)

Carolina Urzua Torres, office: HG G 53.1,
e-mail: carolina.urzua@sam.math.ethz.ch
(2nd year PhD student at Seminar of Applied Mathematics)

Teaching assistants:	Dominik Borer	borerdo@student.ethz.ch
	Nicolas Ochsner	ochsnern@student.ethz.ch
	Alexander Xandeep	xandeepv@student.ethz.ch

Office hours:

- Prof. Ralf Hiptmair, Monday, 17:15-17:45, HG G 58.2

Assignments:

- 11 weekly assignment sheets, made available for download on Monday. Due on Monday one week later: to be deposited in the labeled box at HG G 53.x.
- "Testat" requirement: **NONE**
- Submit your C++ solutions via the online submission interface http://www.math.ethz.ch/ grsam/submit/ (choose course n.5)
- Exercises are marked either as core or non-core problems. Core problems (about 2 per sheet) are supposed to be essential for following the course.
- Correction of homework problems will be done on request for at most two problems per sheet.

Examinations:

- Mid-term quick assessment (∗): **April 12, 2016**, 15:15 - 15:45
- End-term quick assessment (∗): **May 24, 2016**, 15:15 - 15:45
- "Sessionsprüfung": Computer based examination involving coding problems beside theoretical questions. Parts of the lecture slides, C++ documents, and Eigen documentation will be made available during the examination.

3 hour examination

(∗) The mid-term and end-term last 30 minutes, closed book. Achieving 100% of the points in a term exam will yield a BONUS of 10% of the points for the session exam. Passing of either term exam is not required for admission to the session exam. Repetition of term exams is not possible.

Web page: http://www.math.ethz.ch/education/bachelor/lectures/fs2016/other/n_dgl

0.7 Course Wiki

A course wiki can be accessed through

`http://npdeeth.wikispaces.com/`

It serves two purposes

1. **Reporting errors**: Please supply the following information:
   - (sub)section where the error has been found,
   - precise location (e.g, after Equation (4), Thm. 2.3.3, etc.). Refrain from giving page numbers,
   - brief description of the error.
2. **Online discussion**: The wiki has been set up so that you can post questions on the programming exercises that accompany the course. One of the assistants will look at the entries and
   - write an answer in this forum or
   - discuss the question in a consulting session and post an answer later.

   A second purpose of online discussion is that the assistants can collect FAQs and post answers here.

0.8 Credits

- To Thomas Häner and Benjamin Ulmer, MSc students of CSE, for setting up the DUNE based environment used in this course.
- To Baranidharan Mohan, MSc student of CSE, for preparing the text for Section 3.6.1.
- to Federico Danieli, MSc students of CSE, for preparing BETL based finite element demonstration codes for scalar linear elliptic boundary value problems in 2D.

0.9 Implementation

Algorithmic aspect of numerical methods will bulk large in this course. Thus, code samples will even be discussed in class, and homework assignments will involve substantial coding parts.

(0.9.1) Programming language

Programming language

This course will entirely rely on the programming language C++ using the latest standard C++ 11.

➨ For information about C++ 11 and further references please consult [8, Section 0.2].

(0.9.3) Tools for numerical linear algebra

To handle matrices and vectors we will rely on the template library EIGEN, which offers very efficient high-level operations from linear algebra; from the EIGEN home page:

Eigen is a C++ template library for linear algebra:
matrices, vectors, numerical solvers, and related algorithms.

For an introduction to EIGEN refer to [8, Section 1.2.3] and the wealth of online resources available for this widely used library.

EIGEN is an open source community code development project and you are also invited to contribute.

(0.9.4) Visualization tools

A big advantage of MATLAB is the ease with which computational results can be visualized using high level plotting functions. In C++ a similar ease can be achieved by the use of suitable libraries. For this course we opted for MATHGL, a library for creating high-quality graphics in C++.

(0.9.5) Build tools

The use of large template libraries entails complex build procedures, because the compiler and linker have to be informed about the location of numerous files and libraries scattered over many directories. Thus build tools become indispensable and in this course we use the

psframebox[style=melframe]**cmake** ⟶ psframebox[style=melframe]**make**

build chain:

cmake

make

(0.9.6) Git/Gitlab source repository

- Short tutorial for Git with many further links.
- Link to an introduction to Gitlab bu Til Ehrengruber.
- Course Git repository:

  `git@gitlab.math.ethz.ch:NumPDE/NumPDE.git`

  See the instructions on how to clone the repository and for information about its structure.

0.10 Mathematical Modelling with Partial Differential Equations

(0.10.1) Continuum models

Partial differential equations (PDEs) are at the core of most mathematical models arising from a continuum approach, where the configuration or state of a system is described by means of a function on a (multi-dimensional) domain $\Omega \subset \mathbb{R}^d$, $d \geq 1$. In fact, in one dimension, for $d = 1$, the models will involve ordinary differential equations (ODEs) rather than partial differential equations. Yet, in many cases the dimension d can be regarded as a parameter for a family of models and the case $d = 1$ is not really special and shares many traits with models for the genuinely multi-dimensional setting $d > 1$.

Therefore, the title of the course is a slight misnomer; a more appropriate title would be

Numerical Methods for Continuum Models with Local Interactions

but who would find this excitiing?

Next, notations used for stating partial differential equations will be explained. Then a few examples of mathematical models based on PDEs will be presented in a cursory way, in order to convey their diversity and wide scope.

0.10.1 PDEs: Basic Notions

(0.10.2) Formal notion of a partial differential equation (PDE)

A partial differential equation for an unknown function $\mathbf{u} = [u_1, \dots, u_n]^\top : \Omega \subset \mathbb{R}^d \to \mathbb{R}^n$, $d, n \in \mathbb{N}$, depending on the independent variables $x_1, \dots, x_d$ ($\mathbf{u} = \mathbf{u}(x_1, \dots, x_d)$) has the form

$$F(\mathbf{u}, \mathrm{D}\,\mathbf{u}, \mathrm{D}^2\,\mathbf{u}, \dots, \mathrm{D}^m\,\mathbf{u}) = \mathbf{0}\,, \tag{0.10.3}$$

where F is a general function and $\mathsf{D}^j\,\mathbf{u}$ denotes a tensor of dimension $n \times d \times \cdots \times d$ with nd^j entries defined as

$$\left(\mathsf{D}^j\,\mathbf{u}(\boldsymbol{x})\right)_{i,k_1,\ldots,k_j} := \frac{\partial^j u_i}{\partial x_{k_1}\ldots\partial x_{k_j}}(\boldsymbol{x})\,, \quad k_\ell \in \{1,\ldots,d\}\,, k_\ell \in \{1,\ldots,j\}\,. \tag{0.10.4}$$

✎ notation: we write $x_1,\ldots,x_d$ for the independent "spatial" variables, $\boldsymbol{x} = [x_1,\ldots,x_d]^\top$

Note that for $j = 1$ the derivative $\mathsf{D}\,\mathbf{u}$ boills down to the classical Jacobian of $\mathbf{u}$

$$\mathsf{D}\,u(\boldsymbol{x}) = \left[\frac{\partial u_i}{\partial x_j}(\boldsymbol{x})\right]_{i,j=1}^n = \begin{bmatrix} \frac{\partial u_1}{\partial x_1}(\boldsymbol{x}) & \frac{\partial u_1}{\partial x_2}(\boldsymbol{x}) & \cdots & \cdots & \frac{\partial u_1}{\partial x_n}(\boldsymbol{x}) \\ \frac{\partial u_2}{\partial x_1}(\boldsymbol{x}) & & & & \frac{\partial u_2}{\partial x_n}(\boldsymbol{x}) \\ \vdots & & & & \vdots \\ \frac{\partial u_n}{\partial x_1}(\boldsymbol{x}) & \frac{\partial u_n}{\partial x_2}(\boldsymbol{x}) & \cdots & \cdots & \frac{\partial u_n}{\partial x_n}(\boldsymbol{x}) \end{bmatrix}. \tag{0.10.5}$$

For $j = 1$ and $n = 1$ the matrix $\mathsf{D}^2\,\mathbf{u}$ agrees with the Hessian $\mathsf{H}\,u$ of the scalar valued function $u = u_1$:

$$\mathsf{H}\,u(\boldsymbol{x}) = \mathsf{D}^2\,u(\boldsymbol{x}) = \left[\frac{\partial^2 u}{\partial x_i \partial x_j}(\boldsymbol{x})\right]_{i,j=1}^d \in \mathbb{R}^{d,d}\,. \tag{0.10.6}$$

Note that $\mathsf{D}^j\,\mathbf{u}$ may not be well defined in some $\boldsymbol{x} \in \Omega$ in case $\mathbf{u}$ is not "sufficiently smooth" (j-times differentiable).

(0.10.7) Partial derivatives [12, Sect. 7.1]

In (0.10.4) we already used the concept of partial derivatives. The partial derivative of a function $f : \Omega \subset \mathbb{R}^d \to \mathbb{R}$ of d independent variables $x_1,\ldots,x_d$ ($f = f(x_1,\ldots,x_d)$) in an interior point $\boldsymbol{x} = [x_1,\ldots,x_d] \in \Omega$ with respect to x_j, $j = 1,\ldots,d$, is defined as

$$\frac{\partial f}{\partial x_j}(\boldsymbol{x}) := \lim_{h\to 0} \frac{f(x_1,\ldots,x_{j-1},x_j+h,x_{j+1},\ldots,x_d) - f(x_1,\ldots,x_{j-1},x_j,x_{j+1},\ldots,x_d)}{h}\,, \tag{0.10.8}$$

if the limit exists. In other words, the partial derivative with respect to x_j is obtained by differentiating the function $x_j \mapsto f(x_1,\ldots,x_{j-1},x_j,x_{j+1},\ldots,x_d)$ as a function $\mathbb{R} \mapsto \mathbb{R}$, regarding all the other independent variables as mere parameters. Higher-order partial derivatives are simply defined by nesting the above definition, for instance

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\boldsymbol{x}) := \frac{\partial}{\partial x_j}\left(\frac{\partial f}{\partial x_i}\right)(\boldsymbol{x}) \quad 1 \le i,j \le d\,. \tag{0.10.9}$$

A fundamental result about higher order partial derivatives is that there order does not matter in general:

Theorem 0.10.10. Partial derivatives commute

If all second partial derivatives of $f : \Omega \subset \mathbb{R}^d \to \mathbb{R}^n$ are continuous functions on the open domain Ω, then

$$\frac{\partial^2 f}{\partial x_i \partial x_j}(\boldsymbol{x}) = \frac{\partial^2 f}{\partial x_j \partial x_i}(\boldsymbol{x})\,, \quad 1 \le i,j \le d\,, \quad \boldsymbol{x} \in \Omega\,.$$

The assertion of the theorem generalizes to all higher-order partial derivatives:

$$\frac{\partial^j u_i}{\partial x_{k_1} \dots \partial x_{k_j}}(\boldsymbol{x}) \,, \quad k_\ell \in \{1,\dots,d\} \,, k_\ell \in \{1,\dots,j\} \,,$$

is invariant with respect to permutations of $k_1,\dots,k_j$, if u is at least j-times continuously differentiable.

(0.10.11) Differential operators

Differential operators are *special linear combinations of partial derivatives*. As such they spawn linear operators on spaces of differentiable functions defined on a domain $\Omega \subset \mathbb{R}^d$.

Important first-order differential operators are

- the gradient , defined for differentiable scalar functions $u : \Omega \to \mathbb{R}$, is the column vector

$$\mathbf{grad}\, u(\boldsymbol{x}) := \begin{bmatrix} \dfrac{\partial u}{\partial x_1} \\ \vdots \\ \dfrac{\partial u}{\partial x_d} \end{bmatrix} \in \mathbb{R}^d \,, \quad \boldsymbol{x} \in \Omega \,,$$

  see Suppl. 2.2.8 for more details.

- the divergence, defined for differentiable vector fields $\mathbf{u} : \Omega \to \mathbb{R}^d$, is the scalar function

$$\operatorname{div} \mathbf{u}(\boldsymbol{x}) := \frac{\partial u_1}{\partial x_1}(\boldsymbol{x}) + \dots + \frac{\partial u_d}{\partial x_d}(\boldsymbol{x}) \in \mathbb{R} \,, \quad \boldsymbol{x} \in \Omega \,,$$

  refer to Suppl. 2.5.6.

- the rotation, defined for $d = 3$ and differentiable vector fields $\mathbf{u} = [u_1, u_2, u_3]^\top : \Omega \to \mathbb{R}^3$, is the column vector

$$\mathbf{curl}\, \mathbf{u}(\boldsymbol{x}) := \begin{bmatrix} \dfrac{\partial u_3}{\partial x_2} - \dfrac{\partial u_2}{\partial x_3} \\ \dfrac{\partial u_1}{\partial x_3} - \dfrac{\partial u_3}{\partial x_1} \\ \dfrac{\partial u_2}{\partial x_1} - \dfrac{\partial u_1}{\partial x_2} \end{bmatrix} \in \mathbb{R}^3 \,, \quad \boldsymbol{x} \in \Omega \,.$$

These operators have distinct properties that account for their prominent occurrence in many mathematical models. For instance, from Thm. 0.10.10 we conclude

$$\boxed{\mathbf{curl} \circ \mathbf{grad} = 0} \quad , \quad \boxed{\operatorname{div} \circ \mathbf{curl} = 0} \,. \tag{0.10.12}$$

A key second-order differential operator is the Laplacian, defined for a twice differentiable scalar function $u : \Omega \to \mathbb{R}$ as

$$\Delta u(\boldsymbol{x}) := \frac{\partial^2 u}{\partial {x_1}^2}(\boldsymbol{x}) + \dots + \frac{\partial^2 u}{\partial {x_d}^2}(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,.$$

0.10.2 Electromagnetics: Eddy Current Problem

A model for the behavior of low-frequency electromagnetic fields with harmonic dependence on time:

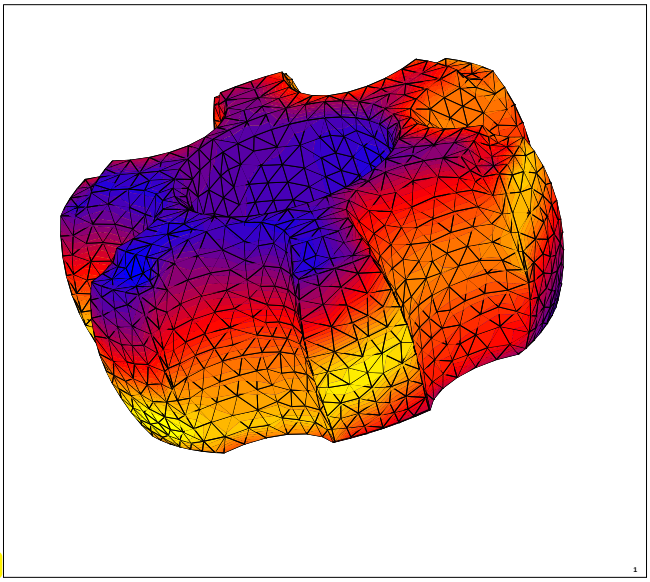


Fig. 2

Eddy current model [2]

$$\begin{aligned} \mathbf{curl}\,\mathbf{E} &= -\imath\omega\mu(\boldsymbol{x})\mathbf{H} && \text{in } \Omega\,, \\ \mathbf{curl}\,\mathbf{H} &= \sigma(\boldsymbol{x})\mathbf{E} + \mathbf{j}_s && \text{in } \Omega\,, \\ \mathbf{E}\times\mathbf{n} &= 0 && \text{on } \partial\Omega\,. \end{aligned} \tag{0.10.13}$$

$\omega > 0$	$\hat{=}$	angular frequency
$\sigma = \sigma(\boldsymbol{x}) \geq 0$	$\hat{=}$	conductivity
$\mu = \mu(\boldsymbol{x}) > 0$	$\hat{=}$	magnetic permeability
$\mathbf{E} : \Omega \mapsto \mathbb{C}^3$	$\hat{=}$	electric field
$\mathbf{H} : \Omega \mapsto \mathbb{C}^3$	$\hat{=}$	magnetic field

(Known: σ, μ, unknowns: $\mathbf{E}, \mathbf{H}$: complex fields!)

◁ induction heating simulation: surface electric field (boundary element simulation [9])

Remark 0.10.14 (Truncation of unbounded domain)

Generically, the electromagnetic equations are posed on the unbounded domain $\Omega = \mathbb{R}^3$ and have to be supplemented by the decay conditions

$$\mathbf{E}(\boldsymbol{x}) \to 0 \quad \text{uniformly for} \quad \|\boldsymbol{x}\| \to \infty\,. \tag{0.10.15}$$

In practice, (0.10.15) is often approximated by switching to a bounded domain $\Omega \subset \mathbb{R}^3$ and imposing vanishing tangential components of the electric field $\mathbf{E}$ on the boundary $\partial\Omega$, as it is done in (0.10.13).

Remark 0.10.16 (Degenerate elliptic boundary value problem)

The eddy current equations in frequency domain (0.10.13) belong to the class of degenerate second-order elliptic boundary value problems. They are called degenerate, because $\mathbf{E}$ is not uniquely determined where $\sigma \equiv 0$. To see this recall (0.10.12): In regions where $\sigma \equiv 0$ we can add any gradient to $\mathbf{E}$ and it will still be a solution of (0.10.13).

0.10.3 Viscous Fluid Flow

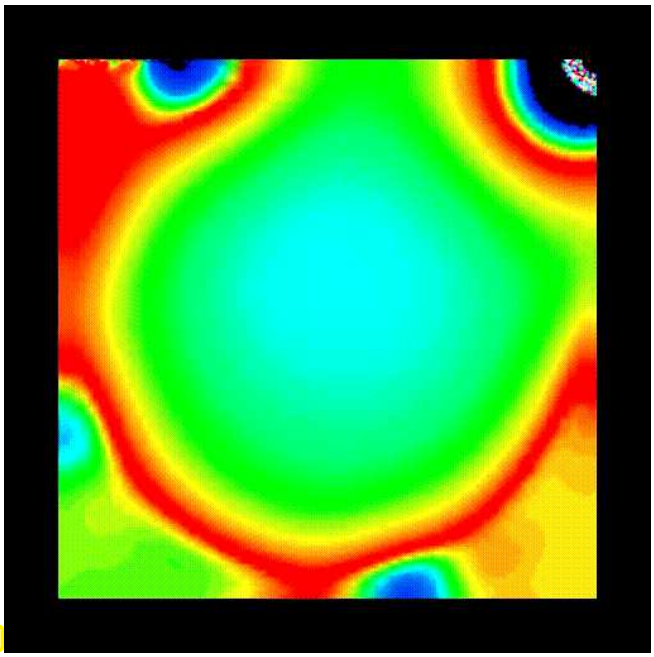


Fig. 3

(Stationary, incompressible) Navier-Stokes equations:

$$\begin{aligned} -\nu\Delta\mathbf{u} + D\mathbf{u}\cdot\mathbf{u} + \mathbf{grad}\, p &= \mathbf{f} \quad \text{in } \Omega\,, \\ \operatorname{div}\mathbf{u} &= 0 \quad \text{in } \Omega\,, \\ \mathbf{u} &= 0 \quad \text{on } \partial\Omega\,. \end{aligned} \tag{0.10.17}$$

$$\begin{aligned} \nu &\mathrel{\hat{=}} \text{dynamic viscosity} \\ \mathbf{f}:\Omega\mapsto\mathbb{R}^3 &\mathrel{\hat{=}} \text{given external force field} \\ \mathbf{u}:\Omega\mapsto\mathbb{R}^3 &\mathrel{\hat{=}} \text{velocity field (unknown)} \\ p:\Omega\mapsto\mathbb{R} &\mathrel{\hat{=}} \text{pressure (unknown)} \end{aligned}$$

If the convective term $D\mathbf{u}\cdot\mathbf{u}$ is omitted we obtain the Stokes-equations, see Chapter 9

◁ Lid driven cavity flow, pressure distribution (finite element simulation with FEATFLOW)

The Navier-Stokes equations (0.10.17) describe the motion of viscuous ("sticky") fluid under external forces. The boundary conditions mean that the fluid sticks to the wall of the container Ω (no-slip boundary conditions). The equations (0.10.17) provide the fundamental model in computational fluid dynamics (CFD).

0.10.4 Micromagnetics

Micromagnetics deals with the evolution of the time-dependent magnetization $\mathbf{m} = \mathbf{m}(\boldsymbol{x},t)$, of a ferromagnetic material under the influence of an external magnetic field. The main quasi-stationary model are the Landau-Livshits-Gilbert equations here given in scaled (non-dimensional) form, see[11]:

$$\begin{aligned} \frac{\partial\mathbf{m}}{\partial t} - \mathbf{m}\times\frac{d\mathcal{E}(\mathbf{m},\psi(\mathbf{m}))}{d\mathbf{m}} - \alpha\mathbf{m}\times(\mathbf{m}\times\frac{d\mathcal{E}(\mathbf{m},\psi(\mathbf{m}))}{d\mathbf{m}}) &= 0 && \text{in } \Omega\times[0,T]\,, \\ -\Delta\psi + \operatorname{div}\mathbf{m} &= 0 && \text{in } \mathbb{R}^3\times[0,T]\,. \\ |\psi(\boldsymbol{x})| &= O(|\boldsymbol{x}|^{-1}) && \text{for } |\boldsymbol{x}|\to\infty\,, \\ \mathbf{m}(\cdot,0) &= \mathbf{m}_0(\cdot) && \text{in } \Omega\,. \end{aligned} \tag{0.10.18}$$

with scaled Gibbs free energy

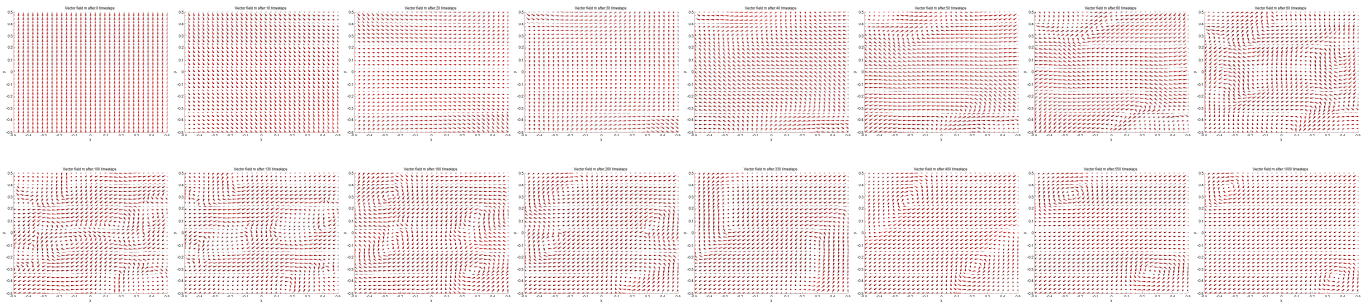
$$\mathcal{E}(\mathbf{m},\psi) = \tfrac{1}{2}\int_\Omega \eta|\,\mathbf{grad}\,\mathbf{m}|^2 + Q(1-(\boldsymbol{d}\cdot\mathbf{m})^2) - 2\mathbf{H}_0\cdot\mathbf{m}\,\mathrm{d}\boldsymbol{x} + \tfrac{1}{2}\int_{\mathbb{R}^3}|\,\mathbf{grad}\,\psi|^2\,\mathrm{d}\boldsymbol{x}\,.$$

The fields and coefficients occurring in the model are

$$\begin{aligned} \mathbf{m}:\Omega\times[0,T]\mapsto\mathbb{S}^2 &\mathrel{\hat{=}} \text{magnetization (direction field, } \|\mathbf{m}\|=1\text{, if } \|\mathbf{m}_0\|=1\text{);} \\ &\phantom{\mathrel{\hat{=}}}\ \text{(the unknown of the model)} \\ \psi:\mathbb{R}^3\to\mathbb{R} &\mathrel{\hat{=}} \text{magnetic scalar potential} \\ \alpha>0 &\mathrel{\hat{=}} \text{damping parameter} \\ Q>0,\ \boldsymbol{d}\in\mathbb{R}^3 &\mathrel{\hat{=}} \text{strength/direction of material anisotropy} \\ \mathbf{m}_0 &\mathrel{\hat{=}} \text{initial magnetization} \end{aligned}$$

The equations (0.10.18) describe a parabolic *gradient flow system* for the Gibbs free energy on the manifold of director fields, that is, vector fields with modulus 1.

Flipping of magnetization, computed by means of a finite element simulation [5], more details about finite element method (FEM) are given in Chapter 3.



We observe the formation of vortices, which finally disappear at the upper left and the lower right corners. In the final state, the elementary magnets tend to point in the same direction.

0.10.5 Reaction-diffusion: Phase Separation

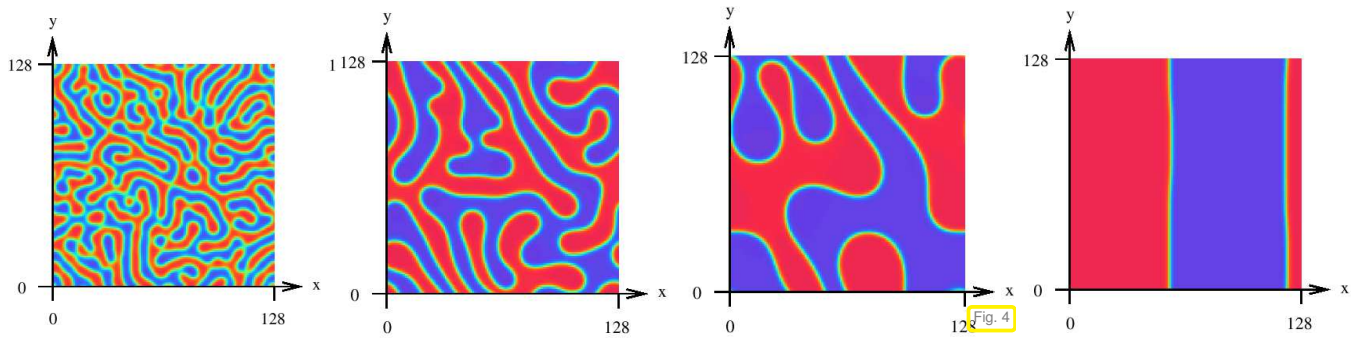
The Cahn-Hillard equation is a PDE of mathematical physics which describes the process of phase separation, by which the two components of a binary fluid spontaneously separate and form domains pure in each component. u is the concentration of one phase of the fluid, with $u = \pm 1$ indicating domains. Here we give a boundary value evolution problem for the Cahn-Hillard equation in scaled (non-dimensional) form:

$$\begin{aligned} \frac{du}{dt} - \alpha\Delta(u^3 - u - \gamma\Delta u) &= 0 && \text{in } \Omega\times]0,T[\,, \\ u(\cdot,0) &= u_0 && \text{in } \Omega\,. \\ \mathbf{grad}(u^3 - u - \gamma\Delta u)\cdot\mathbf{n} &= 0 && \text{on } \partial\Omega\,, \\ \frac{1}{\Gamma_s}\frac{\partial u}{\partial t} + \mathbf{grad}\, u\cdot\mathbf{n} + \sigma\Delta u &= 0 && \text{on } \partial\Omega\,. \end{aligned} \tag{0.10.19}$$

$$\begin{aligned} u = u(\boldsymbol{x},t) &\;\hat{=}\; \text{time-dependent concentration (unknown)} \\ \alpha > 0 &\;\hat{=}\; \text{known diffusion coefficient} \\ \gamma > 0 &\;\hat{=}\; \text{known diffusion length} \end{aligned}$$

The equations (0.10.19) describe a gradient flow system with "mass conservation" for the chemical potential.

Evolution snapshots (finite difference discretization, [10]):



0.10.6 Quantitative Finance: Black-Scholes equation

The task of option pricing for European options leads to the Black-Scholes equation on $\mathbb{R}^d_+$ [1]:

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}\sum_{i,j=1}^{d} q_{ij}\, x_i x_j \frac{\partial^2 u}{\partial x_i \partial x_j} + r\sum_{i=1}^{d} x_i \frac{\partial u}{\partial x_i} - ru &= 0 \quad \text{in } \mathbb{R}^d_+ \times [0,T]\,, \\ \text{+ "exact boundary values" imposed on } \partial\mathbb{R}^d_+\,, & \\ u(T,\cdot) &= g(\cdot) \quad \text{in } \Omega\,. \end{aligned} \tag{0.10.20}$$

- $d \in \mathbb{N}$ = no. of underlying stocks, $\boldsymbol{x} = (x_1,\dots,x_d)^T$, $x_i \leftrightarrow$ price of stock #i
- Unknown $u = u(\boldsymbol{x},t) \mathrel{\hat{=}}$ option price at time t given stock prices x_i:

  $$u(t,\boldsymbol{x}) = \mathbb{E}(\exp(-r(t-T))g(\boldsymbol{S}_T)|\boldsymbol{S}_t = \boldsymbol{x})\,,$$

  with payoff function $g : \mathbb{R}^d_+ \mapsto \mathbb{R}$.
- Coefficients$r > 0$ = interest rate, $\left(q_{ij}\right)_{i,j=1}^{d}$ = s.p.d. covariance matrix

This is a high-dimensional degenerate parabolic initial-boundary value problem. The Stock price fluctuations are modelled by means of a Wiener process (log-normal distribution) $S_i(t) = \exp(rt + X_t^i)$ .Here we give numerical simulations in $d = 2$ with linear finite elements on tensor product mesh (MATLAB computations, C. Winter, SAM, ETH Zürich):

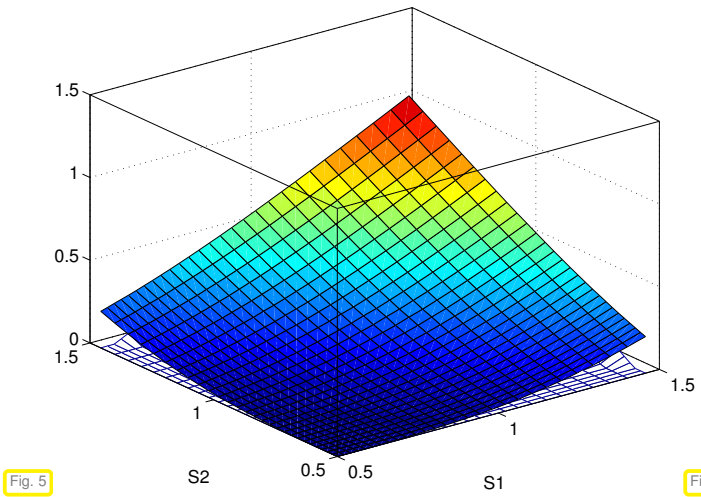


Fig. 5

Call option

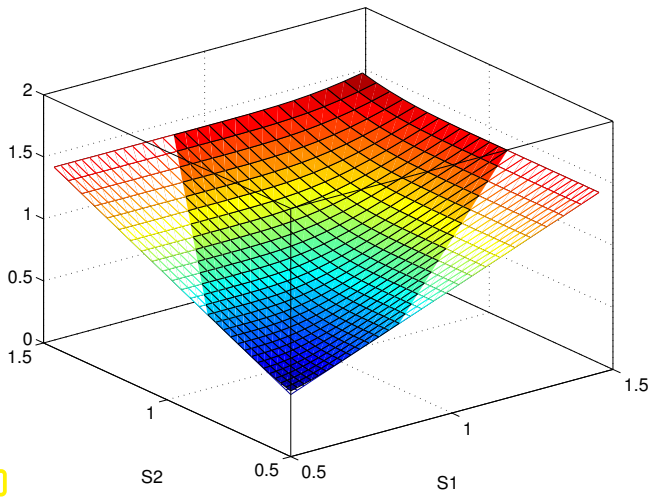


Fig. 6

Better-of-two option

The payoff functions used in the computations are

$$\begin{aligned} \text{call option:} \quad & g(S_1,S_2) := \max\{S_1 + S_2 - K, 0\}\,, \quad \text{with strike price } K\,, \\ \text{better-of-two option:} \quad & g(S_1,S_2) := \max\{S_1, S_2\}\,. \end{aligned}$$

More details ➪ course "Computational Methods for Quantitative Finance" (Ch. Schwab)

0.10.7 Quantum Mechanics: Electronic Schrödinger Equation

The following equations formulate an elliptic eigenvalue problem obtained from the Born-Oppenheimer approxmation of the Schrödinger equation, the fundamental governing equation for quantum phenomena.

Its solutions describe the cloud of electrons around for a molecule at different excited states.

$$\left(-\frac{1}{2}\Delta + \sum_{i=1}^{N}\sum_{j=1}^{P}\frac{Z_j}{|\boldsymbol{x}_i - \boldsymbol{r}_j|} + \sum_{i=1}^{N}\sum_{j>i}^{N}\frac{1}{|\boldsymbol{x}_i - \boldsymbol{x}_j|}\right)u = \lambda u\,, \tag{0.10.21}$$

$$+ \text{ exponential decay of } u \text{ for } |\boldsymbol{x}| \to \infty\,. \tag{0.10.22}$$

- N = number of electrons
- P = number of nuclei (with charges $Z_j \in \mathbb{N}$ and positions $\boldsymbol{r}_j \in \mathbb{R}^3$)
- Unknown: $u = u(\boldsymbol{x}_1, \dots, \boldsymbol{x}_N) \neq 0$, $\boldsymbol{x}_i \in \mathbb{R}^3$! ➥ probability density $|u|^2$
- Unknown: eigenvalue λ = state energy

High-dimensional elliptic eigenvalue problem on $\mathbb{R}^{3N}$!

Numerical simulation: states $(N, P = 1)$ computed with spectral sparse grid Galerkin method [6]:

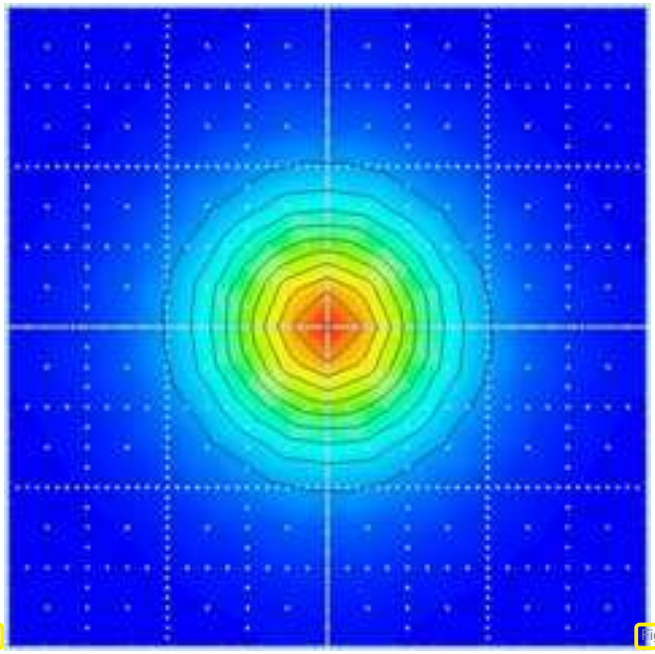


Fig. 7 Symmetric ground state

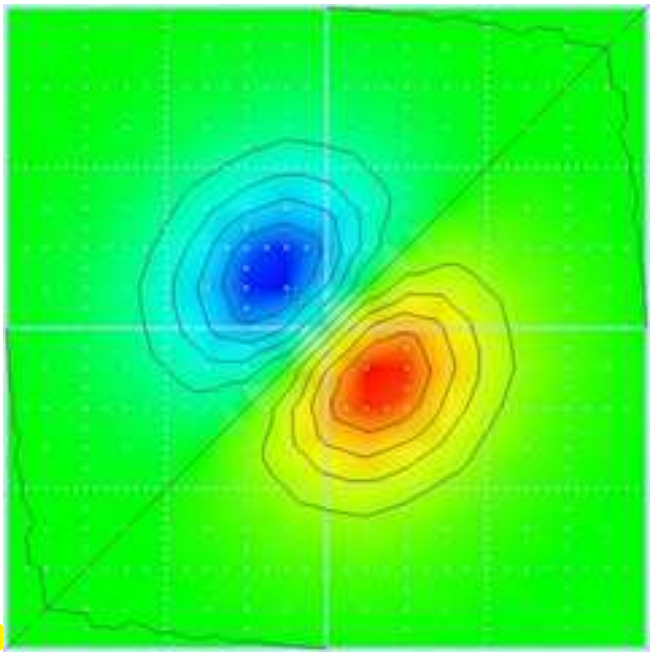


Fig. 8 Anti-symmetric ground state

0.10.8 Rarefied Gas Dynamics: Boltzmann Equation

The state of a rarefied gas occupying the bounded region of space $\Omega \subset \mathbb{R}^3$ can be described by a density function $f = f(\boldsymbol{x}, \boldsymbol{v}, t)$, which is a function of space ($\boldsymbol{x}$), of velocity ($\boldsymbol{v}$), and time (t). Its meaning is the following: the integral

$$\int_{B_x}\int_{B_v} f(\boldsymbol{x}, \boldsymbol{v}, t)\,\mathrm{d}\boldsymbol{v}\mathrm{d}\boldsymbol{x}\,, \quad B_x \subset \Omega\,, \quad B_v \subset \mathbb{R}^3\,,$$

yields the number of gas molecules located inside B_x and travelling with a velocity in B_v at time t. The evolution of the density function is governed by the Boltzmann equation

$$\frac{\partial f}{\partial t} + \boldsymbol{v}\cdot\mathbf{grad}_{\boldsymbol{x}} f = Q(f, f) \quad \text{in } \Omega \times \mathbb{R}^3\,, \tag{0.10.23}$$

supplemented with the inflow boundary conditions

$$u(\boldsymbol{x},\boldsymbol{v},t) = g(\boldsymbol{x},\boldsymbol{v},t) \quad \text{for} \quad \boldsymbol{x} \in \partial\Omega\,, \quad \boldsymbol{v}\cdot\boldsymbol{n}(\boldsymbol{x}) < 0\,, \tag{0.10.24}$$

where g are given boundary data. The collision operator is given by

$$Q(f,g)(\boldsymbol{x},\boldsymbol{v},t) := \int_{\mathbb{R}^3}\int_{\mathbb{S}^2} B(\|\boldsymbol{v}-\boldsymbol{v}_*\|,\cos\theta)(h'_* f' - h_* f)\,\mathrm{d}\sigma\mathrm{d}\boldsymbol{v}_*\,, \tag{0.10.25}$$

$$f := f(\boldsymbol{x},\boldsymbol{v},t)\,, \quad h_* := h(\boldsymbol{x},\boldsymbol{v}_*,t)\,, \quad f' := f(\boldsymbol{x},\boldsymbol{v}',t)\,, \quad h'_* := h(\boldsymbol{x},\boldsymbol{v}'_*,t)\,,$$
$$\boldsymbol{v}' := \tfrac{1}{2}(\boldsymbol{v}+\boldsymbol{v}_* + \|\boldsymbol{v}-\boldsymbol{v}_*\|)\,, \quad \boldsymbol{v}'_* := \tfrac{1}{2}(\boldsymbol{v}+\boldsymbol{v}_* - \|\boldsymbol{v}-\boldsymbol{v}_*\|\boldsymbol{\sigma})\,.$$

The function $B : \mathbb{R}\times\mathbb{R} \to \mathbb{R}$ is the so-called collision kernel, and $\mathbb{S}^2$ stands for the unit sphere. The angle θ is enclosed by the two velocities $\boldsymbol{v}$ and $\boldsymbol{v}'$.

Note: The problem (0.10.23) is moderately high-dimensional, since it is posed on a seven-dimensional unbounded domain $\Omega\times\mathbb{R}\times\mathbb{R}$.

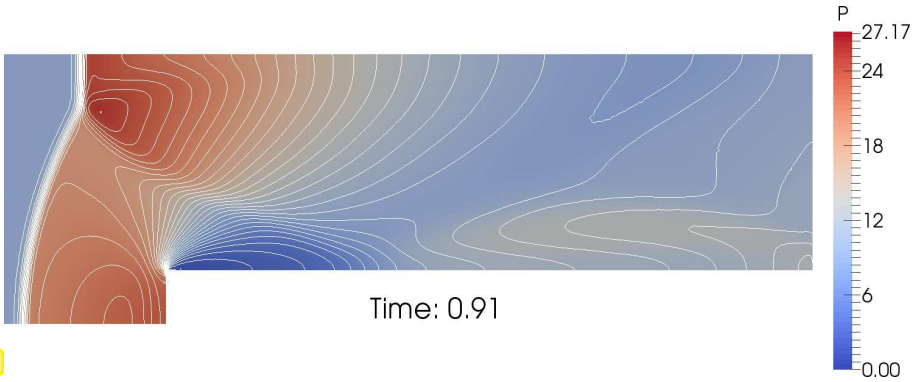


Fig. 9

Mach-3 CFD benchmark problem. Inflow on left boundary, specular reflective wall, outflow boundary conditions on the right. Computation with least squares finite elements in space and spectral polynomial approximation in velocity, see [7].

0.10.9 Wave Propagation: Helmholtz equation

Time-harmonic acoustic waves are described by the spatio-temporal behavior of sound pressure $o = p(\boldsymbol{x},t)$. In linear media without sources it satisfies the homogeneous Helmholtz equation

$$\Delta p + k^2 n(\boldsymbol{x}) p = 0 \quad \text{in } \Omega\,, \tag{0.10.26}$$

where $k > 0$ is the wave number (inversely proportional to the frequency), and $n = n(\boldsymbol{x})$ is a dimensionless spatially varying refractive index of the medium.

Often the Helmholtz equation is posed on unbounded domain, for instance $\Omega = \mathbb{R}^3$. In this case we need a radiation condition at ∞:

$$\lim_{\|\boldsymbol{x}\|\to\infty} \|\boldsymbol{x}\| \left(\frac{\partial p_s}{\partial r}(\boldsymbol{x}) - \imath k p_s(\boldsymbol{x}) \right) = 0 \quad \text{uniformly}\,, \tag{0.10.27}$$

where $p_s := p - p_{\text{inc}}$ is the scattered field, the difference of the pressure field p and another pressure field p_{inc} that belongs to an incident exciting acoustic wave.

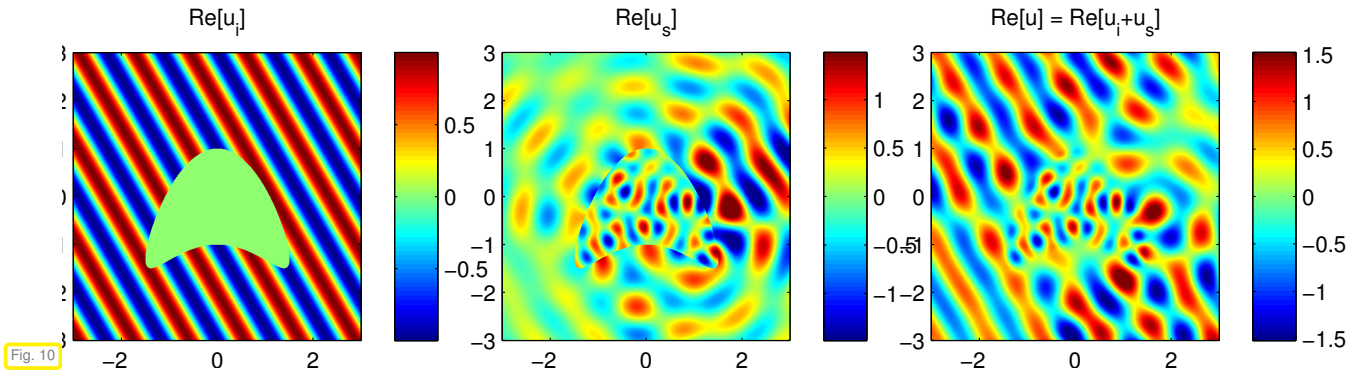


Fig. 10

Left: incident wave; middle: scattered field, right: total pressure p (real parts). Computation with method of particular solutions, using the software MPSPACK, see [3].

?! *Review question(s) 0.10.28.*

- The following is a PDE for a vector field $\mathbf{u} : \Omega \to \mathbb{R}^2$:

$$\mathbf{grad}\,\mathrm{div}\,\mathbf{u} = [f_1 f_2 f_3]^\top \,, \quad f_i : \Omega \to \mathbb{R} \,. \tag{0.10.29}$$

  Write this PDE in detail for the components of $\mathbf{u}$.

- Compute the Hessian according to (0.10.6) for the function $u(x_1, x_2) = \exp(x_1^2 + x_2^2)$.
- Compute the rotation $\mathbf{curl}\,\mathbf{u}$ and the divergence $\mathrm{div}\,\mathbf{u}$ for the vector field $\mathbf{u}(\boldsymbol{x}) = -\frac{\boldsymbol{x}}{\|\boldsymbol{x}\|^2}$, $\boldsymbol{x} \in \mathbb{R}^3 \setminus \{\mathbf{0}\}$.
- Compute the Laplacian of the function $\mathbf{u}(x_1 x_2) = \sin(x_1)\cos(x_2)$. What do you observe?

Bibliography

[1] Y. Achdou and O. Pironneau. *Computational methods for option pricing*. Frontiers in Applied Mathematics. SIAM, Philadelphia, 2005.

[2] A. Alonso-Rodriguez and A. Valli. *Eddy Current Approximation of Maxwell Equations*, volume 4 of *Modelling, Simulation & Applications*. Springer, Milan, 2010.

[3] A.H. Barnett and T. Betcke. An exponentially convergent nonpolynomial finite element method for time-harmonic scattering from polygons. *SIAM J. Sci. Comp.*, 32(3):1417–1441, 2010.

[4] A. Burtscher, E. Fonn, P. Meury, and C. Wiesmayr. *LehrFEM - A 2D Finite Element Toolbox*. SAM, ETH Zürich, Zürich, Switzerland, 2010. http://www.sam.math.ethz.ch/ hiptmair/tmp/LehrFEMManual.pdf.

[5] P. Corboz. Mixed implicit timestepping for micromagnetism. Semesterarbeit rw/cse, SAM, ETH Zürich, Zürich, Switzerland, 2003. http://www.sam.math.ethz.ch/ Ehiptmair/StudentProjects/Corboz/report.pdf.

[6] M. Griebel and J. Hamaekers. Saprse grids for the Schrödinger equation. Preprint 0504, INS, University of Bonn, Bonn, Germany, 2005. http://wissrech.ins.uni-bonn.de/research/pub/hamaekers/GriebelHamaekersINSpreprint0504.pdf.

[7] P. Grohs, R. Hiptmair, and S. Pintarelli. Tensor-product discretization for the spatially inhomogeneous and transient boltzmann equation in 2d. Technical Report 2015-38, Seminar for Applied Mathematics, ETH Zürich, Switzerland, 2015.

[8] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[9] R. Hiptmair and J. Ostrowski. Coupled boundary element scheme for eddy current computation. *J. Engr. Math.*, 51(3):231–250, 2004.

[10] R. Kenzler, F. Eurich, P. Maass, B. Rinn, J. Schropp, E. Bohl, and W. Dietrich. Phase separation in confined geometries: Solving the Cahl-Hilliard equation with generic boundary conditions. Report, Fachbereich für Mathematik, Universität Konstanz, 2000. http://www.uni-konstanz.de/FuF/Physik/Dieterich/.

[11] A. Prohl. *Computational micromagnetism*. Advances in Numerical Mathematics. B.G. Teubner, Stuttgart, 2001.

[12] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

Chapter 1

Case Study: A Two-point Boundary Value Problem

This chapter offers a brief tour of

- mathematical modelling of a physical system based on variational principles (= minimization in infinite dimensional configuration space),
- the derivation of *differential equations* from these variational principles,
- the discretization of the variational problems and/or of the differential equations using various approaches.

Contents

1.1 Preface

(1.1.1) Looking behing a partial differential equation (PDE)

The term "partial differential equation" (PDE) usually conjures up formulas like

$$\operatorname{div}(\sqrt{1+\|\mathbf{grad}\, u(\boldsymbol{x})\|^2}\,\mathbf{grad}\, u(\boldsymbol{x})) + \mathbf{v}\cdot\mathbf{grad}\, u = f(\boldsymbol{x})\,, \quad \boldsymbol{x}\in\Omega\subset\mathbb{R}^d\,.$$

All the mathematical models presented in Section 0.10 were stated in this "PDE form".

This chapter aims to wean you off the impulse to look at a PDE as an equation of the form (0.10.3) linking partial derivatives. Rather it wants to instill the appreciation that

a meaningful PDE encodes structural principles
(like equilibrium, conservation, etc.)

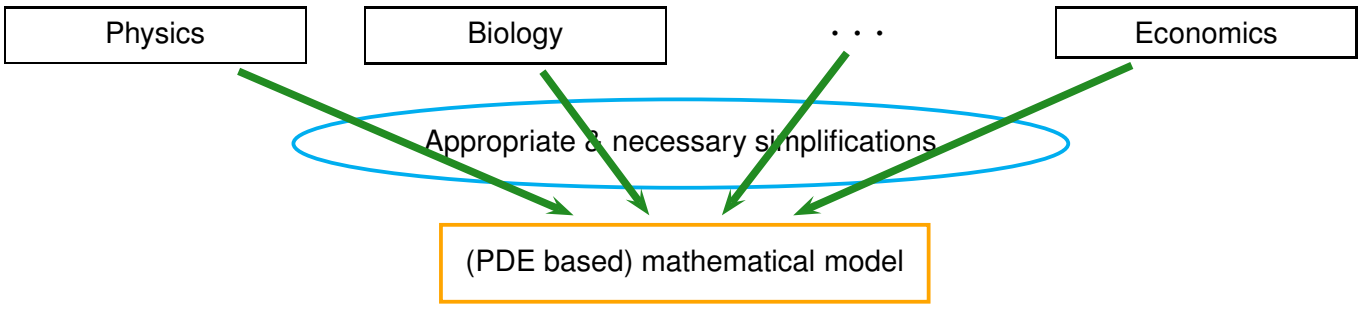
In other words, usually there is a physical system or real phenomenon behind a PDE. The differential equation is meant to capture some of its aspects, in particular those that are important to the "user" of the PDE. In a sense, it provides a mathematical model.

Of course, a numerical solution of a PDE should reflect the structures inherent in the model. Thus, awareness of these structures matters for numerical simulation and we can state the following guideline:

! The design and selection of numerical methods for solving a "PDE" has to take into account its origin and context.

Remark 1.1.2 (Mathematical modelling)

Prerequisite for numerical simulation: Mathematical modelling



Necessary simplification:

$\left\{\begin{array}{c}\text{system}\\ \text{phenomenon}\end{array}\right\}$ described by *a few* variables/functions in a configuration space

The art of modelling: devise "faithful model"

Essential/relevant traits of $\left\{\begin{array}{c}\text{system}\\ \text{phenomenon}\end{array}\right\}$ $\longrightarrow$ structural properties of model

This chapter will offer a glimpse of considerations typical of mathematical modeling approaches that lead to differential equations.

Our notion of PDE based models $\rightarrow$ **§ 0.10.1**

This course uses "partial differential equation" as a synonym for a mathematical model
- ✦ based on an infinite-dimensional function space as configuration space,
- ✦ governed by "local interactions of function values".

The elements of a function space are functions defined on a common *domain*. In analysis and linear algebra you have come across function spaces, for instance the space of continuous functions on an interval, which is the domain in this case, see [5, Sect. 4.1, Bsp. 3].

Theses function spaces will usually be *vector spaces* under (pointwise) addition and multiplication with a scalar; if V is a space of real-valued functions on the domain Ω, we define

$$f,g \in V,\ \alpha \in \mathbb{R}: \quad (f+g)(x) := f(x) + g(x)\ , \quad (\alpha \cdot f)(x) := \alpha f(x)\ , \quad \forall x \in \Omega\ .$$

Remark 1.1.4 (Models based on ordinary differential equations (ODEs))

Mathematical models of *time-dependent* (instationary) systems with finite-dimensional configuration space are often stated in the form of initial value problems [4, § 11.1.19] for ordinary differential equations: with a function $\mathbf{f} : I \times \mathbb{R}^n \rightarrow \mathbb{R}^n$, $I \subset \mathbb{R}$ and interval, $t_0 \in I$, they read [4, Section 11.1]

$$\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y}) \quad , \quad \mathbf{y}(t_0) = \mathbf{y}_0 \in \mathbb{R}^n\ . \tag{1.1.5}$$

Though we seek an unknown function $\mathbf{y} : I \subset \mathbb{R} \rightarrow \mathbb{R}^n$, the configuration space is $\mathbb{R}^n$ in this case. Equally important, the solution is obtained from tracking an evolution from initial time to final time.

Remark 1.1.6 (Evolution partial differential equations)

Systems that evolve with time and whose configuration can be adequately modelled by means of an infinite-dimensional function space are very common, refer to Section 0.10.4, Section 0.10.5, Section 0.10.6, Section 0.10.8 for examples. Their mathematical description leads to evolution problems for partial differential equations, which can be regarded as "ordinary differential equations" in function spaces, which are evolutions with infinite-dimensional configuration spaces.

Remark 1.1.7 ("PDEs" for univariate functions)

The classical notion of a PDE inherently involves functions of several independent variables. However, when one embraces the concept of a PDE as a mathematical model built on a function space, then simple representatives in a univariate setting can be discussed.

☞ some ordinary differential equations (ODEs) do not encode evolutions in time and thus offer simple specimens of important classes of PDEs!

Thus, in this chapter we examine ODEs that are related to the important class of elliptic PDEs.

Note: The presentation in this course cannot completely comply with standards of mathematical rigor, because what has deliberately omitted is the discussion of the functional analytic framework $(*)$ (function space theory) required for a complete statement of, for instance, minimization problems and variational problems.

$(*)$ functional analysis a branch of pure mathematics devoted to the study of infinite dimensional vector spaces and related mappings.

1.2 A Model(ing) Problem

1.2.1 Thin Elastic String

Stationary mechanical problem:

Deformation of elastic "1D" string (rubber band) under its own weight

Constraint: string pinned at endpoints

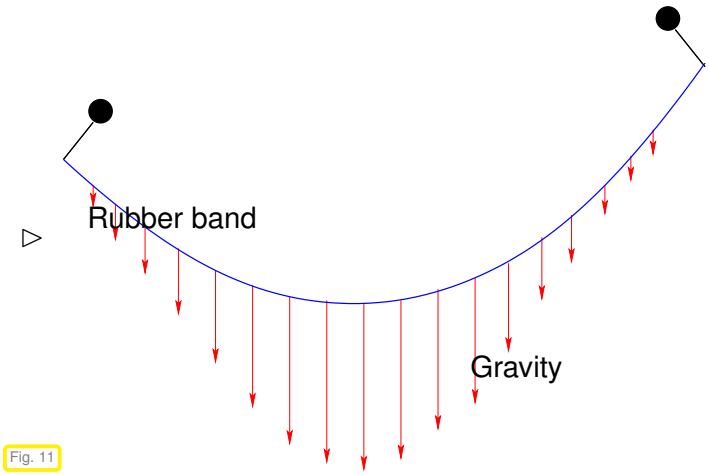


Fig. 11

Photo of a rubber band bent by its own weight ▷

(In this case the elastic deformation is almost negligible due to the light weight of rubber)

Fig. 12

Sought:	(Numerical approximation of) "shape" of elastic string

We want to find the shape of the elastic string based on a continuum approach. The first step is to find a suitable mathematical description of this "shape".

(1.2.1) Configuration space for elastic string

The configuration space of a physical system is a set, for which a single element completely describes the state of the system in the underlying mathematical model.

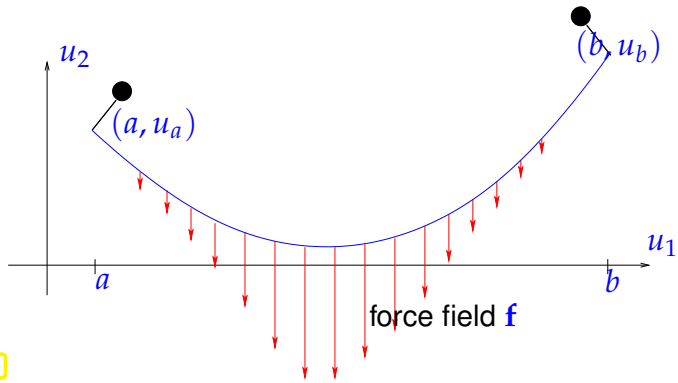


Fig. 13

Configuration space
= space of curves (connecting given points)
☞ shape of string
$\updownarrow$
function $\mathbf{u} : [0,1] \mapsto \mathbb{R}^2$, $\mathbf{u} = \mathbf{u}(\xi)$
(physical units $[\mathbf{u}] = 1\mathrm{m}$)

☞ Pinning conditions (boundary conditions):

$$\mathbf{u}(0) = \begin{bmatrix} a \\ u_a \end{bmatrix} \in \mathbb{R}^2 , \quad \mathbf{u}(1) = \begin{bmatrix} b \\ u_b \end{bmatrix} \in \mathbb{R}^2 . \tag{1.2.2}$$

Terminology: $[0,1] \hat{=}$ parameter domain, ✎ notation Ω

Remark 1.2.3 (Coordinate system)

The description of a curve in the plane by a mapping $[0,1] \mapsto \mathbb{R}^2$ requires a coordinate system. Of course, the choice of the coordinate system must not affect the shape obtained from the mathematical model, a property called frame indifference.

Switching from one coordinate system to another is accomplished by *affine linear* mappings of point coordinates.

(1.2.4) Spaces of continuously differentiable functions $\rightarrow$ [6, Sect. 5.4]

A first family of important function spaces relies on the classical notion of differentiability from calculus:

✎ notation: $C^k([a,b]) \hat{=}$ k-times continuously differentiable functions on $[a,b] \subset \mathbb{R}$,

Clearly, $C^k([a,b])$ is a vector space under pointwise addition and pointwise multiplication with a scalar.

(1.2.5) Parameterization of a curve $\rightarrow$ [6, Sect. 7.4]

(Coordinate system in the plane is taken for granted, see Rem. 1.2.3)

We consider a curve in $\mathbb{R}^2$ $\quad \mathbf{u} : [0,1] \mapsto \mathbb{R}^2$

$\triangleright$

$$\boxed{\begin{array}{c} \mathbf{u} \in (C^0([0,1]))^2 \\ \Updownarrow \\ \text{connected curve} \end{array}}$$

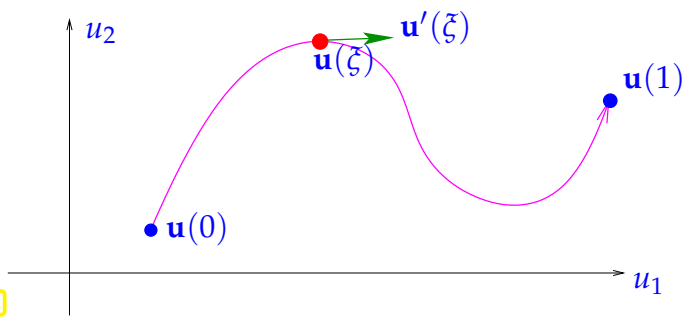


Fig. 14

✎ notation: $(C^k([a,b]))^2 \hat{=}$ k-times continuously differentiable curves $\mathbf{u} : [a,b] \mapsto \mathbb{R}^2$, that is, if $\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$, then $u_1, u_2 \in C^k([a,b])$.

➢ parameterization is supposed to be *locally injective*:

$$\forall \xi \in]0,1[: \quad \exists \epsilon > 0: \quad \forall \eta, \ |\eta - \xi| < \epsilon: \quad \mathbf{u}(\eta) \neq \mathbf{u}(\xi) \ .$$

Meaning of "locally": Global injectivity of parameterization must break down in case of self-intersection of the curve, because in this case there will be $\xi \neq \eta$ with $\mathbf{u}(\eta) = \mathbf{u}(\xi)$, but these parameter values cannot be arbitrarily close.

► For $\mathbf{u} \in (C^1([0,1]))^2$ we demand $\mathbf{u}'(\xi) \neq 0$ for all $0 \leq \xi \leq 1$

✎ notation: $' \hat{=}$ derivative w.r.t. curve parameter, here ξ. It does not affect physical units, because the parameter ξ is dimensionless.

Remark 1.2.6 (Length of a curve)

We consider a curve in the plane (equipped with a coordinate system according to Rem. 1.2.3) described by a parameterization $\mathbf{u} : [0,1] \to \mathbb{R}^2$ as in § 1.2.5.

Geometric intuition: $\mathbf{u}(\xi)$ moves along the curve as ξ increases from 0 to 1.

Interpretation of curve parameter ξ: "virtual time"

➢ $\|\mathbf{u}'\| \hat{=}$ "speed" with which curve is traversed (physical units m!), see Fig. 14.

➢ $\int_0^1 \|\mathbf{u}'(\xi)\| \, \mathrm{d}\xi \hat{=}$ length of curve,
because the length of a path is the integral of speed over time.

✎ notation: $\|\cdot\| \hat{=}$ Euclidean norm of a vector $\in \mathbb{R}^n$

Note: Length remains well defined for merely *piecewise differentiable* curves.

?! *Review question(s) 1.2.7.* (Curves)

1. Give a parameterization over $[0,1]$ of a straight line segment connecting two points $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d$, $d \in \mathbb{N}$.
2. How can you describe a circle in the plane with radius $r > 0$ and center $\boldsymbol{z} \in \mathbb{R}^2$ as a curve parameterized over $[0,1]$.
3. Let a curve in the plane be given by a parameterization $\mathbf{u} =: [u_1, u_2]^\top : [0,1] \to \mathbb{R}^2$. When can it be written as the graph of a function $f : D \subset \mathbb{R} \to \mathbb{R}$, that is $u_2 = f(u_1)$?
4. Let $\mathbf{u} : [0,1] \to \mathbb{R}^d$ be a parameterization of a C^1-curve. Give a formula describing the tangent at the curve in $\mathbf{u}(\xi)$, $0 \leq \xi \leq 1$.

(1.2.8) Non-uniqueness of parameterization of curve

Again, consider a parameterized curve $\mathbf{u} : [0,1] \to \mathbb{R}^2$.

For every *strictly monotone* continuous function $\Phi : [0,1] \to [0,1]$ with $\Phi(0) = 0$ and $\Phi(1) = 1$, the function $\mathbf{v}(\xi) := \mathbf{u}(\Phi(\xi))$, $\xi \in [0,1]$, will provide another parameterization of the *same* curve.

➥

The parameterization of a curve is **not unique**: different functions $[0,1] \to \mathbb{R}^2$ can describe the same curve (reparameterization)!

➥ Seeking a curve in terms of a parameterization leads to models that may fail to possess a unique solution!

We have to constrain the parameterization in order to render it unique. A popular way to do this is arclength parameterization that demands $\|\mathbf{u}'\| \equiv \mathrm{const}$ (constant speed).

Remark 1.2.9 (Material coordinate)

Interpretation of curve parameter ξ:

ξ: unique identifier for each infinitesimal section of the string,
a *label* for each "material point" on the string

$\xi \mathrel{\hat{=}}$ material coordinate, unrelated to "position in space" (= physical coordinate),
ξ has no physical dimension ➤ $'$ does not change physical units.

However, we have great freedom to choose material coordinates for a curve, compare § 1.2.8.

Remark 1.2.10 (Non-dimensional equations)

By fixing reference values for the basic physical units occurring in a model ("scaling"), one can switch to a non-dimensional form of the model equations.

In the case of the elastic string model the basic units are

- unit of length $1\mathrm{m}$,
- unit of force $1\mathrm{N}$.

Thus, non-dimensional equations arise from fixing a reference length ℓ_0 and a reference force f_0.

Below, following a (bad) habit of mathematicians, physical units will routinely be dropped, which tacitly assumes a priori scaling.

Note: Scaling is convenient, but is actually not required for numerical simulation and SI units can be kept for all quantities, owing to the fact that proper implementations of numerical methods should be *scale-invariant*. The code should always produce the same result regardless of chosen physical units (, if potential under-/overflow of floating point numbers is neglected [4, Rem. 1.5.36]).

(1.2.11) External forces

Non-trivial and stable shapes of elastic strings can be expected only if they are subject to external forces. These will usually be supplied by the earth's gravity field. We focus on the more general setting of *conservative force fields* acting on the string.

Assumption 1.2.12. Gravitational potential

We assume that a (differentiable) gravitational potential $V : \mathbb{R}^2 \to \mathbb{R}$ (units $[V] = \frac{\mathrm{m}^2}{\mathrm{s}^2}$) acts on the elastic string; in it a mass m at position $\mathbf{x} \in \mathbb{R}^2$ has "gravitational" potential energy $-mV(\mathbf{x})$.

► Force on mass at $\mathbf{x}$: $\mathbf{f} = m\,\mathbf{grad}\,V(\mathbf{x})\,, \quad \mathbf{x} \in \mathbb{R}^2\,.$

➢ $\mathbf{grad}\,V : \mathbb{R}^2 \to \mathbb{R}^2$ is called the acceleration field or gravitational field (units $\frac{\mathrm{m}}{\mathrm{s}^2}$, acceleration).

Simplest choice: linear potential $\leftrightarrow$ constant acceleration field

$$V(\mathbf{x}) = \mathbf{g}\cdot\mathbf{x} \quad \Leftrightarrow \quad \mathbf{grad}\,V(\mathbf{x}) = \mathbf{g}\,, \quad \mathbf{g} \in \mathbb{R}^2 \quad (\text{units } [\mathbf{g}] = \frac{\mathrm{m}}{\mathrm{s}^2})\,.$$

✎ notation: $\mathbf{u}\cdot\mathbf{v} := \mathbf{u}^T\mathbf{v} = \sum_{j=1}^{n} u_i v_i \mathrel{\hat{=}}$ inner product of vectors in $\mathbb{R}^n$.

(1.2.13) Problem data/parameters

Quantities that have to be specified to allow the unique determination of a configuration in a mathematical model are called problem data/parameters. In the elastic string model the problem parameters are

- the boundary conditions (1.2.2),
- the acceleration field $\mathbf{f} := \mathbf{grad}\,V : \mathbb{R}^2 \mapsto \mathbb{R}^2$, (units $[\mathbf{f}] = \frac{\mathrm{N}}{\mathrm{kg}}$), $\mathbf{f}(\mathbf{x}) \mathrel{\hat{=}}$ force "pulling at" a point $\mathbf{x}$.

  Special case: vertical downward gravitational force $\mathbf{f}(\mathbf{x}) := -g\begin{bmatrix}0\\1\end{bmatrix}$, $g = 9.81\mathrm{m}\,\mathrm{s}^{-2}$.
- local elastic material properties, see Section 1.2.3.

?! *Review question(s) 1.2.14.* (Configuration spaces)

Suggest suitable configuration spaces for

- a fluid model like the stationary Navier-Stokes equations (0.10.17),
- the wave propagation model based on the Helmholtz equation (0.10.26)

1.2.2 Mass-Spring Model

(Fixed coordinate system in the plane is assumed throughout this section, *cf.* Rem. 1.2.3).

Reduction to finite-dimensional configuration space

Idea: model string as a system of finitely many simple components that interact in simple ways

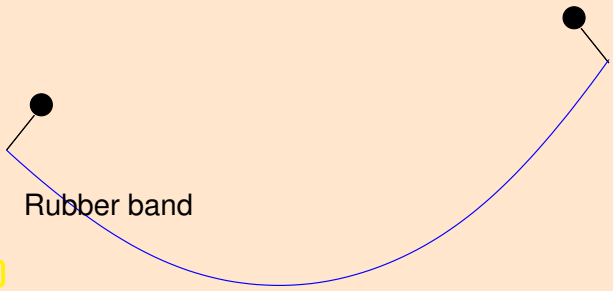


Fig. 15

∞-dimensional: elastic string

$\Longrightarrow$

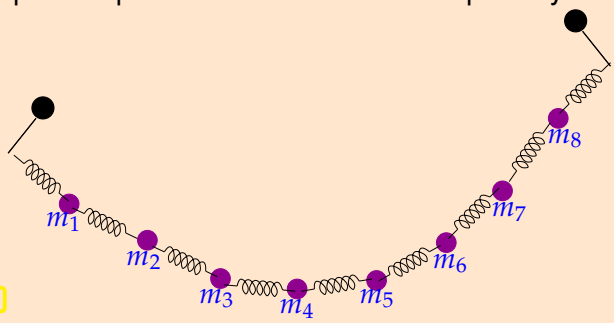


Fig. 16

n point masses connected with $n+1$ springs

Configuration space for mass-spring model:

$$\mathbf{u}^i \in \mathbb{R}^2 \mathrel{\hat{=}} \text{position of } i\text{-th mass point ("bead")}, \; i = 1, \dots, n$$

$$\rightarrow \quad \text{finite-dimensional configuration space} \quad = \quad (\mathbb{R}^2)^n$$

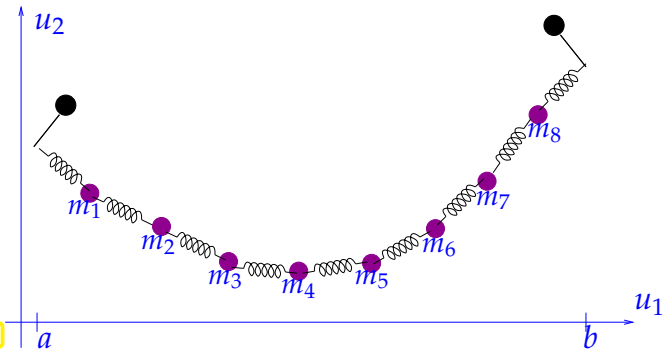


Fig. 17

Convention: We set

$$\mathbf{u}^0 := \begin{bmatrix} a \\ u_a \end{bmatrix}, \quad \mathbf{u}^{n+1} := \begin{bmatrix} b \\ u_b \end{bmatrix} \tag{1.2.16}$$

for the pinning point positions. This will simplify the formulas.

Remark 1.2.17 (Discrete models)

Models, for which configurations can be described by means of finitely many real numbers are called discrete. Hence, the mass-spring model is a discrete model, see Sect. 1.5.

The configuration of the mass-spring system with n masses can uniquely be described by specifying the $2n$ numbers u_1^i, u_2^i, $i = 1, \dots, n$. Thus the customary parlance is that the mass-spring system has $2n$ degrees of freedom.

Still missing: mathematical model for the springs; we opt for the simplest possible.

Assumption 1.2.18. Hooke's law

linear springs $\leftrightarrow$ Hooke's law: force proportional to relative elongation

$$\text{Force} \quad \boxed{F(l) = \kappa\left(\frac{l}{l_0} - 1\right)} \quad \text{(relative elongation)} . \tag{1.2.19}$$

$\kappa \mathrel{\hat{=}}$ spring constant (stiffness), $[\kappa] = 1\mathrm{N}$, $\kappa > 0$,
$l_0 \mathrel{\hat{=}}$ equilibrium length of (relaxed) spring, $[l_0] = 1\mathrm{m}$, $l_0 > 0$.

(1.2.20) Elastic energy

Recall that work can be computed by integrating a (tangential) force along a path.

From (1.2.19): elastic energy stored in linear spring at length $l > 0$

$$E_{\mathrm{el}} = \int_{l_0}^{l} F(\tau)\,\mathrm{d}\tau = \frac{1}{2}\frac{\kappa}{l_0}(l - l_0)^2 \,, \quad [E_{\mathrm{el}}] = 1\mathrm{J} \,. \tag{1.2.21}$$

The elastic energies stored in the individual springs of the mass-spring model can simply be summed up:

Total elastic energy of mass-spring model in configuration $(\mathbf{u}^1, \dots, \mathbf{u}^n) \in (\mathbb{R}^2)^n$:

$$(1.2.21) \quad \Rightarrow \quad J_{\mathrm{el}}^{(n)} = J_{\mathrm{el}}^{(n)}(\mathbf{u}^1, \dots, \mathbf{u}^n) := \frac{1}{2}\sum_{i=0}^{n} \underbrace{\frac{\kappa_i}{l_i}\left(\left\|\mathbf{u}^{i+1} - \mathbf{u}^i\right\| - l_i\right)^2}_{\text{elastic energy of } i\text{-th spring}} \,, \tag{1.2.22}$$

where $\mathbf{u}^0 := \begin{bmatrix} a \\ u_a \end{bmatrix}$, $\mathbf{u}^{n+1} := \begin{bmatrix} b \\ u_b \end{bmatrix}$ (pinning positions according to (1.2.2)),
$\kappa_i \mathrel{\hat{=}}$ spring constant of i-th spring, $i = 0, \dots, n$, see (1.2.19)
$l_i > 0 \mathrel{\hat{=}}$ equilibrium length of i-th spring.

✎ notation: $\|\mathbf{x}\| \mathrel{\hat{=}}$ Euclidean norm (length) of a vector $\mathbf{x} \in \mathbb{R}^d$, $d \in \mathbb{N}$

(1.2.23) Potential energy in external acceleration field

Ass. 1.2.12 ➢ "gravitational energy" of i-th mass $= -m_i V(\mathbf{u}^i)$

Total "gravitational energy" of mass-spring model in configuration $(\mathbf{u}^1, \dots, \mathbf{u}^n)$ due to external acceleration field:

$$J_{\mathrm{f}}^{(n)} = J_{\mathrm{f}}^{(n)}(\mathbf{u}^1, \dots, \mathbf{u}^n) := -\sum_{i=1}^{n} m_i V(\mathbf{u}^i) \,, \tag{1.2.24}$$

where m_i is mass of the i-th bead, $i = 1, \dots, n$.

The total potential energy of a mass spring system as introduced above is obtained by summing the elastic contribution (1.2.22) and gravity potential contribution (1.2.24):

$$J^{(n)} := J_{\mathrm{el}}^{(n)} + J_{\mathrm{f}}^{(n)} = \frac{1}{2}\sum_{i=0}^{n} \frac{\kappa_i}{l_i}\left(\left\|\mathbf{u}^{i+1} - \mathbf{u}^i\right\| - l_i\right)^2 - \sum_{i=1}^{n} m_i V(\mathbf{u}^i) \,. \tag{1.2.25}$$

This total potential energy is key to formulating a selection criterium for the configuration of a mass-spring system that will actually be obtained.

Equilibrium principle

Known from classical mechanics, stationary case

systems attain configuration(s) of minimal (potential) energy
$$J^{(n)}n := J_{\mathrm{el}}^{(n)} + J_{\mathrm{f}}^{(n)}$$

▶ (Global) equilibrium configuration $\mathbf{u}_*^1, \dots, \mathbf{u}_*^n$ of mass-spring system satisfies

$$(\mathbf{u}_*^1, \dots, \mathbf{u}_*^n) = \underset{(\mathbf{u}^1,\dots,\mathbf{u}^n)\in\mathbb{R}^{2n}}{\operatorname{argmin}} \; J^{(n)}(\mathbf{u}^1, \dots, \mathbf{u}^n) \; . \tag{1.2.27}$$

Example 1.2.28 (Single mass system)

Mass-spring system with only one point mass (non-dimensional $l_1 = l_2 = 1$, $\kappa_1 = \kappa_2 = 1$, $\mathbf{u}^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{u}^2 = \begin{bmatrix} 1 \\ 0.2 \end{bmatrix}$, $V(\boldsymbol{x}) = x_2$: vertical gravity)

Plot of $J^{(1)}(\mathbf{u}^1)$ ("energy surface") ▷

("Nice" energy surface: convex and tending to ∞ as $\|\mathbf{u}^1\| \to \infty$)

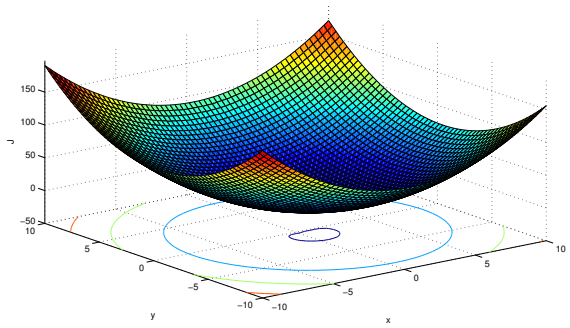


Fig. 18

Remark 1.2.29 (Non-unique solutions)

❶ solutions of (1.2.27) need not be unique !

To see this, consider the case $L := \sum_{i=0}^{n} l_i > \|\mathbf{u}^{n+1} - \mathbf{u}^0\|$ and $\mathbf{f} \equiv 0$ (slack ensemble of springs without external forcing = zero gravity). In this situation many crooked arrangements of the masses will have zero total potential energy.

Experiment 1.2.30 (Computed minimal potential energy configurations of mass-spring systems)

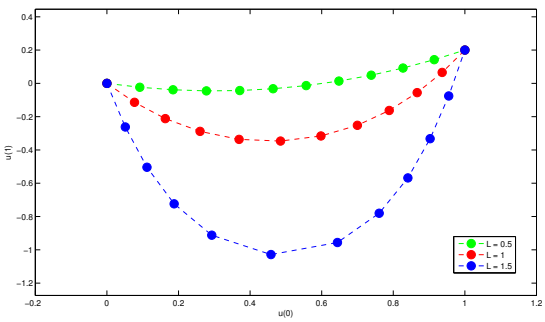


Fig. 19

◁ minimal energy configuration of a mass spring system for variable L.

($n = 10$, non-dimensional $\kappa_i = 1$, $l_i = L/n$, $i = 0, \dots, 10$, $\mathbf{f} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$)

1.2.3 Continuum Limit

Our goal is to derive a truly "∞-dimensional" mathematical model of an elastic string under external loading.

Heuristics: elastic string = spring-mass system with "infinitely many infinitesimal masses" and "infinitesimally short" springs.

(1.2.31) Continuum limit policy

- consider sequence $(\mathcal{SMS}_n)_{n\in\mathbb{N}}$ of spring-mass systems with n masses, $n \to \infty$.
- identify material coordinate ($\to$ Rem. 1.2.9) of point masses.
- choose system parameters with meaningful limits for $n \to \infty$.
- derive expressions for energies as $n \to \infty$,
- use them to define the potential energy functional of a "continuous elastic string model".

Assumption 1.2.32. Equal springs

Equal equilibrium lengths of all springs: $l_i = \dfrac{L}{n+1}, \quad L > 0,$

➢ Here, $L \mathrel{\hat{=}}$ equilibrium length of elastic string: $L = \sum_i l_i, \quad [L] = 1\mathrm{m}.$

Experiment 1.2.33 (Mass-spring equilibrium configurations with increasing number of masses)

Equilibrium configuration of mass-spring system ▷

(non-dimensional $l_i = \frac{L}{n+1}$, $\kappa_i = 1$, $m_i = \frac{1}{n}$, $V(\boldsymbol{x}) = \begin{bmatrix}0\\1\end{bmatrix} \cdot \boldsymbol{x}$, $L = 1$, n varying)

We observe a "visual limit" of the equilibrium configurations of the mass-spring systems for $n \to \infty$: positions of mass points trace out a smooth curve.

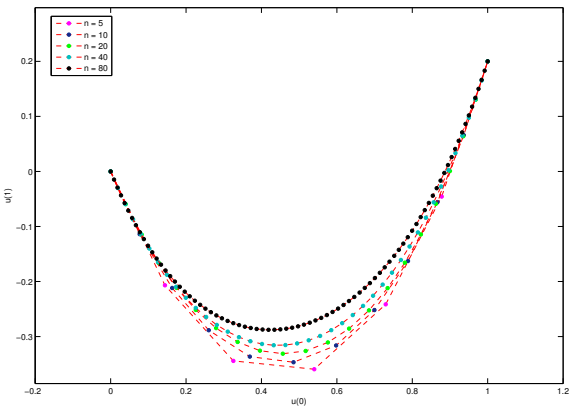


Fig. 20

(1.2.34) Connecting mass-spring model and continuum model

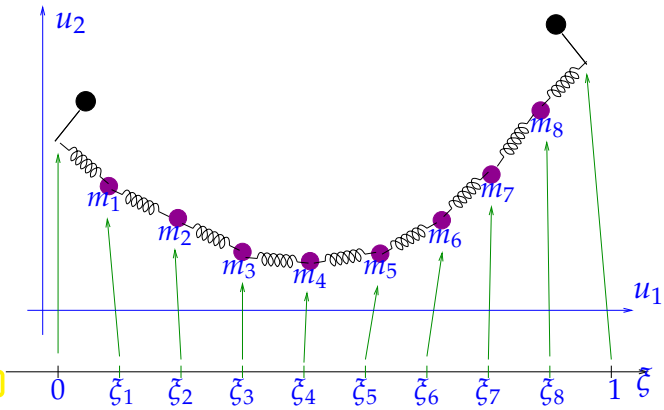


Fig. 21

masses are uniformly spaced **on** string curve $\mathbf{u} : [0,1] \mapsto \mathbb{R}^2$

➢ material coordinate of i-th mass in $\mathcal{SMS}_n$:

$$\xi_i^{(n)} := \frac{i}{n+1}: \quad \mathbf{u}^i := \mathbf{u}(\xi_i^{(n)})$$

$\mathrel{\hat{=}}$ implicit arclength parameterization in force-free state, recall § 1.2.8: constrains parameterization, removes non-uniqueness.

(1.2.35) Limit compatible choice of parameters in mass-spring model

In the spring-mass model each spring has its own stiffness κ_i and every mass point its own mass. When considering the "limit" of a sequence of spring-mass models, we have to detach stiffness and mass from springs and masses and attach them to material points, *cf.* Rem. 1.2.9. In other words, stiffness κ_i and mass m_i have to be induced by a stiffness function $\kappa(\xi)$ and mass density function $\rho(\xi)$. This linkage has to be done in a way to allow for a meaningful limit $n \to \infty$ for the potential energies.

"Limit compatible" system parameters: $(\xi_{i+1/2}^{(n)} := \frac{1}{2}(\xi_{i+1}^{(n)} + \xi_i^{(n)}))$

- $\kappa_i = \kappa(\xi_{i+1/2}^{(n)})$ with stiffness function $\kappa : [0,1] \mapsto \mathbb{R}^+$ *continuous* at $\xi_{i+1/2}^{(n)}$ (string has a stiffness at every point),

- $m_i = \int\limits_{\xi_{i-1/2}^{(n)}}^{\xi_{i+1/2}^{(n)}} \rho(\xi)\,\mathrm{d}\xi$ "lumped mass", with *integrable* mass density $\rho : [0,1] \mapsto \mathbb{R}^2$ (units $[\rho] = \frac{\mathrm{kg}}{\mathrm{m}}$)

  ► For $\rho \equiv 1$ we find $m_i \sim \frac{1}{n}$ (as in Exp. 1.2.33)

► "Limit compatible" potential energy contributions, see (1.2.22), (1.2.24):

$$J_{\mathrm{el}}^{(n)}(\mathbf{u}) = \frac{1}{2}\sum_{i=0}^{n} \frac{n+1}{L}\kappa(\xi_{i+1/2}^{(n)}) \left(\left\| \mathbf{u}(\xi_{i+1}^{(n)}) - \mathbf{u}(\xi_i^{(n)}) \right\| - \frac{L}{n+1} \right)^2 , \tag{1.2.36}$$

$$J_{\mathrm{f}}^{(n)}(\mathbf{u}) = -\sum_{i=1}^{n} \int\limits_{\xi_{i-1/2}^{(n)}}^{\xi_{i+1/2}^{(n)}} \rho(\xi)\,\mathrm{d}\xi\; V(\mathbf{u}(\xi_i^{(n)}))\ . \tag{1.2.37}$$

Assumption 1.2.38.

$\mathbf{u} \in (C^2([0,1]))^2$ (twice continuously differentiable)

(1.2.39) Calculus tools for computing limits

- Riemann sums, see [6, Sect. 6.2]: for continuous $f : [0,1] \to \mathbb{R}$

$$\int_0^1 f(\xi)\,\mathrm{d}\xi = \lim_{n\to\infty} \frac{1}{n+1} \sum_{j=0}^{n} f\left(\frac{j+1/2}{n+1}\right) . \tag{1.2.40}$$

- Taylor expansion: for $f \in C^2([0,1])$ holds

$$f(\xi) = f(\xi_0) + (\xi - \xi_0) f'(\xi_0) + \int_{\xi_0}^{\xi} f''(\tau)(\xi - \tau)\,\mathrm{d}\xi\ , \quad \forall \xi, \xi_0 \in [0,1]\ . \tag{1.2.41}$$

Also recall the Landau symbol $O(\cdot)$:

$$f(h) = O(g(h)) \quad \text{for } h \to 0 \quad \Leftrightarrow \quad |f(h)| \le C|g(h)| \quad \text{for sufficiently small } |h| \,. \tag{1.2.42}$$

It permits us to rewrite (1.2.41) as

$$f(\xi) = f(\xi_0) + (\xi - \xi_0) f'(\xi) + O(|\xi - \xi_0|^2) \,,$$

if the details of the remainder term are of no interest.

(1.2.43) Limits of contributions to potential energy

We apply the tools from § 1.2.39 to both energies in the spring-mass model for $n \to \infty$:

❶ Simple limit for potential energy due to external force (Riemann summation (1.2.40)):

$$J_{\mathrm{f}}(\mathbf{u}) = \lim_{n\to\infty} J_{\mathrm{f}}^{(n)}(\mathbf{u}) = \lim_{n\to\infty} -\sum_{i=1}^{n} \int_{\xi_{i-1/2}^{(n)}}^{\xi_{i+1/2}^{(n)}} \rho(\xi)\,\mathrm{d}\xi \cdot V(\mathbf{u}(\xi_i^n)) = -\int_0^1 \rho(\xi) V(\mathbf{u}(\xi))\,\mathrm{d}\xi \,, \tag{1.2.44}$$

where $V : \mathbb{R}^2 \to \mathbb{R}$ is the gravitational potential according to Ass. 1.2.12.

❷ Limit of elastic energy:

Tool: Taylor expansion (1.2.41): for $\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \in C^2$ with derivative $\mathbf{u}'$, $1 \gg \eta \to 0$

$$\begin{aligned} \|\mathbf{u}(\xi+\eta) - \mathbf{u}(\xi-\eta)\| &= \sqrt{(u_1(\xi+\eta) - u_1(\xi-\eta))^2 + (u_2(\xi+\eta) - u_2(\xi-\eta))^2} \\ &= \sqrt{(2u_1'(\xi)\eta + O(\eta^2))^2 + (2u_2'(\xi)\eta + O(\eta^2))^2} \\ &= 2\eta \|\mathbf{u}(\xi)'\| \sqrt{1 + O(\eta)} = 2\eta \|\mathbf{u}'(\xi)\| + O(\eta^2) \,, \end{aligned} \tag{1.2.45}$$

because

$$\sqrt{1+\tau} = 1 + \frac{1}{2}\tau + O(\tau^2) \quad \text{for} \quad \tau \to 0 \,. \tag{1.2.46}$$

Apply this to (1.2.36) with $\eta = \frac{1}{2}\frac{1}{n+1} \to 0$ for $n \to \infty$, "O-terms" vanish in the limit

$$\begin{aligned} J_{\mathrm{el}}^{(n)}(\mathbf{u}) &= \tfrac{1}{2} \sum_{i=0}^{n} \frac{n+1}{L} \kappa(\xi_{i+1/2}^{(n)}) \left(\frac{1}{n+1} \left\| \mathbf{u}'(\xi_{i+1/2}^{(n)}) \right\| + O(\frac{1}{(n+1)^2}) - \frac{L}{n+1} \right)^2 \\ &= \frac{1}{2L} \frac{1}{n+1} \sum_{i=0}^{n} \kappa(\xi_{i+1/2}^{(n)}) \left(\left\| \mathbf{u}'(\xi_{i+1/2}^{(n)}) \right\| + O(\frac{1}{n+1}) - L \right)^2 \end{aligned} \tag{1.2.47}$$

Recall that an integral can be obtained as the limit of Riemann sums, see (1.2.40),

$$q \in C^0([0,1]): \quad \lim_{n\to\infty} \frac{1}{n+1} \sum_{j=0}^{n} q(\frac{j+1/2}{n+1}) = \int_0^1 q(\xi)\,\mathrm{d}\xi \,, \tag{1.2.48}$$

which immediately gives us the limit of the elastic energy (1.2.47)

$$\Rightarrow \quad J_{\mathrm{el}}(\mathbf{u}) = \lim_{n\to\infty} J_{\mathrm{el}}^{(n)}(\mathbf{u}) = \frac{1}{2L} \int_0^1 \kappa(\xi) \left(\|\mathbf{u}'(\xi)\| - L \right)^2 \mathrm{d}\xi \,. \tag{1.2.49}$$

Equilibrium condition for elastic string model

Equilibrium condition for limit model (minimal total potential energy $J(\mathbf{u}) := J_{\mathrm{f}}(\mathbf{u}) + J_{\mathrm{el}}(\mathbf{u})$):

$$\mathbf{u}_* = \underset{\mathbf{u}\in(C^1([0,1]))^2 \,\&\, (1.2.2)}{\operatorname{argmin}} \underbrace{\int_0^1 \frac{\kappa(\xi)}{2L}\left(\|\mathbf{u}'(\xi)\| - L\right)^2 - \rho(\xi)\, V(\mathbf{u}(\xi))\,\mathrm{d}\xi}_{=:J(\mathbf{u})} \,. \tag{1.2.51}$$

total potential energy functional, $[J] = 1\mathrm{J}$

= a minimization problem in a function space !

Example 1.2.52 (Tense string without external forcing)

We pursue *model validation* of (1.2.51) by making sure that its predictions agree with observations in simple situations.

Setting:
- no external force: $\mathbf{f} \equiv 0$
- homogeneous string: $\kappa = \kappa_0 = \mathrm{const}$
- tense string: $L < \|\mathbf{u}(0) - \mathbf{u}(1)\|$
  (➢ positive elastic energy)

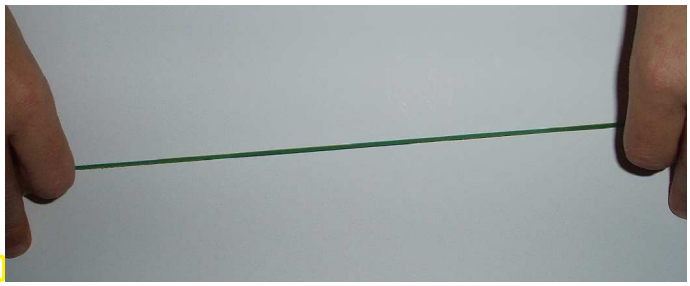


Fig. 22

$$(1.2.51) \quad \Leftrightarrow \quad \mathbf{u}_* = \underset{\mathbf{u}\in(C^1([0,1]))^2 \,\&\, (1.2.2)}{\operatorname{argmin}} \frac{\kappa_0}{2L}\int_0^1 \left(\|\mathbf{u}'(\xi)\| - L\right)^2 \mathrm{d}\xi \,. \tag{1.2.53}$$

Note: in (1.2.53) $\mathbf{u}$ enters J only through $\mathbf{u}'$!

Constraint on $\mathbf{u}'$: by triangle inequality for integrals, see [6, Sect. 6.3]

$$\ell := \|\mathbf{u}(1) - \mathbf{u}(0)\| = \left\|\int_0^1 \mathbf{u}'(\xi)\,\mathrm{d}\xi\right\| \le \int_0^1 \|\mathbf{u}'(\xi)\|\,\mathrm{d}\xi \,. \tag{1.2.54}$$

Note that $\mathbf{u}$ enters only through the norm of its derivative!

➥ Consider related minimization problem ($w \leftrightarrow \|\mathbf{u}'\|$)

$$w_* = \underset{w}{\operatorname{argmin}} \left\{ \frac{\kappa_0}{2L}\int_0^1 (w - L)^2\,\mathrm{d}\xi : \begin{array}{l} w \in C^0([0,1]) \,, \\ \int_0^1 w(\xi)\,\mathrm{d}\xi \ge \ell \end{array} \right\} . \tag{1.2.55}$$

$\Rightarrow$ unique solution $w_*(\xi) = \ell$ (constant solution)

$\|\mathbf{u}'(\xi)\| = \ell$ and the boundary conditions (1.2.2) are satisfied for the straight line solution of (1.2.53)

$$\mathbf{u}_*(\xi) = (1-\xi)\mathbf{u}(0) + \xi\mathbf{u}(1) \,.$$

It is exactly the "straight string" solution that physical intuition suggests. This solution is unique.

?! *Review question(s) 1.2.56.* (Limits of discrete models)

1. Compute the limit

$$\lim_{n\to\infty} \sum_{j=1}^{n} \sqrt{n^{-2} + \left(\sin(j/n) - \sin(j-1/n)\right)^2} \,.$$

2. Some closed curves $\mathcal{C}$ winding around zero once can be written in polar coordinates (r, φ) as

$$\mathcal{C} = \{(r,\varphi) : \; r(\varphi) = F(\varphi) \,, 0 \le \varphi \le 2\pi\} \,, \quad F(0) = F(2\pi) \,,$$

where $F : [0, 2\pi] \to \mathbb{R}^+$ is a continuous function. The area enclosed by $\mathcal{C}$ can be approximated by summing up the areas of slender triangles and taking the limit

$$A = \lim_{n\to\infty} \sum_{j=1}^{n} \tfrac{1}{2} r(2\pi j/n) r(2\pi j-1/n) \sin(2\pi/n) \,.$$

Compute an expression for the limit.

1.3 Variational Approach

We face the task of minimizing a functional over an ∞-dimensional function space. In this section necessary conditions for the minimizer will formally be derived in the form of variational equations. This idea is one of the cornerstone of a branch of analysis called calculus of variations.

We will not dip into this theory, but perform manipulations at a formal level. Yet, all considerations below can be justified rigorously.

1.3.1 Virtual Work Equation

We focus on the elastic string model introduced in Section 1.2.3. In the case of the equilibrium condition (1.2.51) we face a minimization problem for the functional $J : V \to \mathbb{R}$ given by

$$J(\mathbf{u}) := \int_0^1 \frac{\kappa(\xi)}{2L} \left(\|\mathbf{u}'(\xi)\| - L\right)^2 - \rho(\xi)\, V(\mathbf{u}(\xi))\, \mathrm{d}\xi \tag{1.2.51}$$

on the infinite-dimensional *function space*

$$V := \{\mathbf{u} \in (C^1([0,1]))^2, \; \mathbf{u} \text{ satisfies boundary conditions (1.2.2)}\} :$$
$$\text{find } \mathbf{u}_* \in V \text{ such that } \quad \mathbf{u}_* = \operatorname*{argmin}_{\mathbf{u}\in V} J(\mathbf{u}) \,.$$

Remark 1.3.1 (Necessary conditions for minimizers in finite-dimensional setting)

From finite dimensional calculus we know that all partial/directional derivatives of a continuously smooth functional $J : \mathbb{R}^n \to \mathbb{R}$ vanish at a minimum:

$$\mathbf{x}_* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} J(\mathbf{x}) \quad \Rightarrow \quad \mathbf{grad}\, J(\mathbf{x}_*) = 0 \quad \Leftrightarrow \quad \frac{\partial J}{\partial x_k}(\mathbf{x}_*) = 0\,, \quad k = 1,\dots,n\,. \tag{1.3.2}$$

We can view this from a different angle; we consider the section function $\varphi_{\mathbf{v}}(t) := J(\mathbf{x}_* + t\mathbf{v})$, $\mathbf{v} \in \mathbb{R}^d$. If $\mathbf{x}_*$ is a minimizer of J, then $\varphi_{\mathbf{v}}$ has a minimum at $t = 0$ for every $\mathbf{v}$. Necessarily, the derivative of $\varphi_{\mathbf{v}}$, $\frac{d}{dt}\varphi_{\mathbf{v}}(t) = \mathbf{grad}\, J(\mathbf{x}_* + t\mathbf{v}) \cdot \mathbf{v}$ will be zero for $t = 0$ for every $\mathbf{v}$:

$$\frac{d}{dt}\varphi_{\mathbf{v}}(0) = \boxed{\mathbf{grad}\, J(\mathbf{x}_*) \cdot \mathbf{v} = 0 \quad \forall \mathbf{v} \in \mathbb{R}^d}\,. \tag{1.3.3}$$

Hence, (1.3.2) boils down to the special case $\mathbf{v} = \mathbf{e}_k$, $\mathbf{e}_k \mathrel{\hat{=}} k$-th unit vector in $\mathbb{R}^d$, $k = 1,\dots,d$.

Note: For obvious reasons, the expression $\mathbf{grad}\, J(\mathbf{x}_*) \cdot \mathbf{v}$ is called a directional derivative of J in the direction $\mathbf{v}$.

Next we learn the analogue of (1.3.3) in an infinite-dimensional setting.

✎ notation: $C_0^k([0,1]) := \{v \in C^k([0,1]) : v(0) = v(1) = 0\}$, $k \in \mathbb{N}_0$

Main "idea of calculus of variations"

$$\mathbf{u}_* \text{ solves (1.2.51)} \quad \Rightarrow \quad J(\mathbf{u}_*) \le J(\mathbf{u}_* + t\mathbf{v}) \quad \forall t \in \mathbb{R},\ \mathbf{v} \in (C_0^2([0,1]))^2\,. \tag{1.3.5}$$

- $\varphi_{\mathbf{v}}(t) := J(\mathbf{u}_* + t\mathbf{v})$ has global minimum for $t = 0$
- If $\varphi_{\mathbf{v}}$ is differentiable, then $\dfrac{d\varphi_{\mathbf{v}}}{dt}(0) = 0$

(1.3.5) expresses the fact that $\mathbf{u}_*$ can only be a minimal energy configuration, if no admissible perturbation leads to a decrease of the total energy. We conclude this in exactly the same way as we concluded (1.3.3) in a finite-dimensional setting.

Note: We have to impose $\mathbf{v}(0) = \mathbf{v}(1) = 0$, because we must not tamper with the pinning conditions (1.2.2); they must still hold for any perturbed curve.

Rule: Variation $\mathbf{v}$ must vanish where argument function $\mathbf{u}$ is fixed.

Parlance: The computation of $\frac{d\varphi}{dt}(0)$ for J from (1.2.51) amounts to computing a "configurational derivative"/"directional derivative" in direction $\mathbf{v}$:

$$\frac{d\varphi}{dt}(0) = \lim_{t\to 0} \frac{\varphi(t) - \varphi(0)}{t} = \lim_{t\to 0} \frac{J(u_* + t\mathbf{v}) - J(\mathbf{u}_*)}{t}\,. \tag{1.3.6}$$

We pursue a separate treatment of energy contributions (This also demonstrates a simple formal approach to computing configurational derivatives.):

(1.3.7) Configurational derivatives of energies

❶ Potential energy (1.2.44) due to external force according to Ass. 1.2.12:

$$\begin{aligned}\lim_{t\to 0}\frac{J_{\mathrm{f}}(\mathbf{u}_*+t\mathbf{v})-J_{\mathrm{f}}(\mathbf{u}_*)}{t} &= -\lim_{t\to 0}\frac{1}{t}\int_0^1 \rho(\xi)\big(V(\mathbf{u}_*(\xi)+t\mathbf{v}(\xi))-V(\mathbf{u}_*(\xi))\big)\,\mathrm{d}\xi \\ &= -\int_0^1 \rho(\xi)\,\mathbf{grad}\,V(\mathbf{u}_*(\xi))\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi\,,\end{aligned} \tag{1.3.8}$$

where we used (multidimensional) Taylor expansion of V analoguous to (1.2.41)

$$\boxed{V(\mathbf{x}) = V(\mathbf{x}_0)+\mathbf{grad}\,V(\mathbf{x}_0)\cdot(\mathbf{x}-\mathbf{x}_0)+O(\|\mathbf{x}-\mathbf{x}_0\|^2) \text{ for } \mathbf{x}\to\mathbf{x}_0}\,,\quad \mathbf{x},\mathbf{x}_0\in\mathbb{R}^2\,. \tag{1.3.9}$$

❷ Computing the directional derivative of the elastic energy

$$J_{\mathrm{el}}(\mathbf{u}) = \frac{1}{2L}\int_0^1 \kappa(\xi)\big(\|\mathbf{u}'(\xi)\|-L\big)^2\,\mathrm{d}\xi\,. \tag{1.2.49}$$

is more difficult. It can be achieved using the Taylor expansion (1.2.41) as a tool, see also [6, Sect. 5.5]:

Analogous to (1.2.45), for $\mathbf{x}\in\mathbb{R}^2\setminus\{0\}$, $\mathbf{h}\in\mathbb{R}^2$, and $\mathbb{R}\ni t\to 0$ we find

$$\begin{aligned}\|\mathbf{x}+t\mathbf{h}\| &= \sqrt{(x_1+th_1)^2+(x_2+th_2)^2} = \sqrt{\|\mathbf{x}\|^2+2t\mathbf{x}\cdot\mathbf{h}+t^2\|\mathbf{h}\|^2} \\ &= \|\mathbf{x}\|\sqrt{1+2t\frac{\mathbf{x}\cdot\mathbf{h}}{\|\mathbf{x}\|^2}+t^2\frac{\|\mathbf{h}\|^2}{\|\mathbf{x}\|^2}} = \|\mathbf{x}\|+t\frac{\mathbf{x}\cdot\mathbf{h}}{\|\mathbf{x}\|}+O(t^2)\,,\end{aligned} \tag{1.3.10}$$

where we used Taylor expansion for $\sqrt{1+x}$ around 0

$$\sqrt{1+\delta} = 1+\tfrac{1}{2}\delta+O(\delta^2) \quad \text{for} \quad \delta\to 0\,. \tag{1.3.11}$$

Use (1.3.10) with $\mathbf{x}:=\mathbf{u}'(\xi)$ and $\mathbf{h}:=\mathbf{v}'(\xi)$ in the perturbation analysis for the elastic energy:

$$\begin{aligned}(\|\mathbf{u}'(\xi)+t\mathbf{v}'(\xi)\|-L)^2 &= \left(\|\mathbf{u}'(\xi)\|+t\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}+O(t^2)-L\right)^2 \\ &= \big(\|\mathbf{u}'(\xi)\|-L\big)^2+2t\big(\|\mathbf{u}'(\xi)\|-L\big)\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}+O(t^2)\,.\end{aligned}$$

$$J_{\mathrm{el}}(\mathbf{u}+t\mathbf{v})-J_{\mathrm{el}}(\mathbf{u}) = \frac{t}{L}\int_0^1 \kappa(\xi)\big(\|\mathbf{u}'(\xi)\|-L\big)\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}+O(t^2)\,\mathrm{d}\xi\,. \tag{1.3.12}$$

$$\lim_{t\to 0}\frac{J_{\mathrm{el}}(\mathbf{u}_*+t\mathbf{v})-J_{\mathrm{el}}(\mathbf{u}_*)}{t} = \int_0^1 \frac{\kappa(\xi)}{L}\big(\|\mathbf{u}'(\xi)\|-L\big)\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|}\,\mathrm{d}\xi\,. \tag{1.3.13}$$

Here we take for granted $\|\mathbf{u}'(\xi)\|\neq 0$, which is an essential property of a meaningful parameterization of the elastic string, see § 1.2.5.

Theorem 1.3.14. Minimizer solves variational equation

Necessary condition to be satisfied for $\mathbf{u}_* \in V$ *solving* (1.2.51)

$$\int_0^1 \frac{\kappa(\xi)}{L}\left(\|\mathbf{u}_*'(\xi)\| - L\right)\frac{\mathbf{u}_*'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}_*'(\xi)\|} - \rho(\xi)\ \mathbf{grad}\, V(\mathbf{u}_*(\xi))\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = 0 \quad \forall \mathbf{v} \in (C_0^2([0,1]))^2 \,. \tag{1.3.15}$$

This is a non-linear variational equation on the domain $\Omega = [0,1]$.

Example 1.3.16 (Differentiating a functional on a function space)

The directional derivative of a real-valued functional $J : V \to \mathbb{R}$ defined on a vector space V is the plain and simple derivative of a function (denoted by φ above) mapping $\mathbb{R} \mapsto \mathbb{R}$. However, differentiating this function can be challenging and usually involves multiple applications of chain rule and product rule [4, § 2.4.5].

Yet, the easiest way to do *formal* directional differentiation of functionals on a function space may rely on *repeated use of Taylor's expansion*.

Theorem 1.3.17. Multi-dimensional truncated Tayler expansion $\to$ **[6, Satz 7.5.2]**

Let $F : D \subset \mathbb{R}^n \to \mathbb{R}$, $n \in \mathbb{N}$, *a function that is* twice continuously differentiable *in* $\mathbf{x} \in \mathbb{R}^n$. *Then*

$$F(\mathbf{x}+\mathbf{h}) = F(\mathbf{x}) + \mathbf{grad}\, F(\mathbf{x})\cdot\mathbf{h} + O(\|\mathbf{h}\|^2) \quad \textit{for} \quad \mathbf{h} \to 0 \,. \tag{1.3.18}$$

Now we give an example to demonstrate the approach. For a twice continuously differentiable (C^2) function $F : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$, $d \in \mathbb{N}$, consider the functional

$$J : (C^1([0,1]))^d \mapsto \mathbb{R} \quad , \quad J(\mathbf{u}) := \int_0^1 F(\mathbf{u}'(\xi), \mathbf{u}(\xi))\,\mathrm{d}\xi \,.$$

To determine the derivative of F we rely on Thm. 1.3.17

$$F(\mathbf{u}+\delta\mathbf{u}, \mathbf{v}+\delta\mathbf{v}) = F(\mathbf{u},\mathbf{v}) + D_1F(\mathbf{u},\mathbf{v})\delta\mathbf{u} + D_2F(\mathbf{u},\mathbf{v})\delta\mathbf{v} + O(\|\delta\mathbf{u}\|^2 + \|\delta\mathbf{v}\|^2) \,. \tag{1.3.19}$$

Here, D_1F and D_2F are the partial derivatives of F w.r.t the first and second vector argument, respectively. These are *row vectors*.

$$J(\mathbf{u}+t\mathbf{v}) = J(\mathbf{u}) + t\underbrace{\int_0^1 D_1F(\mathbf{u}'(\xi),\mathbf{u}(\xi))\mathbf{v}'(\xi) + D_2F(\mathbf{u}'(\xi),\mathbf{u}(\xi))\mathbf{v}(\xi)\,\mathrm{d}\xi}_{\text{"directional derivative" } (D_\mathbf{v}J)(\mathbf{u})(\mathbf{v})} + O(t^2) \,. \tag{1.3.20}$$

The derivatives $\mathbf{u}', \mathbf{v}'$ are just regular one-dimensional derivatives w.r.t. the parameter ξ. They yield column vectors. Hence, we integrate over products of row vectors and column vectors, that is, we deal with a scalar integrand.

Remark 1.3.21 (Virtual work principle)

In statics, the derivation of variational equations from energy minimization (equilibrium principle, see (1.2.27)) is known as the method of virtual work: Small admissible changes of the equilibrium configuration of the system invariably entail *active* work, that is, energy has to be supplied to the system.

Now we unravel the structure behind the non-linear variational problem (1.3.15).

Recall essential terminology from linear algebra:

Definition 1.3.22. (Bi-)linear forms

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

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

A bilinear form 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 $u_i, v_i \in V$, $\alpha_i, \beta_i \in \mathbb{R}$, $i = 1, 2$.

✎ notation: For bilinear forms we write $\mathsf{a}(\cdot,\cdot)$, $\mathsf{b}(\cdot,\cdot)$, etc.

In the case of (1.3.15) we make a very important observation, namely that, keeping $\mathbf{u}_*$ fixed, the left hand side is a linear functional (linear form) in the test function $\mathbf{v}$:

We recall

$$\int_0^1 \frac{\kappa(\xi)}{L}\left(\|\mathbf{u}_*'(\xi)\| - L\right)\frac{\mathbf{u}_*'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}_*'(\xi)\|} - \rho(\xi)\,\mathbf{grad}\, V(\mathbf{u}_*(\xi))\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = 0 \quad \forall \mathbf{v} \in (C_0^2([0,1]))^2 : \tag{1.3.15}$$

and find the following structure of (1.3.15):

Definition 1.3.23. Non-linear variational equation

An (abstract) non-linear variational equation is an equation of the form

$$\boxed{u \in V: \quad \mathsf{a}(u; v) = 0 \quad \forall v \in V_0} \,, \tag{1.3.24}$$

where

- $V_0 \hat{=}$ is (real) *vector space* of functions,
- $V \hat{=}$ is an *affine space* of functions: $V = u_0 + V_0$, with offset function $u_0 \in V$,
- $\mathsf{a} \hat{=}$ a mapping $V \times V_0 \mapsto \mathbb{R}$ that is *linear in the second argument* v:

$$\mathsf{a}(u; \alpha v + \beta w) = \alpha \mathsf{a}(u; v) + \beta \mathsf{a}(u; w) \quad \forall u \in V \,, v, w \in V_0 \,, \alpha, \beta \in \mathbb{R} \,. \tag{1.3.25}$$

Terminology related to variational problem (1.3.24): V is called trial space

V_0 is called test space

Explanation of terminology:

- trial space $\hat{=}$ the function space in which we seek the solution
- test space $\hat{=}$ the space of eligible test functions v in a variational problem like (1.3.24) = space of admissible variations (shape perturbations) in (1.3.5).

The two spaces need not be the same: $V \neq V_0$ is common and already indicated by the notation. For many variational problem, which are not studied in this course, they may even comprise functions with different smoothness properties.

Example 1.3.26 (Elastic string model as non-linear variational problem)

In concrete terms, for elastic string continuum model in variational form (1.3.15), the entities in Def. 1.3.23 become:

✦ $V_0 := (C_0^2([0,1]))^2$ (infinite dimensional function space) ,

✦ $$V := \{\mathbf{u} \in (C^2([0,1]))^2 : \mathbf{u}(0) = \begin{bmatrix} a \\ u_a \end{bmatrix}, \mathbf{u}(1) = \begin{bmatrix} b \\ u_b \end{bmatrix}\}$$
$$= \underbrace{[\xi \mapsto (1-\xi)\mathbf{u}(0) + \xi\mathbf{u}(1)]}_{=:\mathbf{u}_0} + V_0 \,, \tag{1.3.27}$$

✦ $$\mathsf{a}(\mathbf{u};\mathbf{v}) := \int_0^1 \frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} - \rho(\xi)\,\mathbf{grad}\,V(\mathbf{u}(\xi))\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi \,. \tag{1.3.28}$$

Thus, for variational problem (1.3.15) arising from the elastic string model we find the common pattern $V = V_0 + \mathbf{u}_0$, that is, the trial space is an *affine space*, arising from the test space V_0 by adding an offset function $\mathbf{u}_0$.

Remark 1.3.29 (Offset function technique)

Assume that both V and V_0 are contained in a larger function space W.

If $V = V_0 + u_0$ with an offset function $u_0 \in W$, , that is, V is an affine space, then there is a way to recast the abstract variational problems (1.3.24) as a variational problem with the same trial and test space V_0:

$$\begin{gathered} u \in V = V_0 + u_0 : \quad \mathsf{a}(u;v) = 0 \quad \forall v \in V_0 \\ \Updownarrow \\ \widetilde{u} \in V_0 : \quad \mathsf{a}(\widetilde{u} + u_0; v) = 0 \quad \forall v \in V_0 \,. \end{gathered} \tag{1.3.30}$$

Obviously, if solutions exist and are unique, then $u = \widetilde{u} + u_0$.

➢ Now, (1.3.30) is a variational problem, for which both trial and test spaces are vector spaces.

Remark 1.3.31 (Non-linear variational problem)

Despite the fact that a in (1.3.24) is linear in the *second* argument, the variational problem (1.3.24) is generically non-linear, because a need *not* be *linear* in the first argument (and a from (1.3.28) obviously is not).

?! *Review question(s) 1.3.32.* (Calculus of variations)

1. For the following functionals $J : C^0([0,1]) \to \mathbb{R}$ determine whether they are globally differentiable on $C^0([0,1]])$. If they are not, find open subsets of $C^0([0,1])$, where they are differentiable
   (a) $J(u) = \int_0^1 u^2(x)\,\mathrm{d}x$,
   (b) $J(u) = \int_0^1 |u(x)|\,\mathrm{d}x$,
   (c) $J(u) = u(\frac{1}{2})$,
   (d) $J(u) = \int_0^1 \sqrt{1+u(x)^2}\,\mathrm{d}x$
   (e) $J(u) = \int_0^1 \cosh(u(x))\,\mathrm{d}x$
   (f) $J(u) = \int_0^1 \frac{1}{1+u(x)}\,\mathrm{d}x$
2. For the functionals given in the previous task, compute their directional derivative in direction $v \in C^0([0,1])$ and for an argument u, in which they are differentiable
3. For the functional $J(\boldsymbol{x}) := \exp(\|\boldsymbol{x}\|) - x_1$, $\boldsymbol{x} \in \mathbb{R}^2$, find the non-linear variational equation to be satisfied by the components of a global minimizer. How is this related to the non-linear system of equations spawned by (**??**)?

1.3.2 Regularity (Smoothness) Requirements

For the sake of simplicity we restrict ourselves to forces due to homogeneous gravitational field (linear potentials), see § 1.2.11.

$$V(\mathbf{x}) = \mathbf{g}\cdot\mathbf{x} \quad \text{with} \quad \mathbf{g} \in \mathbb{R}^2 \qquad \Rightarrow \quad \mathbf{grad}\, V(\mathbf{x}) = \mathbf{g}\,. \tag{1.3.33}$$

Issue: The derivation of the continuum models (1.2.51) (→ Sect. 1.2.3) and (1.3.15) was based on the assumption $\mathbf{u} \in (C^2([0,1]))^2$.

Is $\mathbf{u} \in (C^2([0,1]))^2$ required to render the minimization problem (1.2.51) or the variational problem 1.3.15) meaningful?

We will find that curves with less smoothness can still yield relevant solutions of (1.2.51)/(1.3.15).

Obvious (→ § 1.2.5): Minimal requirement $\mathbf{u} \in (C^0([0,1]))^2$
(string must not be torn)

Recall the minimization problem underlying the elastic string continuum model, here stated for homogeneous gravitational field as introduced above in (1.3.33):

$$\boxed{\mathbf{u}_* = \operatorname*{argmin}_{\mathbf{u}\in(C^1([0,1]))^2\,\&\,(1.2.2)} \int_0^1 \frac{\kappa(\xi)}{2L}\left(\|\mathbf{u}'(\xi)\| - L\right)^2 - \rho(\xi)\,\mathbf{g}\cdot\mathbf{u}(\xi)\,\mathrm{d}\xi} \,. \tag{1.3.34}$$

Observations on required smoothness

- $J(\mathbf{u})$ from (1.2.51) and a, from (1.3.28) remain well defined for merely *continuous, piecewise continuously differentiable* functions $\mathbf{u}, \mathbf{v} : [0,1] \mapsto \mathbb{R}^2$!
  ➢ In this case $\mathbf{u}'$ will be piecewise continuous and can be integrated.
- Mere integrability of κ and ρ sufficient in both (1.2.51) and (1.3.28).

(1.3.36) Piecewise C^k-functions

$C^k_{\mathrm{pw}}([a,b]) \hat{=}$ globally C^{k-1} and piecewise k-times continuously differentiable functions on $[a,b] \subset \mathbb{R}$

for each $v \in C^k_{\mathrm{pw}}([a,b])$ there is a finite partition $\{a = \tau_0 < \tau_1 < \cdots < \tau_m = b\}$ such that $v_{|]\tau_{i-1},\tau_i[}$ can be extended to a function $\in C^k([\tau_{i-1},\tau_i])$. $C^0_{\mathrm{pw}}([a,b]) \hat{=}$ piecewise continuous functions with only a finite number of discontinuities.

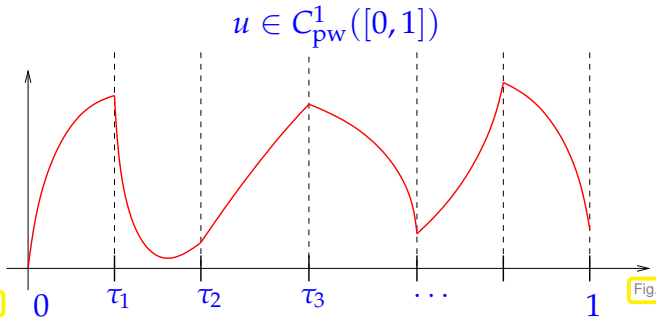


Fig. 23

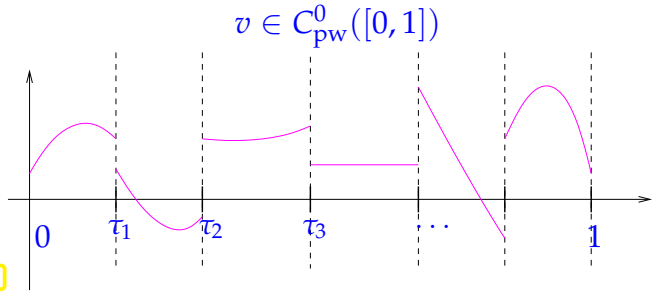


Fig. 24

Clear: If $u \in C^1_{\mathrm{pw}}([0,1])$, then $u' \in C^0_{\mathrm{pw}}([0,1])$

Example 1.3.37 (Non-smooth external forcing)

Above we discovered that piecewise differentiability of the curve parameterization still permits us to obtain a meaningful non-linear variational problem (1.3.15). Curves $\mathbf{u}_*$ with kinks can be accommodated in the variational approach. The question is, whether such curves can describe physically meaningful solutions of the elastic string continuum model. This example will give a resounding YES as an answer.

Setting:

- $\kappa = \mathrm{const}$, $\rho = \mathrm{const}$ (homogeneous string)
- Vertical force field $\mathbf{grad}\, V(\mathbf{x}) = \begin{bmatrix} 0 \\ g(x_1) \end{bmatrix}$,

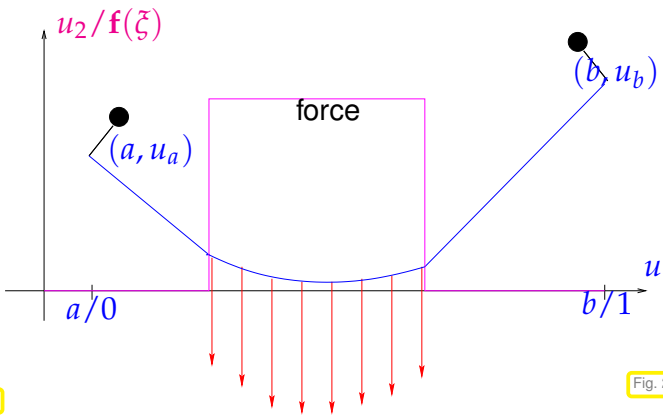


Fig. 25

✔ discontinuous g

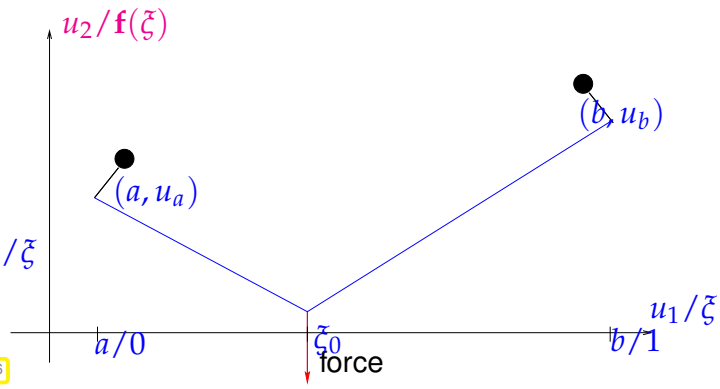


Fig. 26

✔ point force $g(x) = \delta(x - x_p)$, $a < x_p < b$

⇨ $\mathbf{u}_* \notin (C^2([0,1]))^2$ physically meaningful:

✦ $\mathbf{u}_* \in (C^1([0,1]))^2$ for discontinuous g

✦ merely $\mathbf{u}_* \in (C^0([0,1]))^2$ for point force concentrated in x_p: kink at $x_1 = x_p$!

Such solutions **must not** be ruled out by too stringent smoothness requirements on trial functions $\mathbf{u}$.

Fig. 27

Bottom line: $\mathbf{u}, \mathbf{v} \in (C^1_{\mathrm{pw}}([0,1]))^2$ right choice for variational problem (1.3.15)

1.3.3 Elastic String Differential Equation

So far we have not seen a single differential equation, though we are supposed to pursue modelling based on them! This section will disclose the connection between the variational problem from Thm. 1.3.14 and a differential equation.

Again we consider only the simplified non-linear variational equation arising from (1.3.15) when assuming linear potential (1.3.33) ($\mathbf{g} \in \mathbb{R}^2$):

$$\int_0^1 \frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} - \rho(\xi)\,\mathbf{g}\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = 0 \quad \forall \mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2 \,. \tag{1.3.38}$$

Assumption: $\boxed{\mathbf{u} \in (C^2([0,1]))^2 \;\&\; \kappa \in C^1([0,1]) \;\&\; \rho \in C^0([0,1])}$ (1.3.39)

Recall: integration by parts formula [6, Satz 6.1.2]:

$$\int_0^1 u(\xi)v'(\xi)\,\mathrm{d}\xi = -\int_0^1 u'(\xi)v(\xi) + \underbrace{\left(u(1)v(1) - u(0)v(0)\right)}_{\text{boundary terms}} \quad \forall u, v \in C^1_{\mathrm{pw}}([0,1]) \,. \tag{1.3.40}$$

Applying it to the elastic energy contribution in (1.3.15) yields

$$\int_0^1 \left(\frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|}\right)\cdot\mathbf{v}'(\xi) - \rho(\xi)\,\mathbf{g}\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi$$

$$= \int_0^1 \left\{-\frac{d}{d\xi}\left(\frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|}\right) - \rho(\xi)\,\mathbf{g}\right\}\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi \,.$$

Well, boundary terms occur prominently in (1.3.40). Where are they? Note: $\mathbf{v}(0) = \mathbf{v}(1) = 0 \quad \Rightarrow$ boundary terms vanish !

$$\text{(1.3.15)} \quad \Rightarrow \quad \int_0^1 \underbrace{\left\{ -\frac{d}{d\xi}\left(\frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|}\right) - \rho(\xi)\mathbf{g}\right\}}_{\in C^0_{\mathrm{pw}}([0,1])} \cdot \mathbf{v}(\xi)\,\mathrm{d}\xi = 0$$

$$\forall \mathbf{v} \in (C^1_0([0,1]))^2$$

Lemma 1.3.41. fundamental lemma of the calculus of variations

Let $f \in C^0_{\mathrm{pw}}([a,b])$, $-\infty < a < b < \infty$, satisfy

$$\int_a^b f(\xi)v(\xi)\,\mathrm{d}\xi = 0 \quad \forall v \in C^k([a,b]),\ v(a) = v(b) = 0\,.$$

for some $k \in \mathbb{N}_0$. Then $f \equiv 0$.

This lemma can immediately be applied to the equation obtained before:

$$\text{Ass. (1.3.39) \& (1.3.15)} \quad \overset{\text{Lemma 1.3.41}}{\Longrightarrow} \quad -\frac{d}{d\xi}\left(\frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)}{\|\mathbf{u}'(\xi)\|}\right) = \rho(\xi)\mathbf{g} \quad \forall 0 \le \xi \le 1\,.$$

Theorem 1.3.42. Differential equation for elastic string model

If $\kappa \in C^1$, $\rho \in C^0$, then a C^2-minimizer of J or a C^2-solution of (1.3.15)*, respectively, solve the 2nd-order ordinary differential equation*

$$-\frac{d}{d\xi}\left(\frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'\| - L\right)\frac{\mathbf{u}'}{\|\mathbf{u}'\|}\right) = \rho(\xi)\mathbf{g} \quad \text{on } [0;1]\,. \tag{1.3.43}$$

(1.3.44) Summary: policy for obtaining a differential equation from a variational equation

- Use integration by parts to remove all derivatives from test functions and shift them onto expressions containing only the trial function.

  Thus recast variational equation into the form

  $$u\colon \quad \int T(u)\,v\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v\,.$$

- Appeal to Lemma 1.3.41 to conclude $T(u) = 0$, which yields the differential equation. This differential equation is also known as Euler-Lagrange equation for the underlying functional J
- Boundary conditions (here = values of u at endpoints) from the definitions of the trial space.

Terminology:

ODE (1.3.43) + boundary conditions (1.2.2) = two-point boundary value problem (BPV) (on domain $\Omega = [0,1]$)

Remark 1.3.45 (Different incarnations of elastic string model)

We have arrived at different ways to state the elastic string continuum model:

Minimization problem (1.2.51)	❶ $\Rightarrow$	Variational problem (1.3.15)	❷ $\Rightarrow$	Two-point BVP
$\mathbf{u}_* = \operatorname*{argmin}_{\mathbf{v}\in V} J(\mathbf{v})$		$\mathsf{a}(\mathbf{u};\mathbf{v}) = 0 \;\forall \mathbf{v}$		$F(\mathbf{u},\mathbf{u}',\mathbf{u}'') = 0$, $\mathbf{u}(0),\mathbf{u}(1)$ fixed .

❶: equivalence ("$\Leftrightarrow$") holds if minimization problem has unique solution

❷: meaningful two-point BVP stipulates extra regularity (smoothness) of $\mathbf{u}$, see Rem. 1.3.47 below.

(1.3.46) Weak form and strong solutions of 2-point BVP

Terminology: $\left\{ \begin{array}{c} \text{minimization problem (1.2.51)} \\ \text{variational problem (1.3.15)} \end{array} \right\}$ is called the weak form of the string model,

Two-point boundary value problem (1.3.43), (1.2.2) is called the strong form of the string model.

A solution $\mathbf{u}$ of (1.3.43), for which all occurring derivatives are continuous is called a classical solution of the two-point BVP.

Remark 1.3.47 (Extra regularity requirements $\rightarrow$ Ex. 1.3.37)

Here we detail the extra smoothness requirements hinted a in Rem. 1.3.45 in order to ensure equivalence of the variational problem and the 2-point BVP arising from integration by parts for the elastic string model.

Minimization problem (1.2.51):	=	Variational problem (1.3.15):	$\neq$	Two-point BVP:
✦ κ, $\mathbf{f}$ integrable, ✦ $\mathbf{u}$ piecewise C^1		✦ κ, $\mathbf{f}$ integrable, ✦ $\mathbf{u}$ piecewise C^1		✦ $\kappa \in C^1([0,1])$, ✦ $\mathbf{f} \in (C^0([0,1]))^2$, ✦ $\mathbf{u} \in (C^2([0,1]))^2$.

☞ In light of the discussion in Ex. 1.3.37, the formulation as a classical two-point BVP imposes (unduly) restrictive smoothness on solutions and coefficient functions.

Solutions of the two-point BVP with smoothness properties as listed in the right box □ above, are called classical solutions of the elastic string continuum model.

Lemma 1.3.48. Classical solutions are weak solutions

For $\kappa \in C^1([0,1])$, any classical solution of (1.3.43) *also solves* (1.3.15).

Proof. ("Derivation of (1.3.43) reversed")

Multiply (1.3.43) with $v \in C^1_{\mathrm{pw},0}([0,1])$ and integrate over $[0,1]$. The push a derivative onto v by using (1.3.40).

□

1.4 Simplified Models for Elastic String

The variational problem of Thm. 1.3.14 is non-linear (→ Rem. 1.3.31) and posed on a space of vector valued functions. Now we discuss (approximate) simplified variants.

Again, for the sake of simplicity, we assume linear potential (1.3.33)

$$V(\mathbf{x}) = \mathbf{g}\cdot\mathbf{x} \quad \Rightarrow \quad \mathbf{grad}\, V(\mathbf{x}) = \mathbf{g}\,. \tag{1.3.33}$$

This leads to the minimization problem (1.3.34) and the non-linear variational equation (1.3.38).

1.4.1 Taut String

Taut string = elastic string stretched way beyond its equilibrium length

$$L \ll \|\mathbf{u}(0) - \mathbf{u}(1)\|\,. \tag{1.4.1}$$

Recall formula for length of a parameterized curve form Rem. 1.2.6 and the discussion in Ex. 1.2.52, in particular (1.2.54). ➥ expected: $\|\mathbf{u}_*'(\xi)\| \gg L$ for all $0 \le \xi \le 1$ for solution $\mathbf{u}_*$ of (1.2.51)

"Intuitive asymptotics":

- ✦ renormalize stiffness $\kappa \to \widetilde{\kappa} := \frac{\kappa}{L}$, $[\widetilde{\kappa}] = \mathrm{Nm}^{-1}$
- ✦ suppress equilibrium length: $L = 0$ in (1.2.51).

Simplified equilibrium model:

$$\boxed{\widetilde{\mathbf{u}}_* = \operatorname*{argmin}_{\mathbf{u}\in(C^1_{\mathrm{pw}}([0,1]))^2\,\&\,(1.2.2)} \underbrace{\int_0^1 \frac{1}{2}\widetilde{\kappa}(\xi)\left\|\mathbf{u}'(\xi)\right\|^2 - \rho(\xi)\,\mathbf{g}\cdot\mathbf{u}(\xi)\,\mathrm{d}\xi}_{=:\widetilde{J}(\mathbf{u})}}\,. \tag{1.4.2}$$

The functional (= a mapping from a function space to $\mathbb{R}$) $\widetilde{J}$ from (1.4.2) has the structure

$$\boxed{\widetilde{J}(\mathbf{u}) = \tfrac{1}{2}\mathsf{a}(\mathbf{u},\mathbf{u}) - \ell(\mathbf{u})}\,,$$

with a symmetric bilinear form

$$\mathsf{a} : V \times V \mapsto \mathbb{R} \quad , \quad \mathsf{a}(\mathbf{u},\mathbf{v}) := \int_0^1 \widetilde{\kappa}(\xi)\,\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)\,\mathrm{d}\xi\,,$$

and a linear form

$$\ell : V \mapsto \mathbb{R} \quad , \quad \ell(\mathbf{v}) := \int_0^1 \rho(\xi)\,\mathbf{g}\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi\,.$$

This makes $\widetilde{J}$ a representative of an important class of minimization problems:

Definition 1.4.3. Quadratic minimization problem

If a functional $J : V \to \mathbb{R}$ on a vector space V can be written as

$$J(v) = \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v) + \gamma$$

with a symmetric bilinear form $\mathsf{a} : V \times V \mapsto \mathbb{R}$, a linear form $\ell : V \mapsto \mathbb{R}$, and $\gamma \in \mathbb{R}$, then

$$u_* = \operatorname*{argmin}_{v \in V} J(v)$$

is called a quadratic minimization problem on V.

(1.4.2) = a quadratic minimization problem on a function space

(1.4.4) Variational problem corresponding to (1.4.2)

We use the "Taylor expansion" of the square of the Euclidean norm:

$$\|\mathbf{x} + t\mathbf{h}\|^2 = \|\mathbf{x}\|^2 + 2t\mathbf{x}\cdot\mathbf{h} + t^2\|\mathbf{h}\|^2 = \|\mathbf{x}\| + 2t\mathbf{x}\cdot\mathbf{h} + O(t^2)\ .$$

Following the same recipe as in § 1.3.7 we obtain the directional derivative:

$$\lim_{t\to 0}\frac{\widetilde{J}(\mathbf{u}+t\mathbf{v}) - \widetilde{J}(\mathbf{u})}{t} = \int_0^1 \widetilde{\kappa}(\xi)\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi) - \rho(\xi)\,\mathbf{g}\cdot\mathbf{v}(\xi)\ \mathrm{d}\xi = 0\ , \quad \forall \mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2\ .$$

Theorem 1.4.5. Variational equation for taut string model with linear potential

The solution $\widetilde{\mathbf{u}}_*$ *of* (1.4.2) *solves the variational equation*

$$\int_0^1 \widetilde{\kappa}(\xi)\mathbf{u}_*'(\xi)\cdot\mathbf{v}'(\xi)\,\mathrm{d}\xi = \int_0^1 \rho(\xi)\mathbf{g}\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi \quad \forall \mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2\ . \tag{1.4.6}$$

(1.4.7) Linear variational problems $\leftrightarrow$ **Rem. 1.3.31**

The left hand side of the variational problem (1.4.6) is not only linear in the test function argument $\mathbf{v}$, but also in the trial function argument $\mathbf{u}_*$! This distinguishes a special class of variational problems:

Definition 1.4.8. Linear variational problem

A variational problem posed on an affine space V and a vector space V_0 of the form

$$\boxed{u \in V\colon\quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0}\ , \tag{1.4.9}$$

is called a linear variational problem, if

- $\mathsf{a} : V \times V_0 \mapsto \mathbb{R}$ is a bilinear form, that is, linear in both arguments ($\to$ Def. 1.3.22),
- and $\ell : V_0 \to \mathbb{R}$ is a linear form.

In general, *quadratic* minimization problems give rise to *linear* variational problems.

This can be confirmed by an elementary computation:

$$J(\mathbf{u}) = \tfrac{1}{2}\mathsf{a}(\mathbf{u},\mathbf{u}) - \ell(\mathbf{u})$$

$$\lim_{t\to 0}\frac{J(\mathbf{u}+t\mathbf{v})-J(\mathbf{u})}{t} = \lim_{t\to 0}\frac{t\mathsf{a}(\mathbf{u},\mathbf{v})+\frac{1}{2}t^2\mathsf{a}(\mathbf{v},\mathbf{v})-t\ell(\mathbf{v})}{t} = \mathsf{a}(\mathbf{u},\mathbf{v})-\ell(\mathbf{v})\,,$$

where the bilinearity of a and the linearity of ℓ was crucial, see Def. 1.4.3.

(1.4.10) 2-point BVP corresponding to (1.4.2)

Perform integration by parts according to see (1.3.40) on (1.4.6) to shift derivatives off $\mathbf{v}$. Boundary terms do not interfere, because test functions vanish in endpoints.

$$\int_0^1 \widetilde{\kappa}(\xi)\mathbf{u}_*'(\xi)\cdot\mathbf{v}'(\xi) - \mathbf{f}(\xi)\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = \int_0^1 \left\{-\frac{d}{d\xi}\left(\widetilde{\kappa}(\xi)\frac{d}{d\xi}\mathbf{u}(\xi)\right) - \rho(\xi)\mathbf{g}\right\}\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi$$

$$\forall \mathbf{v}\in (C^1_{\mathrm{pw},0}([0,1]))^2\,.$$

Then appeal to Lemma 1.3.41.

Theorem 1.4.11. 2-point boundary value problem for taut string model with linear potential

If $\kappa\in C^1$, $\rho\in C^0$, then a C^2-solution of (1.4.6) *solves the two-point BVP*

$$\begin{aligned} -\frac{d}{d\xi}\left(\widetilde{\kappa}(\xi)\frac{d\mathbf{u}}{d\xi}(\xi)\right) &= \rho(\xi)\mathbf{g}\,, \quad 0\le\xi\le 1\,,\\ \mathbf{u}(0) = \begin{bmatrix} a\\ u_a\end{bmatrix} &\quad , \quad \mathbf{u}(1) = \begin{bmatrix} b\\ u_b\end{bmatrix}\,. \end{aligned} \tag{1.4.12}$$

Remark 1.4.13 (Vertical force)

Special setting: "gravitational force" $\mathbf{g} = -\begin{bmatrix}0\\1\end{bmatrix}$, see also Ex. 1.3.37

(1.4.2) decouples into two minimization problems for the components of $\mathbf{u}$!

$$(1.4.2) \quad\Rightarrow\quad \begin{aligned} \widetilde{u}_{1,*} &= \operatorname*{argmin}_{u\in C^1_{\mathrm{pw}}([0,1]),u(0)=a,u(1)=b} \frac{1}{2}\int_0^1 \widetilde{\kappa}(\xi)(u'(\xi))^2\,\mathrm{d}\xi\,,\\ \widetilde{u}_{2,*} &= \operatorname*{argmin}_{u\in C^1_{\mathrm{pw}}([0,1]),u(0)=u_a,u(1)=u_b} \int_0^1 \tfrac{1}{2}\widetilde{\kappa}(\xi)(u'(\xi))^2 + \rho(\xi)u(\xi)\,\mathrm{d}\xi\,. \end{aligned} \tag{1.4.14}$$

The minimization problem for $\widetilde{u}_{1,*}$ has a closed-form solution:

$$\widetilde{u}_{1,*}(\xi) = a + \frac{b-a}{\int_0^1 \widetilde{\kappa}^{-1}(\tau)\,\mathrm{d}\tau} \int_0^{\xi} \widetilde{\kappa}^{-1}(\tau)\,\mathrm{d}\tau\,, \quad 0 \le \xi \le 1\,. \tag{1.4.15}$$

This solution can easily be found by converting the minimization problem to a 2-point boundary value problem as was done above, *cf.* (1.4.6), (1.4.12). The minimization problem for $\widetilde{u}_{2,*}$ leads to the *linear* variational problem, *cf.* (1.4.6)

$$\begin{array}{c} \widetilde{u}_{2,*} \in C^1_{\mathrm{pw}}([0,1]) \\ \widetilde{u}_{2,*}(0) = u_a,\ \widetilde{u}_{2,*}(1) = u_b \end{array} : \quad \int_0^1 \widetilde{\kappa}(\xi)\widetilde{u}'_{2,*}(\xi)v'(\xi)\,\mathrm{d}\xi = -\int_0^1 \rho(\xi)v(\xi)\,\mathrm{d}\xi \quad \forall v \in C^1_{\mathrm{pw},0}([0,1])\,. \tag{1.4.16}$$

1.4.2 Function Graph Models for String Shape

We aim for a reformulation of the elastic string model in terms of a minimization problem/variational problem for a real-valued function on an interval. Focus is on the taut string limit model discussed in Section 1.4.1. We consider the special situation with a (scaled) linear potential $V(\mathbf{x}) = x_2$ which corresponds to vertical gravitational force, see (1.4.15), (1.4.16) from Rem. 1.4.13. We also define $g(\xi) := \rho(\xi)$ as a "load function".

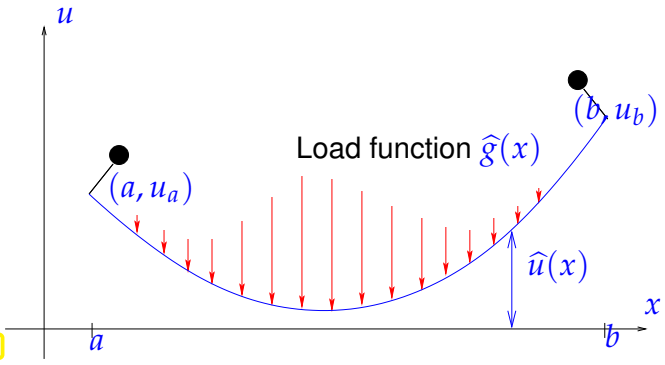


Fig. 28

Goal: Describe shape of string through graph of displacement function $\widehat{u} = \widehat{u}(x)$, $\widehat{u} : [a,b] \mapsto \mathbb{R}$, $a < b$ (physical units $[\widehat{u}] = 1\mathrm{m}$).

boundary conditions:

$$\widehat{u}(a) = u_a \quad , \quad \widehat{u}(b) = u_b\,. \tag{1.4.17}$$

$$\widehat{u}(x) = \widetilde{u}_{2,*}(\Phi^{-1}(x)) \quad \text{with} \quad \Phi(\xi) := \widetilde{u}_{1,*}(\xi)\,. \tag{1.4.18}$$

Here $\widetilde{u}_{1,*}(\xi)$, $\widetilde{u}_{2,*}(\xi)$ are the components of the curve description of the equilibrium shape of the string, see Sect. 1.2.1:

$$\mathbf{u}_*(\xi) = \begin{bmatrix} \widetilde{u}_{1,*}(\xi) \\ \widetilde{u}_{2,*}(\xi) \end{bmatrix}, \quad 0 \le \xi \le 1\,.$$

Of course, the graph description is possible only for special string shapes. It also hinges on the choice of suitable coordinates.

Assumption 1.4.19. Requirements for graph description

Assume that $\xi \mapsto \widetilde{u}_{1,*}(\xi)$ is monotone, $\widetilde{u}'_{1,*}(\xi) > 0$ for all $0 \le \xi \le 1$, $\widetilde{u}_{1,*}(0) = a$, $\widetilde{u}_{1,*}(1) = b$.

In fact, this assumption will always be satisfied in the current setting and can be proved invoking the so-called "maximum principle".

(1.4.20) Derivation of variational equation for graph model of taut elastic string

As above we set $\Phi(\xi) := \widetilde{u}_{1,*}(\xi)$ and take for granted $\Phi(0) = a < \Phi(1) = b$, $\Phi \in C^1_{\mathrm{pw}}([0,1])$, and that $\xi \to \Phi(\xi)$ is strictly monotone.

By the chain rule [6, Thm. 5.1.3]:

$$u(\xi) = \widehat{v}(\Phi(\xi)) \quad \Rightarrow \quad u'(\xi) = \frac{d\widehat{v}}{dx}(x)\Phi'(\xi)\,, \quad x := \Phi(\xi)\,. \tag{1.4.21}$$

Recall: transformation formula for integrals in one dimension (substitution rule [6, Thm. 6.1.5], $x := \Phi(\xi)$, "$\mathrm{d}x = \Phi'(\xi)\mathrm{d}\xi$"):

$$q \in C^0_{\mathrm{pw}}([0,1]): \quad \int_0^1 q(\xi)\,\mathrm{d}\xi = \int_{a=\Phi(0)}^{b=\Phi(1)} \widehat{q}(x)\left|\frac{1}{\Phi'(\Phi^{-1}(x))}\right|\mathrm{d}x\,, \quad \widehat{q}(x) := g(\Phi^{-1}(x))\,. \tag{1.4.22}$$

$\leftarrow$ (1.4.21) & (1.4.22)

Transformation of left hand bilinear form of variational problem of taut string problem described as a function of spatial coordinate x:

$$\begin{aligned}\int_0^1 \widetilde{\kappa}(\xi)\widetilde{u}'_{2,*}(\xi)v'(\xi)\,\mathrm{d}\xi &= \int_a^b \widetilde{\kappa}(\Phi^{-1}(x))\Phi'(\xi)\frac{d\widehat{u}}{dx}(x)\Phi'(\xi)\frac{d\widehat{v}}{dx}(x)\frac{1}{|\Phi'(\xi)|}\,\mathrm{d}x \\ &= \int_a^b \underbrace{\widetilde{\kappa}(\Phi^{-1}(x))|\Phi'(\Phi^{-1}(x))|}_{=:\widehat{\sigma}(x)}\frac{d\widehat{u}}{dx}(x)\frac{d\widehat{v}}{dx}(x)\,\mathrm{d}x\,.\end{aligned}$$

Transformation of right hand side linear form:

$$-\int_0^1 g(\xi)v(\xi)\,\mathrm{d}\xi = -\int_a^b \underbrace{\frac{g(\Phi^{-1}(x))}{|\Phi'(\Phi^{-1}(x))|}}_{=:\widehat{g}(x),\ [\widehat{g}]=\mathrm{Nm}^{-1}}\widehat{v}(x)\,\mathrm{d}x\,.$$

Note that in the case of pure gravitational loading, see (1.4.14), by (1.4.15) Φ is a linear function and Φ' is constant, which means that $\Phi'(\Phi^{-1}(x))$ contributes only a constant to both sides of the variational problem. These constants will be suppressed in the wake of rescaling.

Linear variational problem in physical space coordinate on spatial domain $\Omega = [a,b]$:

$$\begin{array}{c}\widehat{u}_* \in C^1_{\mathrm{pw}}([a,b]),\\ \widehat{u}_*(a) = u_a,\ \widehat{u}_*(b) = u_b\end{array} : \quad \int_a^b \widehat{\sigma}(x)\frac{d\widehat{u}_*}{dx}(x)\frac{d\widehat{v}}{dx}(x)\,\mathrm{d}x = -\int_a^b \widehat{g}(x)\widehat{v}(x)\,\mathrm{d}x \quad \forall \widehat{v} \in C^1_{\mathrm{pw},0}([a,b])\,. \tag{1.4.23}$$

(1.4.24) 2-point BPV from graph model for taut string

As in Section 1.3.3 use integration by parts (1.3.40) to remove derivatives from test function $\widehat{v}$ in (1.4.23).

(assuming $\widehat{\sigma} \in C^1([a,b])$) Scalar two-point BVP

$$(1.4.23) \quad \Rightarrow \quad \begin{cases} \dfrac{d}{dx}\left(\widehat{\sigma}(x)\dfrac{d\widehat{u}_*}{dx}(x)\right) = \widehat{g}(x)\,, \quad a \le x \le b\,, \\ \widehat{u}_*(a) = u_a \quad , \quad \widehat{u}_*(b) = u_b\,. \end{cases} \tag{1.4.25}$$

?! *Review question(s) 1.4.26.* (Two-point boundary value problems)

1. What is a quadratic minimization problem posed on an affine space V?
2. Derive the (formal) two-point boundary value problems induced by the minimization of the following functionals on $C^1_{0,\mathrm{pw}}([-1,1])$:
   (a) $J(u) := \int_{-1}^{1} |(xu)'(x)|^2 \,\mathrm{d}x$,
   (b) $J(u) := \int_{-1}^{1} u^2(x) - \beta(x)u'(x)\,\mathrm{d}x$, with a function $\beta \in C^1([-1,1])$,
   (c) $J(u) := \int_{-1}^{1} (u(x) + u'(x))^2\,\mathrm{d}x$,
   (d) $J(u) := \frac{1}{2}\int_{-1}^{1} \frac{1}{1+u'(x)^2}\,\mathrm{d}x$.
3. Which of the above functionals gives rise to a linear variational problem ($\rightarrow$ Def. 1.4.8) as a necessary conditions for its minimizers?
4. Find the solution of (1.4.25) for $\widehat{\sigma} \equiv 1$, in which case the ODE reduces to $\frac{d^2\widehat{u}}{dx^2} = 0$.
5. Determine the solution of (1.4.25) for $\widehat{\sigma} \equiv 1$, $\widehat{g} \equiv 1$, and $u_a = u_b = 0$.
6. Give an example for a variational problem (1.4.23), for which we cannot find a two-point boundary value problem (1.4.25), which has the same solution, if derivatives in (1.4.25) are read in classical sense.

1.5 Discretization

1.5.1 The Concept of Discretization

Goal: "computation" of a/the solution $\mathbf{u} : [0,1] \mapsto \mathbb{R}^2$ of $\begin{cases} \text{minimization problem (1.2.51)} \\ \text{variational problem (1.3.15)} \\ \text{two-point BVP (1.3.43) \& (1.2.2)} \end{cases}$

a function: infinite amount of information, see [4, Rem. 3.1.4].

Remark 1.5.1 (Analytic solutions)

! Well, just provide a *formula* for u (analytic solution):  in general elusive for the above problems

Only option: Numerical algorithm $\xrightarrow{\text{Computer}}$ approximate solution

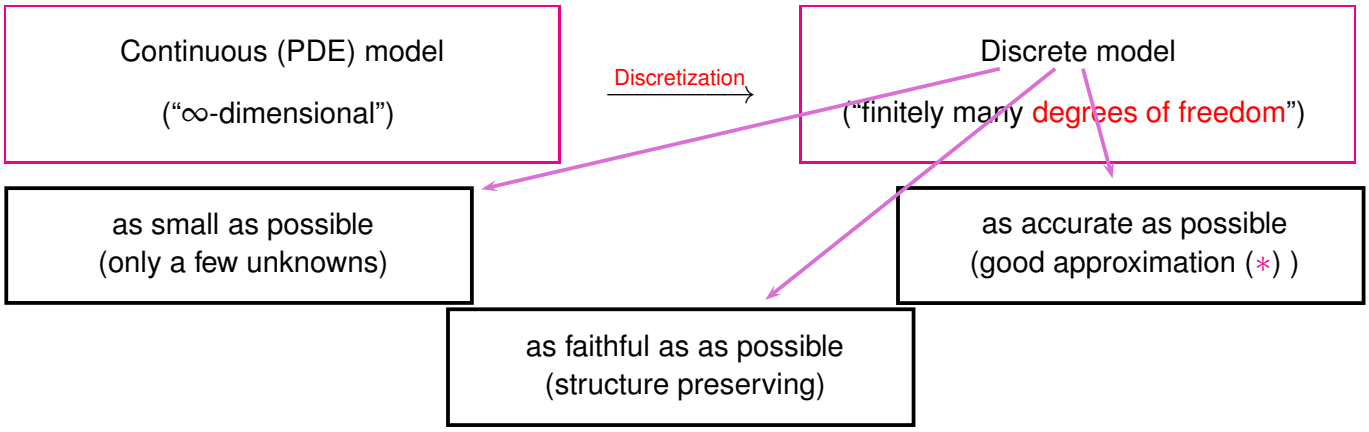
Finitely many floating point operations

Computers are finite automata ▶ Numerical algorithms can only operate on discrete models

Definition 1.5.2. Discrete model

A discrete model for a physical system/phenomenon is a model, for which both data/parameters and unknowns can be described by a finite number of real numbers.

The construction of meaningful discrete models from continuous models whose data/unknowns contain an infinite amount of information is the task of discretization:



(∗): needs a measure for quality of a solution, usually a norm of the error, error = difference of exact/analytic and approximate solution.

Parlance: number of "degrees of freedom" $\hat{=}$ number of reals required to describe discrete configuration space (usually agrees with number of "unknowns" in the discrete model.)

Remark 1.5.3 ("Physics based" discretization)

Mass-spring model (→ Section 1.2.2) = discretization of the minimization problem (1.2.51) describing the elastic string.

This discretization may be called "physics based", because it is inspired by the (physical) context of the model.

Note: Other approaches to discretization discussed below will lead to equations resembling the mass-spring model, see Section 1.5.2.2.

Remark 1.5.4 (Timestepping for ODEs)

For initial value problems for ODEs, whose solutions are functions, too, we also face the problem of discretization: timestepping methods compute a finite number of approximate values of the solutions at discrete instances in time, see [4, Chapter 11].

Remark 1.5.5 (Coefficients/data in procedural form)

For the elastic string model (→ Section 1.2.3) the stiffness $\kappa(\xi)$, and force field $\mathbf{f}$ may not be available in closed form (as formulas).

Instead they are usually given in procedural form, in MATLAB syntax as

```
function k = kappa(xi);,
function f = force(xi);,
```

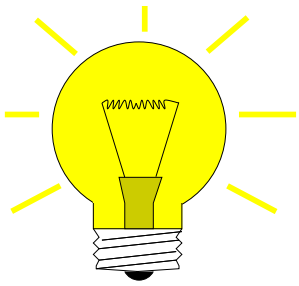
in C++ as a function with signature **double** (**double**). This might be the only way to access the coefficient functions, because they may be obtained

- as results of another computation,
- by interpolation from a table.

➤ viable discretizations must be able to deal with data in procedural form!

Preview. The remainder of this section will present a few strategies on how to derive discrete models for the problem of computing the shape of an elastic string. The different approaches start from different formulations, some target the minimization problem (1.2.51), or, equivalently, the variational problem (1.3.15), while others tackle the ODE (1.3.43) together with the boundary conditions (1.2.2).

1.5.2 Ritz-Galerkin discretization



Simple idea of first step of Ritz-Galerkin discretization

In $\left\{ \begin{array}{c} \text{minimization problem, e.g., (1.2.51)} \\ \Downarrow \\ \text{variational problem, e.g. (1.3.15)} \end{array} \right.$

replace function space V/V_0 with
finite dimensional subspace $V_N/V_{N,0}$

Note that a subscript tag $_N$ distinguishes "discrete functions/quantities", that is, functions/operators etc. that are associated with a finite dimensional space. In some contexts, N will also be an integer designating the dimension of a finite dimensional space.

Formal presentation: V, V_0 : (affine) function spaces, $\dim V_0 = \infty$,
V_N, $V_{N,0}$: subspaces $V_N \subset V$, $V_{N,0} \subset V_0$, $N := \dim V_{N,0}, \dim V_N < \infty$.

Ritz-Galerkin discretization of minimization problem for functional $J : V \mapsto \mathbb{R}$:

Continuous minimization problem		*Discrete* minimization problem
$u = \operatorname*{argmin}_{v\in V} J(v)$. (1.5.6)	$\xrightarrow{\text{Galerkin disc.}}$	$u_N = \operatorname*{argmin}_{v_N\in V_N} J(v_N)$. (1.5.7)

Ritz-Galerkin discretization of abstract (non-linear) variational problem (1.3.24), see Rem. 1.3.31

Continuous variational problem		*Discrete* variational problem
$u \in V$: $\mathsf{a}(u;v) = 0 \quad \forall v \in V_0$. (1.5.8)	$\xrightarrow{\text{Galerkin disc.}}$	$u_N \in V_N$: $\mathsf{a}(u_N;v_N) = 0$ $\forall v_N \in V_{N,0}$. (1.5.9)

Terminology: $u_N \in V_N$ satisfying (1.5.7)/(1.5.9) is called a Galerkin solution of (9.2.9)/(9.3.31)
V_N is called the (Galerkin) trial space, $V_{N,0}$ is the (Galerkin) test space.
(compare with terminology for variational equations → Def. 1.3.23)

(1.5.10) Relationship between discrete minimization problem and discrete variational problem

In Sect. 1.3.1 we discovered the implication (equivalence for unique minimizer)

Continuous minimization problem (9.2.9)	$\Longrightarrow$	*Continuous* variational problem (9.3.31)

Now it seems that we have *two different* strategies for Galerkin discretization:

1. Ritz-Galerkin discretization via the discrete minimization problem (1.5.7),
2. Ritz-Galerkin discretization based on the discrete variational problem (1.5.9).

However, the above implication extends to the discrete problems!

More precisely, we have the *commuting relationship*:

$$\begin{array}{ccc} \text{minization problem} & \xrightarrow{\text{Galerkin discretization}} & \text{discrete minimization problem} \\ \text{Configurational derivative in } V_0 \;\Big\downarrow & & \Big\downarrow\; \text{Configurational derivative in } V_{N,0} \\ \text{variational problem} & \xrightarrow{\text{Galerkin discretization}} & \text{discrete variational problem .} \end{array} \tag{1.5.11}$$

The commuting diagram means that the same discrete variational problem is obtained no matter whether

1. the minimization problem is first restricted to a finite dimensional subspace and the result is converted into a variational problem according to the recipe of Sect. 1.3.1.

2. or whether the variational problem derived from the minimization problem is restricted to the subspace.

To see this, understand that the manipulations of Sect. 1.3.1 can be carried out for infinite and finite dimensional function spaces alike.

(1.5.12) Offset functions and Ritz-Galerkin discretization

Often: $V = u_0 + V_0$, with offset function $u_0 \to$ Eq. (1.3.30), (1.3.27)

If u_0 is sufficiently simple, we may choose a trial space $V_N = u_0 + V_{N,0}$

➢ Discrete variational problem analogous to (1.3.30), same Galerkin trial and test space

$$w_N \in V_{N,0}: \quad \mathsf{a}(u_0 + w_N; v_N) = 0 \quad \forall v_N \in V_{N,0} \quad \Rightarrow \quad u_N := w_N + u_0 \,. \tag{1.5.13}$$

In the case of a linear variational problem ($\to$ Def. 1.4.8, $\mathsf{a}(u,v) = \ell(v)\ \forall v \in V_0$), that is, a bilinear form a, we have

$$(1.5.13) \quad \Leftrightarrow \quad \mathsf{a}(w_N, v_N) = \ell(v_N) - \mathsf{a}(u_0, v_N) \quad \forall v_N \in V_{N,0} \,. \tag{1.5.14}$$

Below we will always make the assumption $\boxed{V = u_0 + V_0}$ ➢ V is an affine space.

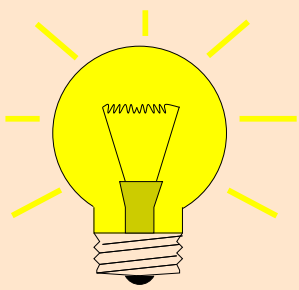
(1.5.15) Towards a (non-linear) system of equations

A computer is clueless about a concept like "finite dimensional subspace". What it can process are arrays of floating point numbers (vectors and matrices). Hence, all discretization methods must yield equations connecting vectors = systems of equations. There is a tool from linear algebra that accomplishes this.

Definition 1.5.16. Basis of a finite dimensional vector space

Let V be a real vector space. A finite subset $\{b^1, \dots, b^N\} \subset V$, $N \in \mathbb{N}$, is a basis of V, if for every $v \in V$ there are *unique* coefficients $\mu_\ell \in \mathbb{R}$, $\ell \in \{1, \dots, N\}$, such that $v = \sum_{\ell=1}^N \mu_\ell b^\ell$. Then N agrees with the dimension of V.

Idea:



- ✦ choose basis $\mathfrak{B}_N = \{b_N^1, \dots, b_N^N\}$ of $V_{N,0}$: $V_{N,0} = \mathrm{Span}\{\mathfrak{B}_N\}$
- ✦ insert basis representation into minimization problem (1.5.7)

$$v_N \in V_{N,0} \quad \Rightarrow \quad v_N = \nu_1 b_N^1 + \dots + \nu_N b_N^N \,, \quad \nu_i \in \mathbb{R} \,, \tag{1.5.17}$$

and variational equation (1.5.9)

$$v_N \in V_{N,0} \quad \Rightarrow \quad v_N = \nu_1 b_N^1 + \dots + \nu_N b_N^N \,, \quad \nu_i \in \mathbb{R} \,, \tag{1.5.18}$$

$$u_N \in V_N \quad \Rightarrow \quad u_N = u_0 + \mu_1 b_N^1 + \dots + \mu_N b_N^N \,, \quad \mu_i \in \mathbb{R} \,. \tag{1.5.19}$$

Remark 1.5.20 (Ordered basis of test space)

Once we have chosen a basis $\mathfrak{B}$ and ordered it, as already indicated in the notation above, the test space $V_{N,0}$ can be identified with $\mathbb{R}^N$: a coefficient vector $\vec{\boldsymbol{\mu}} = (\mu_1, \dots, \mu_N)^T \in \mathbb{R}^N$ provides a *unique* characterization of a function $u_N \in V_{N,0}$ (basis property stated in Def. 1.5.16)

$$u_N = \sum_{j=1}^N \mu_j b_N^j \,. \tag{1.5.21}$$

✎ notation: $\vec{\boldsymbol{v}}$, $\vec{\boldsymbol{\mu}}$ $\hat{=}$ vectors of coefficients $(\nu_i)_{i=1}^N$, $(\mu_i)_{i=1}^N$, in basis representation of functions $v_N, u_N \in V_N$ according to (1.5.17).

Discrete minimization problem

$$u_N = \operatorname*{argmin}_{v_N \in V_N} J(v_N) \,. \tag{1.5.7}$$

$\xrightarrow[\text{representation}]{\text{Basis}}$

Minimization problem on $\mathbb{R}^N$

$$\vec{\boldsymbol{\mu}} = \operatorname*{argmin}_{\vec{\boldsymbol{v}} \in \mathbb{R}^N} F(\vec{\boldsymbol{v}}) \,, \tag{1.5.22}$$

$$F(\vec{\boldsymbol{v}}) := J(u_0 + \nu_1 b_N^1 + \dots + \nu_N b_N^N) \,.$$

amenable to classical optimization techniques

Discrete variational problem

$$u_N \in V_N: \quad \mathsf{a}(u_N; v_N) = 0 \quad \forall v_N \in V_{N,0} \,. \tag{1.5.9}$$

$\xrightarrow[\text{representation}]{\text{Basis}}$

System of equations

$$\mathsf{a}(u_0 + \sum_{j=1}^N \mu_j b_N^j; b_N^k) = 0 \quad \forall k = 1, \dots, N \,. \tag{1.5.23}$$

apply numerical techniques for solving linear/non-linear systems of equations $(*)$

$(*)$: Some numerical methods designed for solving linear and non-linear systems of equations are presented in [4, Section 1.6], [4, Chapter 2].

Note that we owe the above equivalence of the discrete variational problem (left) and the system of equations (right) to the fact that $\mathsf{a}(u; v)$ is linear in its second argument, see (1.3.25). This is a consequence of the following result.

Lemma 1.5.24. Testing with basis vectors

For every linear form $\ell : V \mapsto \mathbb{R}$ ($\rightarrow$ Def. 1.3.22) on a vector space V holds

$$\ell(v) = 0 \quad \forall v \in V \quad \Leftrightarrow \quad \ell(b) = 0 \quad \forall b \in \mathfrak{B}$$

for any basis $\mathfrak{B}$ ($\rightarrow$ Def. 1.5.16) of V.

To understand, why this lemma is relevant, observe that $v \mapsto \mathsf{a}(u;v)$ is a linear form.

Every finite-dimensional vector space has infinitely many different bases. When choosing two different bases of V_N for the Galerkin discretization of a variational problem (9.3.31), then we obtain different systems of equations that have different (sets of) solution vectors. However, when forming linear combinations according to (1.5.21) we obtain the same elements of V_N.

Theorem 1.5.25. Independence of Galerkin solution of choice of basis

The choice of the basis $\mathfrak{B}$ has no impact on the (set of) Galerkin solutions u_N of (1.5.9).

(1.5.26) Recalled: Elastic string variational problems

Below, we apply Galerkin approaches to variational problems that we found in the context of the elastic string problem. We list them for easier recollection:

- (1.4.23) as an example for the treatment of a *linear* variational problem:

$$\begin{array}{c} u \in C^1_{\mathrm{pw}}([a,b]), \\ u(a) = u_a,\ u(b) = u_b \end{array} : \quad \int_a^b \sigma(x)\frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x = -\int_a^b g(x)v(x)\,\mathrm{d}x \quad \forall v \in C^1_{\mathrm{pw},0}([a,b])\ . \tag{1.4.23}$$

  Here: spatial domain $\Omega = [a,b]$, linear offset function $u_0(x) = \frac{b-x}{b-a}u_a + \frac{x-a}{b-a}u_a$,
  function space $V_0 = C^1_{\mathrm{pw},0}([a,b])$.

- (1.3.15) to demonstrate its use in the case of a non-linear variational equation:

$$\begin{array}{c} \mathbf{u} \in C^1_{\mathrm{pw}}([0,1]) \\ \mathbf{u}(0), \mathbf{u}(1) \text{ from (1.2.2)} \end{array} : \quad \int_0^1 \frac{\kappa(\xi)}{L}\left(\|\mathbf{u}'(\xi)\| - L\right)\frac{\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)}{\|\mathbf{u}'(\xi)\|} - \rho(\xi)\ \mathbf{grad}\, V(\mathbf{u}(\xi))\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = 0$$
$$\forall \mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2\ . \tag{1.3.15}$$

  Here: parameter domain $\Omega = [0,1]$, linear offset function $\mathbf{u}_0(\xi) = \xi\mathbf{u}(0) + (1-\xi)\mathbf{u}(1)$,
  function space $V_0 = (C^1_{\mathrm{pw},0}([a,b]))^2$.

?! *Review question(s) 1.5.27.* (Galerkin discretization)

1. Explain the main steps of the Ritz-Galerkin discretization of a variational problem posed on an infinite dimensional space.
2. Describe in your own words the meaning of the commuting diagram (1.5.11).

3. Let $u \in V$ solve the (continuous) minimization problem (9.2.9), and u_N be the solution of the discrete minimization problem (1.5.7). What can be said about $J(u)$ and $J(u_N)$.
4. Given a linear variational problem posed on an *affine* space $V := V_0 + u_0$, how can it be converted to a linear variational problem posed on the *vector space* V_0?
5. Assume that a variational problem posed on a finite-dimensional vector space $V_{N,0}$ has a unique solution. How does the choice of basis for $V_{N,0}$ affect the solution?

1.5.2.1 Spectral Galerkin discretization

Now we look forward to learning about the first complete Galerkin discretization of the (simplified) elastic string model.

When asked for a simple trial/test space the following will probably come to your mind (widely used for interpolation, see [4, Chapter 3], and approximation, see [4, Section 4.1.2]): for interval $\Omega \subset \mathbb{R}$

$$\boxed{\begin{aligned} V_{N,0} = {} & \mathcal{P}_p(\mathbb{R}) \cap C_0^0(\Omega) \\ & \hat{=} \text{ space of univariate polynomials of degree } \leq p \text{ vanishing at endpoints of } \Omega\,, \end{aligned}} \tag{1.5.28}$$

► $N := \dim V_N = p - 1$ [4, Section 3.2.1] for more information.

Obvious: choice (1.5.28) guarantees $V_N \subset C^1_{\mathrm{pw},0}(\Omega)$ (even $V_{N,0} \subset C^\infty(\Omega)$)

Please note that $V_{N,0}$ is a space of *global* polynomials on Ω.

Experiment 1.5.29 (Spectral Galerkin discretization of linear variational problem)

Targetted: linear variational problem (1.4.23) with

- $a = 0$, $b = 1$ ➢ domain $\Omega =]0,1[$,
- constant coefficient function $\sigma \equiv 1$,
- load $g(x) = -4\pi(\cos(2\pi x^2) - 4\pi x^2 \sin(2\pi x^2))$,
- boundary values $u_a = u_b = 0$.

► $u(x) = \sin(2\pi x^2)\,, \quad 0 < x < 1\,.$

because $\frac{d^2u}{dx^2}(x) = g(x)$.

► Concrete variational problem

$$u \in C^1_{\mathrm{pw},0}([0,1]): \quad \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x = -\int_0^1 g(x)v(x)\,\mathrm{d}x \quad \forall v \in C^1_{0,\mathrm{pw}}([0,1])\,. \tag{1.5.30}$$

Polynomial spectral Galerkin discretization, degree $p \in \{4,5,6\}$.

Plots of approximate/exact solutions ▷

Observation: Higher polynomial degree leads to better approximation in the "eyeball norm".

Note that Thm. 1.5.25 permits us to discuss the discrete solution without disclosing the actual basis of $V_{N,0}$.

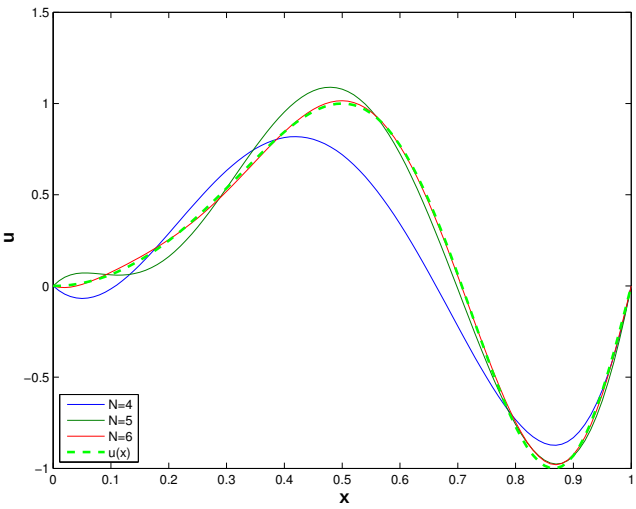


Fig. 29

Remark 1.5.31 (Choice of basis for polynomial spectral Galerkin methods)

In this remark we consider only the "reference interval" $[-1,1]$.

Sought: (ordered) basis of $V_{N,0} := C_0^1([-1,1]) \cap \mathcal{P}_p(\mathbb{R})$

❶ "Tempting": monomial-type basis

$$V_{N,0} = \mathrm{Span}\{1-x^2, x(1-x^2), x^2(1-x^2), \dots, x^{p-2}(1-x^2)\} \,. \tag{1.5.32}$$

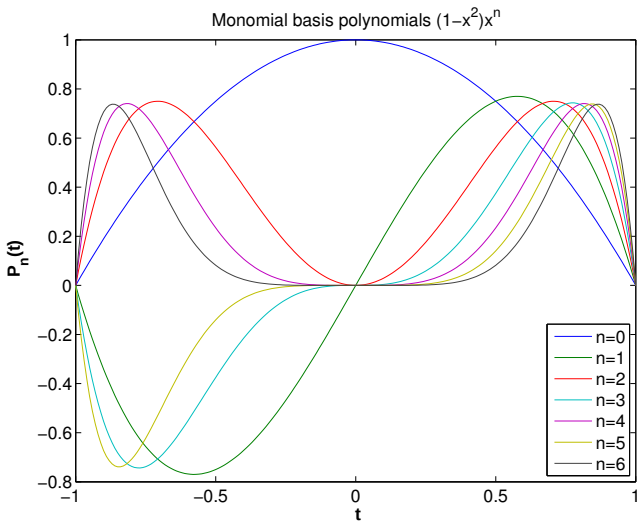


Fig. 30

◁ Monomial basis polynomials

Beware: ill-conditioned !

→ Exp. 1.5.59 below

"Visual instability": for large degree the basis functions look very much alike ↔ "almost linearly dependent".

Note: in the extreme case of linear dependence of basis functions, we certainly lose uniqueness of solutions of the (non-)linear system of equations (1.5.23).

❷ "Popular": integrated Legendre polynomials

$$V_{N,0} = \mathrm{Span}\{x \mapsto M_n(x) := \int_{-1}^{x} P_n(\tau)\,\mathrm{d}\tau,\ n = 1,\dots,p-1\} \,, \tag{1.5.33}$$

where $P_n \mathrel{\hat{=}} n$-th Legende polynomial.

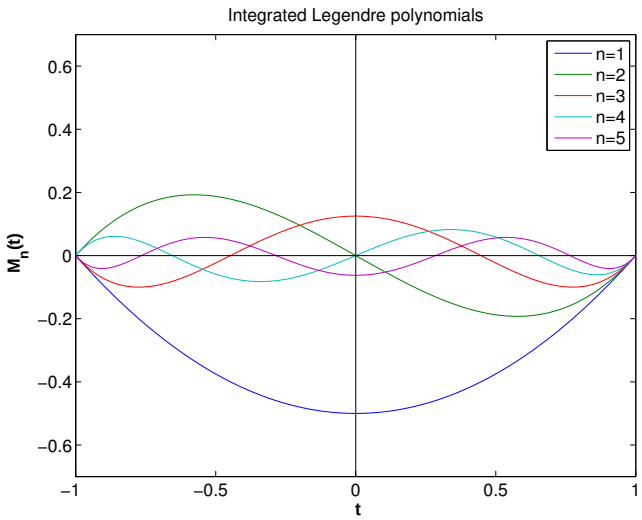


Fig. 31

◁ integrated Legendre polynomials $M_1, \dots, M_5$
"Visual stability": the basis functions are very much distinct, that is, "not nearly linearly dependent".

Integrated Legendre polynomials satisfy

$$M_n(-1) = M_n(1) = 0 \quad \forall n \in \mathbb{N} .$$

Definition 1.5.34. Legendre polynomials → **[4, Def. 5.3.26]**

The n-th Legendre polynomial P_n, $n \in \mathbb{N}_0$, is defined by (Rodriguez formula)

$$P_n(x) := \frac{1}{n!2^n} \frac{d^n}{dx^n}[(x^2-1)^n] .$$

Legendre polynomials $P_0, \dots, P_5$

$$\begin{aligned} P_0(x) &= 1 , \\ P_1(x) &= x , \\ P_2(x) &= \tfrac{3}{2}x^2 - \tfrac{1}{2} , \\ P_3(x) &= \tfrac{5}{2}x^3 - \tfrac{3}{2}x , \\ P_4(x) &= \tfrac{35}{8}x^4 - \tfrac{15}{4}x^2 + \tfrac{3}{8} . \end{aligned}$$

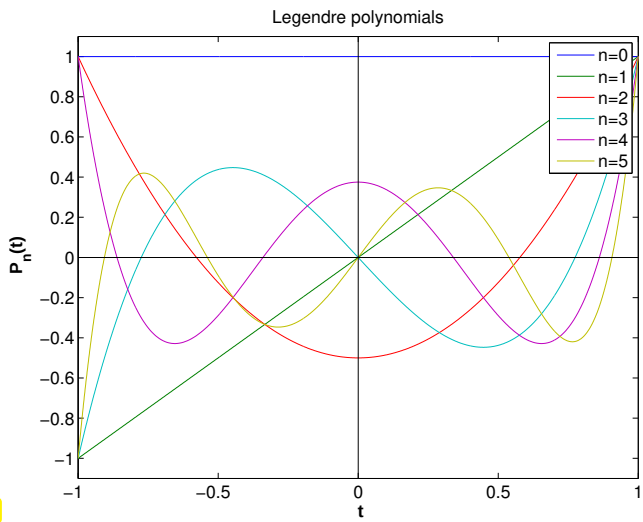


Fig. 32

Some facts about Legendre polynomials:

Lemma 1.5.35. Properties of Legendre polynomials

The Legendre polynomials P_n according to Def. 1.5.34 satisfy

- *Symmetry:*

$$P_n \quad \text{is} \begin{cases} \text{even} \\ \text{odd} \end{cases} \quad \text{for} \quad \begin{cases} \text{even } n \\ \text{odd } n \end{cases} , \quad P_n(1) = 1 , \quad P_n(-1) = (-1)^n . \tag{1.5.36}$$

- *Orthogonality*

$$\int_{-1}^{1} P_n(x)P_m(x)\mathrm{d}x = \begin{cases} \frac{2}{2n+1} & \text{, if } m = n , \\ 0 & \text{else.} \end{cases} \tag{1.5.37}$$

- *3-term recursion*

$$P_{n+1}(x) := \frac{2n+1}{n+1} xP_n(t) - \frac{n}{n+1} P_{n-1}(x) \quad , \quad P_0 := 1 , \quad P_1(x) := x . \tag{1.5.38}$$

The orthogonality (1.5.37) can be seen as a guarantee for "maximal linear independence"/"maximal stability" of a set of basis functions.

The 3-term recurrence formula (1.5.38) paves the way for an efficient evaluation of all Legendre polynomials at many (quadrature) points, see Code 1.5.40.

C++11 code 1.5.39: Computation of Legendre polynomials based on 3-term recursion (1.5.38)

```
function V= legendre(n,x)
% Computes values of Legendre polynomials up to degree n
% in the points x_j passed in the row vector x.
% Exploits the 3-term recursion (1.5.38) for Legendre polynomials
V = ones(size(x)); V = [V; x];
for j=1:n-1
  V = [V; ((2*j+1)/(j+1)).*x.*V(end,:) - j/(j+1)*V(end-1,:)];
end
```

C++ code 1.5.40: Computation of Legendre polynomials based on 3-term recursion (1.5.38)

```
// Compute Legendre polynomials P_i(x) for i = 0,···,n;
// computes values of Legendre polynomials up to degree n
// in the points x_j passed in the row vector x.
// Exploits the 3-term recursion (1.5.38) for Legendre polynomials.
// n :  degree of polynomials
// x:  Points at which the polynomials have to be computed
// return value:  Matrix of size n+1 by x.cols() where i-th row is P_i(x)
Eigen::MatrixXd
legendre(int n, const Eigen::RowVectorXd &x) {
  const std::size_t n_points = x.cols();
  Eigen::MatrixXd V = Eigen::MatrixXd::Zero(n+1, n_points);
  V.row(0) = Eigen::RowVectorXd::Ones(n_points);
  V.row(1) = x;
  for(int i = 1; i < n; i++) {
    auto tmp = (2*i+1.)/(i+1.) * x.array() * V.row(i).array()
```

```
16                                  - i/(i+1.)*V.row(i-1).array();
17      V.row(i+1) = tmp.matrix();
18    }
19    return V;
20  }
```

Code 1.5.40 relies on the EIGEN template library for numerical linear algebra to handle matrices and vectors, see [4, Section 1.2.3] for an introduction and the EIGEN home page for detailed documentation. The 3-term recursion is implemented in Line 15 using componentwise vector operation accessible via the EIGEN**array** data type.

✦ From the 3-term recursion formula (1.5.38) we can infer a particular representation of derivatives and primitives of Legendre polynomials, *cf.* Code 1.5.44

$$P_n(x) = (\tfrac{d}{dx}P_{n+1}(x) - \tfrac{d}{dx}P_{n-1}(x))/(2n+1)\,, \quad n \in \mathbb{N}\,, \tag{1.5.41}$$

$$M_n(x) = \frac{1}{2n+1}(P_{n+1}(x) - P_{n-1}(x)) \quad \text{and} \quad \frac{dM_n}{dx} = P_n\,. \tag{1.5.42}$$

C++11 code 1.5.43: Computation of (integrated) Legendre polynomials using (1.5.38) **and** (1.5.42)

```
function [V,M] = intlegpol(n,x)
% Computes values of the first n+1 Legendre polynomials P_n (returned in
% matrix V) and the first n-1 integrated Legendre polynomials
% M_n (returned in matrix M) in the points x_j passed in the
% row vector x.  Uses the recursion formulas (1.5.38) and
% (1.5.42)
V = ones(size(x)); V = [V; x];
for j=1:n-1, V = [V; ((2*j+1)/(j+1)).*x.*V(end,:) -
   j/(j+1)*V(end-1,:)]; end
M = diag(1./(2*(1:n-1)+1))*(V(3:n+1,:) - V(1:n-1,:));
```

C++ code 1.5.44: Computation of (integrated) Legendre polynomials using (1.5.38) **and** (1.5.42)

```
// Computes values of the first n+1 Legendre polynomials P_n
// and the first n-1 integrated Legendre polynomials M_n
// in the points x_j passed in the row vector x.
// Uses the recursion formulas (1.5.38) and (1.5.42)
// n:  Degree of polynomials
// x:  Points at which the polynomials have to be computed
// return value is a std::pair of matrices, with first one containing
// values of Legendre polynomials and the second one those of the
// integrated Legendre polynomials
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
intlegendrepol(int n, const Eigen::RowVectorXd &x) {
```

```
  const int n_points = x.cols();
  Eigen::MatrixXd V = legendre(n, x);
  Eigen::MatrixXd M = Eigen::MatrixXd::Zero(n-1, n_points);
  Eigen::DiagonalMatrix<double, Eigen::Dynamic> diag(n-1);
  diag.diagonal() =
    1./(2*(Eigen::VectorXd::LinSpaced(n-1,1,n-1)).array()+1);
  M = diag * (V.bottomRows(n-1) - V.topRows(n-1));
  return std::make_pair(V, M);
}
```

(1.5.45) Transformation of basis functions

The Legendre polynomials from Def. 1.5.34 are defined on $[-1,1]$. However, the variational problems (1.4.23) and (1.3.15) are defined on different domains. Can we use the Legendre polynomials on those?

The idea is borrowed from the transformation of quadrature formulas to general intervals as explained in [4, Rem. 5.1.4]. Recipe: On a "general domain $\Omega = [a,b]$", we obtain the basis function by a so-called affine transformation of the basis functions on $[-1,1]$, *cf.* [4, Rem. 5.1.4]. In the case of integrated Legendre polynomials as basis functions on $\Omega = [a,b]$ we use the basis functions

$$b_N^i(x) = M_i\left(2\frac{x-a}{b-a} - 1\right), \quad a \le x \le b\,. \tag{1.5.46}$$

Note the effect of this transformation on the derivative (chain rule!):

$$\frac{db_N^i}{dx}(x) = \frac{dM_i}{dx}\left(2\frac{x-a}{b-a} - 1\right)\cdot\frac{2}{b-a} = P_i\left(2\frac{x-a}{b-a} - 1\right)\cdot\frac{2}{b-a}\,. \tag{1.5.47}$$

(1.5.48) Spectral Galerkin discretization with quadrature

Consider the linear variational problem, *cf.* (1.4.23),

$$u \in C^1_{0,\mathrm{pw}}([a,b]): \quad \int_a^b \sigma(x)\frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x = \int_a^b g(x)v(x)\,\mathrm{d}x \quad \forall v \in C^1_{0,\mathrm{pw}}([a,b])\,. \tag{1.5.49}$$

Assume: σ, g only given in procedural form, see Rem. 1.5.5:

MATLAB: `function s = sigma(xi);` , `function g = gfunc(xi);`, or
C++: `auto sigma = [] (double xi)-> double { };`

► Analytic evaluation of integrals becomes impossible even if u, v polynomials **!**

Only remaining option: Numerical quadrature, see [4, Chapter 5]

Replace integral with m-point quadrature formula on $[a,b]$, $m \in \mathbb{N}$ $\rightarrow$ [4, Section 5.1]:

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_m(f) := \sum_{j=1}^m \omega_j^m f(\zeta_j^m) \,. \tag{1.5.50}$$

ω_j^m: quadrature weights , ζ_j^m: quadrature nodes $\in [a,b]$.

(1.5.49) & (1.5.50) $\succ$ discrete variational problem with quadrature:

$$u_N \in V_N: \quad \sum_{j=1}^m \omega_j^m \sigma(\zeta_j^m) \frac{du_N}{dx}(\zeta_j^m) \frac{dv_N}{dx}(\zeta_j^m) = \sum_{j=1}^m \omega_j^m g(\zeta_j^m) v(\zeta_j^m) \quad \forall v \in V_{N,0} \,. \tag{1.5.51}$$

A popular family of (global) quadrature formulas are the Gauss quadrature formulas [4, Section 5.3]. They enjoy the following exceptional properties:

- The m-point Gauss quadrature formula is exact for polynomials up to degree $2m-1$, that is, it features order $2m$.
- All Gauss quadrature formulas have positive quadrature weights.

Often quadrature formulas are given only on a reference interval, usually $[-1,1]$ in the case of for Gauss rules. A quadrature formula for a general interval can then be obtained by a simple affine transformation, see also [4, Rem. 5.1.4]:

$$\int_a^b f(t)\,\mathrm{d}t \approx \tfrac{1}{2}(b-a) \sum_{j=1}^n \omega_j^m f(\widehat{\zeta}_j^m) \quad \text{with} \quad \widehat{\zeta}_j^m := \tfrac{1}{2}(1-\zeta_j^m)a + \tfrac{1}{2}\zeta_j^m b \,,$$

where ω_j^m and ζ_j^m are the weights and nodes, respectively, of the quadrature rule on $[-1,1]$.

Important: Accuracy of quadrature formula and computational cost (no. m of quadrature nodes) have to be balanced, see below Code 1.5.57.

(1.5.52) Implementation of spectral Galerkin discretization for linear 2nd-order two-point BVP

Setting:

- ✦ linear variational problem (1.5.49) on $[a,b]$ $\succ$ vanishing offset function $u_0 = 0$,
- ✦ coefficients σ, g in procedural form, see Rem. 1.5.5,
- ✦ approximation of integrals by p-point Gaussian quadrature formula,
- ✦ polynomial spectral Galerkin discretization, degree $\leq p$, $p \geq 2$,
- ✦ basis $\mathfrak{B}$: integrated Legendre polynomials, see (1.5.33):

$$V_{N,0} = \mathrm{Span}\{\widetilde{M}_n,\, n = 1,\dots,p-1\} \,.$$

$\widetilde{M}_n \hat{=}$ integrated Legendre polynomials transformed to $[a,b]$ according to (1.5.46).

Trial expression using basis expansion, *cf.* (1.5.17),

$$u_N = \mu_1 \widetilde{M}_1 + \mu_2 \widetilde{M}_2 + \cdots + \mu_N \widetilde{M}_N \,, \quad \mu_i \in \mathbb{R} \,, \quad N := p-1 \,. \tag{1.5.53}$$

Note: By definition of integrated Legendre polynomials and transformation (1.5.47) of derivatives $\frac{d}{dx}\widetilde{M}_n = \frac{2}{b-a}\widetilde{P}_n$, where $\widetilde{P}_n$ is the n-th Legendre polynomial transformed to $[a,b]$ according to (1.5.46).

From (1.5.51) with (1.5.53), ζ_j^m, ω_j^m the nodes/weights of a quadrature formula on $[a,b]$,

$$\left(\tfrac{2}{b-a}\right)^2 \sum_{j=1}^{m} \omega_j^m \sigma(\zeta_j^m) \sum_{l=1}^{N} \mu_l \widetilde{P}_l(\zeta_j^m) \widetilde{P}_k(\zeta_j^m) = \underbrace{\sum_{j=1}^{m} \omega_j^m g(\zeta_j^m) \widetilde{M}_k(\zeta_j^m)}_{=:\varphi_k} \,, \quad k = 1,\dots,N \,. \tag{1.5.54}$$

$$\Updownarrow$$

$$\left(\tfrac{2}{b-a}\right)^2 \sum_{l=1}^{N} \left(\sum_{j=1}^{m} \omega_j^m \sigma(\zeta_j^m) \widetilde{P}_l(\zeta_j^m) \widetilde{P}_k(\zeta_j^m) \right) \mu_l = \varphi_k \,, \quad k = 1,\dots,N \,. \tag{1.5.55}$$

$$\Updownarrow$$

$$\boxed{\mathbf{A}\vec{\mu} = \vec{\varphi}} \quad \text{with} \quad \begin{aligned} (\mathbf{A})_{kl} &:= \left(\tfrac{2}{b-a}\right)^2 \sum_{j=1}^{m} \omega_j^m \sigma(\zeta_j^m) \widetilde{P}_l(\zeta_j^m) \widetilde{P}_k(\zeta_j^m) \,, \quad k,l = 1,\dots,N \,, \\ \vec{\mu} &= (\mu_l)_{l=1}^N \in \mathbb{R}^N \quad , \quad \vec{\varphi} = (\varphi_k)_{k=1}^N \in \mathbb{R}^N \,. \end{aligned} \tag{1.5.56}$$

A linear system of equations !

The Galerkin discretization of a *linear* variational problem always leads to a *linear* system of equations, see Section 3.2 in Chapter 2.

C++11 code 1.5.57: Polynomial spectral Galerkin solution of (1.5.49)

```
function u = lin2pbvpspecgalquad(sigma,g,N,x)
% Polynomial spectral Galerkin discretization of linear 2nd-order two-point BVP
% -d/dx(σ(x) du/dx) = g(x), u(0) = u(1) = 0 on Ω = [0,1]. Trial space of dimension N.
% Values of approximate solution in points x_j are returned in the row vector u
m = N+1;                    % Number of quadrature nodes
[zeta,w] = gaussquad(m);    % Get Gauss quadrature nodes/weights w.r.t [-1,1]
% Compute values of (integrated) Legendre polynomials at Gauss nodes
[V,M] = intlegpol(N+1,zeta');
% Note that the 2-point boudary value problem is posed on [0,1], which entails
% transforming the quadrature rule to this interval, achieved by the following
% transformation, see [4, Rem. 5.1.4] and the related Remark 1.5.45.
zeta = (zeta'+1)/2;
omega = w'.*sigma(zeta)*2;          % Modified quadrature weights
```

```
A = V(2:N+1,:)*diag(omega)*V(2:N+1,:)'; % Assemble Galerkin matrix
phi = M*(0.5*w'.*g(zeta)');        % Assemble right hand side vector
mu = A\phi;                                % Solve linear system
% Compute values of integrated Legendre polynomials at output points
[V,M] = intlegpol(N+1,2*x-1); u = mu'*M;
```

C++ code 1.5.58: Polynomial spectral Galerkin solution of (1.5.49)

```
// Polynomial spectral Galerkin discretization of linear 2nd-order
// two-point BVP
// -d/dx(σ(x) du/dx) = g(x), u(0) = u(1) = 0 on Ω = [0,1].
// Argument sigma: Function object for σ(x)
// Argument g: Function object for g(x)
// Argument N: Trial space dimension
// Argument x: Sampling points (where we want the solution)
// Return value: solution at the points in x
template<typename Function1, typename Function2>
Eigen::RowVectorXd
lin2dBVPSpecGalQuad(Function1 sigma, Function2 g, int N,
                    const Eigen::RowVectorXd &x) {
  // Get the quadrature nodes
  Eigen::RowVectorXd gauss_nodes, gauss_weights;
  std::tie(gauss_nodes, gauss_weights) = NPDE::gaussQuad(N+1);
  // Compute Legendre and integrated Legendre polynomials
  // at quadrature nodes, see Code 1.5.44
  Eigen::MatrixXd V, M;
  std::tie(V, M) = NPDE::intlegendrepol(N+1, gauss_nodes);
  // Note that the 2-point boudary value problem is posed on [0,1],
  // which entails transforming the quadrature rule to this
  // interval, which is achieved by the following transformation,
 // see [4, Rem. 5.1.4] and the related Remark 1.5.45.
  gauss_nodes = gauss_nodes.unaryExpr([](double y) { return (y+1)/2.;
      });
  // Modified quadrature weights along the diagonal matrix
  Eigen::MatrixXd omega = Eigen::MatrixXd::Zero(N+1,N+1);
  omega.diagonal() = 2*gauss_weights.cwiseProduct(
                        NPDE::apply(sigma, gauss_nodes));
  // Assemble Galerkin matrix
  Eigen::MatrixXd A = V.block(1,0,N, N+1)*omega*V.block(1,0, N,
      N+1).transpose();
  // Assemble RHS vector
  Eigen::VectorXd phi = 0.5*M*gauss_weights.cwiseProduct(
                        NPDE::apply(g,gauss_nodes)).transpose();
  // Solve the linear system of equations
  Eigen::VectorXd mu = A.lu().solve(phi);
  // Compute the value of the integrated Legendre polynomials
  // at sampling points
  std::tie(std::ignore, M) = intlegendrepol(N+1,
      x.unaryExpr([&](const double &y) { return 2*y-1.; }));
```

```
39    // Compute the approximate solution
40    return mu.transpose() * M;
41  }
```

- The function call `NPDE:gaussQuad(N)` returns nodes and weights of the N-point Gauss quadrature formula on the reference interval $[-1,1]$. They are computed by means of the Golub-Welsch algorithm [4, Rem. 5.3.34], [4, Code 5.3.35].
- The terms (1.5.56) are evaluated in parallel using compact matrix-vector operations, see Line 27, Line 30, Line 32.

Experiment 1.5.59 (Conditioning of spectral Galerkin system matrices)

Finally we can provide a rationale for preferring integrated Legendre polynomials to plain monomials for polynomial spectral Galerkin discretization: the argument is based on condition number of the system matrix from (1.5.56).

✦ Linear variational problem (1.5.30) with bilinear form

$$\mathsf{a}(u,v) = \int_0^1 \frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x\,, \quad u,v \in C^1_{\mathrm{pw},0}([0,1])\,.$$

✦ Choice of basis functions for Galerkin trial/test space $V_{N,0} := \mathcal{P}_p(\mathbb{R}) \cap C^0_0([0,1])$: monomial basis (1.5.32), integrated Legendre polynomials (1.5.33).

Monitored: condition number (w.r.t. Euclidean matrix norm → [4, Def. 1.6.15]) of Galerkin matrices ▷

Exponential increase with polynomial degree of condition number for monomial basis.

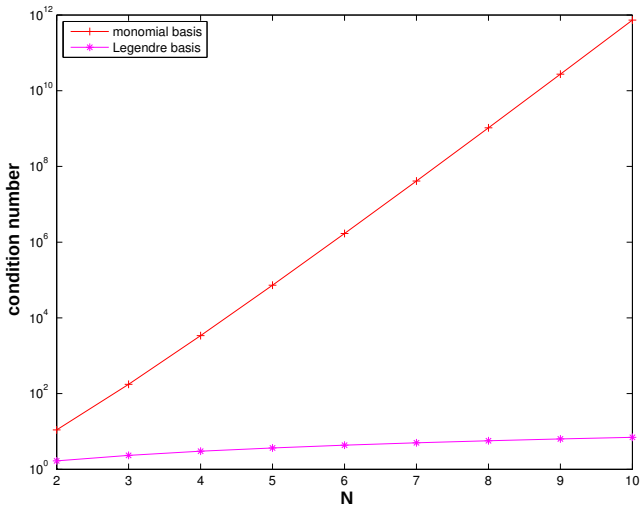


Fig. 33

Recall from [4, Section 1.6.1.2], in particular [4, Thm. 1.6.13], that a condition number of 10^m involves a loss of m digits w.r.t. the precision guaranteed for the right hand side of the linear system. Thus, using the monomial basis for $N > 10$ may no longer produce reliable results.

(1.5.60) Implementation of spectral Galerkin discretization for non-linear elastic string variational problem

Now we target the following *non-linear* variational equation on domain $\Omega = [0,1]$ arising from the equilibrium condition for the elastic string model: Seek $\mathbf{u} \in (C^1_{\mathrm{pw}}([0,1]))^2$, $\mathbf{u}(0), \mathbf{u}(1)$ fixed (pinning conditions), such that

$$\int_0^1 \frac{\kappa(\xi)}{L}\left(1 - \frac{L}{\|\mathbf{u}'(\xi)\|}\right)\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi) - \rho(\xi)\mathbf{grad}\, V(\mathbf{u}(\xi))\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = 0 \tag{1.3.15}$$

for all $\mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2$

For the sake of simplicity we suppress the dependence on $\mathbf{u}$ of the forces, and replace (1.3.15) with Seek $\mathbf{u} \in (C^1_{\mathrm{pw}}([0,1]))^2$, $\mathbf{u}(0), \mathbf{u}(1)$ fixed, such that

$$\int_0^1 \frac{\kappa(\xi)}{L}\left(1 - \frac{L}{\|\mathbf{u}'(\xi)\|}\right)\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi) - \mathbf{f}(\xi)\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi = 0 \quad \forall \mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2\,, \tag{1.3.15}$$

with a suitable force function $\mathbf{f} \in (C^0_{\mathrm{pw}}([0,1]))^2$.

- Data κ, $\mathbf{f}$ given in procedural form, see Rem. 1.5.5.
- Spectral Galerkin discretization of "curve space" $(C^1_{\mathrm{pw},0}([0,1]))^2$: *component-wise discretization*

  Represent each component of $\mathbf{u}$ based on the basis functions $\mathfrak{B} = \left\{\widetilde{M}_n\right\}_{n=1}^K$, $K \in \mathbb{N}$, of integrated Legendre polynomials (transformed to $[0,1]$, *cf.* Rem. 1.5.45), see (1.5.33). This means that as basis we use

  $$\mathfrak{B} := \left\{\begin{bmatrix}\widetilde{M}_1\\0\end{bmatrix},\dots,\begin{bmatrix}\widetilde{M}_K\\0\end{bmatrix},\begin{bmatrix}0\\\widetilde{M}_1\end{bmatrix},\dots\begin{bmatrix}0\\\widetilde{M}_K\end{bmatrix}\right\}, \quad \sharp\mathfrak{B} = 2K\,.$$

  As in the linear case, the rationale for using these basis functions is their excellent stability properties, see Ex. 1.5.59, along with the possibility of fast evaluation by means of the 3-term recurrence (1.5.38), *cf.* Code 1.5.40.

  ➢ basis representation, *cf.* (1.5.53)

  $$\mathbf{u}_N(\xi) = \underbrace{\mathbf{u}(0)(1-\xi) + \mathbf{u}(1)\xi}_{=:\mathbf{u}_0(\xi)\ \text{(offset function)}} + \begin{bmatrix}\mu_1\\\mu_{K+1}\end{bmatrix}\widetilde{M}_1(\xi) + \dots + \begin{bmatrix}\mu_K\\\mu_{2K}\end{bmatrix}\widetilde{M}_K(\xi)\,. \tag{1.5.61}$$

- Approximate evaluation of integrals by m-point Gaussian quadrature on $[0,1]$, $m := K+1$, (Below we write ζ_j for the nodes and ω_j for the weights, $j = 1,\dots,m$).

► Discrete variational problem with m-point Gauss quadrature on $[0,1]$ (nodes ζ_j and weights ω_j):

$$\sum_{j=1}^m \omega_j \frac{\kappa(\zeta_j)}{L}\left(1 - \frac{L}{\|\mathbf{u}_N'(\zeta_j)\|}\right)\mathbf{u}_N'(\zeta_j)\cdot\mathbf{v}_N'(\zeta_j) - \mathbf{f}(\zeta_j)\cdot\mathbf{v}_N(\zeta_j) = 0 \quad \forall \mathbf{v}_N \in \mathrm{Span}\{\mathfrak{B}\}\,. \tag{1.5.62}$$

In analogy to (1.5.54) we arrive at the *non-linear system of equations*: ($M_k' = 2P_k$ because of transformation, see § 1.5.45, (1.5.47)!)

$$x_1\text{-component} \quad \left(\text{test (1.5.62) with } \begin{bmatrix}M_k\\0\end{bmatrix} \mathrel{\hat{=}} k\text{-th basis function}\right)$$

$$\sum_{j=1}^{m} s_j(b-a+\sum_{l=1}^{K}\mu_l 2P_l(\zeta_j))\cdot 2P_k(\zeta_j) = \sum_{j=1}^{m}\omega_j f_1(\zeta_j)\cdot M_k(\zeta_j)\,, \quad k=1,\dots,K\,,$$

x_2-component (test (1.5.62) with $\begin{bmatrix}0\\M_k\end{bmatrix} \hat{=} K+k$-th basis function):

$$\sum_{j=1}^{m} s_j(u_b-u_a+\sum_{l=1}^{K}\mu_{K+l} 2P_l(\zeta_j))\cdot 2P_k(\zeta_j) = \sum_{j=1}^{m}\omega_j f_2(\zeta_j)\cdot M_k(\zeta_j)\,, \quad k=1,\dots,K\,,$$

$$\text{with} \qquad s_j := \omega_j\kappa(\zeta_j)\left(\frac{1}{L}-\frac{1}{\|\mathbf{u}_N'(\zeta_j)\|}\right) \quad (s_j = s_j(\vec{\mu})\ !)\,.$$

This amounts to $N := 2K$ equations for the N unknowns $\mu_1,\dots,\mu_N$ as introduced in (1.5.61). Rewrite in compact form:

$$\boxed{\begin{pmatrix}\mathbf{R}(\vec{\mu}) & 0\\ 0 & \mathbf{R}(\vec{\mu})\end{pmatrix}\vec{\mu} = \begin{pmatrix}\vec{\varphi}_1(\vec{\mu})\\ \vec{\varphi}_2(\vec{\mu})\end{pmatrix}}\,, \tag{1.5.63}$$

$$\text{with} \quad \mathbf{R}(\vec{\mu})\in\mathbb{R}^{K,K}\,, \quad (\mathbf{R}(\vec{\mu}))_{k,l} := \sum_{j=1}^{m} 4s_j(\vec{\mu})P_l(\zeta_j)P_k(\zeta_j)\,,$$

$$(\vec{\varphi}_1(\vec{\mu}))_k = \sum_{j=1}^{m}\omega_j f_1(\zeta_j)\cdot M_k(\zeta_j) - 2(b-a)\sum_{j=1}^{m} s_j(\vec{\mu})P_k(\zeta_j)\,,$$

$$(\vec{\varphi}_2(\vec{\mu}))_k = \sum_{j=1}^{m}\omega_j f_2(\zeta_j)\cdot M_k(\zeta_j) - 2(u_b-u_a)\sum_{j=1}^{m} s_j(\vec{\mu})P_k(\zeta_j)\,.$$

The non-linear system of equations (1.5.63) has to be solved iteratively [4, § 2.1.2]. Newton's method [4, Section 2.4] would be an option and the reader is invited to find the corresponding Newton iteration [4, Eq. (2.4.1)]. However, here we prefer a simpler and widely used strategy to solve (1.5.63), a fixed point iteration, which arises from "freezing" the coefficients $\vec{\mu}$ in the matrices $\mathbf{R}(\vec{\mu})$:

Initial guess $\vec{\mu}^{(0)}\in\mathbb{R}^N$; $k=0$;
repeat
$k\leftarrow k+1$;
Solve the *linear* system of equations $\begin{pmatrix}\mathbf{R}(\vec{\mu}^{(k-1)}) & 0\\ 0 & \mathbf{R}(\vec{\mu}^{(k-1)})\end{pmatrix}\vec{\mu}^{(k)} = \begin{pmatrix}\vec{\varphi}_1(\vec{\mu}^{(k-1)})\\ \vec{\varphi}_2(\vec{\mu}^{(k-1)})\end{pmatrix}$;
until $\left\|\vec{\mu}^{(k)}-\vec{\mu}^{(k-1)}\right\| \le \mathrm{tol}\cdot\left\|\vec{\mu}^{(k)}\right\|$

C++11 code 1.5.64: Polynomial spectral Galerkin discretization of elastic string variational problem

```
function [vu,figsol] = stringspecgal(kappa,f,L,u0,u1,K,xi,tol)
% Solving the non-linear variational problem (1.3.15) for the elastic
  string by means of polynomial
% spectral Galerkin discretization based on K integrated Legendre
  polynomials.  Approximate
% evaluation of integrals by means of Gaussian quadrature.
% kappa, f are handles of type @(xi) providing the coefficient function
% κ and the force field f.  The column vectors u0 and u1 pass the
% pinning points.  M is the number of mesh cells, tol specifies the
```

```
tolerance for the
% fixed point iteration.  return value:  2 × length(xi)-matrix of node
% positions for curve parameter values passed in the row vector xi.
if (nargin < 8), tol = 1E-2; end
m = K+1;                          % Number of quadrature nodes
[zeta,w] = gaussquad(m); % Obtain Gauss quadrature nodes w.r.t [−1,1]
% Compute values of (integrated) Legendre polynomials at Gauss nodes
  and evaluation points
[V,M] = intlegpol(m,zeta');
[Vx,Mx] = intlegpol(m,2*xi-1); Mx = [1-xi;Mx;xi]; %
% Compute right hand side based on m-point Gaussian quadrature on [0,1].
force = f((zeta'+1)/2); phi = M*(0.5*[w';w'].*force)';
sv = kappa((zeta'+1)/2); % Values of coefficient function κ at Gauss
  points in [0,1].
% mu is an 2 × (K+2)-matrix, containing the vectorial basis expansion
  coefficients
% of u_N.  The first and last column are contributions of the two
  functions
% ξ ↦ (1−ξ) and ξ ↦ ξ, which represent the offset function.
% Initial guess for fixed point iteration:  straight string
mu = [u0,zeros(2,K),u1];
figsol = figure; hold on;
for k=1:100  % loop for fixed point iteration, maximum 100 iterations
% Plot shape of string
  vu = mu*Mx; plot(vu(1,:),vu(2,:),'--g'); drawnow;
  title(sprintf('K = %d, iteration #%d',K,k));
  xlabel('{\bf x_1}'); ylabel('{\bf x_2}');
% Compute values of derivatives of u_N and ‖u'_N‖ at Gauss points
  up = mu(:,2:K+1)*V(2:K+1,:) + repmat(u1-u0,1,m);
  lup = sqrt(up(1,:).^2 + up(2,:).^2);
  s = 0.5*(w').*sv.*(1/L - 1./lup);  % Initialization of s_j
% Modification of right hand side due to offset function
  phi1 = phi(:,1) - (2*(u1(1)-u0(1))*V(2:K+1,:)*s');
  phi2 = phi(:,2) - (2*(u1(2)-u0(2))*V(2:K+1,:)*s');
% Assemble K × K-matrix blocks R of linear system, see also Code 1.5.57
  R = 4*V(2:K+1,:)*diag(s)*V(2:K+1,:)';
  mu_new = [u0,[(R\phi1)';(R\phi2)'],u1];
% Check simple termination criterion for fixed point iteration.
  if (norm(mu_new - mu,'fro') < tol*norm(mu_new,'fro')/K)
    vu = mu_new*Mx; fig = plot(vu(1,:),vu(2,:),'r--');
    legend(fig,'spectral Galerkin
      solution','location','southeast');break; end
  mu = mu_new;
end
```

C++ code 1.5.65: Polynomial spectral Galerkin discretization of elastic string variational problem

```
// Solving the non-linear variational problem (1.3.15) for the elastic
   string by means of polynomial
// spectral Galerkin discretization based on K integrated Legendre
   polynomials.  Approximate
```

```
// evaluation of integrals by means of Gaussian quadrature.
// kappa, f are arguments of a function type providing the coefficient function
// κ and the force field f.  The column vectors u0 and u1
// pass the pinning points,tol specifies the tolerance for the
// fixed point iteration.
// Return value:  2 × length(xi)-matrix of node
// positions for curve parameter values passed in the row vector xi.
template<typename Function1, typename Function2>
Eigen::MatrixXd
stringspecgal(int K, double L, double tol, const Eigen::VectorXd &u0, const Eigen::VectorXd &u1,
              Function1 kappa, Function2 f, const Eigen::RowVectorXd &xi) {
  int m = K+1;   // Number of quadrature nodes
  // Obtain Gauss quadrature nodes w.r.t [−1,1]
  Eigen::RowVectorXd gauss_nodes, gauss_weights;
  std::tie(gauss_nodes, gauss_weights) = gaussQuad(m);
  // Compute values of (integrated) Legendre polynomials
  // at Gauss nodes and evaluation points
  Eigen::MatrixXd V, M;  std::tie(V, M) = intlegendrepol(m, gauss_nodes);
  auto res= intlegendrepol(m, (2*xi.array()-1).matrix());
  Eigen::MatrixXd Vx = res.first;
  Eigen::MatrixXd Mx = Eigen::MatrixXd::Zero(m+1, xi.cols());

  Mx.row(0) = (1 - xi.array()).matrix();
  Mx.block(1, 0, m-1, xi.cols()) = res.second;
  Mx.row(m) = xi;

  // Scale the m-pointGaussian quadrature on [0,1].
  gauss_nodes = ((gauss_nodes.array()+1)/2).matrix();

  // Compute the right hand side
  Eigen::MatrixXd force = NPDE::apply(f, gauss_nodes);
  Eigen::MatrixXd phi   = Eigen::MatrixXd::Zero(m-1, 2);
  phi.col(0) =  0.5 * M * (gauss_weights.cwiseProduct(force.row(0))).transpose();
  phi.col(1) =  0.5 * M * (gauss_weights.cwiseProduct(force.row(1))).transpose();

  // Values of coefficient function κ at Gauss points in [0,1].
  // mu is an 2×(K+2)-matrix, containing the vectorial basis expansion coefficients
  // of u_N.  The first and last column are contributions of the two functions
  // ξ ↦ (1−ξ) and ξ ↦ ξ, which represent the offset function.
  Eigen::MatrixXd sv = apply(kappa, gauss_nodes);

  // Initial guess for fixed point iteration:  straight string
  Eigen::MatrixXd mu(2, K+2);
```

```
  mu << u0 , Eigen::MatrixXd::Zero(2, K) , u1;
  Eigen::MatrixXd vu = mu * Mx;

  // Loop through at most 100 iterations
  for(int iter = 0; iter < 100; iter++) {
    vu = mu * Mx;
    // Compute values of derivatives of u_N and ||u'_N|| at Gauss points
    Eigen::MatrixXd up =
       mu.block(0,1,2,K)*V.block(1,0,K,K+1)+(u1-u0).rowwise().replicate(K+1);
    Eigen::MatrixXd lup = (up.row(0).cwiseAbs2() +
       up.row(1).cwiseAbs2()).cwiseSqrt();

    // Initialization of s_j
    Eigen::RowVectorXd s = (0.5 * gauss_weights.array() * sv.array()
       * (1./L - 1/lup.array())).matrix();

    // Modification of right hand side due to offset function
    Eigen::VectorXd phi1 = phi.col(0) - 2*(u1(0) - u0(0)) *
       (V.block(1, 0, K, K+1) * s.transpose());
    Eigen::VectorXd phi2 = phi.col(1) - 2*(u1(1) - u0(1)) *
       (V.block(1, 0, K, K+1) * s.transpose());

    // Assemble K×K-matrix blocks R of linear system,
    // see also Code 1.5.58
    Eigen::MatrixXd omega = Eigen::MatrixXd::Zero(K+1, K+1);
    omega.diagonal() = s;
    Eigen::MatrixXd R = 4 * V.block(1, 0, K, K+1) * omega *
       V.block(1, 0, K, K+1).transpose();

    // Solve the system of equations and build result vector
    Eigen::MatrixXd sol(2, K);
    sol << R.ldlt().solve(phi1).transpose() ,
       R.ldlt().solve(phi2).transpose();

    // Add boundary conditions
    Eigen::MatrixXd mu_new(2, K+2);
    mu_new << u0 ,sol, u1;
    // Compute relative error
    double rel_error = (mu_new - mu).norm()/mu_new.norm() * K;
    mu = mu_new;
    // Check termination condtion for fixed point iterations
    if(rel_error < tol ) { vu = mu_new*Mx; break; }
  }
  return vu;
}
```

Experiment 1.5.66 (Spectral Galerkin computation of elastic string shape)

Test of polynomial spectral Galerkin method for elastic string problem, algorithm of § 1.5.60, Code 1.5.64, with

- pinning positions $\mathbf{u}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{u}(1) = \begin{bmatrix} 1 \\ 0.2 \end{bmatrix}$,
- equilibrium length $L = 0.5$,
- constant coefficient function $\kappa \equiv 1\mathrm{N}$,
- gravitational force field $\mathbf{f}(\xi) = -\begin{bmatrix} 0 \\ 2 \end{bmatrix}$.

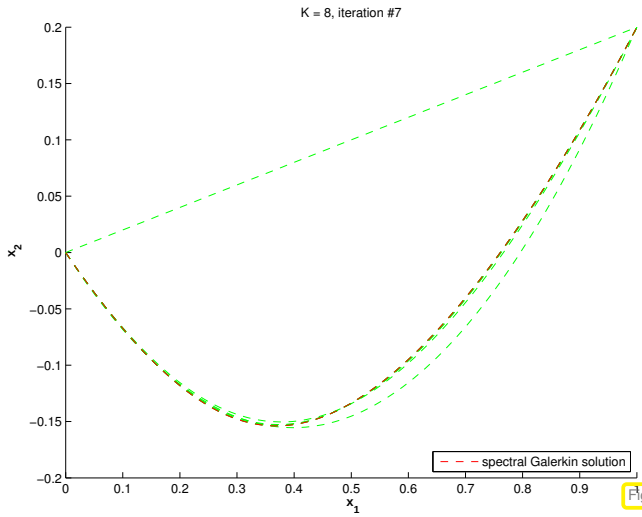


Fig. 34

iteration history

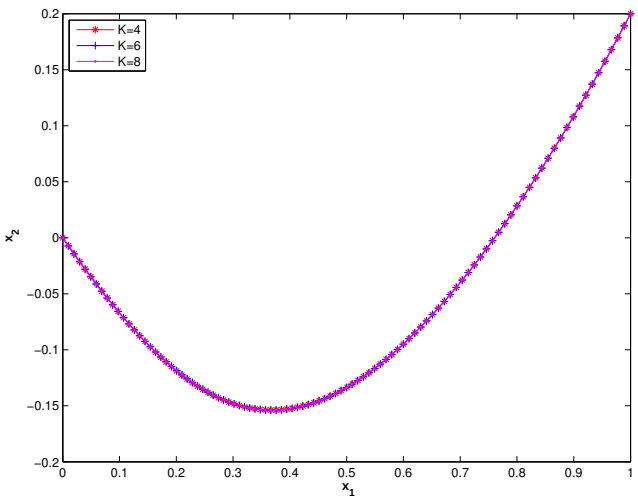


Fig. 35

solutions $\mathbf{u}_N$ for different resolutions

Observation: "Visual convergence" as polynomial degree is increased.

?! *Review question(s) 1.5.67.* (Spectral Galerkin discretization)

1. Determine the dimension of the space

$$V := \{v \in \mathcal{P}_p(\mathbb{R}) : \int_{-1}^{1} v(x)\,\mathrm{d}x = 0\}\,, \quad p \in \mathbb{N}\,,$$

   and describe a convenient and stable basis for Galerkin discretization.

2. What polynomials are integrated exactly by means of an n-point Gauss quadrature formula?
3. Give a formal definition of the finite-dimensional trial and test spaces used in the spectral Galerkin discretization discussed in § 1.5.60.
4. Consider the setting of § 1.5.52 and $\sigma \equiv 1$. For which numbers m of Gauss quadrature points is the Galerkin matrix $\mathbf{A}$ from (1.5.56) symmetric and *positive definite*?

1.5.2.2 Linear finite elements

From elementary numerical methods we know two ways to harness polynomials for the approximation of functions: approximation by

global polynomials [4, Chapter 3]	$\longleftrightarrow$	piecewise polynomials [4, Section 4.5]

The spectral polynomial Galerkin approach presented in Sect. 1.5.2.1 relies on global polynomials. Now let us examine the use of *piecewise polynomials*.

(1.5.68) Finite element mesh

Preliminaries: piecewise polynomials have to be defined w.r.t. partitioning of the domain $\Omega \subset \mathbb{R}$

➢ $\Omega = [a,b]$ equipped with nodes $(M \in \mathbb{N})$

$$\mathcal{X} := \{a = x_0 < x_1 < \cdots < x_{M-1} < x_M = b\} .$$

➤ mesh/grid

$$\mathcal{M} := \{]x_{j-1}, x_j[: 1 \le j \le M\} .$$

Special case:

equidistant mesh: $x_j := a + jh$, $h := \dfrac{b-a}{M}$.

☞ $[x_{j-1}, x_j]$, $j = 1, \dots, M$, $\hat{=}$ cells of $\mathcal{M}$, cell size $h_j := |x_j - x_{j-1}|$, $j = 1, \dots, M$
meshwidth $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|$

(1.5.69) Piecewise linear finite element trial space

Recall from Sect. 1.3.2: merely continuous, piecewise C^1 trial and test functions provide valid trial/test functions for the variational problems (1.3.15) and (1.4.23).

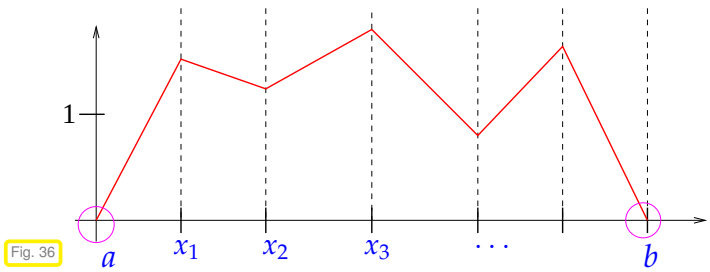


Fig. 36

⇑ function $\in \mathcal{S}^0_{1,0}(\mathcal{M})$

Simplest choice for test space

$$V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M}) := \left\{ \begin{array}{l} v \in C^0([a,b]) : v_{|[x_{i-1},x_i]} \text{ linear,} \\ i = 1, \dots, M, v(a) = v(b) = 0 \end{array} \right\}$$

➢ $N := \dim V_N = M - 1$

Choice of (ordered) basis $\mathfrak{B}_N$ of V_N **?**

1D "tent functions" [4, Ex. 3.1.8] ▷

$$\mathfrak{B} = \{b^1_N, \dots, b^{M-1}_N\} , \tag{1.5.70}$$

$$b^j_N(x_i) = \delta_{ij} := \begin{cases} 1 & \text{, if } i = j , \\ 0 & \text{, if } i \neq j , \end{cases} \tag{1.5.71}$$

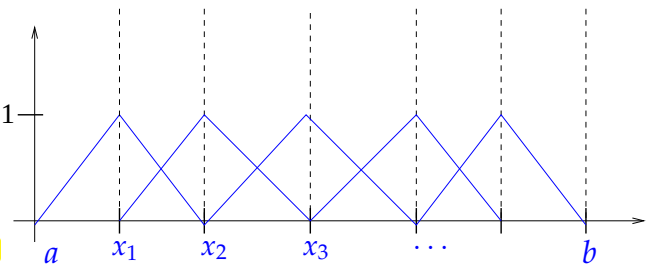


Fig. 37

$$\blacktriangleright \quad \frac{db^j_N}{dx}(x) = \begin{cases} \frac{1}{h_j} & \text{, if } x_{j-1} \le x \le x_j , \\ -\frac{1}{h_{j+1}} & \text{, if } x_j < x \le x_{j+1} , \\ 0 & \text{elsewhere.} \end{cases} \tag{1.5.72}$$

(piecewise derivative!)

Remark 1.5.73 (Benefit of variational formulation of BVPs)

The possibility of using simple piecewise linear trial and test functions is a clear benefit of the variational formulations derived in Sections 1.3 and 1.4, since they still make sense for merely piecewise continuously differentiable functions, also remember Section 1.3.2.

Below, in Section 1.5.3 we will learn about a method that targets the strong form of the 2-point BVP and, thus, has to impose more regularity on the trial functions.

(1.5.74) Simplest case: Linear variational problem with constant coefficients

We apply Galerkin discretization by means of linear finite elements to the linear variational problem (1.4.23) with constant stiffness coefficient σ:

$$u \in C^1_{\mathrm{pw},0}([a,b]): \quad \int_a^b \frac{du}{dx}(x)\frac{dv}{dx}(x)\,\mathrm{d}x = \int_a^b g(x)v(x)\,\mathrm{d}x \quad \forall v \in C^1_{\mathrm{pw},0}([a,b])\ .$$

Follow the policy of Galerkin discretization elaborated in Section 1.5.2, plug in trial functions from the finite element space $\mathcal{S}_1^0(\mathcal{M})$, expand them as a linear combination of tent functions, and test with all tent functions as explained in § 1.5.15. Using $u_N = \mu_1 b_N^1 + \cdots + \mu_N b_N^N$ we arrive as a discrete variational problem with

$$\int_a^b \sum_{l=1}^N \mu_l \frac{db_N^l}{dx}(x)\frac{db_N^k}{dx}(x)\,\mathrm{d}x = \int_a^b g(x)b_N^k(x)\,\mathrm{d}x\ , \quad k=1,\dots,N\ .$$

$$\Updownarrow$$

$$\sum_{l=1}^N \left(\int_a^b \frac{db_N^l}{dx}(x)\frac{db_N^k}{dx}(x)\,\mathrm{d}x\right)\mu_l = \underbrace{\int_a^b g(x)b_N^k(x)\,\mathrm{d}x}_{=:\varphi_k}\ , \quad k=1,\dots,N\ .$$

$$\Updownarrow$$

$$\boxed{\mathbf{A}\vec{\mu} = \vec{\varphi}} \quad \text{with} \quad (\mathbf{A})_{kl} := \int_a^b \frac{db_N^l}{dx}(x)\frac{db_N^k}{dx}(x)\,\mathrm{d}x\,, \quad k,l = 1,\dots,N\,,$$

$$\vec{\mu} = (\mu_l)_{l=1}^N \in \mathbb{R}^N \quad , \quad \vec{\varphi} = (\varphi_k)_{k=1}^N \in \mathbb{R}^N\,.$$

A linear system of equations, *cf.* § 1.5.52!

▷ system matrix $\mathbf{A} = (a_{ij}) \in \mathbb{R}^{M-1,M-1}$, $a_{ij} := \int_a^b \frac{db_N^i}{dx}(x)\,\frac{db_N^j}{dx}(x)\,\mathrm{d}x$, $1 \le i,j \le N$

piecewise derivatives

▷ r.h.s. vector $\vec{\varphi} \in \mathbb{R}^{M-1}$, $\varphi_k := \int_a^b g(x) b_N^k(x)\,\mathrm{d}x$, $k = 1,\dots,N$.

(1.5.75) Computation of entries of Galerkin matrix

We rely on the tent functions b_N^j as basis elements on a mesh as described in § 1.5.68. The detailed computations start with the evident fact that

$$|i-j| \ge 2 \quad \Rightarrow \quad \frac{b_N^j}{dx}(x) \cdot \frac{b_N^i}{dx}(x) = 0 \quad \forall x \in [a,b]\,,$$

because there is *no overlap* of the *supports* of the two basis functions.

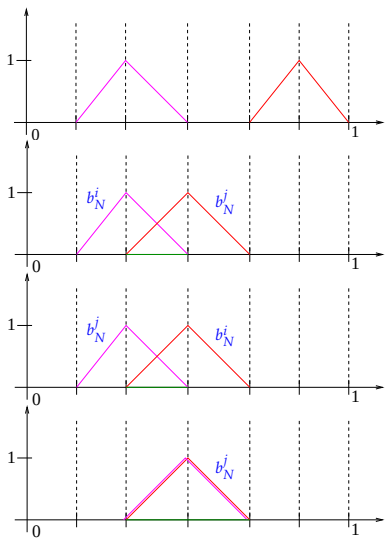
Definition 1.5.76. Support of a function

The support of a function $f : \Omega \mapsto \mathbb{R}$ is defined as

$$\operatorname{supp}(f) := \overline{\{\boldsymbol{x} \in \Omega \colon f(\boldsymbol{x}) \neq 0\}}\,.$$

In addition, we use that the gradients of the tent functions are piecewise constant, see (1.5.72).

$$\int_0^1 \frac{db_N^j}{dx}(x)\,\frac{db_N^i}{dx}(x)\,\mathrm{d}x = \begin{cases} 0 & \text{, if } |i-j| \ge 2 \quad \rightarrow \\ -\frac{1}{h_{i+1}} & \text{, if } j = i+1 \quad \rightarrow \\ -\frac{1}{h_i} & \text{, if } j = i-1 \quad \rightarrow \\ \frac{1}{h_i} + \frac{1}{h_{i+1}} & \text{, if } 1 \le i = j \le M-1 \quad \rightarrow \end{cases}$$



➡ $\mathbf{A}$ symmetric, positive definite and tridiagonal:

$$\mathbf{A} = \begin{bmatrix} \frac{1}{h_1}+\frac{1}{h_2} & -\frac{1}{h_2} & 0 & & & 0 \\ -\frac{1}{h_2} & \frac{1}{h_2}+\frac{1}{h_3} & -\frac{1}{h_3} & & & \\ 0 & \ddots & \ddots & \ddots & & \\ & & & \ddots & & 0 \\ & & & \ddots & \ddots & -\frac{1}{h_{M-1}} \\ 0 & & & 0 & -\frac{1}{h_{M-1}} & \frac{1}{h_{M-1}}+\frac{1}{h_M} \end{bmatrix} \in \mathbb{R}^{N,N}\,, \quad N := M-1\,. \tag{1.5.77}$$

✍ notation: $h_j := |x_j - x_{j-1}|$ $\hat{=}$ local meshwidth, cell size

---

(1.5.78) Properties of linear finite element Galerkin matrix

How can we tell that $\mathbf{A}$ is positive definite?

Application of [4, Lemma 1.8.12] that tells us that a diagonally dominant, regular, and symmetric matrix with positive diagonal is positive definite. Diagonal dominance of $\mathbf{A}$ in the sense of [4, Def. 1.8.8] is obvious.

(1.5.79) Computation of right hand side vector for linear finite element Galerkin discretization

The right hand side linear form of (1.4.23) involves a general coefficient function $g = g(x)$, which may be available only in procedural form, recall Rem. 1.5.5.

▶ Mandatory: computation of right hand side vector by numerical quadrature

Natural choice: piecewise polynomial trial/test spaces $\longleftrightarrow$ composite quadrature rule

e.g, composite trapezoidal rule:
$$\int_a^b f(t)\,\mathrm{d}t \approx \sum_{l=1}^{M} \tfrac{1}{2}h_l\left(f(x_{l-1}) + f(x_l)\right)\,, \tag{1.5.80}$$

▶ $$\varphi_k = \int_a^b g(x) b_N^k(x)\,\mathrm{d}x \approx \tfrac{1}{2}(h_k + h_{k+1})g(x_k)\,, \quad 1 \le k \le N\,.$$

because of property (1.5.71) of b_N^k.

Remark 1.5.81 (Special case: Linear system of equations for linear finite element discretization of equidistant mesh)

From (1.5.77): for equidistant mesh with uniform cell size $h > 0$ we arrive at the linear system of equations:

$$\frac{1}{h}\begin{bmatrix} 2 & -1 & 0 & & & & 0 \\ -1 & 2 & -1 & & & & \\ 0 & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & 0 \\ & & & & -1 & 2 & -1 \\ 0 & & & & 0 & -1 & 2 \end{bmatrix}\begin{pmatrix} \mu_1 \\ \vdots \\ \mu_N \end{pmatrix} = h\begin{pmatrix} g(x_1) \\ \vdots \\ g(x_N) \end{pmatrix} . \tag{1.5.82}$$

This is a symmetric positive definite Töplitz matrix, that is, a matrix with constant entries on diagonals. Its eigenvectors, indexed by $\ell = 1, \dots, N$, have the components

$$\eta_j^\ell = \sin(2\pi \frac{\ell j}{N+1}) , \quad j = 1, \dots, N . \tag{1.5.83}$$

(1.5.84) Linear finite element Galerkin discretization of (1.4.23) **for general stiffness coefficient** σ

We generalize the considerations of § 1.5.74 and perform a piecewise linear finite element Galerkin discretization of the linear variational problem arising from the function graph model for a taut elastic string (1.4.23):

$$u \in C^1_{\mathrm{pw},0}([a,b]): \quad \int_a^b \sigma(x) \frac{du}{dx}(x) \frac{dv}{dx}(x) \, \mathrm{d}x = \int_a^b g(x) v(x) \, \mathrm{d}x \quad \forall v \in C^1_{\mathrm{pw},0}([a,b]) .$$

As above plug in basis expansion of trial function $u_N \in \mathcal{S}^0_{1,0}(\mathcal{M})$, $u_N = \mu_1 b_N^1 + \dots + \mu_N b_N^N$, and test with all tent functions b_N^k:

$$\int_a^b \sigma(x) \sum_{l=1}^N \mu_l \frac{db_N^l}{dx}(x) \frac{db_N^k}{dx}(x) \, \mathrm{d}x = \int_a^b g(x) b_N^k(x) \, \mathrm{d}x , \quad k = 1, \dots, N . \tag{1.5.85}$$

In light of Rem. 1.5.5 numerical quadrature will be required for the (approximate) evaluation of both integrals. We use different quadrature rules.

Quad. rules:

- ✦ composite midpoint rule for left hand side integral $\longrightarrow$ [4, Section 5.4]

$$\int_a^b f(x) \, \mathrm{d}x \approx \sum_{j=1}^M h_j f(m_j) , \quad m_j := \tfrac{1}{2}(x_j + x_{j-1}) . \tag{1.5.86}$$

- ✦ composite trapezoidal rule [4, Eq. (5.4.4)] for right hand side integral, see (1.5.80).

Assumption 1.5.87. Smoothness requirement for stiffness coefficient

σ is piecewise continuous, $\sigma \in C^0_{\mathrm{pw}}([a,b])$, with jumps *only* at grid nodes x_j

Numerical quadrature applied to (1.5.85)

$$\Downarrow$$

$$\sum_{l=1}^{N} \underbrace{\left(\sum_{j=1}^{M} h_j \sigma(m_j) \frac{db_N^l}{dx}(m_j) \frac{db_N^k}{dx}(m_j) \right)}_{=(\mathbf{A})_{k,l}} \mu_l = \underbrace{\tfrac{1}{2}(h_{k+1}+h_k) g(x_k)}_{=:\varphi_k} \;, \quad k = 1,\dots,N \;,$$

$$\Updownarrow$$

$$\mathbf{A}\vec{\mu} = \vec{\varphi} \;.$$

Resulting linear system of equations equidistant mesh with uniform cell size $h > 0$:

$$\frac{1}{h}\begin{bmatrix} \sigma_1+\sigma_2 & -\sigma_2 & 0 & & & & 0 \\ -\sigma_2 & \sigma_2+\sigma_3 & -\sigma_3 & & & & \\ 0 & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & 0 \\ & & & & -\sigma_{M-2} & \sigma_{M-2}+\sigma_{M-1} & -\sigma_{M-1} \\ 0 & & & & 0 & -\sigma_{M-1} & \sigma_{M-1}+\sigma_M \end{bmatrix} \begin{bmatrix} \mu_1 \\ \vdots \\ \mu_N \end{bmatrix} = h \begin{bmatrix} g(x_1) \\ \vdots \\ g(x_N) \end{bmatrix} \;, \tag{1.5.88}$$

with $\sigma_j = \sigma(m_j)$, $j = 1,\dots,m$.

Remark 1.5.89 (Offset function for finite element Galerkin discretization)

Recall the device of an offset function as discussed in Rem. 1.3.29, where it enabled us to return to a vector space as trial space, if the original linear variational problem was posed on an affine space. We face this situation for the variational problem (1.4.23) unless the boundary values u_a and u_b are both zero. What are computationally convenient offset functions in the context of linear finite element Galerkin discretization?

In the case of general boundary conditions

$$u(a) = u_a \;, \quad u(b) = u_b$$

use *piecewise linear* offset function

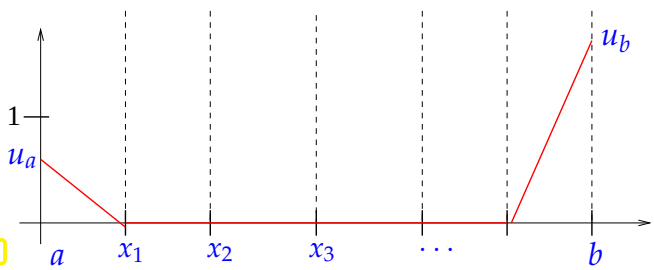


Fig. 38

$$u_0(x) = \begin{cases} u_a(1-\frac{x-a}{h_1}) & \text{, if } a \le x \le x_1 \;, \\ u_b(1-\frac{b-x}{h_M}) & \text{, if } x_{M-1} \le x \le b \;, \\ 0 & \text{elsewhere.} \end{cases} \tag{1.5.90}$$

Of course, we could have used a linear offset function

$$u_0(x) = \frac{b-x}{b-a}u_a + \frac{x-a}{b-a}u_a \,, \quad x \in [a,b] \,,$$

as well, but the above choice (1.5.90) has considerable benefits:

- u_0 is a *simple* function (since p.w. linear),
- u_0 is *locally supported*: contributions from u_0 will alter only first and last component of right hand side vector. To understand why, recall (1.5.14) and verify that $a(u_0, b_N^j) \neq 0$ only for $j = 1, M-1$.

(1.5.91) Linear finite element Galerkin discretization for non-linear elastic string model

Many issues faced in the linear finite element Galerkin discretization of the elastic string variational problem have already been discussed in connection with the spectral Galerkin discretization with global polynomials in § 1.5.60. Understanding that paragraph is essential for this one.

For the sake of simplicity in (1.3.15) we assume a linear gravity potential

$$V(\mathbf{x}) = \mathbf{g}\cdot\mathbf{x} \quad \Rightarrow \quad \mathbf{grad}\, V(\mathbf{x}) = \mathbf{g} \,, \quad \mathbf{g} \in \mathbb{R}^2 \text{ given} \,, \tag{1.3.33}$$

and for ease of notation we set $\mathbf{f}(\xi) := \rho(\xi)\mathbf{g}$. Thus, we target the *non-linear* variational equation on domain $\Omega = [0,1]$

$$\int_0^1 \frac{\kappa(\xi)}{L}\left(1 - \frac{L}{\|\mathbf{u}'(\xi)\|}\right)\mathbf{u}'(\xi)\cdot\mathbf{v}'(\xi)\,\mathrm{d}\xi = \int_0^1 \mathbf{f}(\xi)\cdot\mathbf{v}(\xi)\,\mathrm{d}\xi \quad \forall \mathbf{v} \in (C^1_{\mathrm{pw},0}([0,1]))^2 \,. \tag{1.3.15}$$

- ✦ Data κ, $\mathbf{f}$ given in procedural form, see Rem. 1.5.5.
- ✦ trial space $V_{N,0} = (\mathcal{S}^0_{1,0}(\mathcal{M}))^2$ (→ § 1.5.69) on equidistant mesh $\mathcal{M}$, meshwidth $h := \frac{1}{M}$, $M \in \mathbb{N}$.
- ✦ Basis: 1D tent functions from (1.5.70), lexikographic ordering,

$$\mathfrak{B} = \left\{ \begin{bmatrix} b_N^1 \\ 0 \end{bmatrix}, \begin{bmatrix} b_N^2 \\ 0 \end{bmatrix}, \dots, \begin{pmatrix} b_N^{M-1} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ b_N^1 \end{pmatrix}, \begin{pmatrix} 0 \\ b_N^2 \end{pmatrix}, \dots, \begin{pmatrix} 0 \\ b_N^{M-1} \end{pmatrix} \right\} ,$$

► $$\mathbf{u}_N = \sum_{j=1}^{M-1} \begin{bmatrix} \mu_j \\ \mu_{j+M-1} \end{bmatrix} b_N^j \quad \text{with coefficients} \quad \mu_k \,, \quad k = 1, \dots, 2M-2 \,.$$

- ✦ Evaluation of right hand side by composite trapezoidal rule (1.5.80).
- ✦ Evaluation left hand side by composite midpoint rule (1.5.86).

► Discrete variational problem with quadrature: for all $\mathbf{v}_N \in \mathrm{Span}\{\mathfrak{B}\}$

$$\sum_{j=1}^M h_j \frac{\kappa(m_j)}{L}\left(1 - \frac{L}{\|\mathbf{u}_N'(m_j)\|}\right)\mathbf{u}_N'(m_j)\cdot\mathbf{v}_N'(m_j) = \tfrac{1}{2}\sum_{j=1}^M h_l\left(\mathbf{f}(x_{l-1})\cdot\mathbf{v}_N(x_{l-1}) + \mathbf{f}(x_l)\cdot\mathbf{v}_N(x_l)\right) \,. \tag{1.5.92}$$

Again as in § 1.5.60 we temporarily treat the term $\frac{\kappa(\xi)}{L}\left(1 - \frac{L}{\|\mathbf{u}'(\xi)\|}\right)$ as a coefficient function like $\widehat{\sigma}$ in (1.4.23), and temporarily ignore its dependence on the solution $\mathbf{u}$. Then we arrive at two decoupled linear

variational problems for each component of $\mathbf{u}$, whose discretization by means of linear finite elements can be accomplished as elaborated in § 1.5.84. Details are given now.

Preliminary consideration: the derivative of

$$\mathbf{u}_N := \mathbf{u}_0 + \mu_1 \begin{bmatrix} b_N^1 \\ 0 \end{bmatrix} + \cdots + \mu_{M-1} \begin{bmatrix} b_N^{M-1} \\ 0 \end{bmatrix} + \mu_M \begin{bmatrix} 0 \\ b_N^1 \end{bmatrix} + \cdots + \mu_{2M-2} \begin{bmatrix} 0 \\ b_N^{M-1} \end{bmatrix} \tag{1.5.93}$$

with locally supported offset function $\mathbf{u}_0$ according to Rem. 1.5.89 ($\mathbf{u}(0)$, $\mathbf{u}(1)$ given by pinning conditions (1.2.2)) is piecewise constant on $\mathcal{M}$:

$$\text{in }]x_{j-1}, x_j[: \quad s_j(\vec{\boldsymbol{\mu}}) := \mathbf{u}_N'(\xi) = \frac{\mathbf{u}_N(x_j) - \mathbf{u}_N(x_{j-1})}{h} \tag{1.5.94}$$

$$= \frac{1}{h} \cdot \begin{cases} \begin{bmatrix} \mu_j - \mu_{j-1} \\ \mu_{j+M-1} - \mu_{j+M-2} \end{bmatrix} & \text{, if } 2 \le j \le M-1 \,, \\ \begin{bmatrix} \mu_1 \\ \mu_M \end{bmatrix} - \mathbf{u}(0) & \text{, if } j = 1 \,, \\ \mathbf{u}(1) - \begin{bmatrix} \mu_{M-1} \\ \mu_{2M-2} \end{bmatrix} & \text{, if } j = M \,, \end{cases}$$

because

$$\mathbf{u}_N(x_j) = \begin{cases} \mathbf{u}(0) & \text{, if } j = 0 \,, \\ \begin{bmatrix} \mu_j \\ \mu_{M+j-1} \end{bmatrix} & \text{, if } j \in \{1, \dots, M-1\} \,, \\ \mathbf{u}(1) & \text{, if } j = M \,. \end{cases}$$

For the green expression in (1.5.92) we use the abbreviation

$$r_j = r_j(\vec{\boldsymbol{\mu}}) := h \frac{\kappa(m_j)}{L} \left(1 - \frac{L}{\|s_j(\vec{\boldsymbol{\mu}})\|} \right) \tag{1.5.95}$$

and note the dependence $r_j = r_j(\vec{\boldsymbol{\mu}})$, which renders the system of equations *non-linear*. Thus, we can write (1.5.92) as

$$\sum_{j=1}^{M} h_j r_j(\vec{\boldsymbol{\mu}})\, \mathbf{u}_N'(m_j) \cdot \mathbf{v}_N'(m_j) = \tfrac{1}{2} \sum_{j=1}^{M} h_l \left(\mathbf{f}(x_{l-1}) \cdot \mathbf{v}_N(x_{l-1}) + \mathbf{f}(x_l) \cdot \mathbf{v}_N(x_l) \right) . \tag{1.5.96}$$

Now, we temporarily treat r_j as a mere coefficient. Then we make the important observation that *in each component* (1.5.96) agrees with the (fully, with quadrature) discrete linear variational problem examined in § 1.5.84! Thus we need only recall the Galerkin matrix (1.5.88) for that setting to conclude that a single row of non-linear system of equations arising from Galerkin finite element discretization of (1.3.15) reads

$$\text{row } 1: \quad (r_1 + r_2)\mu_1 - r_2\mu_2 = hf_1(h) + r_1 a \,, \tag{1.5.97}$$

$$\text{row } j: \quad -r_j\mu_j + (r_j + r_{j+1})\mu_{j+1} - r_{j+1}\mu_{j+2} = hf_1(jh) \,, \quad 2 \le j < M-1 \,, \tag{1.5.98}$$

$$\text{row } M-1: \quad -r_{M-1}\mu_{M-2} + (r_{M-1} + r_M)\mu_{M-1} = hf_1((M-1)h) + r_M b \,, \tag{1.5.99}$$

$$\text{row } M: \quad (r_1 + r_2(\vec{\boldsymbol{\mu}}))\mu_M - r_2\mu_{M+1} = hf_2(h) + r_1 u_a \,, \tag{1.5.100}$$

$$\text{row } j: \quad -r_j\mu_{j+M-1} + (r_j + r_{j+1})\mu_{j+M} - r_{j+1}\mu_{j+M+1} = hf_2(jh) \,, \quad 2 \le j < M-1 \,, \tag{1.5.101}$$

$$\text{row } M-1: \quad -r_{M-1}\mu_{2M-3} + (r_{M-1} + r_M)\mu_{2M-2} = hf_2((M-1)h) + r_M u_b \,. \tag{1.5.102}$$

Here the dependence $r_j = r_j(\vec{\mu})$ has been suppressed to simplify the notation.

Please study the derivation of (1.5.88) in order to understand how (1.5.97)-(1.5.102) arise.

These equations can be written in a more compact form, analogous to (1.5.63):

$$\text{(1.5.97)-(1.5.102)} \quad \Leftrightarrow \quad \boxed{\begin{pmatrix} \mathbf{R}(\vec{\mu}) & 0 \\ 0 & \mathbf{R}(\vec{\mu}) \end{pmatrix} \vec{\mu} = \begin{pmatrix} \vec{\varphi}_1(\vec{\mu}) \\ \vec{\varphi}_2(\vec{\mu}) \end{pmatrix}} \; . \tag{1.5.103}$$

with

$$\mathbf{R}(\vec{\mu}) := \begin{pmatrix} r_1 + r_2 & -r_2 & 0 & & & & 0 \\ -r_2 & r_2 + r_3 & -r_3 & & & & \\ 0 & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & 0 \\ & & & & -r_{M-2} & r_{M-2} + r_{M-1} & -r_{M-1} \\ 0 & & & & 0 & -r_{M-1} & r_{M-1} + r_M \end{pmatrix} \in \mathbb{R}^{M-1,M-1} \; ,$$

$$(\vec{\varphi}_1)_j := hf_1(hj) \quad , \quad (\vec{\varphi}_2)_j := hf_2(hj) \; , \quad j = 1, \dots, M-1 \; .$$

Dependence of the right hand side vector on the solution $\vec{\mu}$ is due to the offset function technique, see Rem. 1.5.89.

C++11 code 1.5.104: Linear finite element discretization of elastic string variational problem

```
function [vu,Jrec,figsol,figerg] = stringlinfem(kappa,f,L,u0,u1,M,tol)
% Solving the non-linear variational problem (1.3.15) for the elastic string by means of piecewise
% linear finite elements on an equidistant mesh with M-1 interior nodes.
% kappa, f are handles of type @(xi) providing the coefficient function
% κ and the force field f.  u0 and u1 pass the pinning points.
% M is the number of mesh cells, tol specifies the tolerance for the fixed point
% iteration.  return value:  2×(M+1)-matrix of node positions
if (nargin < 7), tol = 1E-2; end
h = 1/M;                    % meshwidth
phi = h*f(h*(1:M-1));    % Right hand side vector

% Initial guess:  straight string, condition L > ‖u(0) − u(1)‖.
if (L >= norm(u1-u0)), error('String must be tense'); end
vu_new = u0*(1-(0:1/M:1))+u1*(0:1/M:1);
% Meaning of components of vu:  vu(1,2:M) ↔ μ_1,...,mu_{M-1},  vu(2,2:M) ↔ μ_M,...,μ_{2M-2}.
figsol = figure; Jrec = []; hold on;
for k=1:100    % loop for fixed point iteration, maximum 100 iterations
  vu = vu_new;
% Plot shape of string
  plot(vu(1,:),vu(2,:),'--g'); drawnow;
  title(sprintf('M = %d, iteration #%d',M,k));
```

```
  xlabel('{\bf x_1}'); ylabel('{\bf x_2}');
%Compute the cell values s_j, r_j, j=1,...,M, see (1.5.94).
  d = (vu(:,2:end) - vu(:,1:end-1))/h; % derivative, piecewise constant, see (1.5.94)
  s = sqrt(d(1,:).^2 + d(2,:).^2);      % norm of derivative
  r = kappa(h*((1:M)-0.5)).*(1/L - 1./s)/h; % values r_j
% Compute total potential energy
% elastic energy of u_N (1.2.49)
  Jel = h/(2*L)*kappa(h*((1:M)-0.5))*((s-L).^2)';
% graviational energy (1.2.44)
  Jf = - (phi(1,:)*vu(1,2:M)'+phi(2,:)*vu(2,2:M)');
  Jrec = [Jrec; k , Jel, Jf, Jel+Jf];
% Assemble triadiagonal matrix R = R(μ), see (1.5.103)
  R = gallery('tridiag',-r(2:M-1),r(1:M-1)+r(2:M),-r(2:M-1));
% modify right hand side in order to take into account pinning conditions
  phi1 = phi(1,:); phi1(1) = phi1(1) + r(1)*u0(1); phi1(M-1) = phi1(M-1) + r(M)*u1(1);
  phi2 = phi(2,:); phi2(1) = phi2(1) + r(1)*u0(2); phi2(M-1) = phi2(M-1) + r(M)*u1(2);
% Solve linear system by direct elimination and compute new iterate
  vu_new = [u0,[(R\phi1')';(R\phi2')'],u1];
% Check simple termination criterion for fixed point iteration.
  if (norm(vu_new - vu,'fro') < tol*norm(vu_new,'fro')/M)
    plot(vu(1,:),vu(2,:),'r-*'); break; end
end
% Plot of total potential energy in the course of the iteration
figerg = figure('name','total potential energy');
title(sprintf('elastic string, M = %d',M));
plot(Jrec(:,1),Jrec(:,4),'m-*',...
     Jrec(:,1),Jrec(:,2),'b-+',...
     Jrec(:,1),Jrec(:,3),'g-^');
xlabel('{\bf no. of iteration step}'); ylabel('{\bf energy}');
legend('total potential energy','elastic energy','energy in force field','location','east');
```

C++ code 1.5.105: Class implementing the linear finite element discretization of the elastic string model

```
// Class template, template parameters supply types for coefficient functions κ
// (stiffness) and f (force)
template<typename Function1, typename Function2>
class StringLinearFEM {
public:
  // Constructor for passing model parameter and data, L ≙ length of
  // string, u_0,u_1 ≙ pinning positions
  StringLinearFEM(Function1 kappa,Function2 f,double L,
          const Eigen::VectorXd &u0, const Eigen::VectorXd &u1)
  : kappa(kappa),f(f),L(L),u0(u0),u1(u1) {}
  // Actual solution of the discrete non-linear variational problem
```

```
  // based on linear finite elements and an equidistant mesh
  // with M cells.
  Eigen::MatrixXd solve(int M, double tol=1e-2);
private:
  Function1 kappa;  Function2 f;
  Eigen::VectorXd u0,u1;
  double L;
};
```

C++ code 1.5.106: Linear finite element discretization of non-linear elastic string variational problem

```
template<typename Function1, typename Function2>
Eigen::MatrixXd StringLinearFEM<Function1,Function2>::solve(int M, double
    tol=1e-2) {
    double h = 1./M; // meshwidth
    // Compute the rhs vector
    Eigen::MatrixXd phi = h*NPDE::apply(f,
        h*Eigen::RowVectorXd::LinSpaced(M-1, 1, M-1));
    // Initial guess:  straight string, condition L > ||u(0) - u(1)||.
    Eigen::MatrixXd vu = u0 * (1 - Eigen::RowVectorXd::LinSpaced(M+1, 0,
        1).array()).matrix() +
      u1 * Eigen::RowVectorXd::LinSpaced(M+1, 0, 1);
    // Fixed point iteration
    for(int k = 0; k < 100; k++) {
       // derivative, piecewise constant, see (1.5.94)
      auto d = (vu.rightCols(M) - vu.leftCols(M))/h;
      // Compute the cell values s_j, r_j, j=1,...,M, see (1.5.94).
      // Norm of derivative
      Eigen::RowVectorXd s = (d.row(0).cwiseAbs2() +
          d.row(1).cwiseAbs2()).cwiseSqrt();
       // values of r_j
      Eigen::RowVectorXd r = NPDE::apply(kappa,
          h*Eigen::RowVectorXd::LinSpaced(M, 0.5, M-0.5)).cwiseProduct((1./L
          - 1/s.array()).matrix())/h;
      // Assemble triadiagonal matrix R = R(μ), see (1.5.103)
      auto R = NPDE::tridiagonal(-r.block(0, 1, 1, M-2),
                    r.leftCols(M-1)+r.row(0).rightCols(M-1),
                    -r.block(0, 1, 1, M-2));
      // modify right hand side in order to take into account pinning
          conditions
      Eigen::RowVectorXd phi1 = phi.row(0);
      Eigen::RowVectorXd phi2 = phi.row(1);

      phi1(0)   = phi1(0)   + r(0) * u0(0);
      phi1(M-2) = phi1(M-2) + r(M-1) * u1(0);

      phi2(0)   = phi2(0)   + r(0)* u0(1);
      phi2(M-2) = phi2(M-2) + r(M-1) * u1(1);

      // Solve linear system and compute new iterate, do
          LU-decomposition once
```

```
      Eigen::SimplicialLDLT<Eigen::SparseMatrix<double> > solver;
          solver.compute(R);

      auto sol1 = solver.solve(phi1.transpose());
      auto sol2 = solver.solve(phi2.transpose());

      Eigen::MatrixXd sol(2, M-1);
      sol << sol1.transpose() , sol2.transpose();

      // Add the pinning boundary values to the solutions
      Eigen::MatrixXd vu_new = Eigen::MatrixXd::Zero(2, M+1);
      vu_new << u0, sol, u1;

       // Compute the relative error
      double rel_error = (vu_new - vu).norm() / vu_new.norm() * M;
      vu = vu_new;
      if(rel_error < tol) break;
    }
    return vu;
}
```

Remark 1.5.107 (Fixed point iteration for solving non-linear system of equations)

Iterative solution of (1.5.103) by fixed point iteration, see § 1.5.60 and Code 1.5.104.

Initial guess $\vec{\mu}^{(0)} \in \mathbb{R}^N$; $k = 0$;
repeat
$k \leftarrow k+1$;

Solve the *linear* system of equations $\begin{pmatrix} \mathbf{R}(\vec{\mu}^{(k-1)}) & 0 \\ 0 & \mathbf{R}(\vec{\mu}^{(k-1)}) \end{pmatrix} \vec{\mu}^{(k)} = \begin{pmatrix} \vec{\varphi}_1(\vec{\mu}^{(k-1)}) \\ \vec{\varphi}_2(\vec{\mu}^{(k-1)}) \end{pmatrix}$;

until $\left\| \vec{\mu}^{(k)} - \vec{\mu}^{(k-1)} \right\| \leq \mathrm{tol} \cdot \left\| \vec{\mu}^{(k)} \right\|$

Simple termination criterion: stop, when relative change of Euclidean norm of coefficient vector below a prescribed tolerance. Another and better option is to monitor the relative change of the potential energy.

Experiment 1.5.108 (Elastic string shape by finite element discretization)

- Linear finite element discretization of (1.3.15), see § 1.5.91, Code 1.5.104.
- $\kappa \equiv 1$, $L = 0.5$, $\mathbf{u}(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{u}(1) = \begin{bmatrix} 1 \\ 0.2 \end{bmatrix}$
- gravitational force field $\mathbf{f}(\xi) = -\begin{bmatrix} 0 \\ 2 \end{bmatrix}$.

Piecewise linear finite element solution of (1.3.15), equidistant meshes with M cells, $M = 5, 10, 20$ ▷ "Visual convergence" of computed polygon approximating the string shape.

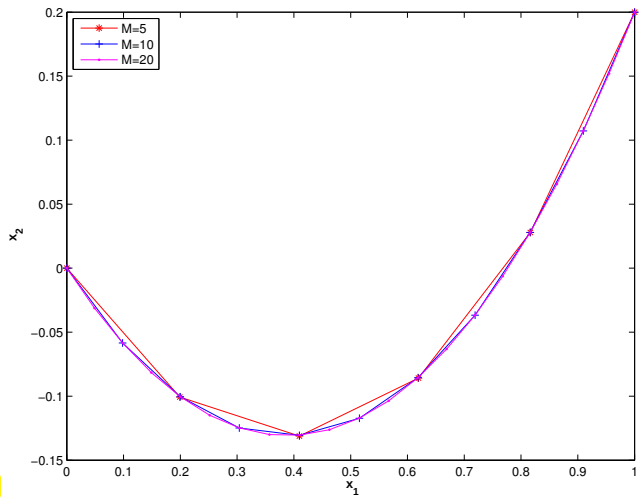


Fig. 39

Convergence of fixed point iteration ($M = 20$):

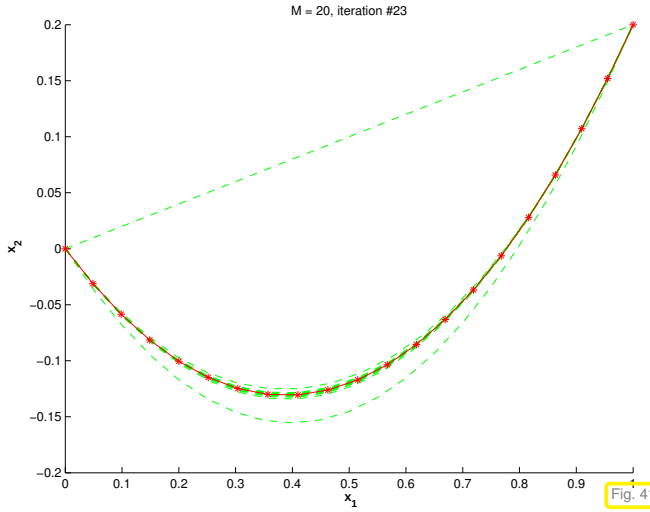


Fig. 40

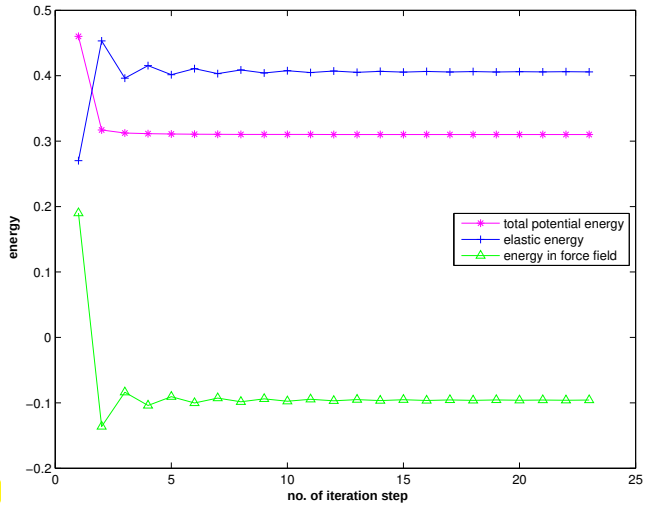


Fig. 41

?! *Review question(s) 1.5.109.* (Galerkin discretization)

1. Argue, why the basis (1.5.32) is notoriously unstable, whereas the integrated Legendre polynomials (1.5.33) provide much more stable bases. N-dimensional
2. How can you obtain a quadrature formula on $[a, b]$ from an m-point quadrature formula on $[0, 1]$ with nodes ζ_j, $j = 1, \dots, m$, and weights ω_j, $j = 1, \dots, m$. How are the oders of both quadrature formulas related?
3. Explain the important consequence of using basis functions for linear finite element Galerkin discretization whose support span two adjacent mesh cells only.
4. For a mesh of $[0, 1]$ with M cells, what is the dimension of the space
$$V_{N,0} := \{v \in C^0([0,1]) : \ v_{|[x_{i-1},x_i]} \in Cp_1, i = 1, \dots, M\} \ ? \tag{1.5.110}$$
Propose a basis of $V_{N,0}$ consisting of basis functions with smallest possible local supports.
5. On a mesh $\mathcal{M}$ with nodes x_i, $i = 0, \dots, M$, consider the space
$$V_{N,0} := \{v \in C_0^0([0,1]) : \ v_{|[x_{i-1},x_i]} \in \mathcal{P}_2, i = 1, \dots, M\} \ . \tag{1.5.111}$$
Determine $\dim V_{N,0}$ and propose a basis of locally supported functions that contains all tent functions from (1.5.70)–(1.5.71).

6. Given an *equidistant* mesh $\mathcal{M}$ of $[a,b]$ as introduced in § 1.5.68, we consider the finite-dimensional space

$$V_{N,0} := \left\{ v \in C^0([a,b]) : v_{|[x_{i-1},x_i]} \in \mathcal{P}_{1(\mathbb{R})}, \int_a^b \right\} .$$

What is $\dim V_{N,0}$? Describe a basis of $V_{N,0}$ consisting of functions with minimal supports.

1.5.3 Collocation

Targeted: Two-point BVP = ODE $\mathcal{L}(u) = f$ + boundary conditions

(1.5.112) Some differential operators of string models

Above, $\mathcal{L}(u)$ stands for a *differential operator*. For instance, for the elastic string model, Thm. 1.3.42 yields

$$\mathcal{L}(\mathbf{u}) = -\frac{d}{d\xi}\left(\frac{\kappa(\xi)}{L}(\|\mathbf{u}'\| - L)\frac{\mathbf{u}'}{\|\mathbf{u}'\|}\right) , \tag{1.5.113}$$

whereas the taut string graph model (1.4.25) yields

$$\mathcal{L}(u) = \frac{d}{dx}\left(\sigma(x)\frac{d\widehat{u}}{dx}(x)\right) . \tag{1.5.114}$$

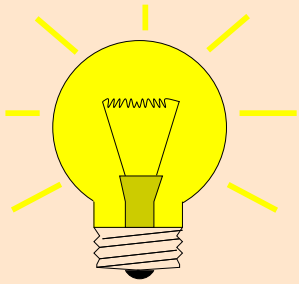
This latter differential operator is *scalar, linear* and *second-order*, *cf.* (1.5.124) below. The one from (1.5.113) is non-linear.

Note: In contrast to the Galerkin approach, collocation techniques do not tackle the weak form of a boundary value problem, but rather the "classical"/strong form (ODE/PDE).

Idea of collocation discretization

Idea: ❶ seek solution in finite-dimensional trial space $V_{N,0}$, $N := \dim V_{N,0} < \infty$

❷ pick collocation nodes $\mathcal{N} := \{x_1, \dots, x_N\} \subset \Omega$ such that



the point values $v(x_j)$, $j = 1, \dots, N$ uniquely determine a $v \in V_{N,0}$

$$\Updownarrow \tag{1.5.116}$$

the points $v(x_j)$, $j = 1, \dots, N$ are suitable for interpolation into $v \in V_{N,0}$

Collocation conditions: $u_N \in V_N$: $\mathcal{L}(u_N)(x_j) = f(x_j)\,, \quad j = 1, \dots, N\,.$ (1.5.117)

❸ choose ordered basis $\mathfrak{B} = \{b_N^1, \dots, b_N^N\}$ of $V_{N,0}$ & plug basis representation

$$u_N = u_0 + \mu_1 b_N^1 + \cdots + \mu_N b_N^N \quad (u_0 \hat{=} \text{ offset function, } \textit{cf.} \text{ Rem. 1.5.12 })$$

into collocation conditions (1.5.117)

$$\boxed{\vec{\mu} = (\mu_l)_{l=1}^N\colon \quad \mathcal{L}(u_0 + \mu_1 b_N^1 + \cdots + \mu_N b_N^N)(x_j) = f(x_j)\,, \quad j = 1, \dots, N}\,. \tag{1.5.118}$$

In general: (1.5.118) is a non-linear system of equation (N equations for N unknowns $\mu_1, \dots, \mu_N$).

It is natural to chose $N = \dim V_N$ collocation points to make the number of unknowns, which agrees with the dimension of the trial space V_N, agree with the number of equations, which is the same as the number of collocation points.

More abstract: bijectivity of point evaluation (1.5.116) $\Rightarrow$ $\boxed{\sharp\{\text{nodes}\} = \dim V_{N,0}}$

In the sequel we present a detailed discussion of the collocation approach for the *linear* two point boundary value problem

$$\begin{gathered} \mathcal{L}(u) := -\frac{d}{dx}\left(\sigma(x)\frac{du}{dx}(x)\right) = g(x)\,, \quad a \le x \le b\,, \\ u(a) = u_a \quad , \quad u(b) = u_b\,, \end{gathered} \tag{1.5.119}$$

on domain $\Omega = [a, b]$, related to variational problem (1.4.23).

This linear model boundary value problem was obtained from the graph description of the taut string model, see Sect. 1.4.

Remark 1.5.120 (Smoothness requirements for collocation trial space)

The collocation equations (1.5.118) will make sense only if the action of the (differential) operator $\mathcal{L}$ is well defined for elements of the trial space V_N. For the differential operator from (1.5.119) with $\sigma \in C^1([a, b])$ this entails $V_{N,0} \subset C^2_{\mathrm{pw}}([a, b])$, that is the trial functions have to be at least continuously differentiable.

Bearing in mind that the collocation conditions are based on point evaluations, it is natural to demand even $\mathcal{L}(u_N) \in C^0([a,b])$, which implies the condition $V_{N,0} \subset C^2([a,b])$, *cf.* Sect. 1.5.3.2.

Impact in a concrete case: For the two-point BVP (1.5.119) it is not possible to build a collocation method on the trial space $V_{N,0} := \mathcal{S}^0_{1,0}(\mathcal{M})$ of $\mathcal{M}$-piecewise linear finite element functions ($\rightarrow$ Sect. 1.5.2.2), because $v_N \in \mathcal{S}^0_{1,0}(\mathcal{M})$ is *not differentiable* in nodes x_j of the mesh, which renders $\mathcal{L}(v_N)$ undefined.

► Trial spaces for collocation methods have to comply with more stringent smoothness conditions that those suitable for Galerkin discretization.

Remark 1.5.121 (Collocation: smoothness requirements for coefficients)

For 2-point BVP (1.5.119): σ must be differentiable in collocation collocation nodes, with known values $\frac{d\sigma}{dx}(x_j)$, $j = 1,\dots,N$, in the sense of Rem. 1.5.5: extra difficulty when σ given in procedural form.

Inability to cope with discontinuous coefficients is another drawback of collocation methods compared to the Galerkin approach.

1.5.3.1 Spectral collocation

Focus: *linear* two point boundary value problem (1.5.119)

Trial space for polynomial spectral collocation:

$$V_{N,0} = \mathcal{P}_p(\mathbb{R}) \cap C^2_0([a,b])\,, \quad p \geq 2\,. \tag{1.5.122}$$

= polynomials of degree $\leq p$, vanishing at endpoints of domain

► No. of degrees of freedom $N := \dim V_{N,0} = p - 1$

➢ same trial space as for polynomial spectral Galerkin approach, see Sect. 1.5.2.1.

(1.5.123) polynomial spectral collocation for (linear) two-point BVP (1.5.119)

Discussion: polynomial spectral collocation for (linear) two-point BVP (1.5.119)

✦ offset function $u_0(x) := \frac{b-x}{b-a}u_a + \frac{x-a}{b-a}u_b$.

✦ Basis $\mathfrak{B} := \{b^j_N := M_j\}$ consisting of integrated Legendre polynomials, see (1.5.33).

The rationale for this choice is the same as for the spectral Galerkin method presented in Section 1.5.2.1: the basis must enjoy *good stability properties* in order to avoid adverse impact of roundoff errors, see Ex. 1.5.59.

Note: $\mathcal{L}$ from (1.5.119) is a linear differential operator!

Terminology: A differential operator is a mapping on a function space involving only values of the function argument and some of its derivatives in the same point.

A differential operator $\mathcal{L}$ is linear, if

$$\boxed{\mathcal{L}(\alpha u + \beta v) = \alpha \mathcal{L}(u) + \beta \mathcal{L}(v) \quad \forall \alpha, \beta \in \mathbb{R},\ \forall \text{functions } u, v} \tag{1.5.124}$$

$$(1.5.118) \overset{(1.5.124)}{\Longrightarrow} \sum_{l=1}^{N} \mathcal{L}(b_N^l)(x_k)\mu_l = f(x_k) - \mathcal{L}(u_0)(x_k)\,, \quad k = 1, \dots, N\,. \tag{1.5.125}$$

$$\Updownarrow$$

$$\mathbf{A}\vec{\mu} = \vec{\varphi}\,, \qquad \begin{aligned} (\mathbf{A})_{k,l} &:= \mathcal{L}(b_N^l)(x_k)\,, \quad k,l \in \{1,\dots,N\}\,, \\ \varphi_k &:= f(x_k) - \mathcal{L}(u_0)(x_k)\,, \quad k \in \{1,\dots,N\}\,. \end{aligned} \tag{1.5.126}$$

An $N \times N$ linear system of equations

For BVPs featuring *linear* differential operators, collocation invariably leads to a *linear* system of equations for the unknown coefficients of the basis representation of the collocation solution.

Remark 1.5.127 (Bases for polynomial spectral collocation)

Same choices as for spectral Galerkin methods, see Rem. 1.5.31, same stability considerations apply.

Remark 1.5.128 (Collocation nodes for polynomial spectral collocation)

Rule of thumb (without further explanation, see [3]):

> choose collocation points x_j, $j = 1, \dots, N$ such that the induced Lagrangian interpolation operator ($\rightarrow$ [4, Cor. 3.2.15]) has a small ∞-norm (Lebesgue constant), see [4, Lemma 3.2.67].

Popular choice (due to [4, Eq. (4.1.82)]): Chebychev nodes (in $[a, b]$)

$$x_k := a + \tfrac{1}{2}(b-a)\left(\cos\left(\frac{2k-1}{2N}\pi\right) + 1\right)\,, \qquad k = 1, \dots, N\,. \tag{1.5.129}$$

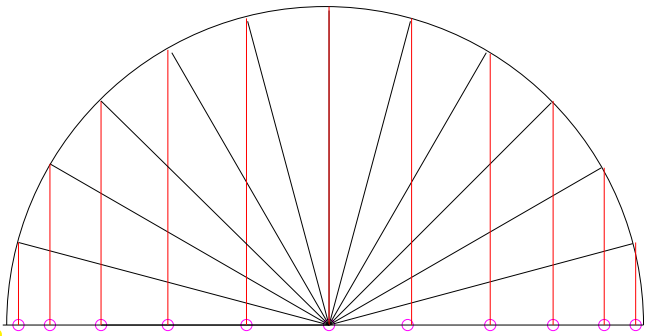


Fig. 42

construction of Chebychev nodes

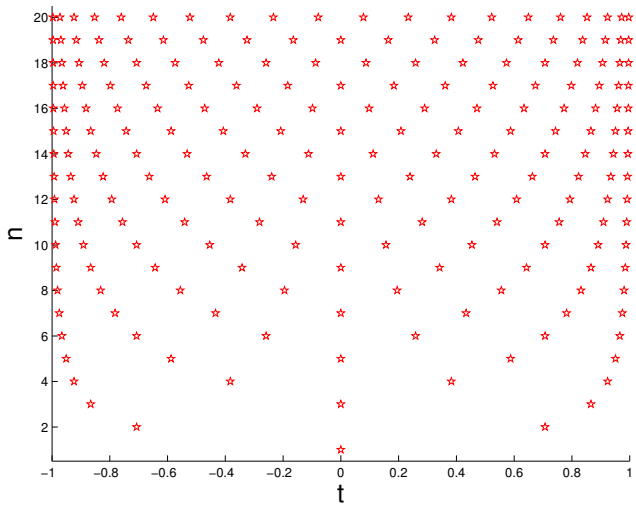


Fig. 43

distribution of Chebychev nodes

C++11 code 1.5.130: Computation of derivatives of Legendre polynomials using (1.5.41)

```
function [V,M,D] = dilegpol(n,x)
% Computes values of the first n+1 Legendre polynomials (returned in matrix V)
% the first n-1 integrated Legendre polynomials (returned in matrix M), and the
% first n+1 first derivatives of Legendre polyomials in the points x_j passed
% in the row vector x.
% Uses the recursion formulas (1.5.38) and (1.5.33)
V = ones(size(x)); V = [V; x];
% recursion (1.5.38) for Legendre polynomials
for j=1:n-1, V = [V; ((2*j+1)/(j+1)).*x.*V(end,:) - j/(j+1)*V(end-1,:)]; end
% Formula (1.5.33) for integrated Legendre polynomials
M = diag(1./(2*(1:n-1)+1))*(V(3:n+1,:) - V(1:n-1,:));
% Recursion formula (1.5.41) for derivatives of Legendre polynomials
if (nargout > 2)
  D = [zeros(size(x)); ones(size(x))];
  for j=1:n-1,  D = [D;(2*j+1)*V(j+1,:)+D(j,:)]; end
end
```

C++ code 1.5.131: Computation of derivatives of Legendre polynomials using (1.5.41)

```
//Compute Legendre and integrated Legendre polynomials and
// derivatives of Legendre polynomials
// n ≙ Degree of polynomials
// x ≙ Points at which the polynomials have to be computed
// return value 3-std::tuple of Eigen::MatrixXd containing
// values of Legendre polynomials, integrated Legendre polynomials,
// and derivatives, respectively
std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd>
dilegendrepol(int n, const Eigen::RowVectorXd &x) {
  const int n_points = x.cols();
```

```
  Eigen::MatrixXd V,M;
  std::tie(V, M) = intlegendrepol(n, x); // see Code 1.5.44
  Eigen::MatrixXd D(n+1, n_points);
  D.row(0) = Eigen::RowVectorXd::Zero(n_points);
  D.row(1) = Eigen::RowVectorXd::Constant(n_points, 1);
  for(int i = 1; i < n; i++) {
    D.row(i+1) = (2*i+1)*V.row(i)+D.row(i-1);
  }
  return std::make_tuple(V, M, D);
}
```

C++11 code 1.5.132: Spectral collocation for linear 2nd-order two-point BVP

```
function u = linspeccol(g,N,x)
% Polynomial spectral collocation discretization of linear 2nd-order two-point BVP
% -d^2u/dx^2 = g(x), u(0) = u(1) = 0
% on Ω = [0,1]. Trial space of dimension N, collocation in Chebychev nodes.
% Values of approximate solution in points x_j are returned in the row vector u
cn = cos((2*(1:N)-1)*pi/(2*N));          % Chebychev nodes, see (1.5.129)
[V,M,D] = dilegpol(N+1,cn);              % Obtain values of (2nd derivatives) of M_m
mu = (-4*D(2:N+1,:))'\(g(0.5*(cn+1))'); % Solve collocation system
% Compute values of integrated Legendre polynomials at output points
[V,M] = dilegpol(N+1,2*x-1); u = mu'*M;
```

C++ code 1.5.133: Spectral collocation for linear 2nd-order two-point BVP

```
// Polynomial spectral collocation discretization of linear 2nd-order two-point BVP
// -d^2u/dx^2 = g(x), u(0) = u(1) = 0
// on Ω = [0,1]. Trial space of dimension N, collocation in Chebychev nodes.
// Values of approximate solution in points x_j are returned in the row vector u
template<typename Function>
static Eigen::RowVectorXd solve(Function g, int N, const Eigen::RowVectorXd& x) {
  // Obtain the collocation nodes
  Eigen::RowVectorXd cn = Collocation::nodes(N);
  // Obtain values of 2nd derivative of integrated Legendre polynomials
  Eigen::MatrixXd V, M, D;
  std::tie(V, M, D) = NPDE::dilegendrepol(N+1, cn);
  // Assemble matrix
  Eigen::MatrixXd R = -4*D.block(1, 0, N, N).transpose();
  // Compute the right hand side
  Eigen::RowVectorXd rhs = NPDE::apply(g, 0.5*(cn.array()+1).matrix());
```

```
    // Solve the collocation system
    Eigen::VectorXd mu = R.lu().solve(rhs.transpose());
    // Compute values of integrated Legendre polynomials at output points
    std::tie(std::ignore, M) = intlegendrepol(N+1, (2*x.array()-1).matrix());
    return mu.transpose() * M;
}
```

Example 1.5.134 (Polynomial spectral collocation for 2-point BVP)

Setting of Ex. 1.5.29, spectral polynomial collocation, on , $N = 5, 7, 10$, basis from integrated Legendre polynomials, plot of solution u_N.

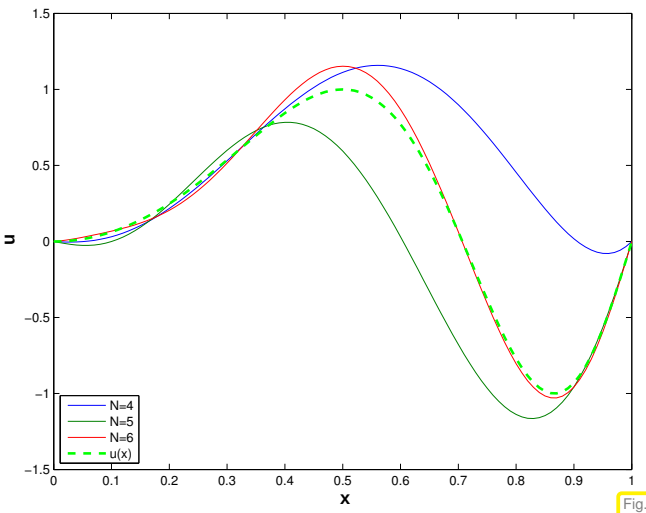


Fig. 44

Collocation in Chebychev nodes

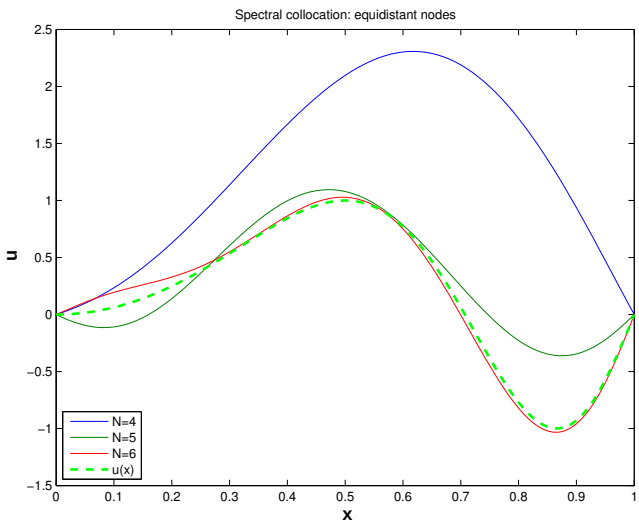


Fig. 45

Collocation in equidistant nodes

1.5.3.2 Spline collocation

Analogous to Sect. 1.5.2.2: collocation based on *piecewise polynomials*

Rem. 1.5.120 ➢ for BVP (1.5.119) smoothness $V_{N,0} \subset C^2([a,b])$ is required.

Which piecewise polynomial spaces offer this kind of smoothness ?

Recall [4, Def. 3.5.1], *cf.* [4, Section 3.5.1]:

Definition 1.5.135. Cubic spline

A function $s : [a,b] \mapsto \mathbb{R}$ is a cubic spline function w.r.t. the ordered knot set $\mathcal{T} := \{a = x_0 < x_1 < x_2 < \ldots < x_{M-1} < x_M = b\}$,

if

(i) $s \in C^2([a,b])$ (twice continuously differentiable),

(ii) $s_{|]x_{j-1},x_j[} \in \mathcal{P}_3(\mathbb{R})$ (piecewise cubic polynomial)

✎ notation: $\mathcal{S}_{3,\mathcal{T}} \hat{=}$ vector space of cubic splines on knot set $\mathcal{T}$

Known: $\boxed{\dim \mathcal{S}_{3,\mathcal{T}} = \sharp\mathcal{T} + 2 = M + 3}$

▶ Spline based trial space for collocation for 2-point BVP(1.5.119)

natural cubic splines: $V_{N,0} := \left\{ s \in \mathcal{S}_{3,\mathcal{T}} : \begin{array}{c} s''(a) = s''(b) = 0 \\ s(a) = s(b) = 0 \end{array} \right\} \Rightarrow \boxed{N := \dim V_N = M - 1}$,

Choice of collocation nodes:

collocation nodes for cubic spline collocation = interior spline nodes x_j: $\boxed{\mathcal{N} = \mathcal{T} \setminus \{a, b\}}$

Example 1.5.136 (Cubic spline collocation discretization of 2-point BVP)

Setting of Exp. 1.5.29

Cubic spline collocation with equidistant nodes, $M = 5, 7, 12$

Solution u_N ▷

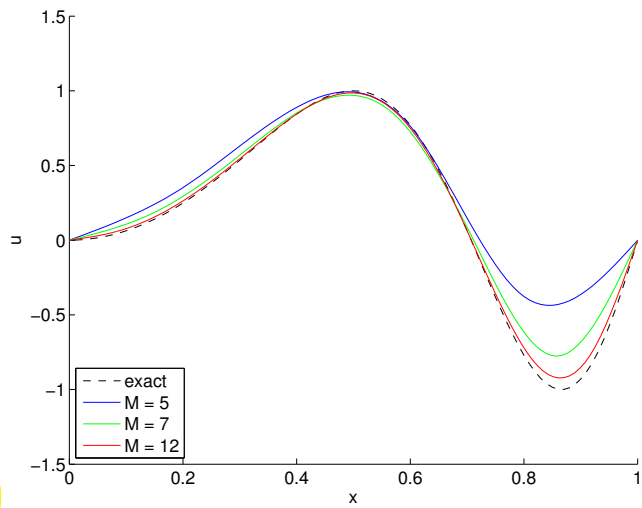


Fig. 46

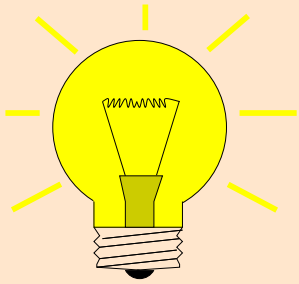
1.5.4 Finite differences

As collocation methods (→ Section 1.5.3), finite difference scheme target the strong form (ODE/PDE) of a boundary value problem. Here we discuss their derivation and implementation for the linear scalar linear 2-point boundary value problem that serves as our model problem.

Focus: 2nd-order linear two-point BVP

$$\mathcal{L}(u) := -\frac{d}{dx}\left(\sigma(x)\frac{du}{dx}(x)\right) = g(x) \,, \quad a \le x \le b \,, \tag{1.5.119}$$
$$u(a) = u_a \quad , \quad u(b) = u_b \,,$$

Construction of finite difference discretization



Idea:

$\boxed{\text{Replace derivatives} \longrightarrow \text{difference quotients}}$

(in finitely many special points = nodes of a mesh/grid)

E.g. $$\frac{d^2u}{dx^2}(x) \approx \frac{u(x+h)-2u(x)+u(x-h)}{h^2}\,, \quad h>0 \text{ "small"}\,. \tag{1.5.138}$$

Remark 1.5.139 (Types of difference quotients)

- Symmetric difference quotient at anchor point x_0

$$\frac{du}{dx}(x_0) \approx \frac{u(x_0+h)-u(x_0-h)}{2h}\,, \quad \text{with span} \quad h>0\,. \tag{1.5.140}$$

- One-sided difference quotients at anchor point x_0

$$\frac{du}{dx}(x_0) \approx \frac{u(x_0+h)-u(x_0)}{h} \approx \frac{u(x_0)-u(x_0-h)}{h}\,, \quad \text{with span} \quad h>0\,. \tag{1.5.141}$$

(1.5.142) Grid for finite difference method as in Sect. 1.5.2.2

➢ $\Omega = [a,b]$ equipped with nodes $(M \in \mathbb{N})$

$$\mathcal{X} := \{a = x_0 < x_1 < \cdots < x_{M-1} < x_M = b\}\,.$$

Fig. 47

➤ mesh/grid

$$\mathcal{M} := \{]x_{j-1}, x_j[: 1 \le j \le M\}\,.$$

Special case:

equidistant mesh: $x_j := a + jh\,, \quad h := \dfrac{b-a}{M}\,.$

☞ $[x_{j-1}, x_j]$, $j = 1,\dots,M$, $\hat{=}$ cells of $\mathcal{M}$, cell size $h_j := |x_j - x_{j-1}$, $j = 1,\dots,M$

meshwidth $h_{\mathcal{M}} := \max\limits_j |x_j - x_{j-1}|$

(1.5.143) Difference quotient approximation

❶ replacement of outer derivative $(x_{j-1/2} = \frac{1}{2}(x_j + x_{j-1}))$:

$$\frac{d}{dx}\left(\sigma(x)\frac{du}{dx}(x)\right)_{|x=x_j} \approx \frac{2}{h_{j-1}+h_j}\left(\sigma(x_{j+1/2})\frac{du}{dx}(x_{j+1/2}) - \sigma(x_{j-1/2})\frac{du}{dx}(x_{j-1/2})\right)\,.$$

Essential: possibility for point evaluation of coefficient function σ: $\sigma \in C^0(]a,b[)$ required.

❷ replacement of inner derivative, e.g.,

$$\frac{du}{dx}(x_{j+1/2}) \approx \frac{u(x_{j+1})-u(x_j)}{h_j}\,.$$

$$-\frac{d}{dx}\left(\sigma(x)\frac{du}{dx}(x)\right)_{|x=x_j} = \frac{\sigma(x_{j-1/2})\dfrac{u(x_j)-u(x_{j-1})}{h_{j-1}} - \sigma(x_{j+1/2})\dfrac{u(x_{j+1})-u(x_j)}{h_j}}{\frac{1}{2}(h_{j-1}+h_j)} \,. \tag{1.5.144}$$

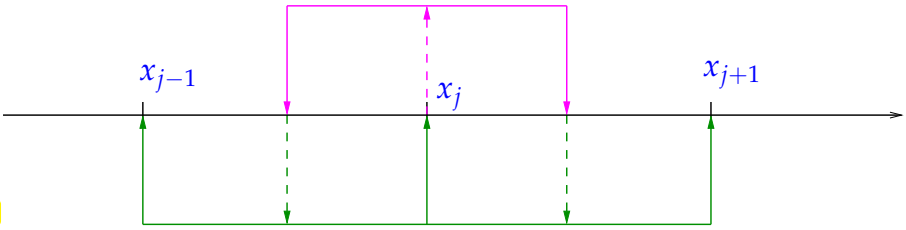


Fig. 48

Construction of (1.5.144):

magenta: outer difference quotient

green: inner difference quotients

From now assume equidistant mesh, uniform meshwidth $h_j = h > 0$, $j = 1,\dots,M$:

$$-\frac{d}{dx}\left(\sigma(x)\frac{du}{dx}(x)\right)_{|x=x_j} = \frac{1}{h^2}\left(-\sigma(x_{j+1/2})u(x_{j+1}) + (\sigma(x_{j+1/2})+\sigma(x_{j-1/2}))u(x_j) - \sigma(x_{j-1/2})u(x_{j-1})\right) \,. \tag{1.5.145}$$

Unknowns in finite difference method: $\boxed{\mu_l = u(x_l), \quad l = 1,\dots,M-1}$

Remark 1.5.146 (Imposing boundary condition in finite difference method)

In (1.5.145) the values $u(x_0)$ and $u(x_M)$ are replaced with the prescribed boundary values u_a and u_b, respectively. Below this is realized by formally setting $\mu_0 := u_a$ and $\mu_M := u_b$.

$$-\frac{d}{dx}\left(\sigma(x)\frac{du}{dx}(x)\right) = g(x) \;, a \le x \le b \,.$$

$\leftarrow$ restriction to $\mathcal{X}$, use (1.5.145)

$$\frac{-\sigma(x_{j+1/2})\mu_{j+1} + (\sigma(x_{j+1/2})+\sigma(x_{j-1/2}))\mu_j - \sigma(x_{j-1/2})\mu_{j-1}}{h^2} = g(x_j) \,, \quad j = 1,\dots,M-1 \,. \tag{1.5.147}$$

$$\Updownarrow$$

$$\boxed{\mathbf{A}\vec{\mu} = \vec{\varphi}} \,, \quad \text{with} \quad (\mathbf{A})_{jl} = h^{-2} \cdot \begin{cases} 0 & \text{, if } |j-l| > 1 \,, \\ -\sigma(x_{j+1/2}) & \text{, if } j = l-1 \,, \\ \sigma(x_{j-1/2}) + \sigma(x_{j-1/2}) & \text{, if } j = l \,, \\ -\sigma(x_{l+1/2}) & \text{, if } l = j-1 \,. \end{cases} \tag{1.5.148}$$

$$\varphi_j = \begin{cases} g(x_1) + \sigma(x_{1/2})u_a & \text{, if } j = 1 \,, \\ g(x_j) & \text{, if } 1 < j < M-1 \,, \\ g(x_{M-1}) + \sigma(x_{M-1/2})u_b & \text{, if } j = M-1 \,. \end{cases}$$

An $(M-1)\times(M-1)$ tridiagonal linear system of equations with

$$\mathbf{A} = \frac{1}{h^2}\begin{pmatrix} \sigma_{\frac{1}{2}}+\sigma_{\frac{3}{2}} & -\sigma_{\frac{3}{2}} & & & & \\ -\sigma_{\frac{3}{2}} & \sigma_{\frac{3}{2}}+\sigma_{\frac{5}{2}} & -\sigma_{\frac{5}{2}} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & -\sigma_{M-\frac{3}{2}} \\ & & & & -\sigma_{M-\frac{3}{2}} & \sigma_{M-\frac{1}{2}}+\sigma_{M-\frac{3}{2}} \end{pmatrix}, \tag{1.5.149}$$

where $\sigma_* := \sigma(x_*)$.

(Up to scaling with h) the finite difference approach and the linear finite element Galerkin scheme ($\rightarrow$ Sect. 1.5.2.2) yield the *same* system matrix for the BVP (1.5.119) and its associated variational problem (1.4.23), *cf.* (1.5.148) and (1.5.88).

1.6 Convergence of Discrete Solutions

After discretization and solution of the resulting system of finitely many equations we obtain an approximate solution. This section discusses how to assess the quality of approximate solutions quantitatively.

For the full elastic string model (1.2.51)/(1.3.15) or the taut string model in graph description (1.4.23) there is an exact solution, a function $\mathbf{u} : [0,1] \mapsto \mathbb{R}^2$ or $u : [a,b] \mapsto \mathbb{R}$, respectively. In general, no algorithm will be able to find it. Instead, we can only approximate it based on discretization.

Discretization schemes (Galerkin approach, Sect. 1.5.2 collocation methods, Sect. 1.5.3)	$\longrightarrow$	Approximate solution $\mathbf{u}_N : [0,1] \mapsto \mathbb{R}^2 / u_N : [a,b] \mapsto \mathbb{R}$ (functions $\in$ trial space V_N)

Desirable: approximation u_N "close to" exact solution u: *rigorous meaning* **?**

$\updownarrow$

How to measure discretization error $u - u_N$ **?**

In this section we will learn ways to measure the distance of $\mathbf{u}/u$ and $\mathbf{u}_N/u_N$, the "size" of the discretization error. In numerical experiments we will explore the behavior of discretization errors for various discretization schemes.

Remark 1.6.1 (Related: Convergence of approximations of functions)

In the course "Numerical Methods for CSE" [4] a whole chapter was devoted to the approximation of functions in 1D [4, Chapter 4]. There, given a function $f : I \subset \mathbb{R} \rightarrow \mathbb{R}$, I a finite interval, we built a "simple" function $\tilde{f} : I \rightarrow \mathbb{R}$ that could be described by a finite number N of degrees of freedom. For instance, $\tilde{f}$ could be a global polynomial encoded through its expansion coefficients with respect to a basis of the space of polynomials [4, Section 4.1]. In this context the counterpart of the discretization error is the approximation error $f - \tilde{f}$, also a function on I.

The size of $f - \tilde{f}$ was naturally measured by computing a suitable norm [4, § 3.2.61]. The supremum norm, the L^2-norm, and L^1-norm were introduced as most important specimens of relevant norms.

Remark 1.6.2 ("Convergence" in other settings)

We encountered the issue of *convergence of approximate solutions* before:

- Numerical quadrature [4, Chapter 5]: study of asymptotic behavior of quadrature error
- Numerical integration [4, Chapter 11]: discretization error of single step methods

---

Remark 1.6.3 (Grid functions)

Note: for finite differences ($\rightarrow$ Sect. 1.5.4) we get no solution function, only a grid function $\mathcal{X} \mapsto \mathbb{R}$ ("point values"). How can we compare u and u_N in this case?

We have two options:

❶ reconstruction of a function u_N through postprocessing, e.g., linear interpolation, or other techniques introduced in [4, Chapter 3].

❷ use of a mesh-depedent norm, for instance, the maximum difference of point values on $\mathcal{X}$ (discrete supremum norm).

1.6.1 Norms on function spaces

Tools for measuring discretization errors: **norms** on function spaces/grid function spaces

Recall from analysis or the introduction to numerical methods:

Definition 1.6.4. Norm (on a vector space) $\rightarrow$ **[4, Def. 1.5.65]**

A norm $\|\cdot\|_V$ on an $\mathbb{R}$-vector space V is a mapping $\|\cdot\|_V : V \mapsto \mathbb{R}_0^+$, such that

$$\begin{aligned} &\text{(definiteness)} && \|v\|_V = 0 \iff v = 0 \quad \forall v \in V && \text{(N1)} \\ &\text{(homogeneity)} && \|\lambda v\|_V = |\lambda| \|v\|_V \quad \forall \lambda \in \mathbb{R},\ \forall v \in V\,, && \text{(N2)} \\ &\text{(triangle inequality)} && \|w+v\|_V \leq \|w\|_V + \|v\|_V \quad \forall w, v \in V\,. && \text{(N3)} \end{aligned}$$

Next we recall important norms on function spaces, *cf.* [4, Eq. (3.2.62)], [4, Eq. (3.2.63)], [4, Eq. (3.2.64)]:

Definition 1.6.5. Supremum norm

The supremum norm of an (essentially) bounded function $\mathbf{u} : \Omega \mapsto \mathbb{R}^n$ is defined as

$$\|\mathbf{u}\|_\infty \left(= \|\mathbf{u}\|_{L^\infty(\Omega)} \right) := \sup_{x \in \Omega} \|\mathbf{u}(x)\|\,, \quad \mathbf{u} \in (L^\infty(\Omega))^n\,. \tag{1.6.6}$$

- $L^\infty(\Omega)$ denotes the vector space of essentially bounded functions. It is the instance for $p = \infty$ of an L^p-space.
- The notation $\|\cdot\|_\infty$ hints at the relationship between the supremum norm of functions and the maximum norm for vectors in $\mathbb{R}^n$.

- For $n = 1$ the Euclidean vector norm in the definition reduces to the modulus $|u(x)|$.
- The norm $\|\mathbf{u} - \mathbf{u}_N\|_{L^\infty(\Omega)}$ measures the maximum distance of the function values of $\mathbf{u}$ and $\mathbf{u}_N$.
- $\|\mathbf{u} - \mathbf{u}_N\|_{L^\infty(\Omega)} \hat{=}$ maximal pointwise error

Definition 1.6.7. Mean square norm/L^2-norm

For a function $\mathbf{u} \in (C^0_{\mathrm{pw}}(\Omega))^n$ the mean square norm/L^2-norm is given by

$$\|\mathbf{u}\|_0 \left(= \|\mathbf{u}\|_{L^2(\Omega)} \right) := \left(\int_\Omega \|\mathbf{u}(x)\|^2 \,\mathrm{d}x \right)^{1/2} , \quad \mathbf{u} \in (L^2(\Omega))^n .$$

- $L^2(\Omega)$ designates the vector space of square integrable functions, another L^p-space (for $p = 2$) and a Hilbert space.
- The "0" in the notation $\|\cdot\|_0$ refers to the absence of derivatives in the definition of the norm.
- Obviously, the L^2-norm is weaker than the supremum norm:

$$\|v\|_{L^2([a,b])} \leq \sqrt{|b-a|}\|v\|_{L^\infty([a,b])} \quad \forall v \in C^0_{\mathrm{pw}}([a,b]) .$$

  In particular, the L^2-norm of the discretization error may be small despite large deviations of u_N from u, provided that these deviations are very much *localized*.
- Parlance: $\|u - u_N\|_{L^2(\Omega)} \hat{=}$ mean square error.

Relevant error norms are suggested by application context/physics!

(1.6.8) Energy norm

We consider the model for a homogeneous taut string in physical space, see (1.4.23), with associated total potential energy functional

$$J(u) := \int_a^b \tfrac{1}{2}\left|\frac{du}{dx}(x)\right|^2 + \widehat{g}(x)u(x)\,\mathrm{d}x , \quad u \in C^1_{\mathrm{pw},0}([a,b]) , \tag{1.6.9}$$

where, for the sake of simplicity, we assume $u_a = u_b = 0$.

A manifestly relevant error quantity of interest is the deviation of energies

$$E_J := |J(u) - J(u_N)| .$$

We adopt the concise notations introduced for abstract (linear) variational problems in Rem. 1.3.31, § 1.4.7:

$$\boxed{J(u) = \tfrac{1}{2}\mathsf{a}(u,u) - \ell(u)} , \qquad \mathsf{a}(u,v) := \int_a^b \frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x , \quad \ell(v) := -\int_a^b \widehat{g}(x)v(x)\,\mathrm{d}x ,$$

where a is a *symmetric* bilinear form, see Def. 1.3.22.

Assumption: $u_N \in V_{N,0} \hat{=}$ Galerkin solution based on discrete trial space $V_{N,0} \subset V_0$.

$$\blacktriangleright \quad \begin{aligned} \mathsf{a}(u,v) &= \ell(v) \quad \forall v \in V_0 := C^1_{\mathrm{pw},0}([a,b]) \;, \\ \mathsf{a}(u_N,v_N) &= \ell(v_N) \quad \forall v_N \in V_{N,0} \subset V_0 \;. \end{aligned} \tag{1.6.10}$$

We can use the defining variational equations for u and u_N to express

$$J(u) - J(u_N) = -\tfrac{1}{2}(\mathsf{a}(u,u) - \mathsf{a}(u_N,u_N)) \overset{(*)}{=} -\tfrac{1}{2}\mathsf{a}(u+u_N, u-u_N) \;. \tag{1.6.11}$$

$(*)$: this is a straightforward consequence of the bilinearity of a, see Def. 1.3.22, *c.f.* the well known identity $a^2 - b^2 = (a+b)(a-b)$ for $a,b \in \mathbb{R}$. These manipulations will be revisited in Remark 2.4.35.

Concretely,

$$\begin{aligned} |J(u) - J(u_N)| &= \tfrac{1}{2}\left| \int_a^b \frac{d}{dx}(u+u_N) \cdot \frac{d}{dx}(u-u_N)\,\mathrm{d}x \right| \\ &\overset{(*)}{\leq} \tfrac{1}{2}\left(\int_a^b |\frac{d}{dx}(u+u_N)|^2\,\mathrm{d}x \right)^{1/2} \left(\int_a^b |\frac{d}{dx}(u-u_N)|^2\,\mathrm{d}x \right)^{1/2} . \end{aligned} \tag{1.6.12}$$

$(*)$: due to Cauchy-Schwarz inequality for inner products

$$\int_\Omega u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \leq \|u\|_{L^2(\Omega)}\|v\|_{L^2(\Omega)} \quad \forall u,v \in L^2(\Omega) \;. \tag{1.6.13}$$

Definition 1.6.14. H^1-seminorm

For a function $u \in C^1_{\mathrm{pw}}([a,b])$ the H^1-seminorm reads

$$|u|^2_{H^1([a,b])} := \int_a^b |\frac{du}{dx}(x)|^2\,\mathrm{d}x \;. \tag{1.6.15}$$

- $|\cdot|_{H^1([a,b])}$ is merely a semi-norm, because it only satisfies norm axioms (N2) and (N3), but fails to be definite: $|\cdot|_{H^1([a,b])} = 0$ for constant functions.
- In the setting of the homogeneous taut string model, we have

  $$|u|^2_{H^1([a,b])} = \mathsf{a}(u,u) \quad \blacktriangleright \quad |\cdot|_{H^1([a,b])} \text{ is called the energy norm for the model.}$$

  More explanations will be given in Sect. 2.2.3.
- On $C^1_{\mathrm{pw},0}([a,b])$ the semi-norm $|\cdot|_{H^1([a,b])}$ is a genuine norm $\rightarrow$ Def. 1.6.4. See proof of Thm. 2.3.31.

From (1.6.12)

$$\|u-u_N\|_{H^1(\Omega)} \leq \epsilon \quad \blacktriangleright \quad \begin{aligned} |J(u) - J(u_N)| &\leq |u+u_N|_{H^1(\Omega)}|u-u_N|_{H^1(\Omega)} \\ &\overset{\text{(N3)}}{\leq} (2|u|_{H^1(\Omega)} + \epsilon)\,\epsilon \;. \end{aligned} \tag{1.6.16}$$

☛ estimate of the energy norm of the discretization error paves the way for bounding the energy deviation.

Remark 1.6.17 (Norms on grid function spaces)

To measure the discretization error for finite difference schemes (→ Sect. 1.5.4) one may resort to mesh-dependent norms, for instance

$$\text{(discrete) } l^2\text{-norm} \quad : \quad \|\vec{\mu}\|^2_{l^2(\mathcal{X})} := \sum_{j=0}^{M} \tfrac{1}{2}(h_j + h_{j+1})|\mu_j|^2 \,, \tag{1.6.18}$$

$$\text{(under convention} \quad h_0 := 0,\, h_{M+1} := 0) \,,$$

$$\text{(discrete) maximum norm} \quad : \quad \|\vec{\mu}\|_{l^\infty(\mathcal{X})} := \max_{j=0,\dots,M} |\mu_j| \,. \tag{1.6.19}$$

Remark 1.6.20 (Approximate computation of norms)

A standard approach to *testing* implementations of numerical methods for 2-point BVP: Examine norm of discretization error for test cases with (analytically) known exact solution u.

Even for numerical methods computing $u_N \in V_N \subset V$ (Galerkin methods → Sect. 1.5.2, collocation methods → Sect. 1.5.3):

usually the exact computation of $\|u - u_N\|$ is impossible or very difficult.

Option: approximate evaluation of norm $\|u - u_N\|$

- supremum norm $\|\cdot\|_\infty$: approximation by sampling on discrete point set.
- L^2-norm, energy norm: numerical quadrature [4, Chapter 5]
  (Gauss quadrature for spectral schemes, composite quadrature for mesh based schemes)

! Error introduced by approximation of norm must be smaller than discretization error
(➢ use "overkill" quadrature/sampling, cost does not matter much when testing).

1.6.2 Algebraic and exponential convergence

In Rem. 1.6.1 we pointed out parallels between studying approximation errors and discretization errors. In the case of approximation errors we discovered rather regular behavior when adopting an asymptotic perspective, which was introduced, e.g., in [4, Section 4.1.2]. It regards suitable norms of approximation errors as functions of the number N of parameters for families of approximation schemes and examines their decay as $N \to \infty$. We do the same for discretization errors.

Convergence: asymptotic perspective

Crucial: convergence is an *asymptotic notion* **!**

sequence of discrete models ⇒ sequence of approximate solutions $(u_N^{(i)})_{i\in\mathbb{N}}$
⇒ study sequence $(\|u_N^{(i)} - u\|)_{i\in\mathbb{N}}$

created by *variation* of a discretization parameter:

(1.6.22) Discretization parameters

The most general discretization parameter is the total number of unknowns (= degrees of freedom) in the discrete model. Often, this is controlled by other discretization parameters, of which the following two are widely used:

- *meshwidth* $h > 0$ for finite differences (→ Section 1.5.4), p.w. linear finite elements (→ Section 1.5.2.2), spline collocation (→ Section 1.5.3.2)
- *polynomial degree* for spectral collocation (→ Section 1.5.3.1), spectral Galerkin discretization (→ Section 1.5.2.1)

Experiment 1.6.23 (Numerical studies of convergence)

Focus: Linear 2-point boundary value problem $-\frac{d^2u}{dx^2} = g(x)$, $u(0) = u(1) = 0$ on $\Omega =]0,1[$, variational form (1.5.30),

exact solution $u(x) = \sin(2\pi x^2)$ (→ setting of Exp. 1.5.29)

❶ finite difference discretization on equidistant mesh, meshwidth $h > 0$ (→ Sect. 1.5.4)

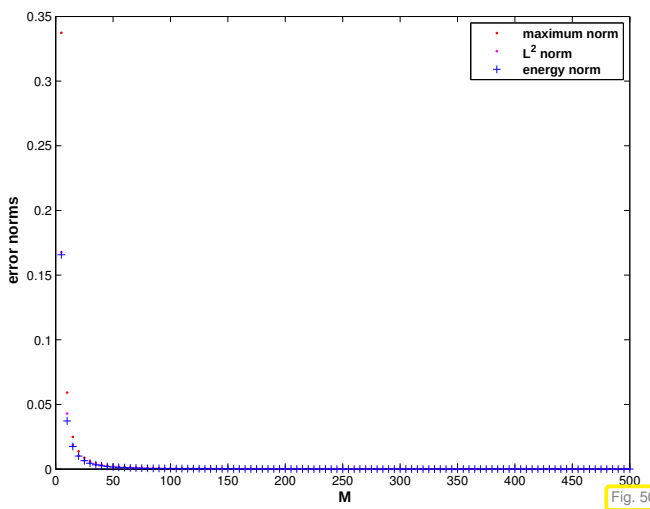


Fig. 49

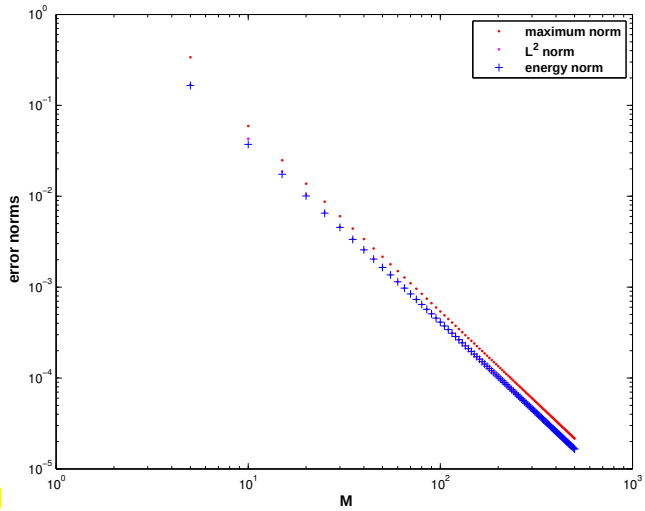


Fig. 50

What is plotted are the discrete versions of the L^2-norm and supremum norm, see Rem. 1.6.17.

The energy norm of the error was computed approximately by means of the midpoint quadrature formula, *cf.* (1.5.86),

$$\text{energy norm}(\text{error})^2 := \sum_{j=1}^{M} h_j \left| \frac{\mu_j - \mu_{j-1}}{h_j} - \frac{du}{dx}(x_{j-1/2}) \right|^2 .$$

→ Rem. 1.6.20 on the approximate computation of norms of the discretization error.

❷ Spectral collocation, polynomial degree $p \in \mathbb{N}$ → Section 1.5.3.1

Monitored: supremum norm (1.6.6), L^2-norm (1.6.1) of discretization error $u - u_N$ (*approximated* by "overkill" Gaussian quadrature with 10^4 nodes)

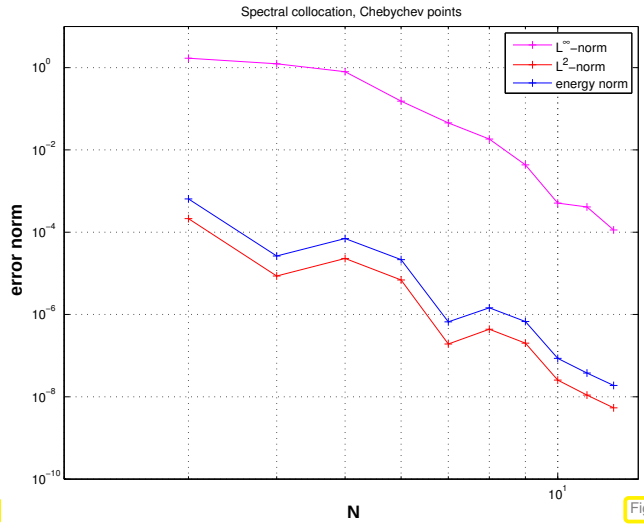


Fig. 51

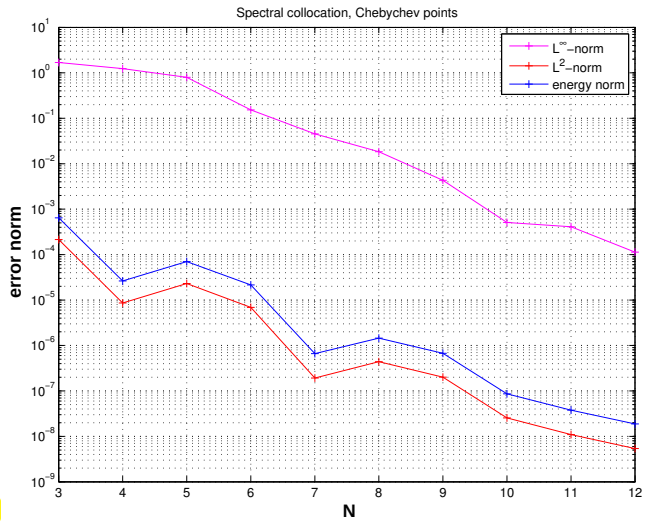


Fig. 52

❸ Spline collocation on equidistant mesh, meshwidth $h > 0$ ($\rightarrow$ Section 1.5.3.2)

Monitored: supremum norm (1.6.6), L^2-norm (1.6.1) of $u - u_N$ (*approximated* by sampling on fine grid with 10^4 points)

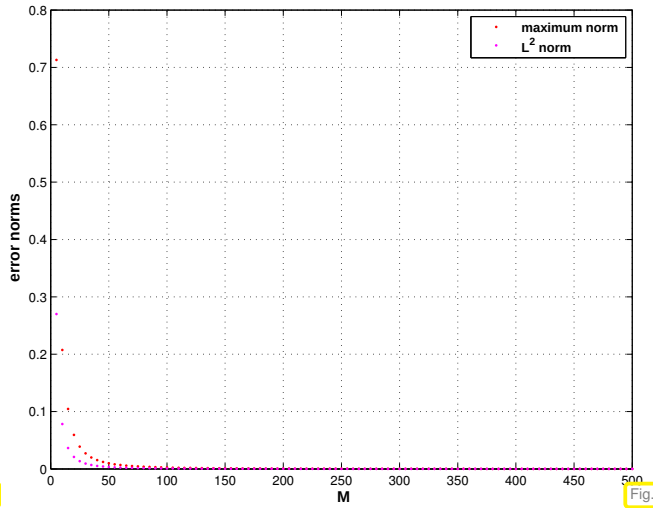


Fig. 53

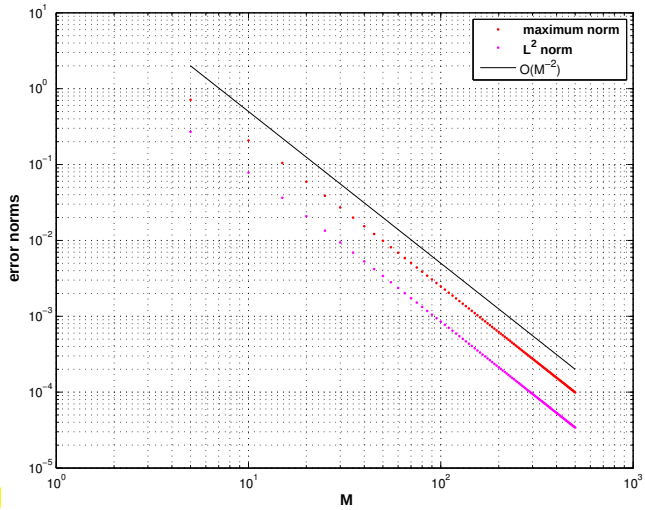


Fig. 54

❹ Spectral Galerkin based on degree $p \in \mathbb{N}$ polynomials $\rightarrow$ Section 1.5.2.1

Monitored: supremum norm (1.6.6), L^2-norm (1.6.1) of discretization error $u - u_N$ (*approximated* by trapezoidal rule on fine grid with 10^4 points)

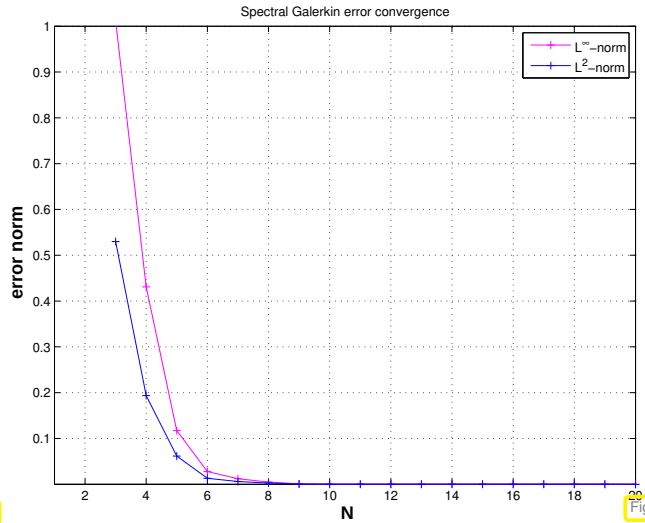


Fig. 55

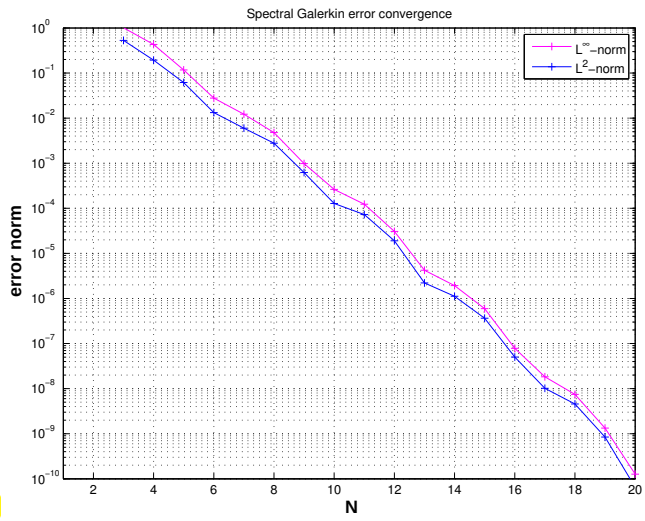


Fig. 56

Observation: ▷ 'Empiric convergence" in all cases
▷ different qualitative behavior (of norm of discretization error)

How to compare different discretizations **?**

Unified view: Study $\|u - u_N\|$ as function of number N of unknowns (degrees of freedom)

measure for costs incurred by method

Definition 1.6.24. Types of convergence → **[4, Section 4.1.2], [4, Def. 4.1.31]**

$\|u - u_N\| = O(N^{-\alpha})$, $\alpha > 0$ $:\Longleftrightarrow$ algebraic convergence with rate α
$\|u - u_N\| = O(\exp(-\gamma N^{\delta}))$, with $\gamma, \delta > 0$ $:\Longleftrightarrow$ exponential convergence
(asymptotically for $N \to \infty$)

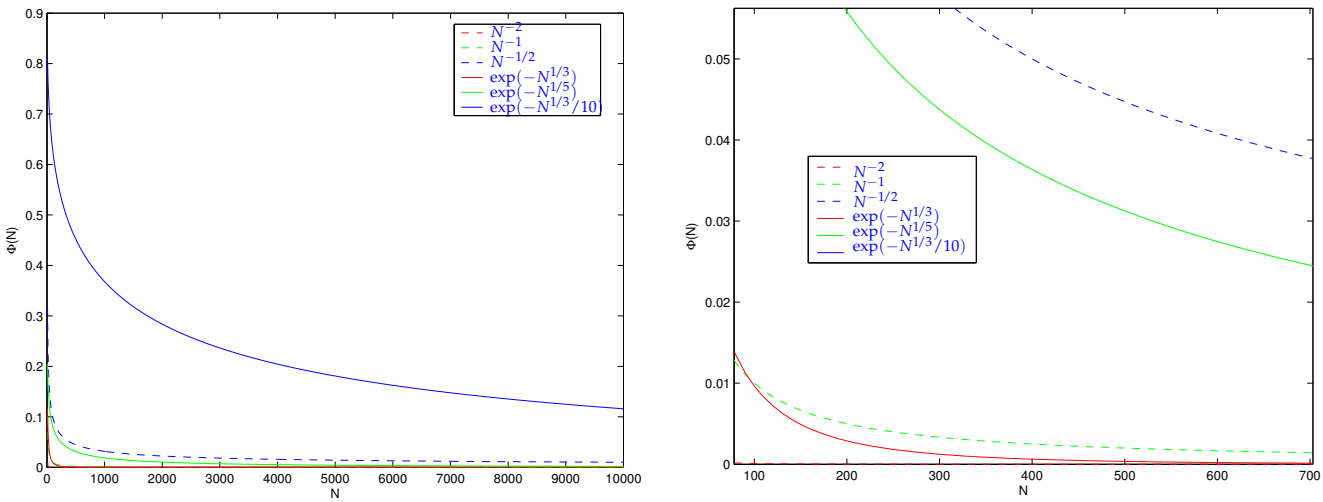
✍ recall notation (Landau-O):

$$f(N) = O(g(N)) \quad :\Leftrightarrow \quad \begin{array}{c} \exists N_0 > 0, \exists C > 0 \text{ independent of } N \\ \text{such that } |f(N)| \leq C g(N) \text{ for } N > N_0 \,. \end{array} \tag{1.6.25}$$

Definition 1.6.26. Rate of convergence

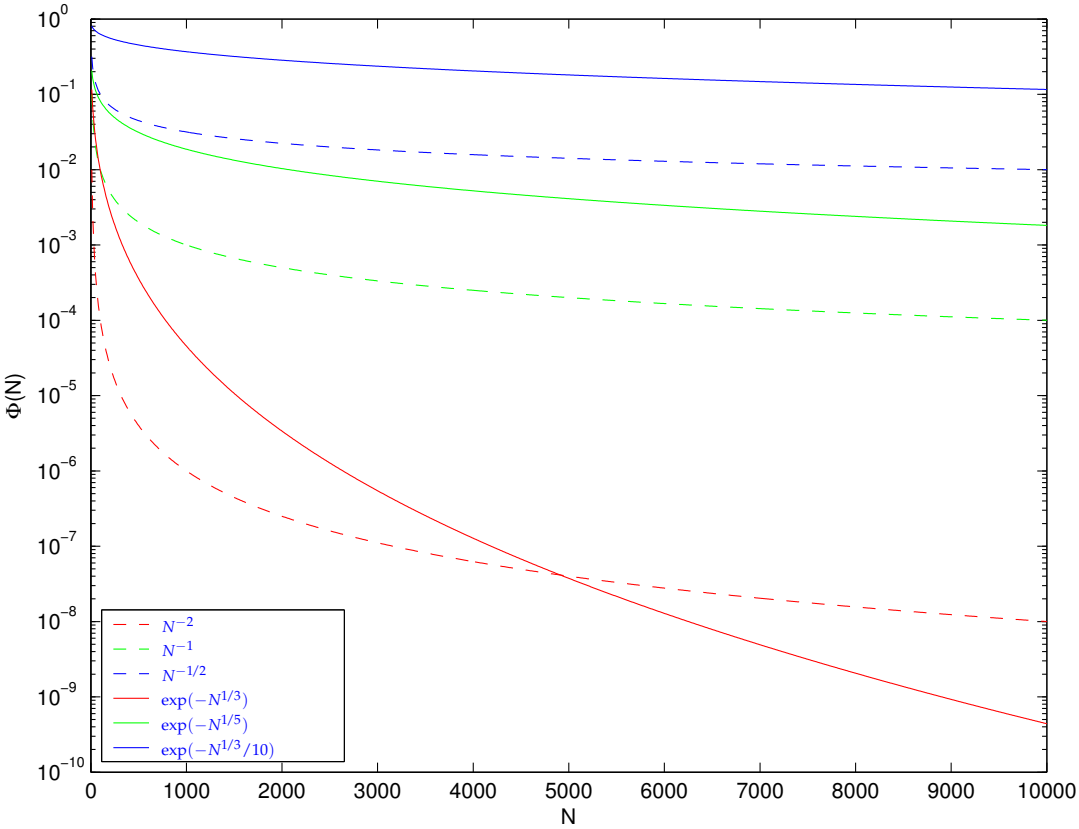
In the case of algebraic convergence the exponent α in Def. 1.6.24 is called the rate of (algebraic) convergence.

The following plots illustrate the qualitative behavior of error norms implied by the two different types of convergence for various parameters.

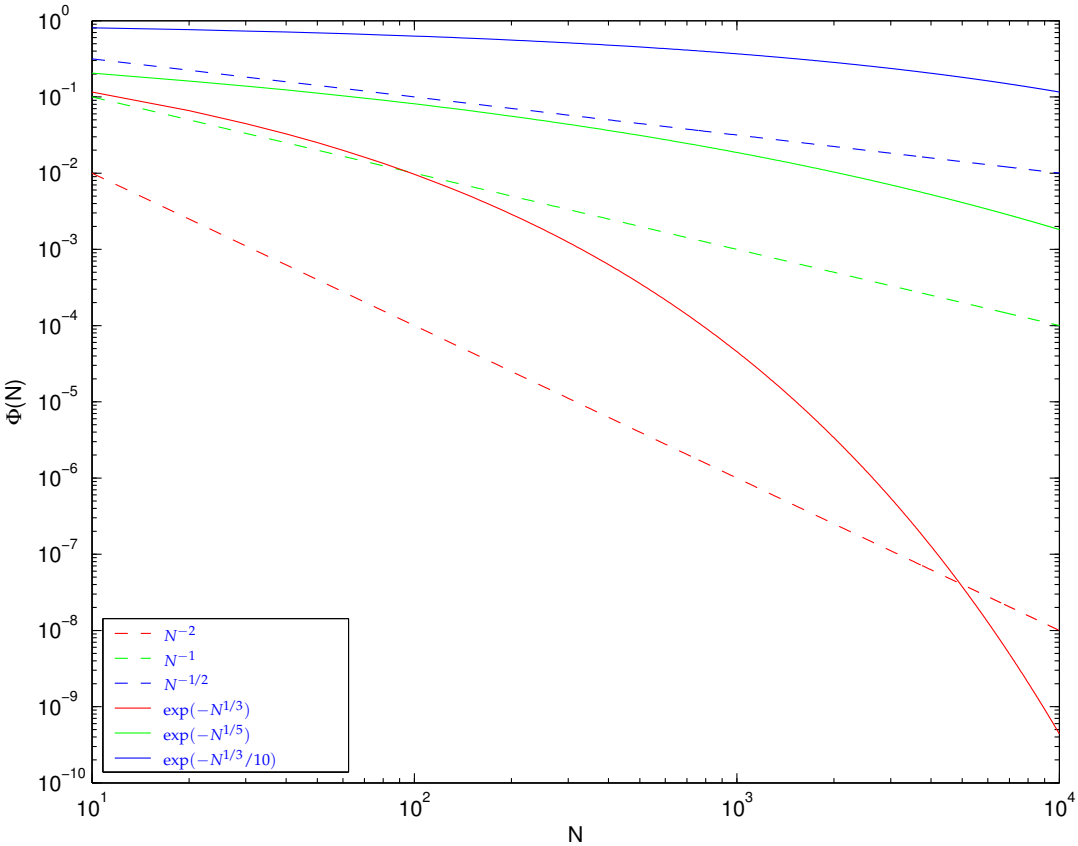


Linear plot of qualitative convergence behavior: algebraic/exponential convergence rates

Exponential convergence will always win (asymptotically)



Log-linear plot of decrease of discretization error for algebraic/exponential convergence rates



Log-log plot of decrease of discretization error for algebraic/exponential convergence rates

(1.6.27) Exploring convergence empirically $\rightarrow$ **[4, Rem. 4.1.33]**

When (in homework problems) you are asked to "investigate the (asymptotic) convergence of a method" in a numerical experiment, you are expected to make a

qualitative and *quantitative* statement

about the asymptotic behavior of a suitable norm ($*$) of the discretization error in the sense of Definition 1.6.24:

- "qualitative": does algebraic or exponential convergence prevail, or none of these?
- "quantitative": the rate α in the case of algebraic convergence, γ, δ in the case of exponential convergence.

($*$): the norm of interest and its evaluation in the sense of Remark 1.6.20 has to be specified as part of the question!

How to tease qualitative/quantitative information about asymptotic convergence out of raw norms of discretization error?

Given: data tuples (N_i, ϵ_i), $i = 1, 2, 3, \ldots$, $N_i \hat{=}$ problem sizes, $\epsilon_i \hat{=}$ error norms

1. Conjecture: algebraic convergence: $\epsilon_i \approx C N_i^{-\alpha}$

$$\log(\epsilon_i) \approx \log(C) - \alpha \log N_i \qquad \text{(affine linear in log-log scale).}$$

   - (almost) linear error plot in doubly logarithmic scale
   - linear regression on data $(\log N_i, \log \epsilon_i)$, $i = 1, 2, 3, \ldots$ to determine rate α, see Code 1.6.30.

2. Conjecture: exponential convergence: $\epsilon_i \approx C \exp(-\gamma N_i^\delta)$

$$\log \epsilon_i \approx \log(C) - \gamma N_i^\delta \,. \tag{1.6.28}$$

$$\log \epsilon_i - \log \epsilon_{i-1} \approx -\gamma (N_i^\delta - N_{i-1}^\delta) \,, \tag{1.6.29}$$

$$\frac{\log \epsilon_{i+1} - \log \epsilon_i}{\log \epsilon_i - \log \epsilon_{i-1}} \approx \frac{N_{i+1}^\delta - N_i^\delta}{N_i^\delta - N_{i-1}^\delta} \,.$$

Special case: geometric increase/decrease of problem size parameters: $N_i = Q N_{i-1}$ for some *known* $Q > 0$.

$$\blacktriangleright \quad \frac{\log \epsilon_{i+1} - \log \epsilon_i}{\log \epsilon_i - \log \epsilon_{i-1}} \approx \frac{Q^\delta - 1}{1 - Q^{-\delta}} \,.$$

From this you can determine δ by solving a quadratic equation. Then you get γ from (1.6.29) or, as above, by linear regression from (1.6.28).

Alternative: non-linear least squares fit ($\rightarrow$ [4, Section 6.5]) to determine δ:

$$(c, \gamma, \delta) = \operatorname{argmin} \left\{ \sum_i |\log \epsilon_i - c + \gamma N_i^\delta|^2 \right\} ,$$

residual $\leftrightarrow$ validity of conjecture. This can be done by a short MATLAB code ($\rightarrow$ exercise)

MATLAB code 1.6.30: Probing for algebraic convergence

```
function alpha = eoc(N,err,fromindex,filename)
% Function for the investigation of algebraic convergence of error
    norms as a
```

```
% function of a problem size parameter.  The argument N has to pass and
% vector of length L > 1 of problem size parameter values, whereas the
% L-vector err contains the corresponding error norms.  The
% argument fromindex∈ {1,...,L − 1} restricts the relevant
% data to fromindex,...,L} in order to suppress the impact of
% possible pre-asymptotic behavior, see Example 1.6.34.
L = length(N); if (L ~= length(err)), error('length mismatch'); end
if (nargin < 3), fromindex = 1; end
N = reshape(N,L,1); err = reshape(err,L,1);
if ((min(N) <= 0) || (min(abs(diff(N))) == 0.0)), error('Invalid sieze parameters'); end
if (min(err) < 0), error('Negative data'); end
% Perform linear regression, aka least squares fitting to a line. polyfit
% returns the coefficients of the linear polynomial, the first of which is its slope
p = polyfit(log(N(fromindex:end)),log(err(fromindex:end)),1); alpha = -p(1);
% Additional graphical output; the "ideal curve" is added as a black solid line
figure('name','error plot');
loglog(N,err,'r-*'); hold on; % plot data
ordcurve = N(fromindex:end).^(-alpha);
ordcurve = ordcurve*err(fromindex)/ordcurve(1);
loglog(N(fromindex:end),ordcurve,'k-'); % plot calibration line
xlabel('problem size N','fontsize',14);
ylabel('error norm','fontsize',14);
legend('error',sprintf('O(N^{%f})',alpha));
if (nargin > 3), print('-depsc2',sprintf('%s.eps',filename)); end
```

C++ code 1.6.31: Estimating the rate of algebraic convergence

```
double eoc(const Eigen::VectorXd &N,const Eigen::VectorXd &err,
      unsigned fromindex = 0, std::string filename = "conv.eps") {
// The argument N has to pass a sorted vector of length L > 1 of
// problem size parameter values, whereas the L-vector err
// contains the corresponding error norms.  The argument
// fromindex∈ {1,...,L − 1} restricts the relevant data to
// fromindex,...,L} in order to suppress the impact of
// possible pre-asymptotic behavior, see Example 1.6.34.
// Returns the estimated rate of convergence.
  const unsigned dim = N.size();
  // Consistency check for arguments
  assert(dim > 1);
  assert(fromindex + 1 < dim);
  assert(err.size() == dim);

  //check if data is proper, that is, positive
  assert(std::none_of(N.data(), N.data()+dim,
```

```
            [] (double d) { return d <= 0; }));
    assert(std::none_of(err.data(), err.data()+dim,
            [] (double d) { return d <= 0; }));

    //check no two elements in N are equal, sorting assumed!
    assert(std::is_sorted(N.data(), N.data()+dim));
    assert(N.data()+dim == std::adjacent_find(N.data(), N.data()+dim));

    //truncate preasymptotic behavior if desired:
    const unsigned int newdim = dim - fromindex;

    //compute log(N) and log(err) componentwise
    auto logfun = [] (double d) {return std::log(d);};
    Eigen::VectorXd Nlog(newdim), errlog(newdim);
    std::transform(N.data()+fromindex,N.data()+dim,Nlog.data(),logfun);
    std::transform(err.data()+fromindex,err.data()+dim,errlog.data(),logfun);

    // perform linear regression, aka least squares fitting to a line. linearFit
    // returns the coefficients of the linear polynomial, the second of which is its slope
    Eigen::VectorXd polyfit = NPDE::linearFit(Nlog, errlog);
    double alpha = -polyfit[1];
    double offset = std::exp(polyfit[0]);

    //evaluate the polynomial
    Eigen::VectorXd polyval(dim);
    std::transform(N.data(), N.data()+dim, polyval.data(),
      [&] (double d) { return offset*std::pow(d,-alpha); });

    //plot the error as red stars and fitted line as blue solid line and save the plot
    mgl::Figure fig; fig.plot(N, err, "r *").label("error");
    std::string lbl = "O(N^{-" + std::to_string(alpha) + "})";
    fig.plot(N, polyval, "b -").label(lbl);
    fig.xlabel("problem size N"); fig.ylabel("error norm");
    fig.setlog(true, true); fig.legend(1, 1); fig.save(filename);

    return alpha;
}
```

Linear fitting of a data vector (x_i, y_i), $i = 1, \dots, n$, means to solve the least squares problem

$$(\beta_*, \alpha_*) := \operatorname*{argmin}_{\alpha,\beta\in\mathbb{R}} \sum_{i=1}^{n} (y_i - \alpha x_i - \beta)^2 \,. \tag{1.6.32}$$

The task of finding (α_*, β_*) is called linear regression. Numerical methods for solving (1.6.32) are covered in [4, Chapter 6].

C++11 code 1.6.33: Linear regression of data

```
// Linear fitting of data passed in vectors x and y of equal length.
// Returns the 2-vector [β*,α*]^T, cf. (1.6.32).
Eigen::VectorXd
linearFit(const Eigen::VectorXd& x, const Eigen::VectorXd& y) {
  assert(x.rows() == y.rows());
  Eigen::MatrixXd X(x.rows(), 2);
  // Set up matrix of overdetermined system of equations
  X.col(0) = Eigen::VectorXd::Constant(x.rows(), 1);
  X.col(1) = x;
  // Solve 2×2 linear system (normal equations)
  return (X.transpose() * X).inverse() * X.transpose() * y;
}
```

Experiment 1.6.34 (Asymptotic nature of convergence)

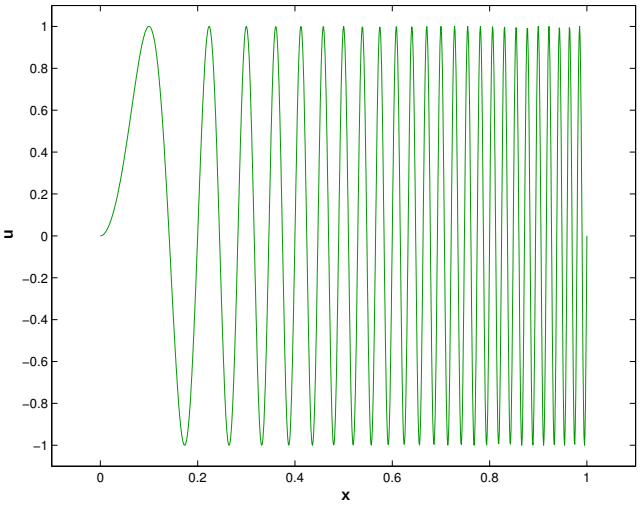


Fig. 57

✦ 2-point BVP $-\frac{d^2u}{dx^2} = g(x)$, $u(0) = u(1) = 0$, $\Omega =]0,1[$,
◁ $u(x) = \sin(50\pi x^2)$

❶ finite difference discretization on equidistant mesh, meshwidth $h > 0$ (→ Sect. 1.5.4)

❷ Spectral Galerkin based on degree $p \in \mathbb{N}$ polynomials → Sect. 1.5.2.1

Evaluations as in Ex. 1.6.23

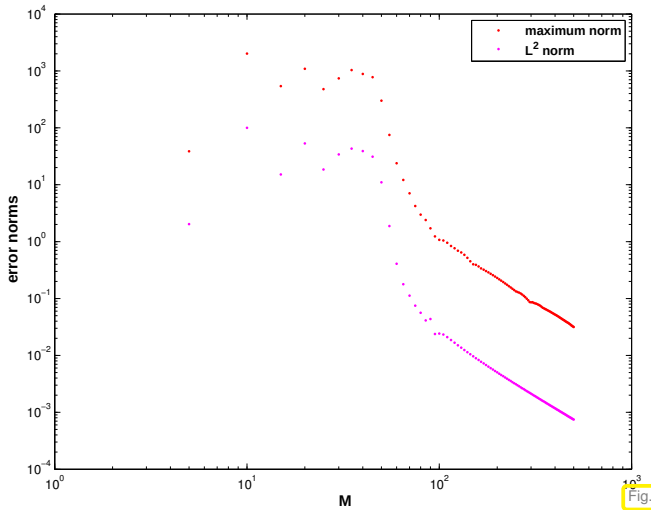


Fig. 58

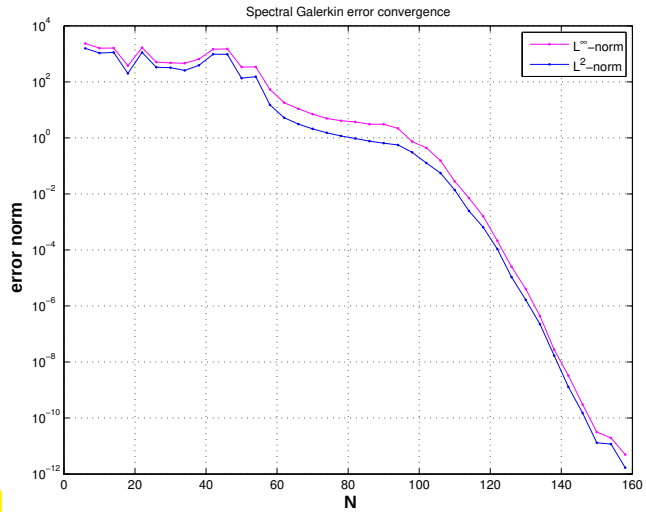


Fig. 59

❶ Finite Difference Method ❷ Spectral Galerkin Method

Observation: for $h \to 0$, $p \to \infty$, algebraic convergence of the finite difference solution and exponential convergenc of the spectral Galerkin solution become conspicuous. This is the "typical" asymptotic behavior of the discretization error norms for these discretization methods.

However, the onset of asymptotic convergence occurs only for rather small meshwidth or large p, respectively, beyond thresholds that may never be reached in a computation. During a long pre-asymptotic phase the error is hardly reduced when increasing the resolution of the discretization.

Experiment 1.6.35 (Convergence and smoothness of solutions)

✦ $\Omega =]0,1[$ (for finite differences), $\Omega =]-1,1[$ (for spectral Galerkin), exact solution of 2-point BVP for ODE $-\frac{d^2u}{dx^2} = g(x)$,

$$u(x) = \begin{cases} \frac{3}{4} - x^2 & \text{, if } |x| < \frac{1}{2}\,, \\ 1 - |x| & \text{, if } |x| \geq \frac{1}{2}\,. \end{cases} \quad \leftrightarrow \quad g(x) = \begin{cases} 2 & \text{, if } |x| < \frac{1}{2}\,, \\ 0 & \text{elsewhere}\,. \end{cases}$$

☞ Solution is still continuous, but no longer smooth: $u \in C^1_{\mathrm{pw}}([0,1])$ only!

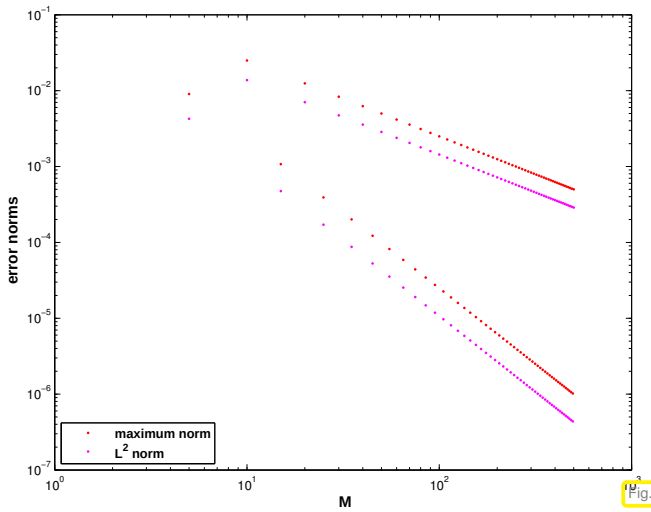


Fig. 60

❶ Finite Difference Method

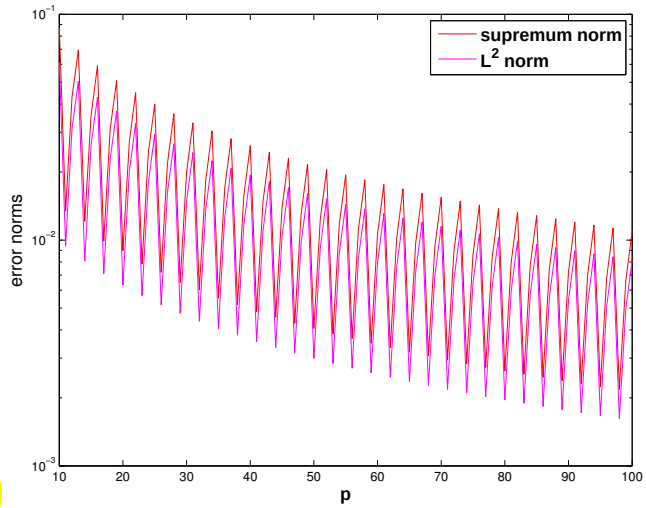


Fig. 61

❷ Spectral Galerkin Method

Observations:
- no more exponential convergence of spectral Galerkin
- FD: different rate of algebraic convergence for even/odd M !

This is a rather familiar observation: For instance, interpolation error estimates for polynomial interpolation of functions in 1D hinge on smoothness assumptions and the interpolation error decays more slowly, when the function enjoys little global smoothness.

Type of asymptotic convergence also depends on data !

?! *Review question(s) 1.6.36.* (Convergence of Galerkin solutions)

1. What is the relationship between the following norms for $v \in C^0([0,1])$: $\|0\|_{L^{\cdot}()}1[]v2$, $\|v\|_{L^\infty(]0,1[)}$, and $\|v\|_{L^1(]0,1[)} := \int_0^1 |v(x)|\,\mathrm{d}x$?
2. Explain how to conclude asymptotic algebraic convergence from tabulated error norms.

3. Consider Chebychev interpolation of a continuous function on a bounded interval. When can you expect exponential convergence, when merely algebraic convergence? Give an example for each case in the form of a non-polynomial continuous function on $[0,1]$.

4. The L^2-norm ϵ_2 of the error of a polynomial spectral Galerkin method as a function of the polynomial degree p behaves like

p	2	3	4	5	6	7	8	9	10
ϵ_2	8.2e-01	4.4e-01	2.9e-01	2.1e-01	2.3e-15	2.4e-15	2.3e-15	2.4e-15	2.3e-15

What can cause such a behavior of the error?

Learning Outcomes

You should be able to ...

- formulate simple mechanical problems as energy minimization problems
- derive variational formulations using the calculus of variations
- know that quadratic minimization problems lead to linear variational equations
- derive a two-point boundary value problem from a variational equation
- understand different smoothness requirements on the solutions for different problem formulations
- understand the principle of Ritz-Galerkin discretization and appreciate the impact of choice of basis.
- apply Ritz-Galerkin approach based on both piecewise polynomials and global polynomials to discretize variational problems in one dimension.
- obtain the linear system of equation that arises from a prescribed Ritz-Galerkin discretization of a linear variational problem.
- Know the principal ideas behind collocation methods and finite difference discretization
- list the advantages of the Galerkin method compared to the collocation approach
- understand the physical relevance of the H^1 semi-norm
- detect algebraic and exponential convergence in numerical experiments.

Bibliography

[1] B. Cockburn and J. Gopalakrishnan. New hybridization techniques. *GAMM-Mitteilungen*, 2:28, 2005.

[2] M. Gander and G. Wanner. From euler, ritz, and galerkin to modern computing. *SIAM Review*, 54(4):627–666, 2012.

[3] W. Hackbusch. *Integral equations. Theory and numerical treatment.*, volume 120 of *International Series of Numerical Mathematics*. Birkhäuser, Basel, 1995.

[4] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[5] K. Nipp and D. Stoffer. *Lineare Algebra*. vdf Hochschulverlag, Zürich, 5 edition, 2002.

[6] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

Chapter 2

Second-order Scalar Elliptic Boundary Value Problems

Contents

2.1 Introduction

(2.1.1) Outline

The previous chapter discussed the transformation of a minimization problem on a function space via a *variational problem* to a differential equation. To begin with, in Section 2.2–Section 2.5, this chapter revisits this theme for models that naturally rely on function spaces over domains in two and three spatial dimensions. Thus the transformation leads to genuine partial differential equations.

Section 2.3 ventures into the realm of *Sobolev spaces*, which provide the framework for rigorous mathematical investigation of variational equations. However, we will approach Sobolev spaces as "spaces of physically meaningful solutions" or "spaces of solutions with finite energy". From this perspective dealing with Sobolev spaces will be reduced to dealing with their norms.

In Section 2.6, we change tack and consider a physical phenomenon (heat conduction) where modelling naturally leads to partial differential equations. On this occasion, we embark on a general discussion of *boundary conditions* in Section 2.7.

Then the fundamental class of second-order elliptic boundary value problems is introduced. Appealing to "intuitive knowledge" about the physical systems underlying the models, key properties of their solutions are presented in Section 2.8.

In Section 2.7 in the context of stationary heat conduction we introduce the whole range of standard boundary conditions for 2nd-order elliptic boundary value problems. The discussion of various *variational formulations* will be resumed in Section 2.10.

Supplementary reading. An excellent *mathematical* introduction to partial differential equations is Evans' book [3]. Chapter 2 gives a very good idea about fundamental properties of various simple PDEs. Chapters 6 and 7 fit the scope of this course chapter, but go way beyond it in terms of mathematical depth.

(2.1.2) Boundary value problems (BVPs)

The traditional concept of a boundary value problem for a partial differential equation (strong form, *cf.* § 1.3.46):

Boundary value problem (BVP)

Given a partial differential operator $\mathcal{L}$, a domain $\Omega \subset \mathbb{R}^d$, a boundary differential operator $\mathcal{B}$, boundary data g, and a source term f, seek a function $u : \Omega \mapsto \mathbb{R}^n$ such that

$$\begin{aligned} \mathcal{L}(u) &= f \quad \text{in} \quad \Omega\,, \\ \mathcal{B}(u) &= g \quad \text{on } \textit{part of (or all)} \text{ boundary } \partial\Omega\,. \end{aligned}$$

Terminology: boundary value problem is scalar $:\Leftrightarrow \quad n = 1$
(in this case the unknown is a real valued function)

(2.1.3) Elliptic boundary value problems

What does elliptic mean **?**

Mathematical theory of PDEs distinguishes three main classes of boundary value problems (BVPs) for partial differential equations (PDE):

- Elliptic BVPs (➢ "equilibrium problems", as discussed in Section 1.2.3, Section 2.2.1, Section 2.2.2)
- Parabolic initial boundary value problems (IBVPs) (➢ evolution towards equilibrium, see Section 6.1)

- Hyperbolic IBVPs, among them wave propagation problems and conservation laws (➢ transport/propagation, see **??**)

The rigorous mathematical definition is complicated and often fails to reveal fundamental properties of, e.g., solutions that are intuitively clear against the backdrop of the physics modelled by a certain PDE. Further discussion of classification in [1, § 1] and [4, Ch. 1].

➢ In the spirit of Section 1.1

Structural properties of a BPV inherited from the modelled system are more important than formal mathematical classification.

2.2 Equilibrium models

We only consider stationary systems. Then, frequently, see Section 1.2.2

equilibrium = minimal energy configuration of a system

Example: elastic string model of Section 1.2 (minimization of energy functional $J(\mathbf{u})$, see (1.2.51))

In this section we study minimization problems for energy functional on spaces of functions $\Omega \mapsto \mathbb{R}$, where $\Omega \subset \mathbb{R}^d$ is a bounded (spatial) domain and $d = 2, 3$.

2.2.1 Taut membrane

Recall: energy functional for pinned *taut* string under gravitational load $\widehat{g}$, see (1.4.14), in terms of displacement (function graph model), see Fig. 28:

$$J(u) := \frac{1}{2}\int_a^b \widehat{\sigma}(x)\left|\frac{du}{dx}(x)\right|^2 - \widehat{g}(x)u(x)\,\mathrm{d}x\,, \qquad \begin{array}{c} u \in C^1_{\mathrm{pw}}([a,b])\,, \\ u(a) = u_a\,,\, u(b) = u_b\,. \end{array} \tag{2.2.1}$$

"2D generalization" of an elastic string

➢ elastic membrane.

Taut drum membranes ▷

Fig. 62

(2.2.2) Configuration space for taut membrane graph model

As in the case of the elastic string (→ § 1.2.1) identifying a suitable configuration space is an essential part of mathematical modeling. Again, we rely on a space of functions, scalar valued for a graph model, *cf.* Fig. 28.

Shape of membrane
⇕
Graph of $u : \Omega \mapsto \mathbb{R}$

"membrane" on spatial domain $\Omega =]0,1[^2$ ▷
(– – – ≙ boundary data)

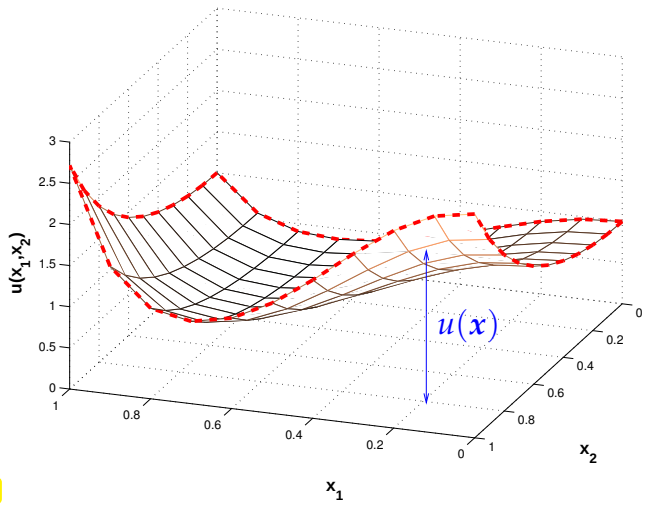


Fig. 63

(2.2.3) Spatial domains

As explained in § 2.2.2 the configuration space for the membrane is a space of functions defined on a spatial domain Ω. In one dimension this was a connected interval and there is not much more to say about it, but in higher dimensions, the boundaries of domains can have special properties, which may affect the well-posedness of boundary value problems and features of their solutions.

General assumptions on spatial domains $\Omega \subset \mathbb{R}^d$:
✎ $d = 1,2,3$ ≙ "dimension" of domain

- ✦ Ω is bounded

  $$\operatorname{diam}(\Omega) := \sup\{\|\boldsymbol{x}-\boldsymbol{y}\| : \boldsymbol{x},\boldsymbol{y} \in \Omega\} < \infty \,,$$

- ✦ Ω has piecewise smooth boundary $\partial\Omega$ ▷
  ("curvilinear polygon/polyhedron")

For $d = 2$ we can distinguish corners (•) and edges (—) of the boundary $\partial\Omega$.

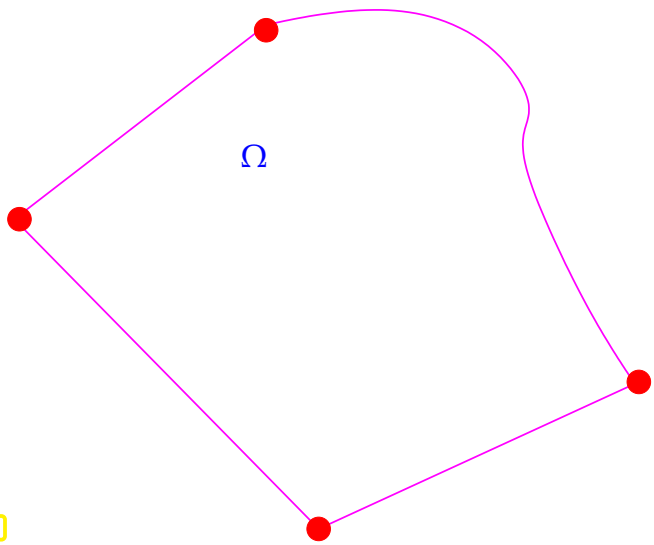


Fig. 64

(2.2.4) Boundary conditions

In the case of the elastic string model we introduced pinning conditions at the endpoint (boundary conditions), *cf.* (1.2.2), (1.4.17). Counterparts for the membrane model:

$$\text{fix} \quad \begin{array}{rcll} u(\boldsymbol{x}) & = & g(\boldsymbol{x}) & \boldsymbol{x} \in \partial\Omega \\ \Updownarrow & & & \\ u_{|\partial\Omega} & = & g & \text{on } \partial\Omega \,. \end{array} \quad \text{for some} \quad g \in C^0(\partial\Omega) \,. \tag{2.2.5}$$

✎ notation: $\partial\Omega$ ≙ boundary of Ω

(2.2.5) means that the displacement of the membrane over $\partial\Omega$ is provided by a prescribed *continuous* function $g : \partial\Omega \mapsto \mathbb{R}$: the membrane is clamped into a rigid frame as illustrated in Fig. 63.

Intuition: g has to be continuous, unless the membrane is to be torn!
(Further discussion in § 2.10.6)

configuration space $V = \left\{ \begin{array}{c} \text{continuous functions } u \in C^0(\overline{\Omega}), \\ \text{with } u_{|\partial\Omega} = g. \end{array} \right\}$

In the notation $C^0(\overline{\Omega})$ the bar above Ω indicates that the functions are supposed to be *continuous up to the boundary*. To understand, why this is important, observe that $x \to \frac{1}{x}$ belongs to $C^0(]0,1[)$, but not to $C^0(\overline{]0,1[})$, because $\overline{]0,1[} = [0,1]$.

Think of the membrane as a grid of taut strings. In light of the considerations of Section 1.4.2 this justifies the following expression for its total potential energy. A detailed derivation can be carried out as for the string model in Section 1.2.3 starting from a mass-spring model.

Taut membrane model: Potential energy functional

Potential energy of a taut membrane (described by graph of $u \in C^0(\Omega)$) under vertical loading:

$$\boxed{J_M(u) := \int_\Omega \underbrace{\tfrac{1}{2}\sigma(\boldsymbol{x})\|\mathbf{grad}\, u\|^2}_{\text{elastic energy}} - \underbrace{f(\boldsymbol{x})u(\boldsymbol{x})}_{\text{potential energy in force field}} \,\mathrm{d}\boldsymbol{x}} \quad . \tag{2.2.7}$$

Here
- ✦ $u : \Omega \mapsto \mathbb{R} \mathrel{\hat{=}}$ displacement function, see Fig. 63, $[u] = \mathrm{m}$,
- ✦ $f : \Omega \mapsto \mathbb{R} \mathrel{\hat{=}}$ force density (pressure), $[f] = \mathrm{N\,m^{-2}}$,
- ✦ $\sigma : \Omega \mapsto \mathbb{R}^+ \mathrel{\hat{=}}$ stiffness, $[\sigma] = \mathrm{J}$.

Supplement 2.2.8 (Gradient, see also § 0.10.11).

Concept from analysis: recall the definition of the gradient of a function $F : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}, F(\boldsymbol{x}) = F(x_1, \dots, x_d)$, see [7, Kap. 7], [5, Eq. (8.1.8)]:

$$\mathbf{grad}\, F(\boldsymbol{x}) := \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_d} \end{bmatrix} \in \mathbb{R}^d \,, \quad \boldsymbol{x} \in \Omega \,.$$

Note: the gradient at $\boldsymbol{x}$ is a column vector of first *partial derivatives*,
read $\mathbf{grad}\, F(\boldsymbol{x})$ as $(\mathbf{grad}\, F)(\boldsymbol{x})$; $\mathbf{grad}\, F$ is a vector valued function $\Omega \mapsto \mathbb{R}^d$.

Also in use (but not in this course) is the "∇-notation": $\boldsymbol{\nabla} F(\boldsymbol{x}) := \mathbf{grad}\, F(\boldsymbol{x})$.

Obviously, as a straightforward consequence of the mean value theorem, a vanishing gradient means that the function has be constant:

$$F \in C^1(\Omega) \quad \text{and} \quad \mathbf{grad}\, F(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} \in \Omega \quad \Rightarrow \quad \exists c \in \mathbb{R}: \quad F(\boldsymbol{x}) = c \quad \forall \boldsymbol{x} \in \Omega \,. \tag{2.2.9}$$

△

Note that

$$\sigma(\boldsymbol{x})\|\mathbf{grad}\, u\|^2 = \sigma(x_1,x_2)\left|\frac{\partial u}{\partial x_1}(x_1,x_2)\right|^2 + \sigma(x_1,x_2)\left|\frac{\partial u}{\partial x_2}(x_1,x_2)\right|^2 ,$$

compare this with the potential energy functional for a taut thin elastic string (graph description)

$$J(u) := \int_a^b \widehat{\sigma}(x)\left|\frac{du}{dx}(x)\right|^2 + \widehat{g}(x)u(x)\,\mathrm{d}x\,, \quad u \in C^1_{\mathrm{pw}}([a,b])\,, \tag{2.2.10}$$

which justifies calling the taut membrane a "two-dimensional string under tension".

Equilibrium principle

Displacement of taut membrane in equilibrium achieves minimal potential energy, *cf.* (1.2.51)

Equilibrium state $$\boxed{u_* = \operatorname*{argmin}_{u\in V} J_M(u)}\,. \tag{2.2.12}$$

Remark 2.2.13 (Minimal regularity of membrane displacement)

Least smoothness required for u, f to render $J_M(u)$ from (2.2.7) meaningful, *cf.* discussion in Section 1.3.2:

- $u \in C^1_{\mathrm{pw}}(\overline{\Omega})$ is sufficient for displacement u,
- $\sigma, f \in C^0_{\mathrm{pw}}(\overline{\Omega})$ already allows integration.

The question what is the largest possible function space on which J_M can still be defined will be examined again in Section 2.3.

2.2.2 Electrostatic fields

In this section we see another important example of a mathematical model

- whose configuration space is a space of functions on a spatial domain,
- and governed by a minimal energy principle as equilibrium condition.

Typical arrangement:

- ✦ metal body in metal box
- ✦ prescribed voltage drop body—box

Sought: electric field $\mathbf{E} : \Omega \mapsto \mathbb{R}^3$ in $\Omega \subset \mathbb{R}^3$
($\Omega \hat{=}$ blue region ▷)

▷

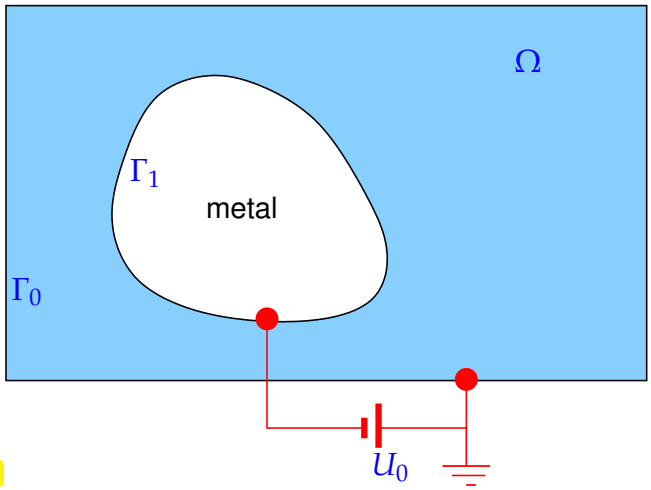


Fig. 65

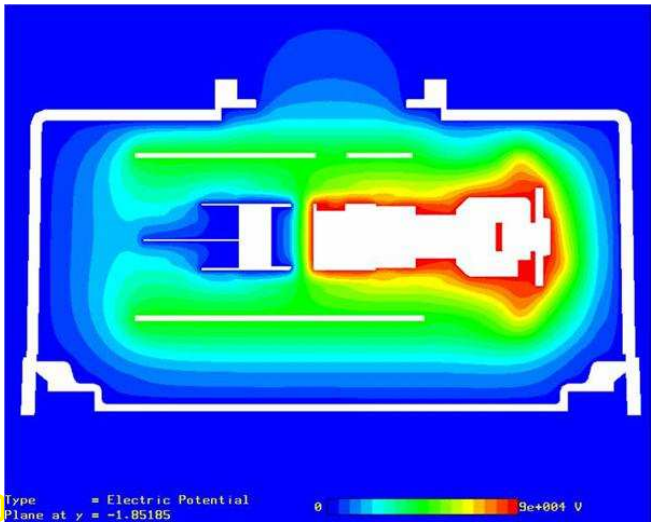


Fig. 66

From Maxwell's equations, static case:

$$\boxed{\mathbf{E} = -\,\mathbf{grad}\, u}\,, \qquad (2.2.14)$$

where $u : \Omega \mapsto \mathbb{R} \hat{=}$ electric (scalar) potential, $[u] = 1\mathrm{V}$

◁ Electric potential in technical device

Electromagnetic field energy: (electrostatic setting)

$$J_E(\mathbf{E}) = \frac{1}{2}\int_\Omega (\epsilon(\boldsymbol{x})\mathbf{E}(\boldsymbol{x}))\cdot\mathbf{E}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \frac{1}{2}\int_\Omega (\epsilon(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\mathbf{grad}\,u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \qquad (2.2.15)$$

where $\epsilon : \Omega \mapsto \mathbb{R}^{3,3} \hat{=}$ dielectric tensor, $\epsilon(\boldsymbol{x})$ symmetric, $[\epsilon] = \frac{\mathrm{As}}{\mathrm{Vm}}$.

(2.2.16) Dielectric tensor

- Symmetry of the dielectic tensor, a matrix valued function on the spatial domain, can always be assumed: if $\epsilon(\boldsymbol{x})$ was not symmetric, then replacing it with $\frac{1}{2}(\epsilon(\boldsymbol{x})^T + \epsilon(\boldsymbol{x}))$ will yield exactly the same field energy.
- In terms of partial derivatives and tensor components $\epsilon(\boldsymbol{x}) = (\epsilon_{ij})_{i,j=1}^3$ we have

$$(\epsilon(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\mathbf{grad}\,u(\boldsymbol{x}) = \sum_{i=1}^3\sum_{j=1}^3 \epsilon_{ij}(\boldsymbol{x})\,\frac{\partial u}{\partial x_i}(\boldsymbol{x})\frac{\partial u}{\partial x_j}(\boldsymbol{x})\,.$$

Fundamental property of dielectric tensor (for "normal" materials):

$$\exists 0 < \epsilon^- \le \epsilon^+ < \infty\colon\quad \epsilon^-\|\mathbf{z}\|^2 \le (\epsilon(\boldsymbol{x})\mathbf{z})\cdot\mathbf{z} \le \epsilon^+\|\mathbf{z}\|^2 \quad \forall \mathbf{z}\in\mathbb{R}^3, \forall \boldsymbol{x}\in\Omega\,. \qquad (2.2.17)$$

Terminology: (2.2.17) $:\Leftrightarrow$ ϵ is bounded and uniformly positive definite

Definition 2.2.18. Uniformly positive (definite) tensor field

An matrix-valued function $\mathbf{A} : \Omega \mapsto \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, is called uniformly positive definite, if

$$\exists \alpha^- > 0: \quad (\mathbf{A}(\boldsymbol{x})\mathbf{z}) \cdot \mathbf{z} \geq \alpha^- \|\mathbf{z}\|^2 \quad \forall\, \mathbf{z} \in \mathbb{R}^n \tag{2.2.19}$$

for *almost all* $\boldsymbol{x} \in \Omega$, that is, only with the exception of a set of volume zero.

If $\mathbf{A}(\boldsymbol{x})$ is symmetric, then we have the equivalence, *cf.* [5, Rem. 8.1.19],

$$(2.2.19) \quad \Leftrightarrow \quad \mathbf{A}(\boldsymbol{x}) \text{ s.p.d. } (\rightarrow [5, \text{Def. } 1.1.8]) \quad \text{and} \quad \lambda_{\min}(\mathbf{A}(\boldsymbol{x})) \geq \alpha^- \,,$$

where $\lambda_{\min}$ stands for the smallers eigenvalue of a matrix.

(2.2.20) Boundary conditions for electrostatic potential

In order to characterize the configuration space for electrostatic field problems completely, we have to identify proper boundary conditions for the scalar potential V. To do so, we have to appeal to physics.

Recall: in electrostatics surfaces of conducting bodies are equipotential surfaces

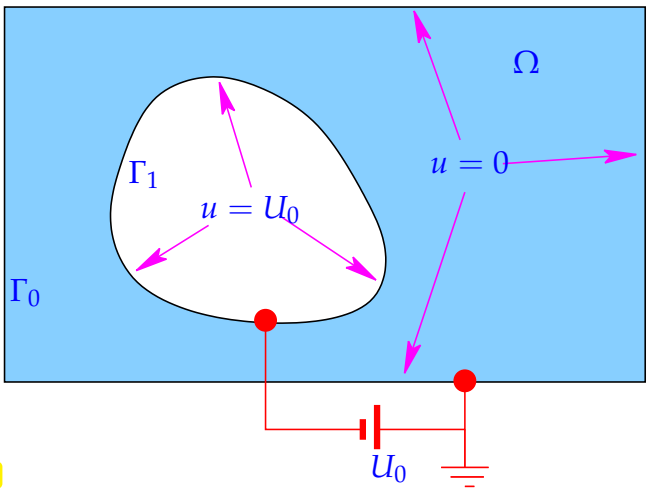


Fig. 67

In the situation of Fig. 65:

Boundary conditions

$$\begin{aligned} u &= 0 \quad \text{on } \Gamma_0 \,, \\ u &= U_0 \quad \text{on } \Gamma_1 \,. \end{aligned} \tag{2.2.21}$$

Configuration space

$$V = \left\{ u \in C^1_{\mathrm{pw}}(\Omega) \,,\, u \text{ satisfies (2.2.21)} \right\} .$$

to render $J_E(u)$ well defined, *cf.* Section 1.3.2.

Below, the notation $u = U$ will designate the boundary conditions (2.2.21).

Remark 2.2.22 (Electromagnetic field problems on $\mathbb{R}^3$)

Generically, Maxwell's equations are posed on the entire unbounded space $\mathbb{R}^3$.

▶ Electrostatic field problems are often posed on *unbounded* domains, for instance the entire space exterior to an electrode.

In this case, boundary conditions for the electric potential have to be replaced with decay conditions "at ∞": we demand

$$|u(\boldsymbol{x})| \leq C\|\boldsymbol{x}\|^{-1} \quad \text{uniformly for} \quad \|\boldsymbol{x}\| \to \infty \,. \tag{2.2.23}$$

Electromagnetic theory (Maxwell's equation) provides a criterion for selecting a unique electric scalar potential from the configuration space:

Equilibrium condition in electrostatic setting: minimal electromagnetic field energy

$$\boxed{u_* = \operatorname*{argmin}_{u\in V} J_E(u)} \quad . \tag{2.2.24}$$

2.2.3 Quadratic minimization problems

Structure of minimization problems (equilibrium problems) encountered above:

Section 2.2.1 [taut membrane]

$$u_* = \operatorname*{argmin}_{\substack{u\in C^1_{\mathrm{pw}}(\Omega)\\ u=g \text{ on } \partial\Omega}} \underbrace{\int_\Omega \tfrac{1}{2}\sigma(\boldsymbol{x})\|\mathbf{grad}\, u(\boldsymbol{x})\|^2 - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:J_M(u),\text{ see (2.2.7)}} \ , \tag{2.2.25}$$

Section 2.2.2 [electrostatics]

$$u_* = \operatorname*{argmin}_{\substack{u\in C^1_{\mathrm{pw}}(\Omega)\\ u=U \text{ on } \partial\Omega}} \underbrace{\int_\Omega \tfrac{1}{2}(\boldsymbol{\epsilon}(\boldsymbol{x})\,\mathbf{grad}\, u(\boldsymbol{x}))\cdot\mathbf{grad}\, u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:J_E(u),\text{ see (2.2.15)}} \ . \tag{2.2.26}$$

Evidently, (2.2.25) and (2.2.26) share a common structure. It is the *same* structure we have already come across in the minimization problem (1.4.2) for the taut string model in Section 1.4. There we identified it as a *quadratic minimization problem*, see Def. 1.4.3. In Section 1.4.1 we also determined the variational equation (1.4.6) arising from the quadratic minimization and saw that it belongs to the class of *linear variational problems*.

In this section we repeat and elaborate the considerations of Section 1.4.1 adopting a more abstract perspective than before.

Definition 2.2.27. Quadratic functional $\rightarrow$ **Def. 1.4.3**

A quadratic functional on a *real vector space* V_0 is a mapping $J : V_0 \mapsto \mathbb{R}$ of the form

$$J(u) := \tfrac{1}{2}\mathsf{a}(u,u) - \ell(u) + c \ , \quad u \in V_0 \ , \tag{2.2.28}$$

where $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ is a symmetric bilinear form ($\rightarrow$ Def. 1.3.22), $\ell : V_0 \mapsto \mathbb{R}$ a linear form, and $c \in \mathbb{R}$.

Recall: A bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ is symmetric, if

$$\mathsf{a}(u,v) = \mathsf{a}(v,u) \quad \forall u,v \in V_0 \ . \tag{2.2.29}$$

Remark 2.2.30 (Quadratic functionals on $\mathbb{R}^N$)

If $V_0 = \mathbb{R}^N$ (finite-dimensional case), then a quadratic functional has the general representation

$$J(\mathbf{u}) = \tfrac{1}{2}\mathbf{u}^T\mathbf{A}\mathbf{u} - \mathbf{b}^T\mathbf{u} + c \quad , \quad \mathbf{A} = \mathbf{A}^T \in \mathbb{R}^{N,N} \; , \quad \mathbf{b} \in \mathbb{R}^N \; , \quad c \in \mathbb{R} \; . \tag{2.2.31}$$

Reminder: quadratic functionals of this forms occur in derivation of steepest descent and conjugate gradient methods for linear systems of equations, see [5, Section 8.1.1].

Further discussion of quadratic functionals on $\mathbb{R}^n$ $\rightarrow$ [5, Section 8.1.1]

Definition 2.2.32. Quadratic minimization problem (II) $\rightarrow$ Def. 1.4.3

A minimization problem

$$w_* = \operatorname*{argmin}_{w \in V_0} J(w)$$

is called a quadratic minimization problem, if J is a *quadratic functional* on a real vector space V_0.

(2.2.33) Offset functions $\rightarrow$ Rem. 1.3.29, § 1.5.12

Objection! Both (2.2.25) and (2.2.26) are not genuine quadratic minimization problems, because they are posed over affine spaces (= "vector space + offset function", *cf.* (1.3.24))!

Proper quadratic minimization problems can be recovered by the "offset function trick", *c.f.* (1.3.30): for quadratic form J from (2.2.28)

$$\begin{aligned} J(u+u_0) &= \tfrac{1}{2}\mathsf{a}(u+u_0, u+u_0) - \ell(u+u_0) + c \\ &= \tfrac{1}{2}\mathsf{a}(u,u) + \underbrace{\mathsf{a}(u,u_0) - \ell(u)}_{=:\widetilde{\ell}(u)} + \underbrace{\tfrac{1}{2}\mathsf{a}(u_0,u_0) - \ell(u_0) + c}_{=:\widetilde{c}} =: \widetilde{J}(u) \; , \end{aligned}$$

due to the bilinearity of a and the linearity of ℓ.

$$\blacktriangleright \quad \operatorname*{argmin}_{u \in u_0 + V_0} J(u) = u_0 + \operatorname*{argmin}_{w \in V_0} J(w + u_0) = u_0 + \operatorname*{argmin}_{w \in V_0} \widetilde{J}(w) \; . \tag{2.2.34}$$

Thus, in the sequel we can focus on quadratic minimization problems posed on genuine vector spaces as in Def. 2.2.32, *cf.* also § 1.5.12.

Both (2.2.25) and (2.2.26) involve quadratic functionals. To see this apply the "offset function trick" from (2.2.34) in this concrete case: write $u = u_0 + w$ with an offset function u_0 that satisfies the boundary conditions and $w \in C^1_{\mathrm{pw},0}(\Omega)$, *cf.* (1.3.30).

(2.2.25) ➥ quadratic minimization problem ($\rightarrow$ Def. 2.2.32) with, *cf.* (2.2.28),

$$\mathsf{a}(w,v) = \int_\Omega \sigma(\boldsymbol{x}) \mathbf{grad}\, w(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \; , \quad \ell(v) := \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} - \mathsf{a}(u_0, v) \; . \tag{2.2.35}$$

(2.2.26) ➥ quadratic minimization problem ($\rightarrow$ Def. 2.2.32) with, *cf.* (2.2.28),

$$\mathsf{a}(w,v) = \int_\Omega \mathbf{grad}\, w(\boldsymbol{x})^T \boldsymbol{\epsilon}(\boldsymbol{x}) \mathbf{grad}\, v(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} \; , \quad \ell(v) := \mathsf{a}(u_0, v) \; . \tag{2.2.36}$$

In both cases: $V_0 = C^1_{\mathrm{pw},0}(\Omega)$
(temporarily before we are going to make a different choice in Section 2.3)

The next issue we have to tackle is the well-posedness of the concrete quadratic minimization problems posed on infinite-dimensional configuration spaces: Can we conclude existence and uniqueness of solutions of the minimization problems (2.2.25) and (2.2.26) **?** If not, this would cast a doubt on the mathematical model of physical reality.

The following property of the bilinear form a is *necessary* for the existence of a minimizer of a quadratic functional.

Definition 2.2.37. Positive semi-definite bilinear form

A (symmetric) bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ on a real vector space V_0 is positive semi-definite, if

$$\mathsf{a}(u,u) \geq 0 \quad \forall u \in V_0 \,. \tag{2.2.38}$$

Necessity of (2.2.38) for the existence of a minimizer can be concluded as follows: In case (2.2.38) fails to hold there is a $u \in V_0$ for which $\mathsf{a}(u,u) < 0$. Hence, for a quadratic functional $J : V_0 \to \mathbb{R}$ as in (2.2.28) we find

$$J(tu) = \tfrac{1}{2}t^2 \underbrace{\mathsf{a}(u,u)}_{<0} - t\ell(u) + c \,, \quad t \in \mathbb{R} \,,$$

such that $J(tu) \to -\infty$ for $t \to \infty$: J has no minimum. The next corollary summarizes this insight.

Corollary 2.2.39. Necessary condition for existence of minimizer

The quadratic functional $J(v) := \frac{1}{2}\mathsf{a}(v,v) - \ell(v)$ ($\to$ Def. 2.2.27) on a vector space V_0 is bounded from below, only if *the bilinear form $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$ is positive semi-definite.*

Next, let us tackle the issue of **uniqueness** of the minimizer of the quadratic functional J from (2.2.28). There is a *necessary and sufficient* condition in terms of a simple property of a, see also **??** below.

Definition 2.2.40. Positive definite bilinear form

A (symmetric) bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ on a real vector space V_0 is positive definite, if

$$u \in V_0 \setminus \{0\} \quad \Longleftrightarrow \quad \mathsf{a}(u,u) > 0 \,.$$

Remark 2.2.41 (Positive definite matrices)

For the special case $V_0 = \mathbb{R}^N$ any matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ induces a bilinear form via

$$\mathsf{a}(\mathbf{u},\mathbf{v}) := \mathbf{u}^T\mathbf{A}\mathbf{v} = (\mathbf{A}\mathbf{v}) \cdot \mathbf{u} \,, \quad \mathbf{u},\mathbf{v} \in \mathbb{R}^N \,. \tag{2.2.42}$$

This connects the concept of a symmetric positive definite bilinear form to the more familiar concept of s.p.d. matrices ($\to$ [5, Def. 1.1.8])

$$\mathbf{A} \text{ s.p.d.} \quad \Leftrightarrow \quad \mathsf{a} \text{ from (2.2.42) is symmetric, positive definite.}$$

Definition 2.2.43. Energy norm cf. [5, Def. 8.1.1]

A symmetric positive definite bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ ($\rightarrow$ Def. 2.2.40) induces the energy norm

$$\|u\|_{\mathsf{a}} := (\mathsf{a}(u,u))^{1/2} .$$

Origin of the term "energy norm" is clear from the connection with potential energy (e.g., in membrane model and in the case of electrostatic fields, see (2.2.35), (2.2.36)), see above.

Next, we have to verify the norm axioms (N1), (N2), and (N3) from Def. 1.6.4:

- (N1) is immediate from Def. 2.2.40,
- (N2) follows from bilinearity of a,
- (N3) is a consequence of the Cauchy-Schwarz inequality: for any symmetric positive definite bilinear form

$$|\mathsf{a}(u,v)| \leq (\mathsf{a}(u,u))^{1/2}(\mathsf{a}(v,v))^{1/2} . \tag{2.2.44}$$

Example 2.2.45 (Quadratic functionals with positive definite bilinear form in 2D)

Analogy between quadratic functionals with positive definite bilinear form and parabolas:

$$\begin{array}{ccccc} J(v) & = & \frac{1}{2}a(v,v) & - & \ell(v) \\ \updownarrow & & \updownarrow & & \updownarrow \\ f(x) & = & \frac{1}{2}ax^2 & - & bx \end{array}$$

with $a > 0$!

graph of quadratic functional $\mathbb{R}^2 \mapsto \mathbb{R}$ $\triangleright$

Geometric intuition suggests unique global and local minimum.

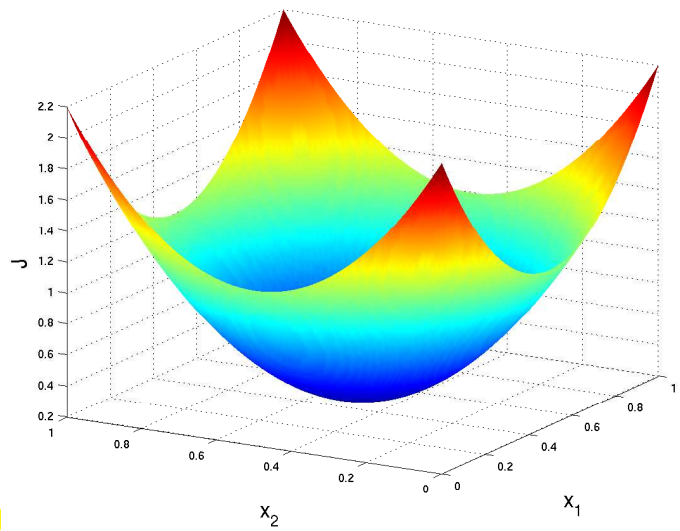


Fig. 68

Theorem 2.2.46. Uniqueness of solutions of quadratic minimization problems

If the bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ is positive definite ($\rightarrow$ Def. 2.2.40), then any solution of

$$u_* = \operatorname*{argmin}_{u \in V_0} J(u) \quad , \quad J(u) := \tfrac{1}{2}\mathsf{a}(u,u) - \ell(u) + c ,$$

is unique for any linear form $\ell : V_0 \mapsto \mathbb{R}$.

Proof. (indirect) Assume that both $u_* \in V_0$ and $w_* \in V_0$, $u_* \neq w_*$ are *global minimizers* of J on V_0.

❶ $\varphi(t) := J(tu_* + (1-t)w_*)$ has *two distinct* global minima in $t = 0, 1$.

❷ However $\varphi(t) = \frac{1}{2}t^2 \underbrace{\mathsf{a}(u_* - w_*, u_* - w_*)}_{>0} + t \ldots$ is a non-degenerate parabola opening towards $+\infty$, which clearly has a unique global minimum at its apex.

Contradiction between ❶ and ❷ $\Rightarrow$ assumption wrong. □

Under the assumptions of the theorem, the quadratic functional J is convex, *cf.* [5, Def. 3.3.5]:

Definition 2.2.47. Convexity of a real-valued function $\rightarrow$ **[7, Def. 5.5.2]**

A function $F : V_0 \to \mathbb{R}$ on a vector space V_0 is called convex, if

$$F(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda F(\mathbf{x}) + (1-\lambda)F(\mathbf{y}) \quad \forall \mathbf{x}, \mathbf{y} \in V_0 \,. \tag{2.2.48}$$

A function is concave, if its negative is convex.

It is well known that a twice continuously differentiable function $F : \mathbb{R} \to \mathbb{R}$ is convex if and only if it second derivative F'' is non-negative everywhere. Thus, the convexity of a quadratic functional based on a positive definite quadratic bilinear form is easily seen by considering the second derivative of the function

$$\varphi(t) := J(u + tv) \quad \Rightarrow \quad \ddot{\varphi}(t) = \mathsf{a}(v,v) > 0 \quad \text{, if } v \neq 0 \,.$$

(2.2.49) Positive definite bilinear form for electrostatic variational problem

? Is $\mathsf{a}(u,v) := \int_\Omega (\epsilon(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x})) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$ positive definite on $V_0 := C^1_{\mathrm{pw},0}(\overline{\Omega})$?

❶: Since ϵ bounded and uniformly positive definite ($\rightarrow$ Def. 2.2.18, (2.2.17))

$$\epsilon^- \int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \leq \mathsf{a}(u,u) \leq \epsilon^+ \int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \quad \forall u \,. \tag{2.2.50}$$

Hence, it is sufficient to examine the simpler bilinear form

$$\mathsf{d}(u,v) := \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \,, \quad u, v \in C^1_{\mathrm{pw},0}(\overline{\Omega}) \,. \tag{2.2.51}$$

❷: Obviously $\mathsf{d}(u,u) = 0 \;\Rightarrow\; \mathbf{grad}\, u = 0 \overset{(2.2.9)}{\Longrightarrow} u \equiv \mathrm{const}$ in Ω

Observe: $u = 0$ on $\partial\Omega \;\Rightarrow\; u = 0$

Zero boundary conditions are essential; otherwise one could add constants to the arguments of a without changing its value.

Next issue: **Existence** of solutions of quadratic minimization problems ($\rightarrow$ Def. 2.2.32) with positive definite bilinear form a.

In a finite dimensional setting this is not a moot point, see Fig. 68 for a "visual proof".

Theorem 2.2.52. Existence of unique minimizer in finite dimensions

Let $J(u) := \frac{1}{2}\mathsf{a}(u,u) - \ell(u) + c$ with symmetric positive definite (→ Def. 2.2.40) bilinear form $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$ (→ Def. 1.3.22), 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 (→ Def. 2.2.32)

$$u_* = \operatorname*{argmin}_{u \in V_0} J(u)$$

always possesses a unique *solution.*

However, infinite dimensional spaces hold a lot of surprises and existence of solutions of quadratic minimization problems becomes a subtle issue, even if the bilinear form is positive definite. Next, we state a necessary condition for the existence of a minimizer. Yet, an example will demonstrate that it may not be sufficient.

(2.2.53) A necessary condition for the existence of a minimizer of a quadratic functional

Consider a quadratic minimization problem (→ Def. 2.2.32) for a quadratic functional (→ Def. 2.2.27)

$$J : V_0 \mapsto \mathbb{R} \quad , \quad J(u) = \frac{1}{2}\mathsf{a}(u,u) - \ell(u) \ ,$$

based on a symmetric positive definite (s.p.d.) bilinear form a → Def. 2.2.40.

Lemma 2.2.54. Boundedness condition on linear form

The quadratic functional J is bounded from below on V_0, if and only if

$$\boxed{\exists C > 0: \quad |\ell(u)| \leq C\|u\|_{\mathsf{a}} \quad \forall u \in V_0} \ , \tag{2.2.55}$$

where $\|\cdot\|_{\mathsf{a}}$ is the energy norm induced by a, see Def. 2.2.43.

Assertion (2.2.55) is written in a way customary in numerical analysis and theory of partial differential equations. It should be read as "there is a constant $C > 0$ such that for all $u \in V_0$ the estimate $|\ell(u)| \leq C\|u\|_{\mathsf{a}}$ holds true.". In logic the quantors would be arranged differently:

$$(2.2.55) \quad \Leftrightarrow \quad \exists C > 0: \quad \{\forall u \in V_0: \quad |\ell(u)| \leq C\|u\|_{\mathsf{a}}\} \ .$$

Proof. (of Lemma 2.2.54)

❶ Condition (2.2.55) ensures that J is bounded from below:

$$J(u) = \frac{1}{2}\mathsf{a}(u,u) - \ell(u) \geq \frac{1}{2}\|u\|_{\mathsf{a}}^2 - C\|u\|_{\mathsf{a}} \geq -\frac{1}{2}C^2 \ .$$

❷ The proof of the other implication is *indirect* (proof by contradiction). Assume that (2.2.55) does not hold. Then, for every $n \in \mathbb{N}$, we can find $u_n \in V_0$ such that

$$\ell(u_n) \geq n\|u_n\|_{\mathsf{a}} \ .$$

By rescaling $u_n \leftarrow \frac{u_n}{\|u_n\|_a}$, we can assume without loss of generality that $\|u_n\|_a = 1$, which implies

$$J(u_n) \leq \tfrac{1}{2} - n \to -\infty \quad \text{for} \quad n \to \infty\,.$$

Hence J can attain values below any threshold. □

Parlance: In mathematical terms (2.2.55) means that ℓ is continuous w.r.t. the energy norm $\|\cdot\|_a$.

Definition 2.2.56. Continuity of a linear form and bilinear form

Consider a normed vector space V_0 with norm $\|\cdot\|$. A linear form $\ell : V_0 \to \mathbb{R}$ (→ Def. 1.3.22) 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 $a : V_0 \times V_0 \to \mathbb{R}$ (→ Def. 1.3.22) on V_0 is continuous, if

$$\exists K > 0: \quad |a(u,v)| \leq K\|u\|\|v\| \quad \forall u,v \in V_0\,.$$

Remark 2.2.57 (Necessary continuity of linear form)

Note that, due to the variational formulation, see Section 2.4 below, we have

$$|\ell(v)| = |a(u_*,v)| \leq \|u_*\|_a\|v\|_a = C\|v\|_a \quad \forall v \in V_0\,, \tag{2.2.58}$$

where u_* is solution of the minimization problem and $C := \|u_*\|_a$.

Whenever a finite energy solution u_* to a quadratic optimization problem exists, then l must be continuous with $C = \|u_*\|_a < \infty$!

Under the conditions that a is positive definite and ℓ is bounded the quadratic minimization problem for J should have a (unique, due to Thm. 2.2.46) solution, if it is considered on a space that is "large enough". However, on infinite-dimensional function spaces this remains a subtle issue, as is strikingly illustrated by the next example.

Example 2.2.59 (Non-existence of solutions of positive definite quadratic minimization problem)

We consider the quadratic functional

$$J(u) := \int_0^1 \tfrac{1}{2}u^2(x) - u(x)\,\mathrm{d}x = \tfrac{1}{2}\int_0^1 \{(u(x)-1)^2 - 1\}\,\mathrm{d}x\,,$$

on the space $\boxed{V_0 := C_0^0([0,1])}$

It fits the abstract form from Def. 2.2.27 with

$$a(u,v) = \int_0^1 u(x)v(x)\,\mathrm{d}x \quad , \quad \ell(v) = \int_0^1 v(x)\,\mathrm{d}x\,.$$

The function $\varphi(\xi) = \frac{1}{2}\xi^2 - \xi = \frac{1}{2}\xi(1-2\xi) = \frac{1}{2}(\xi-1)^2 - \frac{1}{2}$ has a global minimum at $\xi = 1$ and $\varphi(\xi) - \varphi(1) = \frac{1}{2}(\xi-1)^2$.

➤ $$|\eta - 1| > |\xi - 1| \quad \Rightarrow \quad \varphi(\eta) > \varphi(\xi).$$

Assume that $u \in V_0$ is a global minimizer of J. Then

$$w(x) := \min\{1, 2\max\{u(x), 0\}\}\ , \qquad 0 \le x \le 1\ ,$$

is another function $\in C_0^0([0,1])$, which satisfies

$$u(x) \neq 1 \quad \Rightarrow \quad |w(x) - 1| < |u(x) - 1| \quad \Rightarrow \quad J(w) < J(u) \quad !$$

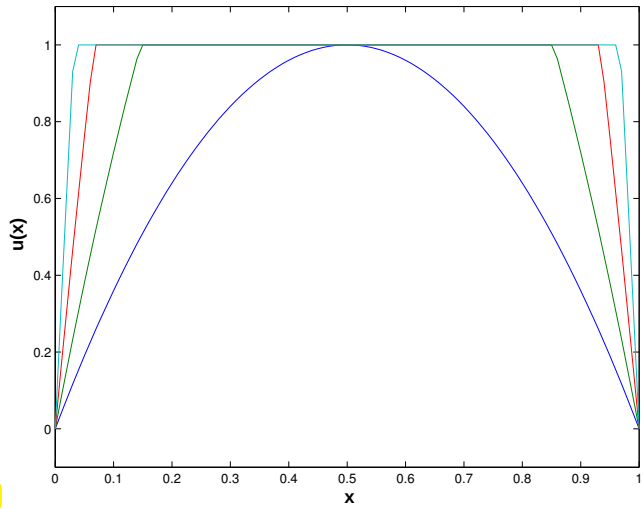


Fig. 69

Hence, whenever we think we have found a minimizer $\in C_0^0([0,1])$, the formula provides another eligible function for which the value of the functional is even smaller! Therefore we can find sequences $(u_n)_n$ of functions in $C_0^0([0,1])$ for which $J(u_n)$ tends to the minimum, whereas the sequence itself has no limit in $C_0^0([0,1])$.

The problem in this example seems to be that we have chosen "too small" a function space, *c.f.* Section 2.3 below.

?! *Review question(s) 2.2.60.* (Quadratic minimization problems)

1. What is a quadratic functional on a vector space V_0?
2. Why can we always assume the bilinear forms to be symmetric when considering the minimization of quadratic quadratic functionals?
3. Argue why a quadratic functional can have a *unique* minimizer, *if and only if* its bilinear form is positive definite ($\rightarrow$ Def. 2.2.40).
4. Show that the continuity condition $\exists C > 0$: $\ell v \le C\|v\|_{\mathsf{a}}\ \forall v \in V_0$ is necessary for the existence of a minimizer of the quadratic functional $J(v) := \frac{1}{2}\mathsf{a}(v,v) - \ell(v)$ on the vector space V_0.
5. State the Cauchy-Schwarz inequality for a symmetric positive definite bilinear form a on a real vector space V_0.
6. Which of the following mappings defined on $C^1(\overline{\Omega})$, $\Omega \subset \mathbb{R}^2$ are linear?
   (a) $v \mapsto v(\boldsymbol{x}) - v(\boldsymbol{y})$, $\boldsymbol{x}, \boldsymbol{y} \in \Omega$,
   (b) $v \mapsto \int_\Omega \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$
   (c) $v \mapsto \int_\Omega 1 + \boldsymbol{a} \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$, $\boldsymbol{a} \in \mathbb{R}^2$,
   (d) $v \mapsto \int_\Omega v(\boldsymbol{x})\, \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$.

2.3 Sobolev spaces

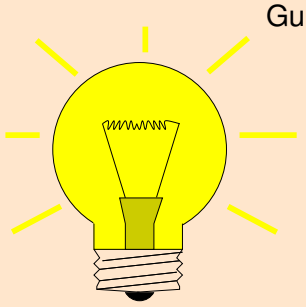
(2.3.1) Preview

Mathematical theory is much concerned about proving existence of suitably defined solutions for minimization problems. As demonstrated in Ex. 2.2.59 this can encounter profound problems.

In this section we will learn about a class of *abstract function spaces* that has been devised to deal with the question of existence of solutions of quadratic minimization problems like (2.2.25) and (2.2.26). We can only catch of glimpse of the considerations; thorough investigation is done in the mathematical field of *functional analysis*.

2.3.1 Function spaces for variational problems

Now we return to the question of how to choose the right function space for a linear variational problem. Bounded energy norm will be the linchpin of the argument:



Guideline: for a quadratic minimization problem ($\rightarrow$ Def. 2.2.32) with

- ✦ symmetric positive definite (s.p.d.) bilinear form a,
- ✦ a linear form ℓ that is continuous w.r.t. $\|\cdot\|_{\mathsf{a}}$, see (2.2.55),

posed over a function space follow the advice:

> consider it on the largest space of functions
> for which a still makes sense **!**

(and which complies with boundary conditions)

In the concrete case of quadratic minimization problems like (2.2.25) (minimization of potential energy of a membrane) and (2.2.26) (minimization of the energy of an electrostatic field) we arrive at the following recommendation:

Choose "$V_0 := \{$functions v on Ω: $\mathsf{a}(v,v) < \infty\}$"

➤ "Reasoning turned upside down": now we first look at the quadratic functional J or, equivalently, the bilinear form a, they determine the function space on which the minimization problem/variational problem is posed!

2.3.2 The function space $L^2(\Omega)$

Consider the quadratic functional (related to J from Ex. 2.2.59)

$$J(u) := \int_\Omega \tfrac{1}{2}|u(\boldsymbol{x})|^2 - u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,. \quad \left(u \in C^0_{\mathrm{pw}}(\Omega)\ ?\right) \tag{2.3.2}$$

In Section 1.3.2 we came to the conclusion that the space of piecewise continuous functions might provide the right setting for treating this functional. Now we follow the above recipe, which suggests that we choose

$$V_0 := \{v : \Omega \mapsto \mathbb{R} \text{ integrable: } \int_\Omega |v(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x} < \infty\} \tag{2.3.3}$$

Definition 2.3.4. Space $L^2(\Omega)$ $\rightarrow$ **Def. 1.6.7**

The function space defined in (2.3.3) is the space of square-integrable functions on Ω and denoted by $L^2(\Omega)$.

It is a normed space with norm $\left(\|v\|_0 := \right) \|v\|_{L^2(\Omega)} := \left(\int_\Omega |v(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x}\right)^{1/2}$.

✎ Notation: $L^2(\Omega)$ ← superscript "2", because square in the definition of norm $\|\cdot\|_0$

Note: obviously $C^0_{\mathrm{pw}}(\Omega) \subset L^2(\Omega)$.

Remark 2.3.5 (Mathematical notion of $L^2(\Omega)$)

Here, we do make an attempt to provide a rigorous mathematical definition of $L^2(\Omega)$. This is done the *measure theory* and involves quotient spaces; a rather accessible presentation is given in [6, Ch. 3].

Remark 2.3.6 (Boundary conditions and $L^2(\Omega)$)

Ex. 2.2.59 vs. Eq. (2.3.3): Something has been forgotten! (boundary conditions $u(0) = u(1) = 0$ in Ex. 2.2.59, but none in Eq. (2.3.3)!)

Consider $u \in C^0([0,1])$ and try to impose boundary values $u_0, u_1 \in \mathbb{R}$ by "altering" u:

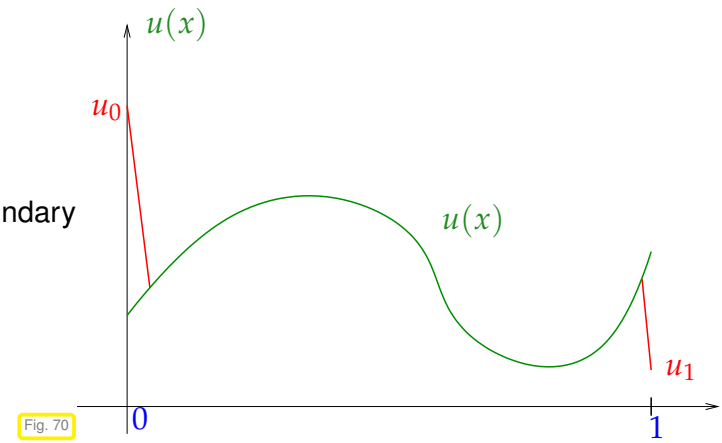


Fig. 70

$$\widetilde{u}(x) = \begin{cases} u(x) + (1 - nx)(u_0 - u(0)) & \text{, for } 0 \le x \le \frac{1}{n}\,, \\ u(x) & \text{, for } \frac{1}{n} < x < 1 - \frac{1}{n}\,, \\ u(x) - n(1 - \frac{1}{n} - x)(u_1 - u(1)) & \text{, for } 1 - \frac{1}{n} < x \le 1\,. \end{cases}$$

- $\widetilde{u}(0) = u_0$, $\widetilde{u}(1) = u_1$, $\|\widetilde{u} - u\|^2_{L^2(]0,1[)} = \frac{1}{3n}(u_0 + u_1 - u(0) - u(1)) \to 0$ for $n \to \infty$.
- Tiny perturbations of a function $u \in L^2(]0,1[)$ (in terms of changing its L^2-norm) can make it attain any value at $x = 0$ and $x = 1$.
- Mathematically this means that the space $V = \{u \in L^2(]0,1[) : u(0) = u(1) = 0\}$ is *not* closed in the energy space $L^2(]0,1[)$, meaning that there exist functions which are not in V but which can be arbitrarily well approximated by elements of V. Ex. 2.2.59 makes this concrete: the solution is approximated better and better but it is never reached because the trial space is too small.

Boundary conditions cannot be imposed in $L^2(\Omega)$!

2.3.3 Quadratic minimization problems on Hilbert spaces

In this section we let you catch a glimpse of the rigorous functional analytic treatment of minimization problems on infinite dimensional spaces. We review the mathematical arguments that confirm the existence of minimizers of quadratic minimization problems

$$u_* = \operatorname*{argmin}_{v \in V_0} J(v) \ , \quad J(v) = \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v) \ , \tag{2.3.7}$$

where

- V_0 is a real vector space, possibly of infinite dimension,
- $\mathsf{a} : V_0 \times V_0 \to \mathbb{R}$ is a symmetric positive definite bilinear form ($\to$ Def. 2.2.40) inducing an energy norm ($\to$ Def. 2.2.43) $\|\cdot\|_{\mathsf{a}}$ on V_0,
- $\ell : V_0 \to \mathbb{R}$ is a linear form, which is bounded with respect to the energy norm ($\to$ Def. 2.2.56).

(2.3.8) Completeness of normed vector spaces

The entire theory is based on a key matching condition for the space V_0 and its energy norm. To express this, we need a fundamental concept from analysis:

Definition 2.3.9. Cauchy sequence $\to$ [7, Def. 3.5.1]

Consider a normed vector space V_0 equipped with a norm $\|\cdot\|$ ($\to$ Def. 1.6.4). A sequence $(v_n)_{n\in\mathbb{N}}$ of vectors of V_0 is called a Cauchy sequence, if

$$\forall \epsilon > 0: \quad \exists n = n(\epsilon) \in \mathbb{N}: \quad \|v_k - v_m\| \le \epsilon \quad \forall k,m \ge n \ .$$

Clearly, every convergent sequence is a Cauchy sequence. The converse is true only in exceptional cases, which are of enormous importance in mathematical modelling, however, which has earned them a particular name.

Definition 2.3.10. Complete normed vector spaces and Hilbert spaces $\to$ [8, Def. I.1.2 & V.1.4]

A normed vector space is called complete, if every Cauchy sequence converges. A complete normed vector space is known as Banach space.
If the norm of a complete normed vector space V_0 is an energy norm ($\to$ Def. 2.2.43) associated with a symmetric positive definite bilinear form ($\to$ Def. 2.2.40), then V_0 is called a Hilbert space.

Example 2.3.11 (Important Banach spaces and Hilbert spaces)

- The real numbers $\mathbb{R}$ equipped with the modulus as norm $|\cdot|$ are complete.
- Every *finite dimensional* normed real or complex vector space is complete.
- For every bounded ($\Rightarrow$ compact) domain $\Omega \subset \mathbb{R}^d$ the space $C^0(\overline{\Omega})$, equipped with the supremum norm $\|\cdot\|_\infty$ ($\rightarrow$ Def. 1.6.5) is a Banach space [8, I.1 Bsp. (c)].
- For any domain $\Omega \subset \mathbb{R}^d$ the function space $L^2(\Omega)$ ($\rightarrow$ Def. 2.3.4) is a Hilbert space [8, I.1 Bsp. (h)].

The main existence theorem is given next.

Theorem 2.3.12. Existence of minimizers in Hilbert spaces

On a real Hilbert space V_0 with (energy) norm $\|\cdot\|_{\mathsf{a}}$ for any $\|\cdot\|_{\mathsf{a}}$-bounded linear functional $\ell : V_0 \to \mathbb{R}$ the quadratic minimization problem

$$u_* = \operatorname*{argmin}_{v\in V_0} J(v) \ , \quad J(v) := \tfrac{1}{2}\|v\|_{\mathsf{a}}^2 - \ell(v) \ , \tag{2.3.13}$$

has a unique solution.

Proof. Owing to the assumptions on ℓ, by Lemma 2.2.54 the quadratic functional J is bounded from below. Hence, there is a minimizing sequence $(v_n)_{n\in\mathbb{N}}$, which satisfies

$$|J(v_n) - \mu| \leq 1/n \quad \text{where} \quad \mu := \inf_{v\in V_0} J(v) \ . \tag{2.3.14}$$

Write $\mathsf{a}(\cdot,\cdot)$ for the bilinear form spawning $\|\cdot\|_{\mathsf{a}}$, that is $\|v\|_{\mathsf{a}}^2 = \mathsf{a}(v,v)$. From (bi-)linearity it is immediate that

$$\tfrac{1}{2}(J(v)+J(w)) - J(\tfrac{1}{2}(v+w)) = \tfrac{1}{4}\Big(\mathsf{a}(v,v)+\mathsf{a}(w,w) - 2\mathsf{a}(\tfrac{1}{2}(v+w),\tfrac{1}{2}(v+w))\Big) = \tfrac{1}{8}\|v-w\|_{\mathsf{a}}^2 \ .$$

This implies

$$\tfrac{1}{8}\|v_k - v_m\|_{\mathsf{a}}^2 \leq \tfrac{1}{2}(J(v_k)+J(v_m)) - \underbrace{J(\tfrac{1}{2}(v_k+v_m))}_{\geq\mu} \overset{(2.3.14)}{\leq} \tfrac{1}{2}(1/k+1/m) \leq \max\{1/k,1/m\} \ .$$

Hence, $(v_n)_{n\in\mathbb{N}}$ is a Cauchy sequence ($\rightarrow$ Def. 2.3.9) and

$$u_* := \lim_{n\to\infty} v_n \in V_0$$

exists and satisfies

$$J(u_{ast}) = \inf_{v\in V_0} J(v) \ .$$

In other words, the limit u_* is a global minimizer of J on V_0. Its uniqueness is established by the arguments of the proof of Thm. 2.2.46.

□

Remark 2.3.15 (Quadratic minimization problem in $L^2(\Omega)$)

Since $L^2(\Omega)$ is a Hilbert space, the previous theorem guarantees that the quadratic minimization problem for the quadratic functional from (2.3.2) on the function space $V_0 = L^2(\Omega)$, $\Omega :=]0,1[$, possesses a solution.

Conversely, though the minimization problem of Ex. 2.2.59 was considered on the Banach space $V_0 := C_0^0([0,1])$, the bilinear form in the quadratic functional failed to be related to the (supremum) norm, which rules out the application of Thm. 2.3.12.

(2.3.16) Completion of a normed vector space

The powerful Thm. 2.3.12 is available only in Hilbert spaces, which makes it very desirable to put quadratic minimization problems in a Hilbert space setting. Surprisingly, this can always be achieved by the procedure of completion.

Completion can be used to "fill the pores" of any normed vector spaces with potential limits of Cauchy sequences so that the resulting augmented space is complete in the sense of Def. 2.3.10.

Theorem 2.3.17. Completion of a normed vector space

For every normed vector space V_0 there is a unique (up to isomorphism) complete vector space $\widetilde{V}_0$ that contains V_0 as a dense subspace.

Definition 2.3.18. Dense subset

A subset $U \subset V_0$ is said to be dense in a normed vector space V_0, if every element of V_0 is the limit of a sequence in U.

Hence, when tackling the minimization of a quadratic functional (2.3.7) with positive definite bilinear form a on a vector space V_0, we can first switch to the completion $\widetilde{V}_0$ of V_0 with respect to the energy norm $\|\cdot\|_{\mathsf{a}}$ induced by a. Them, Thm. 2.3.12 will ensure the existence of a unique minimizer in $\widetilde{V}_0$; the existence issue is no longer moot.

What will we get when we apply the completion trick to the quadratic functional of Ex. 2.2.59, for which $V_0 = C_0^0([0,1])$ and $\mathsf{a}(u,v) = \int_0^1 u(x)v(x)\,\mathrm{d}x$? The next theorem gives an answer.

Theorem 2.3.19. $L^2(\Omega)$ by completion

For any domain $\Omega \subset \mathbb{R}^d$ the completion of $C_0^0(\overline{\Omega})$ equipped wit the norm $\|\cdot\|_{L^2(\Omega)}$ is the function space $L^2(\Omega)$.

As a consequence, when we resort to completion in Ex. 2.2.59, we end up in $L^2(]0,1[)$, inevitably loose the boundary conditions, *cf.* Rem. 2.3.6, but get the unique solution $u \equiv 1$.

Remark 2.3.20 (Boundary conditions and density)

In Rem. 2.3.6 we have seen that we cannot impose boundary conditions in $L^2(\Omega)$. This is evident from **??**: Since functions that vanish on the boundary $\partial\Omega$ are dense in $L^2(\Omega)$, any $v \in L^2(\Omega)$ can be approximated to arbitrary precision (in $L^2(\Omega)$-norm, of course) by a function, which is zero on $\partial\Omega$. This was, what $\widetilde{u}$ did in **??**.

2.3.4 The Sobolev space $H^1(\Omega)$

Now consider a quadratic minimization problem for the functional, *c.f.* (2.2.25),

$$J(u) := \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, u\|^2 - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \left(u \in C^1_{\mathrm{pw},0}(\Omega)\ ?\right) \tag{2.3.21}$$

What is the natural function space for this minimization problem? In Section 1.3.2 we would have opted for $C^1_{\mathrm{pw},0}(\Omega)$. Now, again, we follow the above recipe, which suggests that we choose

$$V_0 := \{v : \Omega \mapsto \mathbb{R} \text{ integrable: } v = 0 \text{ on } \partial\Omega,\ \int_\Omega |\,\mathbf{grad}\, v(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x} < \infty\} \tag{2.3.22}$$

Definition 2.3.23. Sobolev space $H^1_0(\Omega)$

The space of integrable functions on Ω with square integrable gradient that vanish on the boundary $\partial\Omega$,

$$V_0 := \{v : \Omega \mapsto \mathbb{R} \text{ integrable: } v = 0 \text{ on } \partial\Omega,\ \int_\Omega |\,\mathbf{grad}\, v(\boldsymbol{x})|^2\,\mathrm{d}\boldsymbol{x} < \infty\}\ , \tag{2.3.22}$$

is the Sobolev space $H^1_0(\Omega)$ with norm

$$|v|_{H^1(\Omega)} := \left(\int_\Omega \|\mathbf{grad}\, v\|^2\,\mathrm{d}\boldsymbol{x}\right)^{1/2}.$$

Notation: $H^1_0(\Omega)$ ← superscript "1", because first derivatives occur in norm
← subscript "0", because zero on $\partial\Omega$

Note: $|\cdot|_{H^1(\Omega)}$ is the energy norm (→ Def. 2.2.43) associated with the bilinear form in the quadratic functional J from (2.3.21), *cf.* (2.2.28).

☛ See § 1.6.8 for a discussion of the relevance of the energy norm.

Remark 2.3.24 (Boundary conditions in $H^1_0(\Omega)$)

Rem. 2.3.6 explained why imposing boundary conditions on functions in $L^2(\Omega)$ does not make sense.

Yet, in (2.3.22) zero boundary conditions are required for v **!**

Discussion parallel to Rem. 2.3.6, but now with the norm $|\cdot|_{H^1(\Omega)}$ in mind: Consider $u \in C^1([0,1])$ and try to impose boundary values $u_0, u_1 \in \mathbb{R}$ by "altering" u, see Fig. 70:

$$\widetilde{u}(x) = \begin{cases} u(x) + (1-nx)(u_0 - u(0)) & \text{, for } 0 \le x \le \frac{1}{n} \text{ ,} \\ u(x) & \text{, for } \frac{1}{n} < x < 1 - \frac{1}{n} \text{ ,} \\ u(x) - n(1-\frac{1}{n} - x)(u_1 - u(1)) & \text{, for } 1 - \frac{1}{n} < x \le 1 \text{ .} \end{cases}$$

► $\widetilde{u}(0) = u_0$, $\widetilde{u}(1) = u_1$, BUT $|\widetilde{u} - u|^2_{H^1(]0,1[)} = n(u_0 + u_1 - u(0) - u(1)) \to \infty$ for $n \to \infty$.

► Enforcing boundary values at $x = 0$ and $x = 1$ cannot be done without significantly changing the "energy" of the function.

However, the solutions of the quadratic minimization problems (2.2.25), (2.2.26) are to satisfy non-zero boundary conditions. The belong to an affine space $u_0 + V_0$ for a suitable offset function u_0, see § 2.2.33. This affine space will be contained in a larger Sobolev space, which arises from $H^1_0(\Omega)$ by dispensing with the requirement "$v = 0$ on $\partial\Omega$".

Definition 2.3.25. Sobolev space $H^1(\Omega)$

The Sobolev space

$$H^1(\Omega) := \{v : \Omega \mapsto \mathbb{R} \text{ integrable: } \int_\Omega |\,\mathbf{grad}\, v(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x} < \infty\}$$

is a normed function space with norm

$$\|v\|^2_{H^1(\Omega)} := \|v\|^2_0 + |v|^2_{H^1(\Omega)} \text{ .}$$

► $H^1(\Omega)$ is the "maximal function space" on which both J_M and J_E from (2.2.25), (2.2.26) are defined.

Remark 2.3.26 ($H^1(\Omega)$ through completion)

In § 2.3.16 we elaborated how one can build a complete function space as suitable setting for a quadratic minimization problem. In fact, the heuristic construction of $H^1_0(\Omega)$ and $H^1(\Omega)$ given above fits this technique.

Theorem 2.3.27. Sobolev spaces by completion

For domains as described in § 2.2.3 the function space $H^1(\Omega)$ can be obtained through completion ($\to$ Thm. 2.3.17) of $C^\infty(\overline{\Omega})$ equipped with the norm $\|\cdot\|_{H^1(\Omega)}$.
For any domain $\Omega \subset \mathbb{R}^d$, the space $H^1_0(\Omega)$ arises from the completion of $C^\infty_0)(\Omega)$ under the norm $\|\cdot\|_{H^1(\Omega)}$.

As a consequence, the spaces of smooth functions $C^\infty(\overline{\Omega})$ and $C^\infty_0(\Omega)$ are dense ($\to$ Def. 2.3.18) in $H^1(\Omega)$ and $H^1_0(\Omega)$, respectively.

Remark 2.3.28 ($|\cdot|_{H^1(\Omega)}$-seminorm)

Note that $|\cdot|_{H^1(\Omega)}$ alone is no longer a norm on $H^1(\Omega)$, because for $v \equiv \text{const}$ obviously $|v|_{H^1(\Omega)} = 0$, which violates (N1), *cf.* the discussion after Def. 1.6.14.

Lemma 2.2.54 tells us that a quadratic functional with s.p.d. bilinear form a is bounded from below, if its linear form ℓ satisfies the continuity (2.2.55). Now, we discuss this for the quadratic functional J from (2.3.21) in lieu of J_M and J_E.

$$J(u) := \int_\Omega \tfrac{1}{2}\|\mathbf{grad}\, u\|^2 - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad u \in H_0^1(\Omega)\,. \tag{2.3.21}$$

This quadratic functional J involves the linear form

$$\ell(u) := \int_\Omega f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,. \tag{2.3.29}$$

$f \mathrel{\hat{=}}$ load function ➢ $f \in C^0_{\mathrm{pw}}(\Omega)$ should be admitted.

Crucial question: Is ℓ as given in (2.3.29) continuous on $H_0^1(\Omega)$ **?**

$\Updownarrow$ (*c.f.* (2.2.55))

$$\exists C > 0\colon \quad |\ell(u)| \le C|u|_{H^1(\Omega)} \quad \forall u \in H_0^1(\Omega)\ ?\,.$$

(Again, recall Lemma 2.2.54 to appreciate the importance of this continuity: it is a necessary condition for the existence of a minimizer.)

To answer the question, we use the Cauchy-Schwarz inequality (2.2.44) for integrals in the form (1.6.13), which implies

$$|\ell(u)| = \left|\int_\Omega f(\boldsymbol{x})u(\boldsymbol{x})\mathrm{d}\boldsymbol{x}\right| \le \left(\int_\Omega |f(\boldsymbol{x})|^2\mathrm{d}\boldsymbol{x}\right)^{1/2}\left(\int_\Omega |u(\boldsymbol{x})|^2\mathrm{d}\boldsymbol{x}\right)^{1/2} = \underbrace{\|f\|_0}_{<\infty}\|u\|_0\,. \tag{2.3.30}$$

This reduces the problem to bounding $\|u\|_0$ in terms of $|u|_{H^1(\Omega)}$.

Theorem 2.3.31. First Poincaré-Friedrichs inequality

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

$$\|u\|_0 \le \operatorname{diam}(\Omega)\,\|\mathbf{grad}\, u\|_0 \quad \forall u \in H_0^1(\Omega)\,.$$

Proof. The proof employs a powerful technique in the theoretical treatment of function spaces: exploit density of smooth functions (which, by itself, is a deep result).

It boils down to the insight:

In order to establish inequalities between continuous functionals on Sobolev spaces of functions on Ω it often suffices to show the target inequality for smooth functions in $C_0^\infty(\Omega)$ or $C^\infty(\Omega)$, respectively.

✎ notation: $C_0^\infty(\Omega) \mathrel{\hat{=}}$ smooth functions with (compact) support ($\to$ Def. 1.5.76) *inside* Ω

In the concrete case (note the zero boundary values inherent in the definition of $H_0^1(\Omega)$) we have to establish the first Poincaré-Friedrichs inequality for functions $u \in C_0^\infty(\Omega)$ only.

For the sake of simplicity the proof is elaborated for $d = 1$, $\Omega = [0,1]$. It merely employs elementary results from calculus throughout, namely the Cauchy-Schwarz inequality (2.3.30) and the fundamental theorem of calculus [7, Satz 6.3.4], see (2.5.2):

$$\forall u \in C_0^\infty([0,1]): \qquad u(x) = \underbrace{u(0)}_{=0} + \int_0^x \frac{du}{dx}(\tau)\,\mathrm{d}\tau\,, \quad 0 \le x \le 1\,.$$

$$\blacktriangleright \quad \|u\|_0^2 = \int_0^1 \left| \int_0^x \frac{du}{dx}(\tau)\,\mathrm{d}\tau \right|^2 \mathrm{d}x \overset{(2.3.30)}{\le} \int_0^1 \left(\int_0^x 1\,\mathrm{d}\tau \cdot \int_0^x \left| \frac{du}{dx}(\tau) \right|^2 \mathrm{d}\tau \right) \mathrm{d}x \le \left\| \tfrac{du}{dx} \right\|_0^2 .$$

Taking the square root finishes the proof in 1D. □

The elementary proof in higher dimensions can be found in [4, Sect. 6.2.2] and in even greater generality in [3, Sect. 5.6.1].

Corollary 2.3.32. Admissible right hand side functionals for linear 2nd-order elliptic problems

If $f \in L^2(\Omega)$, then $\ell(u) = \int_\Omega f u\,\mathrm{d}\boldsymbol{x}$ is a continuous linear functional on $H_0^1(\Omega)$.

In this lemma "continuity" has to be read as

$$\boxed{\exists C > 0: \quad |\ell(u)| \le C|u|_{H^1(\Omega)} \quad \forall u \in H_0^1(\Omega)}\,. \tag{2.2.55}$$

How to "work with" Sobolev spaces

Most concrete results about Sobolev spaces boil down to relationships between their norms. The spaces themselves remain intangible, but the norms are very concrete and can be computed and manipulated as demonstrated above.

Do not be afraid of Sobolev spaces!

It is only the norms that matter for us, the 'spaces" are irrelevant!

Sobolev spaces = "concept of convenience": the minimization problem seeks its own function space.

Minimization problem		"Maximal" function space on which J is defined (Sobolev space)
$u = \operatorname*{argmin}_{v:\Omega\mapsto\mathbb{R}} J(v)$	$\blacktriangleright$	

"seek" $\leftrightarrow$ in more rigorous terms: completion with respect to energy norm, see § 2.3.16.

Remark 2.3.34 (Justification for teaching Sobolev spaces)

Then, why do you bother me with these uncanny "Sobolev spaces" after all **?**

- Anyone involved in CSE must be able to understand mathematical publications on numerical methods for PDEs, Those regularly resort to the concept of Sobolev spaces to express their findings

We will also need the following spaces (see [19]):

$$V_0 = H_{0,\Gamma_\tau}\left(\mathrm{curl}^0;\Omega\right) = \left\{\underline{v} \in V \,\middle|\, \mathrm{curl}\,\underline{v} = 0\right\} \tag{3}$$

$$H_{0,\Gamma_\nu}\left(\mathrm{div}^0,\Omega,\epsilon\right) = \left\{\underline{v} \in L^2\left(\Omega\right)^3 \,\middle|\, \mathrm{div}\,\epsilon\,\underline{v} = 0, \epsilon\,\underline{v}\cdot\underline{n}\big|_{\Gamma_\nu} = 0\right\} \tag{4}$$

$$H_1 = \epsilon^{-1}\mathrm{curl}\left(H_{0,\Gamma_\nu}\left(\mathrm{curl},\Omega\right)\right) \subset H_{0,\Gamma_\nu}\left(\mathrm{div}^0,\Omega,\epsilon\right) \tag{5}$$

$$V_1 = V \cap H_1 \tag{6}$$

$$\mathbb{H} = \mathbb{H}(\Omega,\Gamma_\tau,\epsilon) = H_{0,\Gamma_\tau}\left(\mathrm{curl}^0,\Omega\right) \cap H_{0,\Gamma_\nu}\left(\mathrm{div}^0,\Omega,\epsilon\right) \tag{7}$$

$$H^1_{0,\Gamma_\tau}\left(\Omega\right) = \left\{\phi \in L^2\left(\Omega\right) \,\middle|\, \mathrm{grad}\,\phi \in L^2\left(\Omega\right)^3,\ \phi\big|_{\Gamma_\tau} = 0\right\}. \tag{8}$$

We will indicate by $\|\ \|_{0,\Omega}$ the norm in H corresponding to $(\ ,\ )_{0,\Omega}$, by $(\ ,\ )_{\mathrm{curl},\Omega}$ and $\|\ \|_{\mathrm{curl},\Omega}$ the standard inner product and norm in $H(\mathrm{curl},\Omega)$, respectively, and by $\|\ \|_{s,\Omega}$, $0 < s \leq 1$, the natural norm in $H^s(\Omega)$ or $H^s(\Omega)^3$. For $s = 1$ we will also use the natural seminorm $|\ |_{1,\Omega}$ [16]. Finally, we define the following inner products and norms in H and V:

Fig. 71

- The statement that a function belongs to a certain Sobolev space can be regarded as a concise way of describing quite a few of its essential properties.

The next result elucidates the second point:

Theorem 2.3.35. 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.*

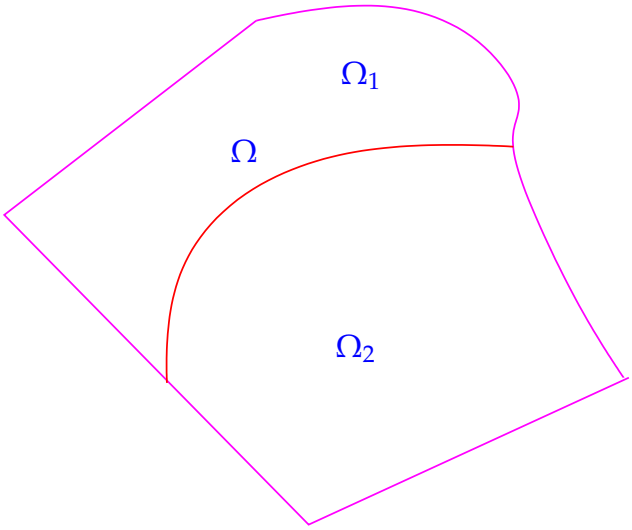
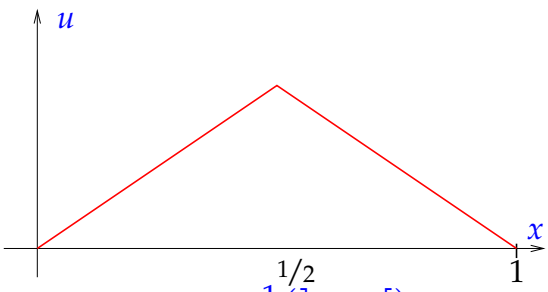


Fig. 72

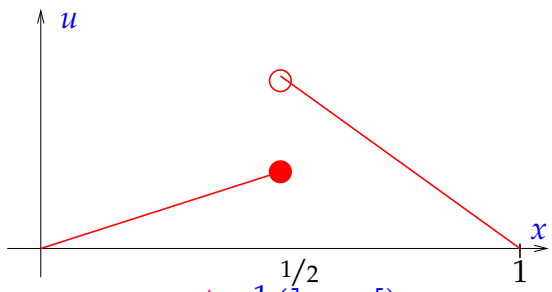
The proof of this theorem requires the notion of *weak derivatives* that will not be introduced in this course.

Example 2.3.36 (Piecewise linear functions (not) in $H^1_0(]0,1[)$)

We conclude from Thm. 2.3.35 applied in 1D:



$u \in H_0^1(]0,1[)$



$u \notin H_0^1(]0,1[)$

(2.3.37) Piecewise smooth functions contained in Sobolev space

From Thm. 2.3.35 we conclude that the function spaces we opted for in Section 1.3.2 were not far off:

- $C^1_{\mathrm{pw}}([a,b]) \subset H^1(]a,b[)$ and $C^1_{\mathrm{pw},0}([a,b]) \subset H^1_0(]a,b[)$,
- but $C^0_{\mathrm{pw}}([a,b]) \not\subset H^1(]a,b[)$.

On more general domains $\Omega \subset \mathbb{R}^d$ still holds true

- $\boxed{C^1_{\mathrm{pw}}(\overline{\Omega}) \subset H^1(\Omega) \text{ and } C^1_{\mathrm{pw},0}(\overline{\Omega}) \subset H^1_0(\Omega)}$,
- but $\boxed{C^0_{\mathrm{pw}}(\overline{\Omega}) \not\subset H^1(\Omega)}$.

Thm. 2.3.35 also provides a simple recipe for computing the norm $|u|_{H^1(\Omega)}$ of a piecewise C^1-function that is continuous in all of Ω.

Corollary 2.3.38. H^1-norm of piecewise smooth functions

Under the assumptions of Thm. 2.3.35 we have for a continuous, piecewise smooth function $u \in C^0(\Omega)$

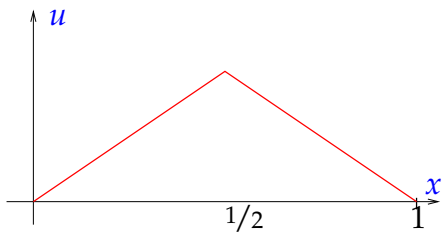
$$|u|^2_{H^1(\Omega)} = |u|^2_{H^1(\Omega_1)} + |u|^2_{H^1(\Omega_2)} = \int_{\Omega_1} |\,\mathbf{grad}\, u(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x} + \int_{\Omega_2} |\,\mathbf{grad}\, u(\boldsymbol{x})|^2 \,\mathrm{d}\boldsymbol{x}\,.$$

Actually, this is not new, see Section 1.3.2: earlier we already evaluated the elastic energy functionals (1.2.49), (1.4.2) for functions in $C^1_{\mathrm{pw}}([0,1])$ by "piecewise differentiation" followed by integration of the resulting discontinuous function.

Example 2.3.39 (Non-differentiable function in $H^1_0(]0,1[)$)

$d = 1, \Omega =]0,1[$:

"Tent function" $\quad u(x) = \begin{cases} 2x & \text{for } 0 < x < 1/2\,, \\ 2(1-x) & \text{for } 1/2 < x < 1\,. \end{cases}$



Compute
$$|u|^2_{H^1(\Omega)} = \int_0^1 |u'(x)|^2 \,\mathrm{d}x = 4 < \infty \,.$$

Example for a $u \in H^1_0(]0,1[)$, which is not globally differentiable.

Recall: we cheerfully computed the derivative of a piecewise smooth function already in Section 1.5.2.2 when differentiating the basis functions, *cf.* (1.5.72). Now this "reckless" computations have found their rigorous justification, because Thm. 2.3.35 tells us that for $u \in H^1(\Omega) \cap C^1_{\mathrm{pw}}(\overline{\Omega})$ we can compute its gradient $\mathbf{grad}\, u \in (L^2(\Omega))^d$ by means of *piecewise differentiation*, that is, $\mathbf{grad}\, u$ agrees with the classical gradient wherever u is differentiable.

☞ The (generalized) gradient is a continuous mapping
$$\mathbf{grad} : H^1(\Omega) \to L^2(\Omega) \,,$$
which can be computed by patching together "classical gradients" on a partition of Ω.

If you are still feeling uneasy when dealing with Sobolev spaces, do not hesitate to resort to the following substitutions in your thinking:
$$L^2(\Omega) \quad \to \quad C^0_{\mathrm{pw}}(\Omega) \quad , \quad H^1_0(\Omega) \quad \to \quad C^1_{\mathrm{pw},0}(\Omega) \,.$$

?! *Review question(s) 2.3.40.* (Sobolev spaces)

1. Which of the following functions belong to the spaces $L^2(]-1,1[)$ and $H^1(]0,1[)$, respectively?
   $$\bullet\ f(x) = |x| \quad \bullet\ f(x) = \log|x| \quad \bullet\ f(x) = \mathrm{sgn}(x) \quad \bullet\ f(x) = \sqrt{|x| + x}.$$
2. Show that the point evaluation $v \mapsto v(\frac{1}{2})$ is an unbounded lineare functional on $L^2(]0,1[)$.
3. Define the Sobolev space fitting the quadratic minimization problem for the functional
   $$J(\mathbf{v}) := \int_\Omega |\operatorname{div} \mathbf{v}(\boldsymbol{x})|^2 + \|\mathbf{v}\|^2 \,\mathrm{d}\boldsymbol{x} \,, \quad \mathbf{v} = (C^1(\overline{\Omega}))^2 \,.$$

2.4 Variational formulations

In this section we establish variational formulations for the minimization problems of Section 2.2, namely (2.2.12) and (2.2.24). Concepts and techniques from Section 1.3 will be discussed and used again. Thus, the reader is advised to repeat

- the main idea of the calculus of variations: (1.3.5) from Section 1.3.1 ("virtual work principle"),
- the computation of configurational derivatives of functionals defined on vector spaces of functions, see § 1.3.7,
- the notion of a linear variational problem, see Def. 1.4.8.

2.4.1 Linear variational problems

(2.4.1) Configurational derivative

Recall: derivation of variational formulation (1.4.6) from taut string minimization problem (1.4.2) in Section 1.4.

No surprise: (2.2.25) & (2.2.26) are amenable to the same approach:

Calculus of variations → Section 1.3.1: Compute "Directional/configurational derivative" of J_E:

$$\begin{aligned} J_E(u+tv) - J_E(u) &= \tfrac{1}{2}\int_\Omega (\epsilon(x)\,\mathbf{grad}(u+tv))\cdot\mathbf{grad}(u+tv)\,\mathrm{d}x \\ &\quad - \tfrac{1}{2}\int_\Omega (\epsilon(x)\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,u\,\mathrm{d}x \\ &\overset{(*)}{=} \tfrac{1}{2}\int_\Omega \begin{array}{l}(\epsilon(x)\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,u + 2t(\epsilon(x)\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v + \\ t^2(\epsilon(x)\,\mathbf{grad}\,v)\cdot\mathbf{grad}\,v - (\epsilon(x)\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,u\,\mathrm{d}x\end{array} \\ &= t\int_\Omega (\epsilon(x)\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v\,\mathrm{d}x + O(t^2) \quad \text{for } t\to 0\,. \end{aligned}$$

$(*)$: due to the symmetry of $\epsilon(x)$: $(\epsilon\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v = (\epsilon\,\mathbf{grad}\,v)\cdot\mathbf{grad}\,u$!

$$\blacktriangleright \quad \lim_{t\to 0}\frac{J_E(u+tv)-J_E(u)}{t} = \int_\Omega (\epsilon(x)\,\mathbf{grad}\,u(x))\cdot\mathbf{grad}\,v(x)\,\mathrm{d}x\,,$$

for perturbation/test functions $\boxed{v\in H_0^1(\Omega)}$, see Def. 2.3.23

The requirement $v = 0$ on $\partial\Omega$ reflects the fact that we may not perturb u on the boundary, lest the prescribed boundary values be violated ↔ whereas the configuration u may belong to an affine space, the test functions must always be chosen from a vector space, see **??** and the considerations on offset functions in Rem. 1.3.29 and § 2.2.33.

(2.4.2) Linear 2nd-order elliptic variational problem

As explained in Section 1.3.1 ("idea of calculus of variations"), setting the configurational derivative to zero leads to the following variational problem equivalent $(*)$ to (2.2.26)

$$\begin{array}{l} u\in H^1(\Omega)\,, \\ u = U \text{ on } \partial\Omega \end{array}: \quad \int_\Omega (\epsilon(x)\,\mathbf{grad}\,u(x))\cdot\mathbf{grad}\,v(x)\,\mathrm{d}x = 0 \quad \forall v\in H_0^1(\Omega)\,. \tag{2.4.3}$$

For the membrane problem (2.2.25) we arrive at

$$\begin{array}{l} u\in H^1(\Omega)\,, \\ u = g \text{ on } \partial\Omega \end{array}: \quad \int_\Omega \sigma(x)\,\mathbf{grad}\,u(x)\cdot\mathbf{grad}\,v(x)\,\mathrm{d}x = \int_\Omega f(x)v(x) \quad \forall v\in H_0^1(\Omega)\,. \tag{2.4.4}$$

($*$) equivalence of minimization problem and variational problem in the sense of equal sets of solutions holds, if existence and uniqueness of a minimizer is known.

Both, (2.4.3) and (2.4.4) have a common structure, expressed in the following variational problem:

Variational formulation of 2nd-order elliptic (Dirichlet ($*$)) minimization problems:

$$\begin{array}{l} u \in H^1(\Omega)\,, \\ u = g \text{ on } \partial\Omega \end{array}: \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,. \tag{2.4.5}$$

Symmetric uniformly positive definite material tensor $\boldsymbol{\alpha} : \Omega \mapsto \mathbb{R}^{d,d}$

($*$) The attribute "Dirichlet" refers to a setting, in which the function u is prescribed on the entire boundary. This is a particular type of boundary condition, which will be studied in detail in Section 2.7.

Some more explanations and terminology:

- $\Omega \subset \mathbb{R}^d$, $d = 2,3 \mathrel{\hat{=}}$ (spatial) domain, bounded, piecewise smooth boundary
- $g \in C^0(\partial\Omega) \mathrel{\hat{=}}$ boundary values (Dirichlet data)
- $f \in C^0_{\mathrm{pw}}(\Omega) \mathrel{\hat{=}}$ loading function, source function
- $\boldsymbol{\alpha} : \Omega \mapsto \mathbb{R}^{d,d} \mathrel{\hat{=}}$ material tensor, stiffness function, diffusion coefficient (uniformly positive definite, bounded $\to$ Def. 2.2.18):

$$\exists 0 < \alpha^- \leq \alpha^+ : \quad \alpha^-\|\boldsymbol{z}\|^2 \leq (\boldsymbol{\alpha}(\boldsymbol{x})\boldsymbol{z})\cdot\boldsymbol{z} \leq \alpha^+\|\boldsymbol{z}\|^2 \quad \forall \boldsymbol{z} \in \mathbb{R}^d\,, \tag{2.4.6}$$

for almost all $\boldsymbol{x} \in \Omega$.

Rewriting (2.4.5), using offset function u_0 with $u_0 = g$ on $\partial\Omega$, *cf.* (2.2.34),

$$w \in H_0^1(\Omega): \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,w(\boldsymbol{x}))\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$
$$= \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x}) - (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u_0(\boldsymbol{x}))\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,. \tag{2.4.7}$$

➥ (2.4.7) is a linear variational problem, see Def. 1.4.8

(2.4.8) Linear variational problem from quadratic minimization problem

We can lift the above discussion to an abstract level, *cf.* discussion in § 1.4.7 after Def. 1.4.8. Variational formulation of a quadratic minimization problem ($\to$ Def. 2.2.32)

$$J(u) := \tfrac{1}{2}\mathsf{a}(u,u) - \ell(u) + c \quad \Rightarrow \quad J(u+tv) = J(u) + t(\mathsf{a}(u,v) - \ell(v)) + \tfrac{1}{2}t^2\mathsf{a}(v,v)\,,$$

for all $u, v \in V_0$.

For a quadratic functional ($\to$ Def. 2.2.32) on the real vector space V_0 holds

$$\lim_{t\to 0}\frac{J(u+tv) - J(u)}{t} = \mathsf{a}(u,v) - \ell(v)\,. \tag{2.4.9}$$

Linear variational problem (→ § 1.4.7) arising from quadratic minimization problem for functional $J(u) := \frac{1}{2}\mathsf{a}(u,u) - \ell(u) + c$:

$$\boxed{w \in V_0: \quad \mathsf{a}(w,v) - \ell(v) = 0 \quad \forall v \in V_0} \; . \tag{2.4.10}$$

Concretely, for (2.4.7): $V_0 = H_0^1(\Omega)$ and

$$\mathsf{a}(w,v) = \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,w(\boldsymbol{x})) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\;, \tag{2.4.11}$$

$$\ell(v) = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x}) + (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u_0(\boldsymbol{x})) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\;. \tag{2.4.12}$$

2.4.2 Well-posedness of linear variational problems

Generic notion of a well-posed mathematical problem according to Jacques Hadamard (1902):

Definition 2.4.13. Well-posed mathematical problem

A mathematical problem of the form $F(x) = y$ based on a mapping $F : X_0 \subset X \mapsto Y_0 \subset Y$, where X and Y are normed vector spaces, and the solution space X_0 and data space Y_0 are suitable open subsets thereof, is called well-posed, if the following properties are satisfied.

❶ Existence: for all $y \in Y_0$ there exists a solution $x \in X_0$.

❷ Uniqueness: for all $y \in Y_0$ the solution is unique.

❸ Continuous dependence: for all $y \in Y_0$ there is a $\delta = \delta(y)$ and $L = L(y) > 0$ such that

$$\|\widetilde{x} - x\|_X \leq L(y)\|y - \widetilde{y}\|_Y \quad \forall \widetilde{y} \in Y_0 \colon \|\widetilde{y} - y\|_Y \leq \delta(y)\;,$$

where $\widetilde{x} \in X_0$ is the solution for data vector $\widetilde{y} \in Y_0$.

The requirement ❸ (continuous dependence) means that

small perturbations of the data cause only "small" ($*$) perturbations of the solution.

($*$): "small" in the sense that there is only a finite amplification of the perturbation, whatever practical significance this carries.

If ❸ is violated, arbitrarily small perturbations of the data, which are usually inevitable in numerical computations owing to round-off [5, **??**], can lead to big changes in the solution. In this case any attempt to solve the problem numerically with finite precision machine arithmetic is pointless.

Note: It might be possible to endow solution and data spaces with different norms. This choice may determine, whether the problem is well-posed or not, because ❸ will depend on it.

(2.4.14) Well-posedness of linear operator equations

Special case: linear problem ($X_0 = X$, $Y_0 = Y$, F linear mapping)

From Def. 2.4.13 we conclude that a *linear* problem is well-posed, if

1. the linear mapping $F : X \mapsto Y$ is bijective,
2. its inverse is bounded:

$$\begin{aligned} &\exists L > 0: \quad \left\| F^{-1}(y) - F^{-1}(\widetilde{y}) \right\|_X \leq L \|y - \widetilde{y}\|_Y \quad \forall y, \widetilde{y} \in Y \,, \\ &\qquad\qquad\qquad \Updownarrow \leftarrow \text{by linearity} \\ &\exists L > 0: \quad \left\| F^{-1}(y) \right\|_X \leq L \|y\|_Y \quad \forall y \in Y \,. \end{aligned} \tag{2.4.15}$$

(2.4.16) Choice of norms for linear 2nd-order elliptic BVPs

In this section we study well-posedness in the case of the *linear* variational (model) problem, *cf.* (2.4.5) and the related quadratic minimization problems (2.2.36), (2.2.35)

$$u \in H_0^1(\Omega): \quad \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,. \tag{2.4.17}$$

Of course, this is a linear problem. For solution and data spaces we make the following (natural) choices, also fixing the norms in the process

- solution space $X = H_0^1(\Omega)$ for u (norm given in Def. 2.3.23),
- data space $Y = L^2(\Omega)$ for loading function/source function f (norm given in Def. 2.3.4)

2.4.2.1 Existence and uniqueness of solutions

As discussed in Section 2.4.1, pp. 132, (2.4.17) is a linear variational problem of the form (2.4.10)

$$u \in V_0: \quad \mathsf{a}(u, v) = \ell(v) \quad \forall v \in V_0 \,, \tag{2.4.10}$$

posed on the Hilbert space $V_0 = H_0^1(\Omega)$, with a symmetric, positive definite ($\rightarrow$ Def. 2.2.40) bilinear form, *cf.* (2.4.11),

$$\mathsf{a}(u, v) := \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,,$$

and linear form, *cf.* (2.4.12) and (2.3.29)

$$\ell(v) := \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \,.$$

Next, recall the discussion in the beginning of Section 2.3 about the existence of solutions of quadratic minimization Thanks to the *equivalence* of quadratic minimization problems and linear variational problems these insights also apply to (2.4.10).

(2.4.18) Linear variational problems in Hilbert spaces

Thm. 2.3.12 ensures the existence of unique solutions of quadratic minimization problems of the form (2.3.13) on a Hilbert space V_0 (equipped with the nergy norm $\|\cdot\|_{\mathsf{a}}$)), provided that the linear functional

ℓ is continuous. By the arguments given in § 1.4.7 every minimizer $u \in V_0$ of J *must satisfy* (necessary condition!) the variational equation

$$\mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,. \tag{2.4.19}$$

Thus we get an existence result for (2.4.19) for free:

Corollary 2.4.20. Riesz representation theorem

For any bounded ($\to$ Def. 2.2.56) linear functional $\ell : V_0 \to \mathbb{R}$ on a real Hilbert space ($\to$ Def. 2.3.10) V_0 (with inner product $\mathsf{a}(\cdot,\cdot)$ and induced norm $\|\cdot\|_{\mathsf{a}}$) there exists a unique $u \in V_0$ such that

$$\mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{2.4.19}$$

$$\text{and} \qquad \|u\|_{\mathsf{a}} = \sup_{v \in V_0 \setminus \{0\}} \frac{|\ell(v)|}{\|v\|_{\mathsf{a}}} \,.$$

Since $H_0^1(\Omega)$ is a Hilbert space by Thm. 2.3.27, we can apply this result to (2.4.5)/(2.4.7). Here, recall that for (2.4.17) the energy norm is equivalent to the norm $|\cdot|_{H^1(\Omega)}$ on $H_0^1(\Omega)$, see Def. 2.3.23.

Theorem 2.4.21. Existence and uniqueness of solutions of s.p.d. linear variational problems

The linear variational problem (2.4.7) *has a unique solution in $H_0^1(\Omega)$ provided that $f \in L^2(\Omega)$ and $u_0 \in H^1(\Omega)$.*

The continuity of right hand side linear functionals $\ell(v)$ of the form $\ell(v) = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ has already been investigated in Section 2.3, pp. 142: the Cauchy-Schwarz estimate (2.3.30) together with the first Poincaré-Friedrichs inequality showed that

$$\boxed{\text{if } f \in L^2(\Omega), \text{ then } \ell \text{ is continuous on } H_0^1(\Omega).}$$

The next example will show that the use of source "functions" (rather, distributions) outside $L^2(\Omega)$ can really destroy existence and uniqueness of solutions:

Example 2.4.22 (Needle loading)

Now we inspect a striking manifestation of instability for a 2nd-order elliptic variational problem caused by a right hand side functional that fails to satisfy (2.2.55).

Consider the taut membrane model, see Section 2.2.1 for details, (2.2.25) for the related minimization problem, and (2.4.4) for the associated variational equation.

Let us assume that a needle is poked at the membrane: loading by a force f "concentrated in a point $\boldsymbol{y}$", often denoted by $f = \delta_{\boldsymbol{y}}$, $\boldsymbol{y} \in \Omega$, where δ is the so-called Dirac delta function (delta distribution).

In the variational formulation this can be taken into account as follows ($u_{|\partial\Omega} = 0$, $\sigma \equiv 1$ is assumed):

$$u \in H_0^1(\Omega): \quad \underbrace{\int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(u,v)} = \underbrace{v(\mathbf{y})}_{=:\ell(v)} \quad \forall v \in H_0^1(\Omega) \,. \tag{2.4.23}$$

Recall the discussion of Section 2.3: is the linear functional ℓ on the right hand side continuous w.r.t. the $H_0^1(\Omega)\cdot$-norm (= energy norm, see Def. 2.2.43) in the sense of (2.2.55)?

Consider the function $v(\boldsymbol{x}) = \log|\log\|\boldsymbol{x}\||$, $\boldsymbol{x} \neq 0$, on $\Omega = \{\boldsymbol{x} \in \mathbb{R}^2 \colon \|\boldsymbol{x}\| < \frac{1}{e}\}$.

(2.4.24) Polar coordinates

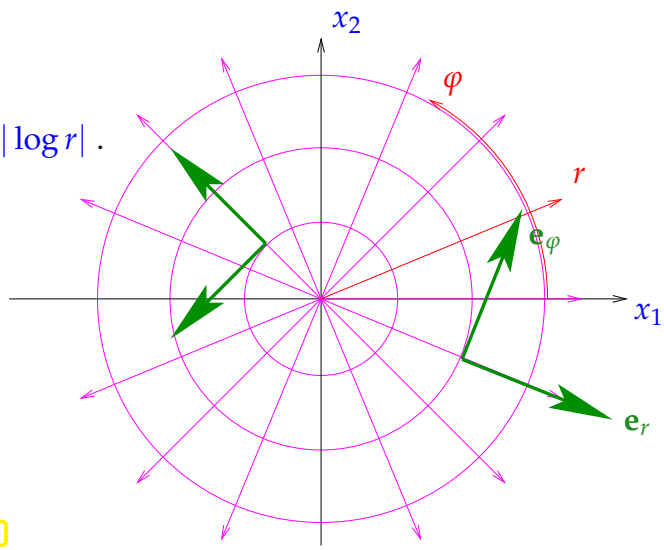
First, we express this function in polar coordinates (r, φ)

$$x_1 = r\cos\varphi \quad , \quad x_2 = r\sin\varphi \quad \blacktriangleright \quad v(r,\varphi) = \log|\log r| \; . \tag{2.4.25}$$

Then we recall the expression for the gradient in polar coordinates

$$\mathbf{grad}\, v(r,\varphi) = \frac{\partial v}{\partial r}(r,\varphi)\mathbf{e}_r + \frac{1}{r}\frac{\partial v}{\partial \varphi}(r,\varphi)\mathbf{e}_\varphi \; , \tag{2.4.26}$$

where $\mathbf{e}_r$ and $\mathbf{e}_\varphi$ are orthogonal unit vectors in the polar coordinate directions. Fig. 73



Also recall integration in polar coordinates, see [7, Bsp. 8.5.3]:

$$\int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_0^{1/e}\int_0^{2\pi} v(r,\varphi)\, r\,\mathrm{d}\varphi\mathrm{d}r \; . \tag{2.4.27}$$

Using polar coordinates and (2.4.27), we compute $|v|_{H^1(\Omega)}$ by evaluating an improper integral,

$$\int_\Omega \|\mathbf{grad}\, v(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x} = \int_0^{1/e}\int_0^{2\pi}\left\|-\frac{1}{\log r\, r}\,\mathbf{e}_r\right\|^2 r\,\mathrm{d}\varphi\mathrm{d}r = 2\pi\int_0^{1/e}\frac{1}{\log^2 r}\cdot\frac{1}{r}\,\mathrm{d}r$$
$$= 2\pi\left[-1/\log r\right]_0^{1/e} = \frac{2\pi}{\log e} = 2\pi < \infty \; .$$

This is allowed, because the improper integral has a finite value. This means that v has "finite elastic energy" , that is $v \in H^1(\Omega)$, see Def. 2.3.25.

On the other hand, $v(0) = \infty$!

$\blacktriangleright$ $H^1(\Omega)$ contains unbounded functions !

Corollary 2.4.28. Point evaluation on $H^1(\Omega)$

The point evaluation $v \mapsto v(\boldsymbol{y})$, $\mathbf{y} \in \Omega$ is not a continuous linear form on $H^1(\Omega)$.

In view of (2.2.58), this means that no solution of (2.4.23) with finite energy can exist. The energy must blow up which results in a bursting of the membrane.

This is the mathematics behind the observation that a needle can easily prick a taut membrane: a point load leads to configurations with "infinite elastic energy". Of course, this does not correspond to "real physics", but indicates that point loads are outside the scope of the simple linear continuum membrane model.

Fig. 74

Another implication of Cor. 2.4.28:

The quadratic functional $J(u) := \int_\Omega \|\mathbf{grad}\, u\|^2 \,\mathrm{d}\boldsymbol{x} - u(\boldsymbol{y}),\ \boldsymbol{y} \in \Omega$

is *not* bounded from below on $H_0^1(\Omega)$!

Thus, it is clear that the attempt to minimize J will run into difficulties. Yet, this is the quadratic functional underlying the variational problem (2.4.23).

2.4.2.2 Continuous dependence

Again we study continuous dependence of the solution u on the data f, see Def. 2.4.13 and the paragraphs after it for this notion, in the case of the model elliptic variational problem (2.4.17).

Recall the Cauchy-Schwarz estimate

$$|\ell(u)| = \left|\int_\Omega f(\boldsymbol{x})u(\boldsymbol{x})\mathrm{d}\boldsymbol{x}\right| \leq \left(\int_\Omega |f(\boldsymbol{x})|^2\mathrm{d}\boldsymbol{x}\right)^{1/2}\left(\int_\Omega |u(\boldsymbol{x})|^2\mathrm{d}\boldsymbol{x}\right)^{1/2} = \underbrace{\|f\|_{L^2(\Omega)}}_{<\infty}\|u\|_{L^2(\Omega)}\,, \tag{2.3.30}$$

which, when combined with (2.4.17) for $v = u$, immediately yields

$$\mathsf{a}(u,u) = |u|^2_{H^1(\Omega)} = |\ell(u)| \leq \|f\|_{L^2(\Omega)}\|u\|_{L^2(\Omega)}\,. \tag{2.4.29}$$

Next, we combine this with the first Poincaré-Friedrichs inequality of Thm. 2.3.31 and obtain

$$|u|_{H^1(\Omega)} \leq \mathrm{diam}(\Omega)\|f\|_{L^2(\Omega)}\,. \tag{2.4.30}$$

$\leftrightarrow$ (2.4.15) with $L = \mathrm{diam}(\Omega)$ for the particular problem (2.4.17).

(2.4.31) Sensitivity of elliptic BVP

Recall a notion introduced in [5, Section 1.6.1.2]:

Sensitivity of a problem (for given data) gauges
impact of small perturbations of the data on the result.

We first study the propagation of perturbations of the source function f (data) to the solution u for the *linear* variational problem (2.4.17) on a Hilbert space V_0. Denote by δf and δu the respective perturbations and adopt the abstract notation of (2.4.10).

$$\begin{aligned} \mathsf{a}(u,v) &= \ell(v) & \forall v \in V_0 \,, \\ \mathsf{a}(u+\delta u,v) &= (\ell+\delta\ell)(v) & \forall v \in V_0 \,. \end{aligned} \tag{2.4.32}$$

$$\overset{\text{a bilinear !}}{\Longrightarrow} \quad \mathsf{a}(\delta u,v) = (\delta\ell)(v) = \int_\Omega (\delta f)(\boldsymbol{x})v(\boldsymbol{x}) \quad \forall v \in V_0 \,. \tag{2.4.33}$$

The perturbation δu solves the same variational problem with source function δf!

Now we can directly apply the estimate (2.4.30) and get

$$|\delta u|_{H^1(\Omega)} \leq \operatorname{diam}(\Omega)\|\delta f\|_{L^2(\Omega)} \,. \tag{2.4.34}$$

There is an abstract principle behind these manipulations:

More general: propagation of perturbations from the data to the solution for an abstract *linear* problem, see p. 150. Here, as well, linearity enables sweeping simplifications:

As in (2.4.15) we write F^{-1} for the linear solution operator and observe

$$\begin{array}{llcll} \text{data} & y & \to & \text{solution} \quad x & : \quad x = F^{-1}y \,, \\ \text{perturbed data} & y+\delta y & \to & \text{perturbed solution} \quad x+\delta x & : \quad x+\delta x = F^{-1}(y+\delta y) \,. \end{array}$$

$$\overset{\text{Linearity !}}{\Longrightarrow} \quad \delta x = F^{-1}\delta y \quad \overset{(2.4.15)}{\Longrightarrow} \quad \|\delta x\|_X \leq L\|\delta y\|_Y \,.$$

(2.4.35) Sensitivity of energy

Again consider the perturbation of the right hand side of the linear variational problem (2.4.10) as in (2.4.32).

Now we study the impact of a perturbation of the right hand side functional on the energy $J(u) = \frac{1}{2}\mathsf{a}(u,u) - \ell(u)$ of the solution. By computations analogous to those in § 1.6.8 we find

$$\left\{ \begin{array}{rcll} \mathsf{a}(u,v) &=& \ell(v) & \forall v \in V_0 \,, \\ \mathsf{a}(u+\delta u,v) &=& (\ell+\delta\ell)(v) & \forall v \in V_0 \,, \end{array} \right. \Rightarrow \left\{ \begin{array}{rcl} J(u) &=& -\frac{1}{2}\mathsf{a}(u,u) \,, \\ J(u+\delta u) &=& -\frac{1}{2}\mathsf{a}(u+\delta u,u+\delta u) \,. \end{array} \right.$$

$$J(u+\delta u) - J(u) = \tfrac{1}{2}(\mathsf{a}(u,u) - \mathsf{a}(u+\delta u,u+\delta u)) \overset{(1.6.11)}{=} \tfrac{1}{2}\mathsf{a}(2u+\delta u,\delta u) \,.$$

$$\overset{(2.2.44)}{\Longrightarrow} \quad |J(u+\delta u) - J(u)| \leq \tfrac{1}{2}\|2u+\delta u\|_{\mathsf{a}}\|\delta u\|_{\mathsf{a}} \leq (2\|u\|_{\mathsf{a}} + \|\delta u\|)\cdot\|\delta u\|_{\mathsf{a}} \,.$$

The concrete meaning for the elliptic model problem (2.4.17) is

$$\begin{aligned} |J(u+\delta u) - J(u)| &\leq \left(2|u|_{H^1(\Omega)} + |\delta u|_{H^1(\Omega)}\right)|\delta u|_{H^1(\Omega)} \\ &\overset{(2.4.34)}{\leq} \operatorname{diam}(\Omega)^2\left(2\|f\|_{L^2(\Omega)} + \|\delta f\|_{L^2(\Omega)}\right)\|\delta f\|_{L^2(\Omega)} \,. \end{aligned} \tag{2.4.36}$$

The bottom line is that *small* perturbations of the source function/load f causes only small perturbations of the energy, with $2\operatorname{diam}(\Omega)^2\|f\|_{L^2(\Omega)}$ providing the amplification factor.

?! *Review question(s) 2.4.37.* (Linear variational problems)

1. State the linear variational problems arising as necessary conditions for the minimizers of the following functionals on $H^1(\Omega)$ for av bounded domain $\Omega \subset \mathbb{R}^2$
   (a) $J(v) := \int_\Omega \left|\frac{\partial u}{\partial x_1}(\boldsymbol{x})\right|^2 + \gamma\left|\frac{\partial u}{\partial x_1}(\boldsymbol{x})\right|^2 \mathrm{d}\boldsymbol{x} - \int_\Omega f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, $\gamma > 0$, $f \in L^2(\Omega)$,
   (b) $J(v) := \int_\Omega \|gradu(\boldsymbol{x}) - \boldsymbol{a}\|^2\,\mathrm{d}\boldsymbol{x}$, $\boldsymbol{a} \in \mathbb{R}^2$,
   (c) $J(v) := \int_\Omega |\mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{a} - 1|^2\,\mathrm{d}\boldsymbol{x}$, $\boldsymbol{a} \in \mathbb{R}^2$.
2. For what values of $\alpha \in \mathbb{R}$ will $x \mapsto x^\alpha$ belong to the spaces $L^1(]0,1[)$?
3. For what values of $\alpha \in \mathbb{R}$ will $v \mapsto \int_0^1 x^\alpha v(x)\,\mathrm{d}x$ be a continuous linear functional on $H_0^1(]0,1[)$?

2.5 Equilibrium Models: Boundary Value Problems

Recall the derivation of an ODE from a variational problem on a 1D domain (interval) in Section 1.3.3:

Tool: Integration by parts (1.3.40)

This section elucidates how to extend this approach to domains $\Omega \subset \mathbb{R}^d$, $d \geq 1$ (usually $d = 2,3$).

Crucial issue: Integration by parts in higher dimensions **?**

(2.5.1) Integration by parts in 1D $\rightarrow$ (1.3.40)

Remember the origin of integration by parts: fundamental theorem of calculus [7, Satz 6.3.4]: for $F \in C^1_{\mathrm{pw}}([a,b])$, $a,b \in \mathbb{R}$,

$$\int_a^b F'(x)\,\mathrm{d}x = F(b) - F(a)\ , \tag{2.5.2}$$

where $'$ stands for differentiation w.r.t x. This formula is combined with the product rule [7, Satz 5.2.1 (ii)]

$$F(x) = f(x)\cdot g(x) \quad \Rightarrow \quad F'(x) = f'(x)g(x) + f(x)g'(x)\ . \tag{2.5.3}$$

$$\blacktriangleright \quad \int_a^b f'(x)g(x) + f(x)g'(x)\,\mathrm{d}x = f(b)g(b) - f(a)g(a)\ ,$$

which amounts to (1.3.40).

2.5.1 Integration by parts in higher dimensions

There is a product rule in higher dimensions, see [7, Sect. 7.2]

Lemma 2.5.4. General product rule

For all $\mathbf{j} \in (C^1(\overline{\Omega}))^d$, $v \in C^1(\overline{\Omega})$ *holds*

$$\operatorname{div}(\mathbf{j}v) = v \operatorname{div} \mathbf{j} + \mathbf{j} \cdot \mathbf{grad}\, v \,. \tag{2.5.5}$$

Supplement 2.5.6 (**Divergence** operator, see also § 0.10.11).

From § 0.10.11 recall the definition of another important first-order *differential operator*, see also [7, Def. 8.8.1]:

The divergence of a C^1-vector field $\mathbf{j} = (f_1, \dots, f_d)^T : \Omega \mapsto \mathbb{R}^d$ is

$$\operatorname{div} \mathbf{j}(\boldsymbol{x}) := \frac{\partial f_1}{\partial x_1}(\boldsymbol{x}) + \cdots + \frac{\partial f_d}{\partial x_d}(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,.$$

A widely used "∇-notation" for the divergence is: $\nabla \cdot \mathbf{j}(\boldsymbol{x}) := \operatorname{div} \mathbf{j}(\boldsymbol{x})$.

The importance of the divergence for the mathematical modelling of flow fields will be explained in Section 7.1.3.

△

A truly fundamental result from differential geometry provides a multidimensional analogue of the fundamental theorem of calculus:

Theorem 2.5.7. Gauss' theorem $\rightarrow$ **[7, Sect. 8.8]**

With $\boldsymbol{n} : \partial\Omega \mapsto \mathbb{R}^d$ *denoting the exterior unit normal vectorfield on* $\partial\Omega$ *and* $\mathrm{d}S$ *indicating integration over a surface, we have*

$$\int_\Omega \operatorname{div} \mathbf{j}(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} \mathbf{j}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \quad \forall \mathbf{j} \in (C^1_{\mathrm{pw}}(\overline{\Omega}))^d \,. \tag{2.5.8}$$

Note: In (2.5.8) integration again allows to relax smoothness requirements, *cf.* Section 1.3.2.

Theorem 2.5.9. 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*

$$\int_\Omega \mathbf{j} \cdot \mathbf{grad}\, v\, \mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div} \mathbf{j}\, v\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j} \cdot \mathbf{n}\, v\, \mathrm{d}S \,. \tag{2.5.10}$$

Note that the dependence on the integration variable $\boldsymbol{x}$ is suppressed in the formula (2.5.10) to achieve a more compact notation. The first Green formula could also have been written as

$$\int_\Omega \mathbf{j}(\boldsymbol{x}) \cdot (\mathbf{grad}\, v)(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = -\int_\Omega (\operatorname{div} \mathbf{j})(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \mathbf{j}(\boldsymbol{x}) \cdot \mathbf{n}(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}S(\boldsymbol{x}) \,. \tag{2.5.10}$$

Proof. (of Thm. 2.5.9) Straightforward from Lemma 2.5.4 and Thm. 2.5.7. □

2.5.2 Scalar Second-order elliptic partial differential equations

Now we apply Green's first formula to the variational problem (2.4.5), which covers the membrane model and electrostatics:

The role of $\mathbf{j}$ in (2.5.10) is played by the *vector field* $\boldsymbol{\alpha}\,\mathbf{grad}\,u : \Omega \mapsto \mathbb{R}^d$.

$$\int_\Omega \underbrace{\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})}_{=:\mathbf{j}(\boldsymbol{x})} \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$$
$$= -\int_\Omega \operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) \cdot \boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})\,.$$

(2.4.5) ⟹

$$-\int_\Omega \operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \underbrace{\int_{\partial\Omega} (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) \cdot \boldsymbol{n}(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x})}_{=0\text{ , since } v_{|\partial\Omega}=0} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\,, \tag{2.5.11}$$

where we have to *assume* that u, $\boldsymbol{\alpha}$ are sufficiently smooth: $\boxed{\boldsymbol{\alpha}\,\mathbf{grad}\,u \in C^1_{\mathrm{pw}}(\overline{\Omega})}$

$$\int_\Omega (\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) + f(\boldsymbol{x}))\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^1_{\mathrm{pw},0}(\overline{\Omega})\,.$$

Now we can invoke the multidimensional analogue of the fundamental lemma of the calculus of variations, see Lemma 1.3.41

Lemma 2.5.12. Fundamental lemma of calculus of variations in higher dimensions

If $f \in L^2(\Omega)$ satisfies

$$\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in C^\infty_0(\Omega)\,,$$

then $f \equiv 0$ can be concluded.

(2.4.5) $\xRightarrow{\boldsymbol{\alpha}\,\mathbf{grad}\,u \in C^1_{\mathrm{pw}}(\Omega)}$ Partial differential equations (PDE)

$$-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega\,. \tag{2.5.13}$$

Again, for the sake of brevity, dependence $\mathbf{grad}\, u = \mathbf{grad}\, u(\boldsymbol{x})$, $f = f(\boldsymbol{x})$ is not made explicit in the PDE in (2.5.13).

Remark 2.5.14 (Laplace operator)

If $\boldsymbol{\alpha}$ agrees with a positive *constant*, by rescaling of (2.6.10) we can achieve

$$-\Delta u = f \quad \text{in } \Omega\,. \tag{2.5.15}$$

$$\Delta = \operatorname{div} \circ \mathbf{grad} = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2} = \text{Laplace operator}$$

(2.5.15) is called Poisson equation, $\Delta u = 0$ in Ω is called Laplace equation

---

Finally: PDE (2.5.13) + boundary conditions

$$-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\, u) = f \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega\,. \tag{2.5.16}$$

(2.5.16) = second-order elliptic BVP with Dirichlet boundary conditions Short name for BVPs of the type (2.5.16): "Dirichlet problem"

Remark 2.5.17 (Extra smoothness requirement for PDE formulation)

Same situation as in Section 1.3.3, *cf.* Assumption (1.3.39):

Transition from variational equation to PDE requires
extra assumptions on smoothness of solution and coefficients.

For instance, in the case of (2.5.13) we demand $\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\, u) \in C^0(\overline{\Omega})$, which is an implicit smoothness requirement for u, provided that the smoothness of the coefficient σ is known.

Terminology:

Terminology: A function $u \in C^1(\overline{\Omega})$, for which the partial differential equation (2.5.13) holds pointwise in $\overline{\Omega}$ and all derivatives exist in the sense of classical analysis, is called a classical solution, *cf.* § 1.3.46.

Example 2.5.18 (Taut membrane with free boundary values)

(Graph description of membrane shape by $u : \Omega \mapsto \mathbb{R}$, see Section 2.2.1)

Now: taut membrane clamped only on a part $\Gamma_0 \subset \partial\Omega$ of its edge.

- - - : prescribed boundary values here (Γ_0)

——: "free boundary"

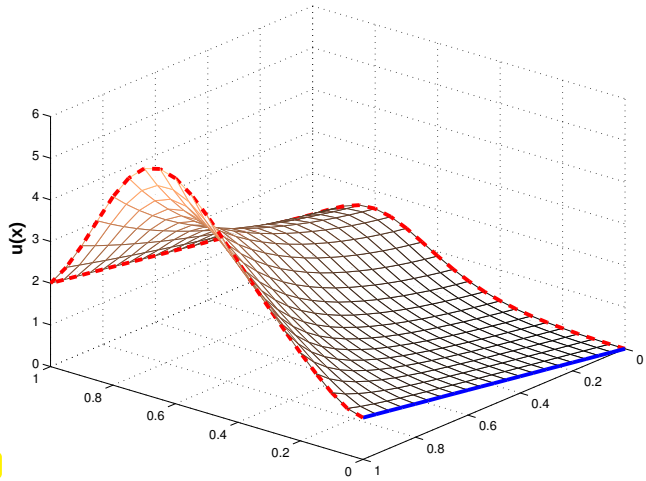


Fig. 75

▶ Configuration space $V := \{u \in H^1(\Omega) \colon u_{|\Gamma_0} = g\}$ → Def. 2.3.25

The expression for the total potential energy remains the same as in (2.2.7):

$$\boxed{J_M(u) := \int_\Omega \tfrac{1}{2}\sigma(\boldsymbol{x})\|\mathbf{grad}\, u\|^2 - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}} \quad . \tag{2.2.7}$$

Next we we derive the variational formulation for the quadratic minimization problem for J_M. The only change compared to Section 2.4.1 concerns a modified test and trial space. To understand the choice of the test space V_0 remember that it may contain only "admissible perturbations" of configurations, *cf.* Section 1.3.1. Concretely, adding any variation from the test space $V_)$ to a configuration $\in V$ must yield another valid configuration $\in V$. Therefore test functions have to vanish on Γ_0!

▶ test space in variational formulation $\boxed{V_0 := \{u \in H^1(\Omega) \colon u_{|\Gamma_0} = 0\}}$

▶ Variational formulation, *c.f.* (2.4.4)

$$\begin{array}{l} u \in H^1(\Omega) \\ u = g \text{ on } \Gamma_0 \end{array}\text{:} \quad \int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x}) \quad \forall v \in V_0 \,. \tag{2.5.19}$$

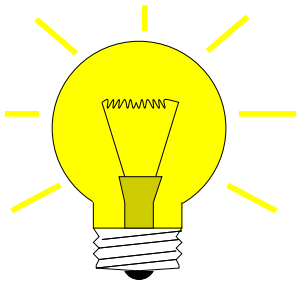
Our goal is to extract a second-order boundary value problem from this variational formulation. To begin with, an application of Green's first formula (2.5.10) to (2.5.19) leads to

$$-\int_\Omega (\operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\, u(\boldsymbol{x})) + f(\boldsymbol{x}))\, v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega\setminus\Gamma_0} ((\sigma(\boldsymbol{x})\,\mathbf{grad}\, u(\boldsymbol{x})) \cdot \boldsymbol{n}(\boldsymbol{x}))\, v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in V_0 \,. \tag{2.5.20}$$

Note that, unlike in (2.5.11), the boundary integral term cannot be dropped entirely, because the test function v need not vanish on all of $\partial\Omega$: $v \neq 0$ on $\partial\Omega \setminus \Gamma_0$ is possible!

In the sequel we assume (→ Rem. 2.5.17) extra smoothness $u \in C^2_{\mathrm{pw}}(\Omega)$, $\sigma \in C^1_{\mathrm{pw}}(\Omega)$

How to deal with the boundary term in (2.5.20) **?**



(Note that test functions in Lemma 2.5.12 vanish on $\partial\Omega$.)

Idea: ❶ First restrict test function v to $C_0^\infty(\Omega) \subset V$

➢ Boundary term vanishes !

Then, apply Lemma 2.5.12.

PDE: $$\operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) + f(\boldsymbol{x}) = 0 \quad \text{in } \Omega\,. \tag{2.5.21}$$

❷ Then test (2.5.20) with generic $v \in V_0$ and *make use of* (2.5.21). More precisely, plugging (2.5.21) into (2.5.20) makes the domain integral $\int_\Omega \dots$ disappear and only boundary terms remain.

$$\int\limits_{\partial\Omega\setminus\Gamma_0} ((\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\boldsymbol{n}(\boldsymbol{x}))\,v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = 0 \quad \forall v \in V_0\,.$$

$$\xRightarrow{\text{Lemma 2.5.12 on } \partial\Omega\setminus\Gamma_0} \quad (\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \text{on } \partial\Omega\setminus\Gamma_0\,. \tag{2.5.22}$$

When removing pinning conditions on $\partial\Omega\setminus\Gamma_0$ the equilibrium conditions imply the (homogeneous) Neumann boundary conditions $(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\boldsymbol{n}(\boldsymbol{x}) = 0$ on $\partial\Omega\setminus\Gamma_0$.

Boundary value problem for membrane clamped at $\Gamma_0 \subset \partial\Omega$

$$-\operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega\,, \qquad \begin{aligned} u &= g \quad \text{on } \Gamma_0\,,\\ (\sigma(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\boldsymbol{n} &= 0 \quad \text{on } \partial\Omega\setminus\Gamma_0\,. \end{aligned} \tag{2.5.23}$$

(2.5.23) = Second-order elliptic BVP with Neumann boundary conditions on $\partial\Omega\setminus\Gamma_0$ Short name for BVPs of the type (2.5.23): "Mixed Neumann–Dirichlet problem"

?! *Review question(s) 2.5.24.* (Elliptic boundary value problems)

1. State Gauss' theorem for a vector field $\mathbf{j} \in (C^1(\overline{\Omega}))^d$ on a domain $\Omega \subset \mathbb{R}^d$.
2. State the 2-point boundary value problem satisfied by the solution of the variational equation
$$u \in H^1(]0,1[): \quad \int_0^1 (1+x^2)\frac{du}{dx}(x)\left(\frac{dv}{dx}(x) - v(x)\right) = v(0) \quad \forall v \in H^1(]0,1[)\,.$$
3. We consider the variational problem
$$\mathbf{u} : \Omega \to \mathbb{R}^2: \quad \int_\Omega \operatorname{div}\mathbf{u}(\boldsymbol{x})\operatorname{div}\mathbf{v}(\boldsymbol{x}) + \mathbf{u}(\boldsymbol{x})\cdot\mathbf{v}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int\limits_{\partial\Omega} \mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{v} : \Omega \to \mathbb{R}^2\,. \tag{2.5.25}$$
What is a suitable Sobolev space and what boundary value problem is satisfied by the vector field $\mathbf{u}$?
4. Which boundary value problem does the minimizer of the functional
$$J(v) = \int\limits_\Omega \left|\frac{\partial u}{\partial x_1}(\boldsymbol{x}) - \frac{\partial u}{\partial x_2}(\boldsymbol{x})\right|^2 + |u(Bx)|^2 - \|\boldsymbol{x}\|u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad v \in H_0^1(\Omega)\,,$$
solve? Here, $\Omega \subset \mathbb{R}^2$ is a bounded domain.

2.6 Diffusion models (Stationary heat conduction)

Now we look at a class of physical phenomena, for which models are based on two building blocks

1. a conservation principle (of mass, energy, etc.),
2. a potential driven flux of the conserved quantity.

Mathematical modelling for these phenomena naturally involves partial differential equations in the first steps, which are supplemented with boundary conditions. Hence, second-order elliptic boundary value problems arise first, while variational formulations are deduced from them, thus reversing the order of steps followed for equilibrium models in Section 2.2 through Section 2.5.

(2.6.1) Heat flux

In order to keep the presentation concrete, the discussion will target heat conduction, about which everybody should have a sound "intuitive grasp".

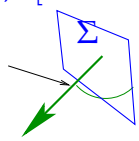
✎ notation: $\Omega \subset \mathbb{R}^3$: bounded open region occupied by solid object
($\hat{=}$ $\Omega \to$ computational domain)

Fundamental concept: heat flux, modelled by vector field $\mathbf{j} : \Omega \mapsto \mathbb{R}^3$

Heat flux = power flux: $[\mathbf{j}] = \frac{\mathrm{W}}{\mathrm{m}^2}$

Vector field $\mathbf{j} : \Omega :=]0,1[^2 \mapsto \mathbb{R}^3$ ▷

normal vector $\boldsymbol{n}$



Total heat flux through oriented surface $\Sigma \subset \mathbb{R}^3$

Power $$P_\Sigma = \int_\Sigma \mathbf{j} \cdot \boldsymbol{n} \, \mathrm{d}S \,. \qquad (2.6.2)$$

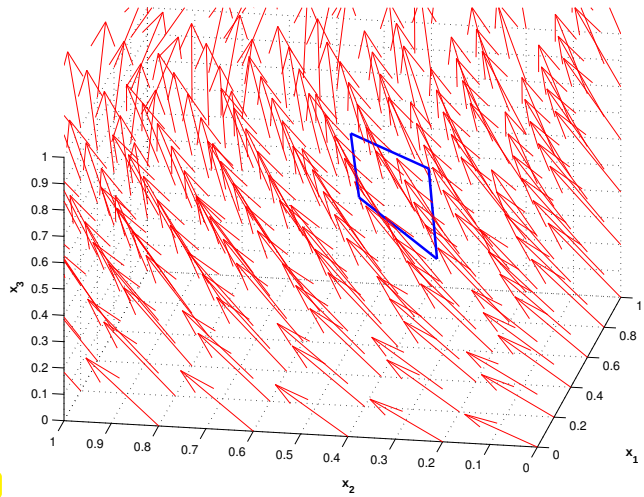


Fig. 76

P_Σ ($[P_\Sigma] = 1\mathrm{W}$): *directed* total power flowing through the oriented surface Σ per unit time. Note that the sign of P_Σ will change when flipping the normal of Σ!

Conservation of energy

$$\int_{\partial V} \mathbf{j} \cdot \boldsymbol{n} \, \mathrm{d}S = \int_V f \, \mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes" } V \,. \qquad (2.6.3)$$

power flux through surface of V heat production inside V

f = heat source/sink ($[f] = \frac{\mathrm{W}}{\mathrm{m}^3}$), $f = f(\boldsymbol{x})$ and f can be discontinuous ($f \in C^0_{\mathrm{pw}}(\Omega)$)

(2.6.4) Flux law

A flow of heat is triggered by temperature differences. Now we aim to quantify this relationship.

Intuition:
- ✦ heat flows from hot zones to cold zones
- ✦ the larger the temperature difference, the stronger the heat flow

Experimental evidence supports this intuition and, for many materials, yields the following quantitative relationship:

Fourier's law

$$\mathbf{j}(\boldsymbol{x}) = -\kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Omega\,. \tag{2.6.5}$$

Meaning of the quantities:

$\mathbf{j}$	= heat flux	$([\mathbf{j}] = 1\frac{\mathrm{W}}{\mathrm{m}^2})$	
u	= temperature	$([u] = 1\mathrm{K})$	(all functions of $\boldsymbol{x} \in \Omega$)
κ	= heat conductivity	$([\kappa] = 1\frac{\mathrm{W}}{\mathrm{Km}})$	

(2.6.5) $\Rightarrow$ Heat flow from hot to cold regions is linearly proportional to gradient of temperature

Some facts about the heat conductivity κ:

☞
- $\kappa = \kappa(\mathbf{x})$ for non-homogeneous materials. (spatially varying heat conductivity)
- κ can even be discontinuous for composite materials.
- κ may be $\mathbb{R}^{3,3}$-valued (heat conductivity tensor).

The most general form of the heat conductivity (tensor) enjoys the very same properties as the dielectric tensor introduced in Section 2.2.2:

From thermodynamic principles, *cf.* (2.2.17):

$$\exists \kappa^-, \kappa^+ > 0: \quad 0 < \kappa^- \leq \kappa(\mathbf{x}) \leq \kappa^+ < \infty \quad \text{for almost all } \mathbf{x} \in \Omega\,. \tag{2.6.6}$$

Terminology: (2.6.6) $\leftrightarrow$ κ is bounded and uniformly positive, see Def. 2.2.18.

(2.6.7) Derivation of 2nd-order linear elliptic PDE

From (2.6.3) by Gauss' theorem Thm. 2.5.7

$$\int_V \operatorname{div} \mathbf{j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_V f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes"} \quad V \subset \Omega\,.$$

Now appeal to another version of the fundamental lemma of the calculus of variations, see Lemma 2.5.12, this time sporting piecewise constant test functions.

► local form of energy conservation:

$$\operatorname{div} \mathbf{j} = f \quad \text{in } \Omega \,. \tag{2.6.8}$$

(2.6.9)

Combine equations (2.6.8) & (2.6.5)

$$\mathbf{j} = -\kappa(\mathbf{x})\,\mathbf{grad}\,u \quad (2.6.5) \qquad + \qquad \operatorname{div} \mathbf{j} = f \quad (2.6.8)$$

$$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \,. \tag{2.6.10}$$

► *Linear* scalar second order elliptic PDE (for unknown temperature u)

?! *Review question(s) 2.6.11.* (Stationary heat conduction)

1. Why is Fourier's law called a *linear* material law?
2. What is the physical meaning of the right hand side function f of the stationary heat equation.
3. Consider the functions

2.7 Boundary conditions

In the examples from Section 2.2.1, Section 2.2.2 we fixed the value of the unknown function $u : \Omega \mapsto \mathbb{R}$ on the boundary $\partial\Omega$: Dirichlet boundary conditions in (2.5.16)

$$u = g \quad \text{on} \quad \partial\Omega \quad \text{for given } g \in C^0(\partial\Omega) \,.$$

Exception: free edge of taut membrane, see Ex. 2.5.18: (homogeneous) Neumann boundary conditions in (2.5.23):

$$(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) \cdot \boldsymbol{n} = 0 \quad \text{on} \quad \partial\Omega \,.$$

In this section we resume the discussion of boundary conditions and examine them for stationary heat conduction, see previous section. This has the advantage that for this everyday physical phenomenon boundary conditions have a very clear intuitive meaning.

Fundamental boundary conditions for 2nd-order elliptic BVPs
Boundary conditions on surface/boundary $\partial\Omega$ of Ω:

(i) Temperature u is fixed: with $g : \partial\Omega \mapsto \mathbb{R}$ prescribed

$$u = g \quad \text{on } \partial\Omega \,. \tag{2.7.2}$$

▶ Dirichlet boundary conditions

(ii) Heat flux $\mathbf{j}$ through $\partial\Omega$ is fixed: with $h : \partial\Omega \mapsto \mathbb{R}$ prescribed ($\boldsymbol{n} : \partial\Omega \mapsto \mathbb{R}^3$ exterior unit normal vectorfield) on $\partial\Omega$

$$\mathbf{j} \cdot \boldsymbol{n} = -h \quad \text{on} \quad \partial\Omega \,. \tag{2.7.3}$$

▶ Neumann boundary conditions

(iii) Heat flux through $\partial\Omega$ depends on (local) temperature: with increasing function $\Psi : \mathbb{R} \mapsto \mathbb{R}$

$$\mathbf{j} \cdot \boldsymbol{n} = \Psi(u) \quad \text{on } \partial\Omega \tag{2.7.4}$$

▶ radiation boundary conditions

Example 2.7.5 (Convective cooling (simple model))

Heat is carried away from the surface of the body by a fluid at bulk temperature u_0. A crude model assumes that the heat flux depends *linearly* on the temperature difference between the surface of Ω and the bulk temperature of the fluid.

$$\mathbf{j} \cdot \boldsymbol{n} = q(u - u_0) \quad \text{on } \partial\Omega \,, \quad \text{where } 0 < q^- \leq q(\mathbf{x}) \leq q^+ < \infty \quad \text{for almost all } \mathbf{x} \in \partial\Omega \,.$$

When combined with Fourier's law (2.6.5), the convective cooling boundary conditions become

$$\kappa(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) + q(u(\boldsymbol{x}) - u_0) = 0 \,, \quad \boldsymbol{x} \in \partial\Omega \,,$$

and in this form they are known as Robin boundary conditions.

Example 2.7.6 (Radiative cooling (simple model))

A hot body emits electromagnetic radiation (blackbody emission), which drains thermal energy. The radiative energy loss is roughly proportional to the 4th power of the temperature difference between the surface temperature of the body and the ambient temperature.

$$\mathbf{j} \cdot \boldsymbol{n} = \alpha |u - u_0| (u - u_0)^3 \quad \text{on } \partial\Omega \,, \quad \text{with } \alpha > 0$$

$\rightarrow$ Non-linear boundary condition

Terminology: If $g = 0$ or $h = 0 \rightarrow$ homogeneous Dirichlet or Neumann boundary conditions

Remark 2.7.7 (Mixed boundary conditions)

Different boundary conditions can be prescribed on different parts of $\partial\Omega$
($\rightarrow$ mixed boundary conditions, *cf.* Ex. 2.5.18)

Example 2.7.8 ("Wrapped rock on a stove")

We consider a solid cylinder mounted on a heating plate whose temperature can be controlled. The vertical walls of the cylinder are covered with an insulating layer, which is assumed to be perfect. The top face is in contact with air and, thus, heat is transported away by convective cooling, see Ex. 2.7.5.

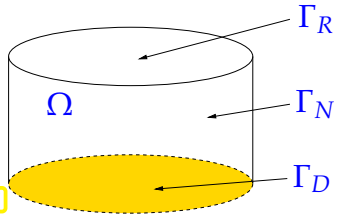


Fig. 77

- Non-homogeneous Dirichlet boundary conditions on $\Gamma_D \subset \partial\Omega$
- Homogeneous Neumann boundary conditions on $\Gamma_N \subset \partial\Omega$
- Convective cooling boundary conditions on $\Gamma_R \subset \partial\Omega$

Partition: $\partial\Omega = \overline{\Gamma}_D \cup \overline{\Gamma}_N \cup \overline{\Gamma}_R$, $\Gamma_D, \Gamma_N, \Gamma_R$ mutually disjoint

$$\underbrace{-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f}$$

$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f$ + boundary conditions $\Rightarrow$ elliptic boundary value problem (BVP)

For second order elliptic boundary value problems exactly one boundary condition is needed on any part of $\partial\Omega$.

Remark 2.7.9 (Linear BVP)

Observe that the solution mapping $\begin{pmatrix} f \\ g \end{pmatrix} \mapsto u$ for (2.6.10), (2.7.2) is linear.

This means that if u_i solves the Dirichlet problem with source function f_i and Dirichlet data g_i, $i = 1, 2$, then $u_1 + u_2$ solves (2.6.10) & (2.7.2) for source $f_1 + f_2$ and boundary values $g_1 + g_2$.

?! *Review question(s) 2.7.10.* (Boundary conditions for 2nd-order elliptic BVPs)

In this quizz we consider *stationary electric currents* in a conducting body occupying $\Omega \subset \mathbb{R}^3$. In this model a vector field $\mathbf{j} : \Omega \to \mathbb{R}^3$ describes the electric current density (units $[\mathbf{j}] = \frac{\mathrm{A}}{\mathrm{m}^2}$) obeying *Ohm's law* $\mathbf{j} = -\sigma\,\mathbf{grad}\,u$, which corresponds to Fourier's law (2.6.5). Here, u is the electric potential, *cf.* (2.2.14) (units $[u] = \mathrm{V}$), and $\sigma : \Omega \to \mathbb{R}^+$ stands for the uniformly positive conductivity (units $[\sigma] = \frac{\mathrm{A}}{\mathrm{Vm}}$).

1. What is the meaning of $\operatorname{div}\mathbf{j}$?
2. Argue, why the *normal component* of $\mathbf{j}$ has to be continuous across any smooth surface.
3. What is the physical meaning of Dirichlet and Neumann boundary conditions in the stationary current model?
4. What could be described by a linear radiation boundary condition (2.7.3) for the stationary current model.

2.8 Characteristics of elliptic boundary value problems

Some qualitative insights gained from heat conduction model:

- continuity: the temperature u must be continuous (jump in $u \to \mathbf{j} = \infty$).
- normal component of $\mathbf{j}$ across surfaces inside Ω must be continuous (jump in $\mathbf{j} \cdot \mathbf{n} \to$ heat source f of infinite intensity).
- interior smoothness of u: u smooth where f and D smooth.
- non-locality: local alterations in f, g, h affect u everywhere in Ω.
- quasi-locality: If local changes in f, g, h confined to $\Omega' \subset \Omega$, their effects decay away from Ω'.
- maximum principle: (in the absence of heat sources extremal temperatures are on the boundary)

$$\boxed{\text{if } f \equiv 0, \text{ then} \qquad \inf_{\mathbf{y}\in\partial\Omega} u(\mathbf{y}) \leq \mathbf{u}(\mathbf{x}) \leq \sup_{\mathbf{y}\in\partial\Omega} u(\mathbf{y}) \qquad \text{for all } \mathbf{x} \in \Omega}$$

Typical features of solutions of 2nd-order scalar elliptic boundary value problems

Example 2.8.1 (Scalar elliptic boundary value problem in one space dimension)

In one dimension the properties of solutions claimed above are obvious. Consider the Poisson equation (2.5.15) in 1D, which boils down to $-u'' = f$. Solutions of the associated two-point boundary value problems can be obtained by integrating f twice.

➢ f discontinuous, piecewise C^0 $\Rightarrow$ $u \in C^1$, piecewise C^2

Example 2.8.2 (Smoothness of solution of scalar elliptic boundary value problem)

Here we give "visual evidence" in 2D that solutions of Poisson's equation (2.5.15) enjoy enhanced smoothness compared to that of the right hand side f. We consider the following boundary value problem:

$$\begin{gathered} -\Delta u = f(\mathbf{x}) \quad \text{in } \Omega :=]0,1[^2 \quad , \quad u = 0 \quad \text{on } \partial\Omega \, , \\ f(\mathbf{x}) := \operatorname{sign}(\sin(2\pi k_1 x_1)\sin(2\pi k_2 x_2)) \, , \quad \mathbf{x} \in \Omega \, , \quad k_1, k_2 \in \mathbb{N} \, . \end{gathered} \tag{2.8.3}$$

Approximate solution computed by means of linear Lagrangian finite elements + lumping
($\to$ Chapter 3 below, details in Section 3.3, Section 3.6.5)

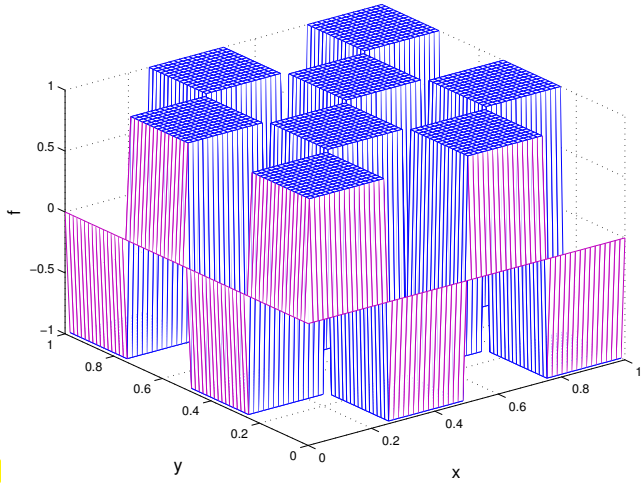


Fig. 78

Source term $f(\mathbf{x})$, $k_1 = k_2 = 2$

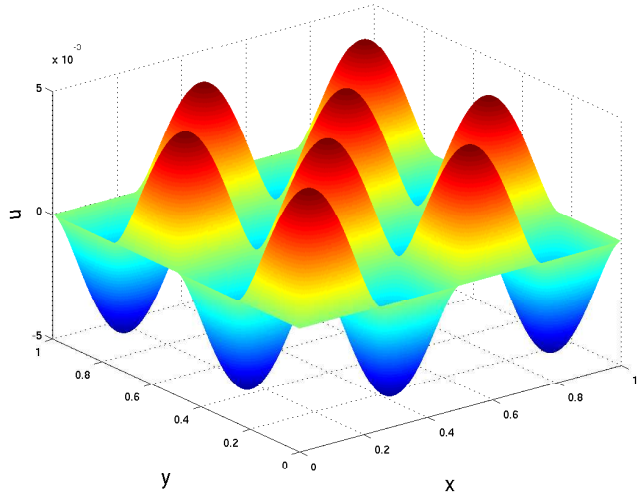


Fig. 79

Solution of (2.8.3)

➢ "Smooth" u despite "rough" f !

Example 2.8.4 (Quasi-locality of solution of scalar elliptic boundary value problem)

Now we give a demonstration that the influence of a local heat source decays away from its support. We look at Poisson problem, where the source function f is non-zero only on a small disk.

$$-\Delta u = f_\delta(\mathbf{x}) \quad \text{in } \Omega :=]0,1[^2 \quad , \quad u = 0 \quad \text{on } \partial\Omega \,, \tag{2.8.5}$$

$$f_\delta(\mathbf{x}) = \begin{cases} \delta^{-2} & \text{, if } \left\| \mathbf{x} - \left(\begin{smallmatrix} 1/2 \\ 1/2 \end{smallmatrix}\right) \right\|_2 \leq \delta \,, \\ 0 & \text{elsewhere.} \end{cases} \quad , \quad \delta > 0 \,. \tag{2.8.6}$$

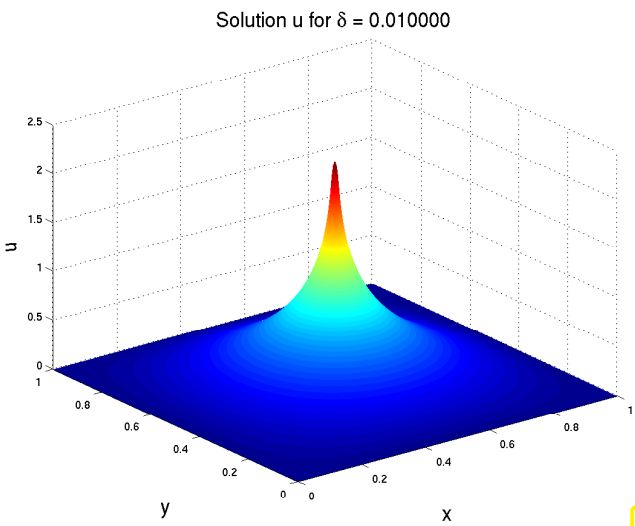


Fig. 80

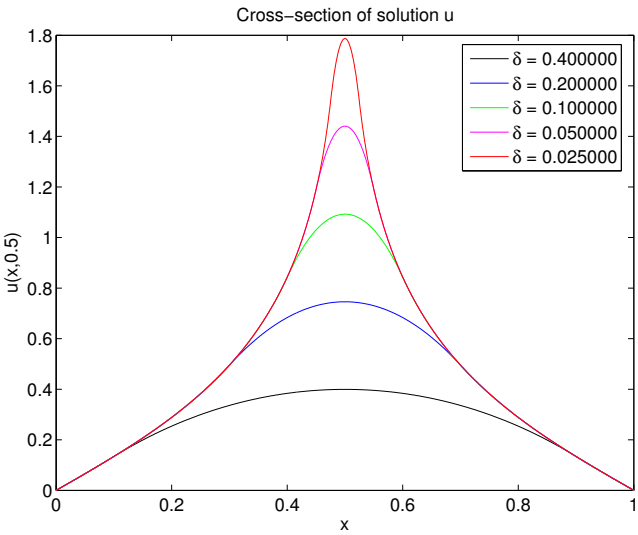


Fig. 81

2.9 Second-order elliptic variational problems

In Chapter 1 and Section 2.2 through Section 2.5 we pursued the derivation:

Minimization problem (e.g., (2.2.12), (2.2.24))	➢	Variational problem (e.g., (2.4.3), (2.4.4))	➢	BVP for PDE (e.g., (2.5.16), (2.5.23))

Now we are proceeding in the opposite direction:

PDE (e.g. (2.6.10))	+	boundary conditions (e.g., (2.7.2), (2.7.3), (2.7.4))	➢	variational problem

Formal transition from boundary value problem for PDE to variational problem

STEP 1:	*test PDE with smooth functions* (do not test, where the solution is known, e.g., on the boundary)
STEP 2:	*integrate over domain*
STEP 3:	*perform integration by parts* (e.g. by using Green's first formula, Thm. 2.5.9)
STEP 4: [optional]	*incorporate boundary conditions into boundary terms*
STEP 5:	*Choose suitable function spaces (Sobolev spaces)* (Section 2.3.1: largest function space on which variational problem well posed)

Example 2.9.2 (Variational formulation for heat conduction with Dirichlet boundary conditions)

$$\text{Targeted BVP:} \quad -\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega\,. \tag{2.9.3}$$

Here the solution is fixed on $\partial\Omega$. Therefore, we test with functions that vanish there.

STEP 1 & 2: test the PDE with $v \in C_0^\infty(\Omega)$ and integrate over Ω

$$\blacktriangleright \quad -\int_\Omega \operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u)\,v\,\mathrm{d}\boldsymbol{x} = \int_\Omega f v\,\mathrm{d}\boldsymbol{x}\,. \tag{2.9.4}$$

Again note: $v_{|\partial\Omega} = 0$ for test function, because u already fixed on $\partial\Omega$.

STEP 3: use Green's formula from Thm. 2.5.9 on $\Omega \subset \mathbb{R}^d$ (multidimensional integration by parts): Apply

$$\int_\Omega \mathbf{j}\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} = -\int_\Omega \operatorname{div}\mathbf{j}\,v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\mathbf{j}\cdot\mathbf{n}\,v\,\mathrm{d}S\,. \tag{2.5.10}$$

to (2.9.4) choosing the vector field as $\mathbf{j} := \kappa(\boldsymbol{x})\,\mathbf{grad}\,u$:

$$\blacktriangleright \quad \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} - \underbrace{\int_{\partial\Omega}\kappa(\boldsymbol{x})\,\mathbf{grad}\,u\cdot\boldsymbol{n}\,v\,\mathrm{d}S}_{=0,\text{because } v_{|\partial\Omega}=0} = \int_\Omega f v\,\mathrm{d}\boldsymbol{x} \quad \forall v \in C_0^\infty(\Omega)\,.$$

This gives the variational formulation after we switch to "maximal admissible function spaces" (Sobolev spaces, see Section 2.3, as spaces of functions with finite energy).

Variational form of (2.9.3): seek

$$\begin{array}{c} u \in H^1(\Omega) \\ u = g \text{ on } \partial\Omega \end{array} : \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,. \tag{2.9.5}$$

Example 2.9.6 (Variational formulation: heat conduction with general radiation boundary conditions)

In this case the appropriate treatment of boundary conditions in STEP 4 can be demonstrated.

$$\text{BVP:} \quad -\operatorname{div}(\kappa(\boldsymbol{x})\, \mathbf{grad}\, u) = f \quad \text{in } \Omega \quad , \quad -\kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \boldsymbol{n} = \Psi(u) \quad \text{on } \partial\Omega \,. \tag{2.9.7}$$

STEP 1 & 2: $u_{|\partial\Omega}$ not fixed $\Rightarrow$ test with $v \in C^\infty(\overline{\Omega})$

$$\blacktriangleright \quad -\int_\Omega \operatorname{div}(\kappa(\boldsymbol{x})\, \mathbf{grad}\, u)\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^\infty(\overline{\Omega}) \,.$$

STEP 3 & 4: apply Green's first formula (2.5.10) and incorporate boundary conditions:

$$\blacktriangleright \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} \underbrace{\kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{n}}_{=-\Psi(u) \text{ (STEP 4)}} v \,\mathrm{d}S = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in C^\infty(\overline{\Omega}) \,.$$

$$\blacktriangledown$$

Variational formulation of (2.9.7): seek

$$u \in H^1(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \Psi(u)\, v \,\mathrm{d}S = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \,. \tag{2.9.8}$$

Theorem 2.9.9. Classical solutions are weak solutions

If $\kappa \in C^1(\Omega)$, classical solutions $u \in C^2(\Omega)$ of the boundary value problems (2.9.3) *and* (2.9.7) *also solve the associated variational problems.*

Proof. Apply Thm. 2.5.9 as in the derivation of the weak formulations. □

Example 2.9.10 (Variational formulation for Neumann problem)

2nd-order elliptic (inhomogeneous) Neumann problem

$$\text{BVP:} \quad \begin{array}{l} -\operatorname{div}(\kappa(\boldsymbol{x})\, \mathbf{grad}\, u) = f \quad \text{in } \Omega \,, \\ \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \boldsymbol{n} = h(\boldsymbol{x}) \quad \text{on } \partial\Omega \,. \end{array} \tag{2.9.11}$$

We confront Neumann boundary conditions (2.7.3) (prescribed heat flux) on the whole boundary.

Variational formulation derived as in Ex. 2.9.6, with $\Psi(u) = -h$.

$$u \in H^1(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} - \int_{\partial\Omega} h\, v \,\mathrm{d}S = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,. \tag{2.9.12}$$

Observation: when we test (2.9.8) with $v \equiv 1$ ▶ $\boxed{-\int_{\partial\Omega} h \,\mathrm{d}S = \int_\Omega f \,\mathrm{d}\boldsymbol{x}}$ (2.9.13)

This is a compatibility condition for the existence of (variational) solutions of the Neumann problem!

Interpretation of (2.9.13) against the backdrop of the stationary heat conduction model:

conservation of energy $\rightarrow$ (2.6.3): Heat generated inside Ω ($\leftrightarrow f$) must be offset by heat flux through $\partial\Omega$ ($\rightarrow h$).

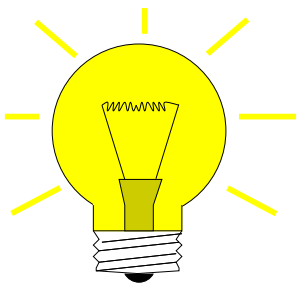
Remark 2.9.14 (Uniqueness of solutions of Neumann problem)

Observation: if compatibility condition (2.9.13) holds true, then

$$v \in H^1(\Omega) \quad \text{solves (2.9.8)} \quad \Longleftrightarrow \quad v + \gamma \quad \text{solves (2.9.8)} \quad \forall \gamma \in \mathbb{R}\,,$$

we say, "the solution is unique only up to constants".

Complementary observation: $\mathsf{a}(u,v) := \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$ is *not* s.p.d ($\rightarrow$ Def. 2.2.40) on $H^1(\Omega)$.



Idea: Restore uniqueness of solutions by enforcing average temperature to be zero $\int_\Omega u(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0$

This amounts to posing the variational problem (2.9.8) over the constrained function space

$$H^1_*(\Omega) := \{v \in H^1(\Omega): \int_\Omega v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = 0\}\,. \tag{2.9.15}$$

The norm on $H^1_*(\Omega)$ is the same as on $H^1_0(\Omega)$, see Def. 2.3.25. Obviously (why ?), the norm property (N1) is satisfied. These arguments also show that a is s.p.d ($\rightarrow$ Def. 2.2.40) on $H^1_*(\Omega)$, *cf.* Thm. 2.9.20.

▶ Uniquely solvable variational formulation of Neumann problem:

$$u \in H^1_*(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\, v \,\mathrm{d}S \quad \forall v \in H^1_*(\Omega)\,. \tag{2.9.16}$$

Remark 2.9.17 (Well-posedness of variational Neumann problem)

For the sake of simplicity we consider the homogeneous Neumann problem with constant coefficients, that is (2.9.16) with vanishing Neumann data $h = 0$ and $\kappa \equiv 1$:

$$u \in H^1_*(\Omega): \quad \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_*(\Omega)\,. \tag{2.9.18}$$

General Neumann data will be discussed below in § 2.10.7.

Question ($\rightarrow$ Section 2.4.2.2): How do perturbations δf in the source function f, measured in $L^2(\Omega)$-norm, affect the energy norm (= $|\cdot|_{H^1(\Omega)}$) of the solution. As in Section 2.4.2.2, see (2.4.32) and (2.4.33), we find that the induced perturbation δu of the solution complies with the variational equations

$$\delta u \in H^1_*(\Omega): \quad \int_\Omega \mathbf{grad}\, \delta u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega \delta f\, v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_*(\Omega)\,. \tag{2.9.19}$$

Recall: (2.4.30) was a consequence of the first Poincaré-Friedrichs inequality of Thm. 2.3.31, which makes a statement about norms on $H^1_0(\Omega)$. However, now we deal with a variational problem posed on $H^1_*(\Omega)$. Thus, we need a counterpart of Thm. 2.3.31 on that space:

Theorem 2.9.20. Second Poincaré-Friedrichs inequality

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

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

✎ notation: $C = C(\Omega)$ indicates that the constant C may depend on the shape of the domain Ω.

Proof. (for $d = 1$, $\Omega = [0,1]$ only, technically difficult in higher dimensions, see [2, Thm. 1.6.6])

As in the proof of Thm. 2.3.31, we employ a density argument and assume that u is sufficiently smooth, $u \in C^1([0,1])$.

By the fundamental theorem of calculus (2.5.2)

$$u(x) = u(y) + \int_y^x \frac{du}{dx}(\tau)\,\mathrm{d}\tau\,, \quad 0 \leq x, y \leq 1\,.$$

$$\blacktriangleright \quad u(x) = \int_0^1 u(x)\,\mathrm{d}y = \underbrace{\int_0^1 u(y)\,\mathrm{d}y}_{=0} + \int_0^1 \int_y^x \frac{du}{dx}(\tau)\,\mathrm{d}\tau\,\mathrm{d}y\,.$$

Then use the Cauchy-Schwarz inequality (2.3.30)

$$u(x)^2 \leq \int_0^1 \int_y^x 1\,\mathrm{d}\tau\,\mathrm{d}y \int_0^1 \int_y^x \left|\frac{du}{dx}(\tau)\right|^2 \mathrm{d}\tau\,\mathrm{d}y \leq \int_0^1 \left|\frac{du}{dx}(\tau)\right|^2 \mathrm{d}\tau\,.$$

Integrate over Ω yields the estimate

$$\|u\|_0^2 = \int_0^1 u^2(x)\,\mathrm{d}x \leq \int_0^1 \left|\frac{du}{dx}(\tau)\right|^2 \mathrm{d}\tau = |u|^2_{H^1(\Omega)} \,.$$

By (2.3.30), Thm. 2.9.20 implies the continuity of the first term in ℓ. □

An immediate consequence of Thm. 2.9.20 and the Cauchy-Schwarz inequality (2.2.44) for integrals in the form (1.6.13) is

$$|\delta u|_{H^1(\Omega)} \leq \operatorname{diam}(\Omega)\, C\, \|\delta f\|_{L^2(\Omega)} \,,$$

where δu solves (2.9.19) and $C > 0$ depends on the shape of Ω only.

?! *Review question(s) 2.9.21.* (Elliptic variational problems)

1. Consider the partial differential equation

$$\mathbf{grad}\,\operatorname{div}\mathbf{u} + c(\boldsymbol{x})\mathbf{u} = \mathbf{f} \quad \text{in } \Omega \subset \mathbb{R}^3 \,,$$

   where $c : \Omega \to \mathbb{R}$ is a bounded and uniformly positive definite coefficient function. Derive the formal variational formulations for boundary value problems for this PDE when equipped with the boundary conditions

   (a) $\mathbf{u} \cdot \boldsymbol{n} = 0$ on $\partial\Omega$, where $\boldsymbol{n}$ is the exterior unit normal vectorfield on $\partial\Omega$.

   (b) $\operatorname{div}\mathbf{u} = 0$ on $\partial\Omega$.

2. What is the meaning and relationship of *classical* and *weak* solutions of 2nd-order elliptic boundary value problems?

2.10 Essential and natural boundary conditions

(2.10.1) A synopsis of scalar 2nd-order linear elliptic boundary value problems

BVPs in strong and weak form, see Section 2.7 for a discussion of boundary conditions and both Section 2.9 and Section 2.5 for how to connect weak and strong forms.

☛ 2nd-order elliptic Dirichlet problem:

$$-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega \,. \tag{2.5.16}$$

with variational formulation

$$\begin{array}{l} u \in H^1(\Omega) \,, \\ u = g \text{ on } \partial\Omega \end{array} : \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_0(\Omega) \,. \tag{2.4.5}$$

☛ 2nd-order elliptic Neumann problem:

$$-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\boldsymbol{n} = -h \quad \text{on } \partial\Omega\,. \tag{2.10.2}$$

with variational formulation

$$u \in H^1_*(\Omega): \quad \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} = \int_\Omega f v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\,v\,\mathrm{d}S \quad \forall v \in H^1_*(\Omega)\,. \tag{2.9.16}$$

☛ 2nd-order elliptic mixed Neumann-Dirichlet problem, see Ex. 2.5.18:

$$-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad \begin{array}{rl} u = g & \text{on } \Gamma_0 \subset \partial\Omega\,, \\ (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\boldsymbol{n} = -h & \text{on } \partial\Omega\setminus\Gamma_0\,. \end{array} \tag{2.10.3}$$

with variational formulation

$$\begin{array}{l} u \in H^1(\Omega)\,, \\ u = g \text{ on } \Gamma_0 \end{array}: \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega\setminus\Gamma_0} h\,v\,\mathrm{d}S \tag{2.10.4}$$

for all $v \in H^1(\Omega)$ with $v_{|\Gamma_0} = 0$.

Natural and essential boundary conditions

A pattern emerges: In the variational formulations of 2nd-order elliptic BVPs of Section 2.9:

Dirichlet boundary conditions are *directly imposed* on trial space and (in homogeneous form) on test space.

Terminology: essential boundary conditions

Neumann boundary conditions are enforced *only* through the variational equation.

Terminology: natural boundary conditions

The attribute "natural" has been coined, because Neumann boundary conditions "naturally" emerge when removing constraints on the boundary, as we have seen for the partially free membrane of Ex. 2.5.18.

(2.10.6) Admissible Dirichlet data

Requirement for "Dirichlet data" $g : \partial\Omega \mapsto \mathbb{R}$ in (2.5.16):

$$\text{there is } u \in H^1(\Omega) \text{ such that } \quad u_{|\partial\Omega} = g$$

Analogous to Thm. 2.3.35:

If $g : \partial\Omega \mapsto \mathbb{R}$ is piecewise continuously differentiable (and bounded with bounded piecewise derivatives), then it can be extended to an $u_0 \in H^1(\Omega)$, *if and only if* it is continuous on $\partial\Omega$.

Bottom line: Dirichlet boundary values have to be continuous

This is also stipulated by physical insight, e.g. in the case of the taut membrane model of Section 2.2.1: discontinuous displacement on $\partial\Omega$ would entail ripping apart the membrane.

(2.10.7) Admissible Neumann data

In the variational problem (2.9.16) Neumann data $h : \partial\Omega \mapsto \mathbb{R}$ enter through the linear form on the right hand side

$$\ell(v) := \int_{\Omega} f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) \;.$$

Remember the discussion in the beginning of Section 2.3, also Ex. 2.4.22: we have to establish that ℓ is continuous on $H^1_*(\Omega)$ defined in (2.9.15). This is sufficient, because the coefficient function κ is uniformly positive and bounded, see (2.6.6). Thus, the energy $\|\cdot\|_{\mathsf{a}}$ associated with the bilinear form

$$\mathsf{a}(u,v) = \int_{\Omega} \kappa(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}$$

can be bounded from above and below by $|\cdot|_{H^1(\Omega)}$.

Continuity of the boundary contribution to the right hand side linear functional ℓ is ensured by a trace theorem:

Theorem 2.10.8. Multiplicative trace inequality

$$\exists C = C(\Omega) > 0\colon \quad \|u\|^2_{L^2(\partial\Omega)} \leq C\|u\|_{L^2(\Omega)} \cdot \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega) \;.$$

Proof. (for $d = 1$, $\Omega = [0,1]$ only, technically difficult in higher dimensions)

As in the proof of Thm. 2.3.31 and Thm. 2.9.20, we employ a density argument and assume that u is sufficiently smooth, $u \in C^1([0,1])$.

By the fundamental theorem of calculus (2.5.2):

$$u(1)^2 = \int_0^1 \frac{dw}{d\xi}(x)\,\mathrm{d}x \;, \quad \text{with} \quad w(\xi) := \xi u^2(\xi) \;,$$

$$\blacktriangleright \qquad u(1)^2 = \int_0^1 u^2(x) + 2u(x)\frac{du}{dx}(x)x\,\mathrm{d}x \;.$$

Then use the Cauchy-Schwarz inequality (2.3.30):

$$u(1)^2 \leq \int_0^1 u^2(x)\,\mathrm{d}x + 2\int_0^1 |x||u(x)|\left|\frac{du}{dx}(x)\right|\mathrm{d}x \leq \|u\|_0^2 + 2\|u\|_0\left\|\frac{du}{dx}\right\|_0 \;.$$

A similar estimate holds for $u(0)^2$. □

Now we can combine

- the Cauchy-Schwarz inequality (2.3.30) on $\partial\Omega$,

- the 2nd Poincaré-Friedrichs inequality of Thm. 2.9.20,
- the multiplicative trace inequality of Thm. 2.10.8:

$$\int_{\partial\Omega} hv \,\mathrm{d}S \overset{(2.3.30)}{\leq} \|h\|_{L^2(\partial\Omega)}\|v\|_{L^2(\partial\Omega)} \overset{\text{Thm. 2.10.8}}{\leq} C\|h\|_{L^2(\partial\Omega)}\|v\|_{H^1(\Omega)}$$
$$\overset{\text{Thm. 2.9.20}}{\leq} C\|h\|_{L^2(\partial\Omega)}|v|_{H^1(\Omega)} \quad \forall v \in H^1_*(\Omega) \,.$$

$h \in L^2(\partial\Omega)$ provides valid Neumann data for the 2nd order elliptic BVP (2.10.2).

In, particular Neumann data h can be *discontinuous*.

?! *Review question(s) 2.10.9.* (Essential and natural boundary conditions)

1. For a scalar 2nd-order elliptic boundary value problem for $-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) = f$ the Robin boundary conditions read, *cf.* Ex. 2.7.5,
$$\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u + \gamma(\boldsymbol{x})u = 0 \quad \text{on } \partial\Omega \,,$$
with $\gamma : \partial\Omega \to \mathbb{R}$ uniformly positive. Are these boundary conditions essential or natural.
2. Describe the minimal regularity of Dirichlet and Neumann data for scalar 2nd-order elliptic BVPs on $\Omega \subset \mathbb{R}^d$ in terms of classical smoothness spaces $C^k_{\mathrm{pw}}(\partial\Omega)$.
3. In the case of the PDE $-\mathbf{grad}\,\mathrm{div}\,\mathbf{u} + \mathbf{u} = \mathbf{f}$ on $\Omega \subset \mathbb{R}^3$, what are essential, what are natural boundary conditions. To answer this questions apply Thm. 2.5.9 and determine and study the boundary terms arising from it.

Learning outcomes

After having studied this chapter you should (be able to)

- convert a quadratic minimization problem into a linear variational problem
- use the formal calculus of variations to find the variational problem induced by a minimization problem posed on a space of functions in two or three dimensions.
- know the norms of the Sobolev spaces $L^2(\Omega)$, $H^1(\Omega)$, and $H^1_0(\Omega)$ and how to use them in the statement of variational problems.
- state the continuity featured by piecewise smooth functions in a Sobolev space.
- appreciate the importance of the continuity in the energy norm of right hand side functionals of variational problems.
- extract a PDE and boundary conditions from a variational problem using integration by parts.

- recast a boundary value problem for a 2nd-order PDE in variational form (using suitable Sobolev spaces).
- tell which boundary conditions make sense for a given 2nd-order PDE.
- distinguish essential and natural boundary conditions for a PDE in variational form.
- know sufficient conditions for admissible Dirichlet- and Neumann data in the case of scalar 2nd-order elliptic variational problems.
- know the compatibility conditions for the data in the case of a pure Neumann problem.

Bibliography

[1] D. Braess. *Finite Elements*. Cambridge University Press, 2nd edition, 2001.

[2] S. Brenner and R. Scott. *Mathematical theory of finite element methods*. Texts in Applied Mathematics. Springer–Verlag, New York, 2nd edition, 2002.

[3] L.C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1998.

[4] W. Hackbusch. *Elliptic Differential Equations. Theory and Numerical Treatment*, volume 18 of *Springer Series in Computational Mathematics*. Springer, Berlin, 1992.

[5] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[6] W. Rudin. *Real and Complex Analysis*. McGraw–Hill, 3rd edition, 1986.

[7] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

[8] D. Werner. *Funktionalanalysis*. Springer, Berlin, 1995.

Chapter 3

Finite Element Methods (FEM)

Contents

3.1 Introduction

(3.1.1) Focus, goals, and prerequisites

Problem	:	linear scalar second-order elliptic boundary value problem $\rightarrow$ Chapter 2
Perspective	:	variational interpretation in Sobolev spaces $\rightarrow$ Section 2.9
Objective	:	algorithm for the computation of an approximate numerical solution as a
	:	*function* belonging to a subspace of the variational function space

This chapter heavily relies on the material covered in Section 2.3, Section 2.9, and Section 2.10. The reader will not be able to understand what follows, unless she or he is familiar with these earlier sections of the course.

(3.1.2) Outline

Section 1.5.2 introduced the fundamental ideas of the Galerkin discretization of variational problems, or, equivalently, of minmization problems, posed over function spaces. A key ingredient are suitably chosen finite-dimensional trial and test spaces, equipped with ordered bases.

In Section 1.5.2.2 the abstract approach was discussed for two-point boundary value problems and the concrete case of piecewise linear trial and test spaces, built upon a partition (mesh/grid $\rightarrow$ § 1.5.68) of the interval (domain). In this context the locally supported tent functions ($\rightarrow$ Fig. 37) lent themselves as natural basis functions.

This chapter is devoted to extending the linear finite element method in 1D to

- 2nd-order linear variational problems on bounded spatial domains Ω in two and three dimensions,
- piecewise polynomial trial/test functions of higher degree.

The leap from $d = 1$ to $d = 2$ will encounter additional difficulties and many new aspects will emerge. This chapter will elaborate on them and present details of algorithms that tackle them. The discussion will even dip into details of implementation in C++.

In Section 3.2 we review the ideas and crucial steps of the abstract Galerkin discretization of a general linear variational problem. This refreshes **??**.

(3.1.3) Targeted boundary value problems

Except for Section 3.8, we will restrict ourselves to *linear* 2nd-order elliptic variational problems on spatial domains $\Omega \in \mathbb{R}^d$, $d = 2, 3$, with the properties listed in § 2.2.3.

☛ 2nd-order elliptic Dirichlet problem:

$$\begin{array}{l} u \in H^1(\Omega)\,, \\ u = g \text{ on } \partial\Omega \end{array} : \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,, \tag{2.4.5}$$

with *continuous* (→ § 2.10.6) Dirichlet data $g \in C^0(\partial\Omega)$.

☛ 2nd-order elliptic Neumann problems:

$$u \in H^1_*(\Omega): \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} = \int_\Omega f v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h\,v\,\mathrm{d}S \quad \forall v \in H^1_*(\Omega)\,, \tag{2.9.16}$$

posed on the constrained Sobolev space

$$H^1_*(\Omega) := \{v \in H^1(\Omega): \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0\}\,, \tag{2.9.15}$$

and with *piecewwise continuous* (→ § 2.10.7) Neumann data $h \in C^0_{\mathrm{pw}}(\partial\Omega)$ that satisfy the compatibility condition

$$-\int_{\partial\Omega} h\,\mathrm{d}S = \int_\Omega f\,\mathrm{d}\boldsymbol{x}\,. \tag{2.9.13}$$

A simpler version with homogeneous Neumann data and reaction term reads

$$u \in H^1(\Omega): \quad \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u\cdot\mathbf{grad}\,v + c(\boldsymbol{x})u\,v\,\mathrm{d}\boldsymbol{x} = \int_\Omega f v\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{3.1.4}$$

with *uniformly positive* reaction coefficient $c : \Omega \mapsto \mathbb{R}^+$, $c \in C^0_{\mathrm{pw}}(\Omega)$, *cf.* (2.2.17), Def. 2.2.18:

$$\exists 0 < \gamma^- \le \gamma^+ < \infty: \quad \gamma^- \le c(\boldsymbol{x}) \le \gamma^+ \quad \text{for almost all} \quad \boldsymbol{x} \in \Omega\,.$$

Supplement 3.1.5 (Almost all/almost everywhere).

In (mathematical) articles on function spaces and variational formulations in them you will often encounter phrases like "almost all" or "almost everywhere". The designate statements about point values of functions that remain true, if the function is changed on sets of points that "do not matter for integration".

For instance, since above c occurs only in an integrand, we do not care about it being positive on "sets of measure zero" like lines in 2D or surfaces in 3D. Whether domains of integration are open or closed is immaterial, too.

△

The benefit of (3.1.4) is that no compatibility condition like (2.9.13) is needed to ensure existence of solutions.

The considerations in Section 2.9 and Section 2.10 established the following key properties of these variational problems:

> The linear variational problems (→ Def. 1.4.8) (2.4.5), (2.9.16), and (3.1.4) feature symmetric positive definite bilinear forms (→ Def. 2.2.40) and right hand side linear forms that are continuous (→ Def. 2.2.56) with respect to the energy norm (→ Def. 2.2.43).
>
> ► existence and uniqueness of solutions (→ Rem. 2.3.15)

Please remember that all the variational problems are connected with quadratic minimization problems, see Section 2.2.3, Def. 2.2.32.

Remark 3.1.6 (Data in procedural form)

Rem. 1.5.5 still applies: all functions (coefficient $\boldsymbol{\alpha}$, source function f, Dirichlet data g, Neumann data h) may be given only in procedural form.

E.g., in MATLAB, as function `function y = f(x)` whose implementation is kept secret.

In C++ functions are encapsulated into function objects [14, Section 0.2.3]:

```
template<typename ReturnType,typename PointCoordinates>
class Function {
  using value_type = ReturnType;
  using arg_type = PointCoordinates;
  Function(void);
  // evaluation operator
  value_type operator ()(const PointCoordinates &x) const;
};
```

Recall the discussion of the consequences this procedural form in § 1.5.48 and Section 1.5.2.2

3.2 Galerkin Discretization

(3.2.1) Recalled: concept of discretization

Recall the concept of "discretization", see Section 1.5:

Not a moot point: any computer can handle only a finite amount of information (reals)

Variational boundary value problem	$\xrightarrow{\text{DISCRETIZATION}}$	System of a finite number of equations for (real) unknowns

---

(3.2.2) Recalled: linear variational problems

Abstract target of discretization in this chapter: linear variational problem (1.4.9)

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{3.2.3}$$

✦ $V_0 \hat{=}$ vector space (Hilbert space) (usually a Sobolev space → Section 2.3) with norm $\|\cdot\|_V$,
✦ $\mathsf{a}(\cdot,\cdot) \hat{=}$ bilinear form, continuous on V_0, which means

$$\exists C > 0: \quad |\mathsf{a}(u,v)| \leq C\|u\|_V\|v\|_V \quad \forall u,v \in V \,. \tag{3.2.4}$$

✦ $\ell \hat{=}$ continuous linear form in the sense of Def. 2.2.56, *cf.* (2.2.55),

$$\exists C > 0: \quad |\ell(v)| \leq C\|v\|_V \quad \forall v \in V_0 \,. \tag{3.2.5}$$

The importance of this *continuity* is discussed in the beginning of Section 2.3, see also Ex. 2.4.22. (The Cs in (3.2.4) and (3.2.5) are so-called "generic constants", whose values need not agree though they are designated by the same symbol, see Rem. 5.3.58 below.)

If a is symmetric and positive definite ($\rightarrow$ Def. 2.2.40), we may choose $\|\cdot\|_V := \|\cdot\|_{\mathsf{a}}$, "energy norm", see Def. 2.2.43. Continuity of a w.r.t. $\|\cdot\|_{\mathsf{a}}$ is clear.

In § 1.4.7 and also in (2.4.5) we encountered more general linear variational problems posed on an affine space

$$u \in u_0 + V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{3.2.6}$$

This can be converted into the form (3.2.3) with a modified right hand side functional through the "offset function trick", already discussed in § 1.5.12, on page 148, and page 128 in the context of quadratic minimization problems.

Recall: Galerkin discretization $\rightarrow$ Section 1.5.2

Idea of Galerkin discretization:

Replace V_0 in (3.2.3) with a finite dimensional subspace.

($V_{0,N} \subset V_0$ called Galerkin (or discrete) trial space/test space)

Notation: Twofold nature of symbol "N", *cf.* Section 1.5.2:

- N = formal index, tagging "discrete entities" ($\rightarrow$ "finite amount of information")
- $N = \dim V_{N,0} \in \mathbb{N} \mathrel{\hat{=}}$ dimension of Galerkin trial/test space

Discrete variational problem, *cf.* (1.5.9),

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N} \,. \tag{3.2.8}$$

Galerkin solution

We begin with a simple consequence of Thm. 2.2.52 and **??**, respectively.

Theorem 3.2.9. Existence and uniqueness of solutions of discrete variational problems

If the bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ is symmetric and positive definite ($\rightarrow$ Def. 2.2.40) and the linear form $\ell : V_0 \mapsto \mathbb{R}$ is continuous in the sense of

$$\exists C_\ell > 0: \quad |\ell(u)| \leq C_\ell \|u\|_{\mathsf{a}} \quad \forall u \in V_0 \,, \tag{2.2.55}$$

then the discrete variational problem has a unique Galerkin solution $u_N \in V_{0,N}$ that satisfies the stability estimate ($\rightarrow$ Section 2.4.2)

$$\|u_N\|_{\mathsf{a}} \leq C_\ell \,. \tag{3.2.10}$$

Proof. Uniqueness of u_N is clear:

$$\begin{array}{l} \mathsf{a}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N} \\ \mathsf{a}(w_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N} \end{array} \quad \Rightarrow \quad \mathsf{a}(u_N - w_N, v_N) = 0 \quad \forall v_N \in V_{N,0}$$

$$\xRightarrow{v_N := u_N - w_N \in V_{0,N}} \quad \|u_N - w_N\|_a = 0 \quad \xRightarrow{\text{a s.p.d.}} \quad u_N - w_N = 0 \,.$$

The discrete linear variational problem (3.2.8) is set in the *finite-dimensional* space $V_{0,N}$. Thus, uniqueness of solutions is equivalent to existence of solutions ($\rightarrow$ linear algebra).

If you do not like this abstract argument, wait and see the equivalence of (3.2.8) with a linear system of equations. It will turn out that under the assumptions of the theorem, the resulting system matrix will be symmetric and positive definite in the sense of [14, Def. 1.1.8].

The estimate (3.2.10) is immediate from setting $v_N := u_N$ in (3.2.8)

$$|\mathsf{a}(u_N, u_N)| = |\ell(u_N)| \leq C_\ell (\mathsf{a}(u_N, u_N))^{1/2} .$$

Then cancel a square root of $\mathsf{a}(u_N, u_N)$.

□

Recalled: second step of Galerkin discretization

Recall from Section 1.5.2: 2nd step of Galerkin discretization:

Introduce (ordered) basis $\mathfrak{B}_N$ of $V_{0,N}$:

$$\mathfrak{B}_N := \{b_N^1, \dots, b_N^N\} \subset V_N \quad , \quad V_{0,N} = \mathrm{Span}\{\mathfrak{B}_N\} \quad , \quad N := \dim(V_{0,N}) .$$

► Unique basis representations:

$$\begin{aligned} u_N &= \mu_1 b_N^1 + \cdots + \mu_N b_N^N , \quad \mu_i \in \mathbb{R} \\ v_N &= \nu_1 b_N^1 + \cdots + \nu_N b_N^N , \quad \nu_i \in \mathbb{R} \end{aligned} : \quad \text{plug into (3.2.8).}$$

Of course, there are infinitely many ways to choose the basis $\mathfrak{B}_N$. On the one hand, Thm. 1.5.25 tells us that

the choice of the basis does do not make a difference for the Galerkin solution u_N.

On the other hand, we saw in Exp. 1.5.59 that different bases lead to (non-)linear systems with vastly different properties.

Later in this chapter, we will take a close look at the impact of different choices of bases, see Rem. 3.2.15.

Now we repeat the derivation of (1.5.23) and, in particular, (1.5.56). Key is the bi-linearity of a,

$$\begin{aligned} &\mathsf{a}(\alpha_1 v_1 + \beta_1 u_1, \alpha_2 v_2 + \beta_2 u_2) = \\ &\quad \alpha_1\alpha_2\, \mathsf{a}(v_1, v_2) + \alpha_1\beta_2\, \mathsf{a}(v_1, u_2) + \beta_1\alpha_2\, \mathsf{a}(u_1, v_2) + \beta_1\beta_2\, \mathsf{a}(u_1, u_2) , \end{aligned}$$

for all $u_i, v_i \in V_0$, $\alpha_i, \beta_i \in \mathbb{R}$, and the linearity of ℓ

$$\ell(\alpha u + \beta v) = \alpha \ell(u) + \beta \ell(v) ,$$

for all $u, v \in V_0$, $\alpha, \beta \in \mathbb{R}$, see Def. 1.3.22.

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N} . \tag{3.2.8}$$

$$\Updownarrow \left[\begin{aligned} u_N &= \mu_1 b_N^1 + \cdots + \mu_N b_N^N , \mu_i \in \mathbb{R} \\ v_N &= \nu_1 b_N^1 + \cdots + \nu_N b_N^N , \nu_i \in \mathbb{R} \end{aligned} \right]$$

$$\sum_{k=1}^N \sum_{j=1}^N \mu_k \nu_j \mathsf{a}(b_N^k, b_N^j) = \sum_{j=1}^N \nu_j \ell(b_N^j) \quad \forall \nu_1, \dots, \nu_N \in \mathbb{R} ,$$

$$\Updownarrow$$

$$\sum_{j=1}^{N} v_j \left(\sum_{k=1}^{N} \mu_k \mathsf{a}(b_N^k, b_N^j) - \ell(b_N^j) \right) = 0 \quad \forall v_1, \dots, v_N \in \mathbb{R} \,,$$

$$\Updownarrow (*)$$

$$\sum_{k=1}^{N} \mu_k \mathsf{a}(b_N^k, b_N^j) = \ell(b_N^j) \quad \text{for } j = 1, \dots, N \,.$$

$$\Updownarrow [\vec{\boldsymbol{\mu}} = (\mu_1, \dots, \mu_N)^\top \in \mathbb{R}^N]$$

$$\boxed{\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}} \,, \quad \text{with} \quad \begin{aligned} \mathbf{A} &= \left(\mathsf{a}(b_N^k, b_N^j) \right)_{j,k=1}^{N} \in \mathbb{R}^{N,N} \,, \\ \vec{\boldsymbol{\varphi}} &= \left(\ell(b_N^j) \right)_{j=1}^{N} \,. \end{aligned}$$

A linear system of equations

Note that equivalence $(*)$ amounts to the statement of Lemma 1.5.24.

Summary: notions connected with Galerkin discretization

Linear discrete variational problem

$u_N \in V_{0,N}$: $\mathsf{a}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N}$

$\xrightarrow{\text{Choosing basis } \mathfrak{B}_N}$

Linear system of equations $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}$

Galerkin matrix: $\mathbf{A} = \left(\mathsf{a}(b_N^k, b_N^j) \right)_{j,k=1}^{N} \in \mathbb{R}^{N,N}$,

Right hand side vector: $\vec{\boldsymbol{\varphi}} = \left(\ell(b_N^j) \right)_{j=1}^{N} \in \mathbb{R}^N$,

Coefficient vector: $\vec{\boldsymbol{\mu}} = (\mu_1, \dots, \mu_N)^\top \in \mathbb{R}^N$,

Recovery of solution: $u_N = \sum_{k=1}^{N} \mu_k b_N^k$.

(3.2.13) Alternative (legacy) terminology

(Legacy) terminology for FEM:	Galerkin matrix	=	stiffness matrix
	Right hand side vector	=	load vector
	Galerkin matrix for $(u,v) \mapsto \int_\Omega uv \,\mathrm{d}\boldsymbol{x}$	=	mass matrix

This hails from the times (lates 60s and early 70s), when finite element methods were mainly applied to solid mechanics (linear elasticity).

A consequence of the equivalence of the linear system of equations $\mathbf{A}\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\varphi}}$ and the discrete variational problem (3.2.8).

Corollary 3.2.14.

(3.2.8) *has unique solution* $\Leftrightarrow$ $\mathbf{A}$ *nonsingular*

Remark 3.2.15 (Recalled: impact of choice of basis)

Thm. 1.5.25: choice of $\mathfrak{B}_N$ in theory does not affect u_N $\Rightarrow$ No impact on discretization error !

But: Key properties (e.g., conditioning) of matrix $\mathbf{A}$ crucially depend on basis $\mathfrak{B}_N$!

We have seen a striking example of the impact of the choice of basis functions in Exp. 1.5.59 (for a polynomial spectral Galerkin discretization), where the "unstable" monomial basis (1.5.32) made the condition number of the Galerkin matrix source exponentially in the problem size parameter B, whereas the stable basis composed of integrated Legendre polynomials (1.5.33) gave only mildly growing condition numbers.

For different bases $\mathfrak{B}_N$ we get different Galerkin matrices. How are these matrices related? What do they have in common? The next lemma gives answers:

Lemma 3.2.16. Effect of change of basis on Galerkin matrix

Consider (3.2.8) *and two bases of* $V_{0,N}$,

$$\mathfrak{B}_N := \{b_N^1, \dots, b_N^N\} \quad , \quad \widetilde{\mathfrak{B}}_N := \{\widetilde{b}_N^1, \dots, \widetilde{b}_N^N\} \,,$$

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

$$\widetilde{b}_N^j = \sum_{k=1}^N s_{jk} b_N^k \quad \text{with} \quad \mathbf{S} = (s_{jk})_{j,k=1}^N \in \mathbb{R}^{N,N} \text{ regular.} \tag{3.2.17}$$

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*

$$\widetilde{\mathbf{A}} = \mathbf{S}\mathbf{A}\mathbf{S}^T \quad , \quad \widetilde{\vec{\boldsymbol{\varphi}}} = \mathbf{S}\vec{\boldsymbol{\varphi}} \quad , \quad \widetilde{\vec{\boldsymbol{\mu}}} = \mathbf{S}^{-T}\vec{\boldsymbol{\mu}} \,. \tag{3.2.18}$$

Proof. Make use of the bilinearity of a ($\rightarrow$ Def. 1.3.22), (3.2.17) and the definition of the entries of the Galerkin matrix:

$$\widetilde{\mathbf{A}}_{lm} = \mathsf{a}(\widetilde{b}_N^m, \widetilde{b}_N^l) = \sum_{k=1}^N \sum_{j=1}^N s_{mk} \mathsf{a}(b_N^k, b_N^j) s_{lj} = \sum_{k=1}^N \underbrace{\Big(\sum_{j=1}^N s_{lj} \mathbf{A}_{jk}\Big)}_{(\mathbf{S}\mathbf{A})_{lk}} s_{mk} = (\mathbf{S}\mathbf{A}\mathbf{S}^T)_{lm} \,,$$

where we used the rules for the product of square matrices. □

Reminder of linear algebra:

Definition 3.2.19. 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 $\mathbf{B} = \mathbf{S}\mathbf{A}\mathbf{S}^T$.

Congruence is an equivalence relation on square matrices

Lemma 3.2.20. Congruent Galerkin matrices

Matrix property invariant under congruence $\Leftrightarrow$ *Property of Galerkin matrix invariant under change of basis* $\mathfrak{B}_N$

(3.2.21) Properties of congruent matrices

Matrix properties invariant under congruence :
- regularity $\rightarrow$ [14, Def. 1.6.8]
- symmetry
- positive definiteness $\rightarrow$ [14, Def. 1.1.8]

Proving the invariance of these properties is straightforward from the definition of congruence.

Not invariant are
- ✦ sparsity and bandstructure, *cf.* linear finite elements ($\rightarrow$ Section 1.5.2.2) and spectral Galerkin ($\rightarrow$ Section 1.5.2.1)
- ✦ conditioning, *cf.* Exp. 1.5.59

Nevertheless, these latter properties have fundamental consequences for the numerical solution of the linear system of equations (required storage, computational effort, and impact of roundoff errors), as was already remarked above.

?! *Review question(s) 3.2.22.* (Abstract Galerkin discretization)

1. Let $V_N \subset V$ be a finite-dimensional subspace of a normed vector space V. Show that that any best approximant
$$u_N \in \operatorname*{argmin}_{v_N \in V_N} \|u - v_N\|_V$$
satisfies $\|u_N\|_V \leq 2\|u\|_V$.
2. Give the formulas for the entries of the Galerkin matrix and the right hand side vector arising from the Galerkin discretization of a linear variational problem (3.2.3).
3. Using the notations of (3.2.3) and (3.2.8) and under the assumptions of Thm. 3.2.9, show that
$$\mathsf{a}(u - u_N, v_N) = 0 \quad \forall v_N \in V_0 \,.$$
4. Show that the solutions u and u_N of (3.2.3) and (3.2.8), respectively, satisfy
$$\tfrac{1}{2}\mathsf{a}(u,u) - \ell(u) \leq \tfrac{1}{2}\mathsf{a}(u_N,u_N) - \ell(u_N) \,,$$
if a is symmetric positive definite.
5. Explain the offset function trick converting (3.2.6) into the form (3.2.3); derive the modified right hand side functional.
6. We consider a linear variational problem: seek $u \in V_0$, $\mathsf{a}(u,v) = \ell(v)$ for all $v \in V_0$, with s.p.d. bilinear form $\mathsf{a}(\cdot,\cdot)$. Show that for every finite dimensional subspace $V_{0,N} \subset V_0$ there is a basis that yields a diagonal Galerkin matrix.
7. We consider the Galerkin discretization of a linear variational: seek $u \in V_0$, $\mathsf{a}(u,v) = \ell(v)$ for all $v \in V_0$, with s.p.d. bilinear form $\mathsf{a}(\cdot,\cdot)$. How do you have to modify the basis $\mathfrak{B} = \{b_N^1, \dots, b_N^N\}$ of a finite-dimensional subspace $V_{0,N}$ so that you obtain a Galerkin matrix with all diagonal entries $= 1$?

3.3 Case Study: Triangular Linear FEM in Two Dimensions

This section elaborates how to extend the linear finite element Galerkin discretization of Section 1.5.2.2 to two dimensions. Familiarity with the 1D setting is essential for understanding the current section.

Parts of the presentation are based on a simple C++ finite element code that is available on the course Git repository in lecture_codes/SimpleLinearFEM2D.

(3.3.1) Model problem

Initial focus: well-posed 2nd-order linear variational problem posed on $H^1(\Omega)$ ($\rightarrow$ Def. 2.3.25)

Example: Neuman problem with homogeneous Neumann data and reaction term

$$u \in H^1(\Omega): \quad \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v + c(\boldsymbol{x})u\,v\,\mathrm{d}\boldsymbol{x} = \int_\Omega f v\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{3.1.4}$$

$\Updownarrow\leftarrow$ see Section 2.5

BVP:
$$\begin{aligned} -\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) + c(\boldsymbol{x})u &= f \quad \text{in } \Omega\,, \\ \mathbf{grad}\,u \cdot \boldsymbol{n} &= 0 \quad \text{on } \partial\Omega\,. \end{aligned}$$

Remember that the reaction coefficient $c = c(\boldsymbol{x})$ is supposed to be uniformly positive definite, see (2.2.17) and Def. 2.2.18.

(3.3.2) Polygonal computational domain

For assumptions on the domain $\Omega \subset \mathbb{R}^2$ see § 2.2.3:

Here: Ω is a polygon

polygon with 10 corners $\triangleright$

By default, the domain Ω is assumed to be an *open* set, that is, $\boldsymbol{x} \in \Omega$ implies $\boldsymbol{x} \notin \partial\Omega$!

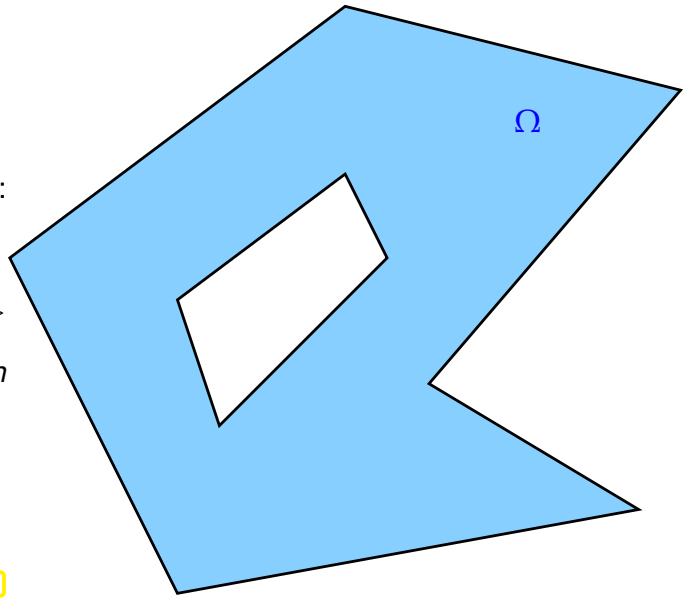


Fig. 82

3.3.1 Triangulations

Question: What is the 2D counterpart of the 1D mesh/grid $\mathcal{M}$ from Sect. (1.5.2.2) **?**

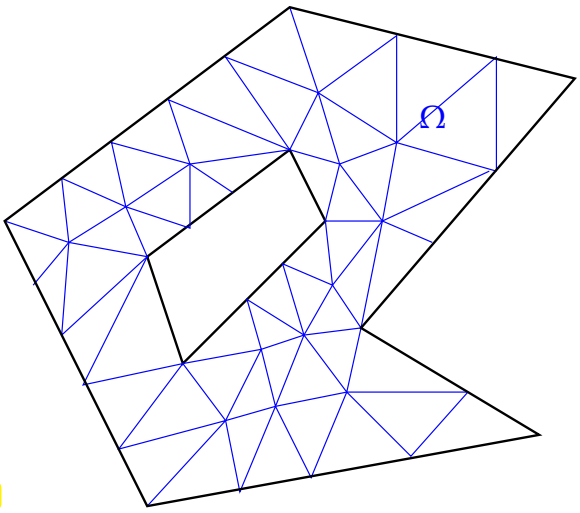


Fig. 83

Triangulation $\mathcal{M}$ of Ω satisfies

(i) $\mathcal{M} = \{K_i\}_{i=1}^{M}$, $M \in \mathbb{N}$, $K_i \hat{=}$ open triangle

(ii) disjoint interiors: $i \neq j \Rightarrow K_i \cap K_i = \emptyset$

(iii) tiling/partition property: $\bigcup_{i=1}^{M} \overline{K}_i = \overline{\Omega}$

(iv) intersection $\overline{K}_i \cap \overline{K}_j$, $i \neq j$,
is – either $\emptyset$
– or an edge of both triangles
– or a vertex of both triangles

✎ notation: $\overline{} \hat{=}$ a subset of $\mathbb{R}^d$ together with its boundary ("closure")

Parlance: vertices of triangles = nodes of mesh (= set $\mathcal{V}(\mathcal{M})$)
triangles of the mesh = cells or elements of mesh (= set $\mathcal{M}$)

A mesh that does not comply with the property (iv) from above.

Parlance: • $\hat{=}$ "hanging node"

▷

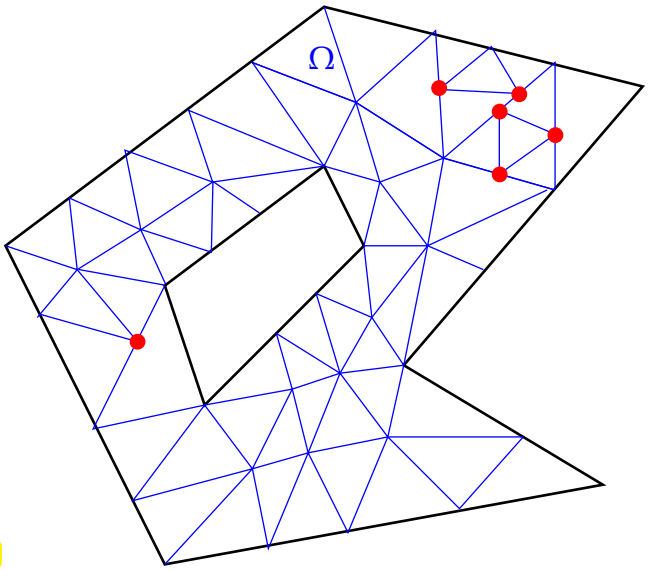


Fig. 84

(3.3.3) Data describing a (triangular) triangulation in 2D

Thanks to the constraints imposed on the triangles of a triangulation with $M \in \mathbb{N}$ triangles and N vertices, its full description requires only two matrices, see Code 3.3.4:

(I) `Coordinates` $\hat{=}$ $N \times 2$ matrix $\in \mathbb{R}^{N,2}$, i-th row containing the coordinates of the i-th vertex, $i \in \{0, \dots, N-1\}$

(II) `Elements` $\hat{=}$ $M \times 3$-matrix $\in \mathbb{N}^{M,3}$, j-th row containing the index numbers of the vertices of the j-th triangle, $j \in \{0, \dots, M-1\}$.

Note: A *local ordering*/numbering of the vertices of every triangle of the triangulation is implicitly provided by this data structure.

(Here we follow the C++ convention of numbering objects from 0.)

The following C++ class **TriaMesh2D** stores this minimal information of a planar triangular mesh. This is a rudimentary implementation; a proper object oriented design would call for many more access and

manipulation methods. For this and all other C++ codes listed in this document a **using namespace** `std;` is tacitly assumed.

C++ code 3.3.4: Class handling planar triangular mesh ➙ GITLAB

```
1  using t_TriGeo = Eigen::Matrix<double,3,2>;
2  struct TriaMesh2D
3  {
4    // Constructor: reads mesh data from file
5    TriaMesh2D(const string &filename);
6    virtual ~TriaMesh2D(void) {}
7
8    // Creates EPS rendering of mesh geometry using MathGL
9    void plotMesh(const string &epsfile,int drawvertices =0) const;
10
11   // Retrieve coordinates of vertices of a triangles as rows of a 3x2
       matrix
12   t_TriGeo getVtCoords(size_t i) const;
13
14   // Data members describing geometry and topolgy
15   Eigen::Matrix<double,Eigen::Dynamic,2> Coordinates;
16   Eigen::Matrix<int,Eigen::Dynamic,3>    Elements;
17 };
```

The constructor defined in Line 5 reads the mesh from a file with this format ▷

```
N vertices
x y
....
x y
M triangles
v1 v2 v3
....
v1 v2 v3
```

1st line: positive integer `N` followed by keyword `vertices`, `N` ≙ number of vertices.

line 2↔`N`+1: pairs `x y` of reals ≙ coordinates of vertices

line `N`+2: positive integer `M` and keyword `triangles`, `M` ≙ number of triangles.

line `N`+3↔`N`+2+`M`: triplets `v1 v2 v3` of positive integers $\in \{1,\dots,N\}$, indices of vertices of triangles.

C++11 code 3.3.5: Constructor of TriaMesh2D reading mesh from file ➙ GITLAB

```
TriaMesh2D::TriaMesh2D(const std::string &filename)
{
  std::ifstream mesh_file(filename,std::ifstream::in);
  if (!mesh_file.good()) {
    throw std::runtime_error("Cannot open mesh file! File not found");
      return;
    }
  // Read number of vertices
  int nVertices; mesh_file >> nVertices;
  char keyword[LINEMAX];
  mesh_file.getline(keyword,LINEMAX);
  if (!strcmp(keyword,"Vertices")) {
    throw std::runtime_error("Keyword 'Vertices' not found. Wrong
        file format");
    return;
```

```
  }
  // Read vertex coordinates
  Coordinates.resize(nVertices,2); int nV=0;
  while (nV<nVertices) {
    mesh_file >> Coordinates(nV,0);
    mesh_file >> Coordinates(nV,1);
    nV++;
    }
  // Read number of elements
  int nElements; mesh_file >> nElements;
  mesh_file.getline(keyword,LINEMAX);
  if (!strcmp(keyword,"Elements")) {
    throw std::runtime_error("Keyword 'Elements' not found. Wrong
        file format");
    return;
  }
  // Read vertex indices of triangles
  Elements.resize(nElements,3); int nE=0;
  while (nE<nElements) {
    mesh_file >> Elements(nE,0);
    mesh_file >> Elements(nE,1);
    mesh_file >> Elements(nE,2);
    nE++;
  }
  mesh_file.close();
}
```

The member function **getVtCoords** stores the coordinates of the three vertices of a triangle in the rows of a 3×2-matrix:

$$\begin{bmatrix} a_1^1 & a_2^1 \\ a_1^2 & a_2^2 \\ a_1^3 & a_2^3 \end{bmatrix} .$$

This format is used in the C++ code below and the fixed size matrix data type `t_triGeo` is introduced for storing triplets of triangle vertex coordinates.

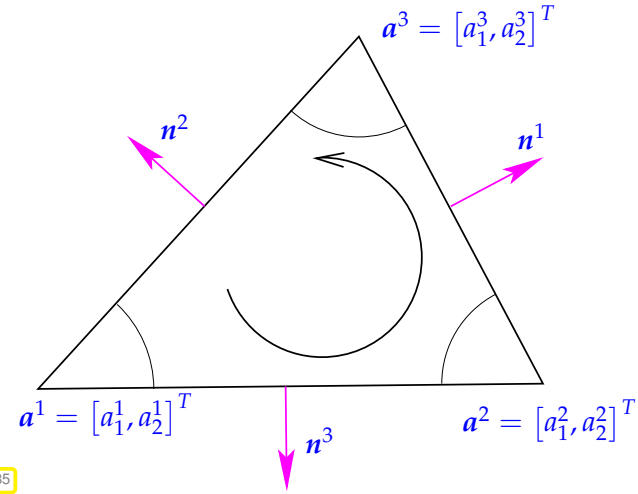


Fig. 85

C++ code 3.3.6: Retrieve coordinates of vertices of a triangles as rows of a $3x2$-matrix
➙ GITLAB

```
t_TriGeo TriaMesh2D::getVtCoords(size_t i) const {
  // Check whether valid cell index (starting from zero!)
  assert(i < Elements.rows());
  // Obtain numbers of vertices of triangle i
  const Eigen::RowVector3i idx = Elements.row(i);
  // Build matrix of vertex coordinates
```

```
  t_TriGeo vtc;
  vtc << Coordinates.row(idx[0]),
         Coordinates.row(idx[1]),
         Coordinates.row(idx[2]);
  return vtc;
}
```

Example 3.3.7 (Internal array representation of 2D triangular mesh)

We consider the triangulation of the square $\Omega =]-1,1[^2$ drawn in Fig. 86.

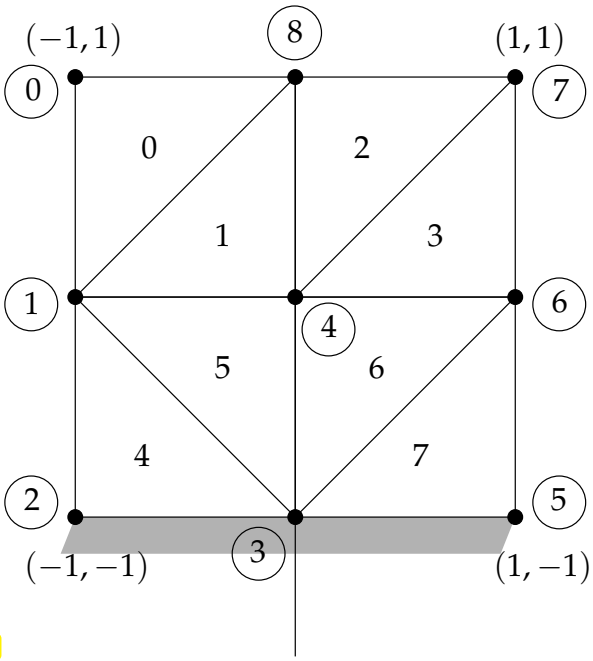


Fig. 86

i	(x_1^i, x_2^i)	
0	-1	1
1	-1	0
2	-1	-1
3	0	-1
4	0	0
5	1	-1
6	1	0
7	1	1
8	0	1

Matrix **Coordinates**

K_j	Vertex indices		
0	0	1	8
1	1	4	8
2	4	7	8
3	4	6	7
4	2	3	1
5	3	4	1
6	3	6	4
7	3	5	6

Matrix **Elements**

(Here: C++ counting, starts from 0!)

3.3.2 Linear finite element space

Recall the spline space $\mathcal{S}_1^0(\mathcal{M}) \subset H^1([a,b])$ of piecewise linear functions on a 1D grid $\mathcal{M}$ with M cells, see § 1.5.69, that was used as Galerkin trial/test space in 1D in Section 1.5.2.2.

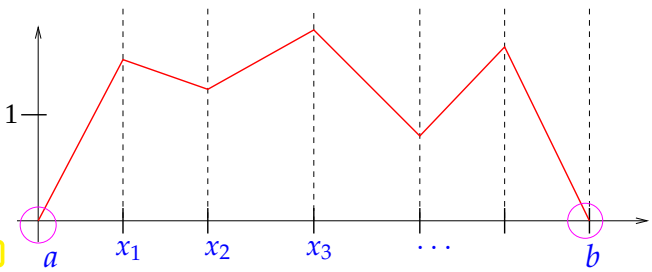


Fig. 87

$\Uparrow$ function $\in \mathcal{S}_{1,0}^0(\mathcal{M})$

1D linear finite element trial space on mesh $\mathcal{M} := \{]x_{j-1}, x_j[: j = 1, \dots, M\}$ of $\Omega :=]a,b[\subset \mathbb{R}$:

$$V_{0,N} = \mathcal{S}_{1,0}^0(\mathcal{M}) := \left\{ \begin{array}{l} v \in C^0([a,b]) : v_{|[x_{i-1},x_i]} \text{ linear,} \\ i = 1, \dots, M, v(a) = v(b) = 0 \end{array} \right\}$$

The goal is to generalize this space to 2D and 3D. To do so we first extend the concept of (affine) linear scalar-valued functions.

	$d=1$	$d=2$
Grid/mesh cells:	intervals $]x_{i-1},x_i[$, $i=1,\dots,M$	triangles K_i, $i=1,\dots,M$
Linear functions:	$x\in\mathbb{R}\mapsto\alpha+\beta\cdot x$, $\alpha,\beta\in\mathbb{R}$	$\boldsymbol{x}\in\mathbb{R}^2\mapsto\alpha+\boldsymbol{\beta}\cdot\boldsymbol{x}$, $\alpha\in\mathbb{R}$, $\boldsymbol{\beta}\in\mathbb{R}^2$

$$V_{0,N}=\mathcal{S}_1^0(\mathcal{M}):=\left\{v\in C^0(\overline{\Omega}):\forall K\in\mathcal{M}:\ \begin{array}{l} v_{|K}(\boldsymbol{x})=\alpha_K+\boldsymbol{\beta}_K\cdot\boldsymbol{x},\\ \alpha_K\in\mathbb{R},\boldsymbol{\beta}_K\in\mathbb{R}^2,\boldsymbol{x}\in K\end{array}\right\}\subset H^1(\Omega)$$

see Thm. 2.3.35

Recall that Thm. 2.3.35 tells us that a function that is piecewise (w.r.t to a "nice" partition of Ω) smooth and bounded belongs to $H^1(\Omega)$, *if and only if* it is continuous on the entire domain $\overline{\Omega}$. This accounts for the requirement $v\in C^0$ in the above definition.

Parlance: Functions of the form $\boldsymbol{x}\mapsto\alpha_K+\boldsymbol{\beta}_K\cdot\boldsymbol{x}$, $\alpha_K\in\mathbb{R},\boldsymbol{\beta}_K\in\mathbb{R}^2$ are called (affine) linear.

✎ notation: $\mathcal{S}_1^0(\mathcal{M})$

continuous functions, *cf.* $C^0(\Omega)$

$\mathcal{S}$calar functions

locally 1st degree polynomials

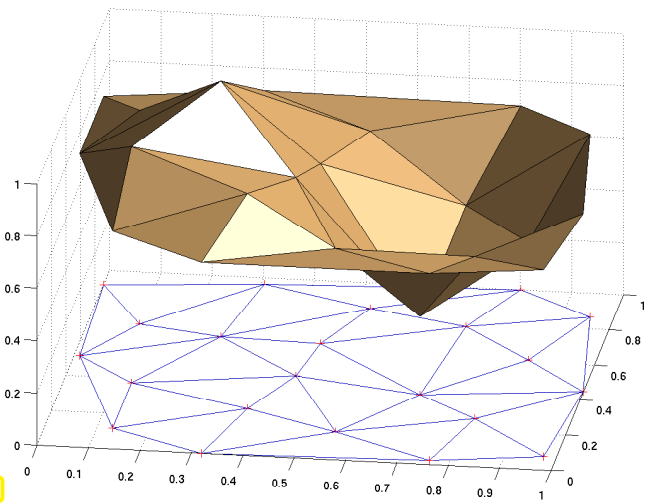


Fig. 88

◁ continuous piecewise affine linear function $\in\mathcal{S}_1^0(\mathcal{M})$ on a triangular mesh $\mathcal{M}$.

(Created with MATLAB function `trisurf`)

Remark 3.3.8 (Piecewise gradient → Section 2.3, p. 146)

Functions in $\mathcal{S}_1^0(\mathcal{M})$ will usually have kinks across intercell interfaces, which rules out global differentiability. However, we can differentiate them nevertheless:

Thm. 2.3.35 $\Rightarrow$ $\mathcal{S}_1^0(\mathcal{M})\subset H^1(\Omega)$, because $\mathcal{S}_1^0(\mathcal{M})\subset C^0(\overline{\Omega})$ and piecewise smooth.

$\Rightarrow$ for $u_N\in\mathcal{S}_1^0(\mathcal{M})$ the gradient $\mathbf{grad}\,u_N$ can be computed on each triangle as piecewise constant function, *cf.* Ex. 2.3.39.
(Easy: on $K\in\mathcal{M}$: $\mathbf{grad}(\alpha_K+\boldsymbol{\beta}_K\cdot\boldsymbol{x})=\boldsymbol{\beta}_K$)

3.3.3 Nodal basis functions

Next goal: generalization of "tent functions", see (1.5.70).

1D "tent functions" [14, Ex. 3.1.8] ▷

$$\mathfrak{B} = \{b_N^1, \dots, b_N^{M-1}\}\,, \tag{1.5.70}$$

$$b_N^j(x_i) = \delta_{ij} := \begin{cases} 1 & \text{, if } i = j\,, \\ 0 & \text{, if } i \neq j\,, \end{cases} \tag{1.5.71}$$

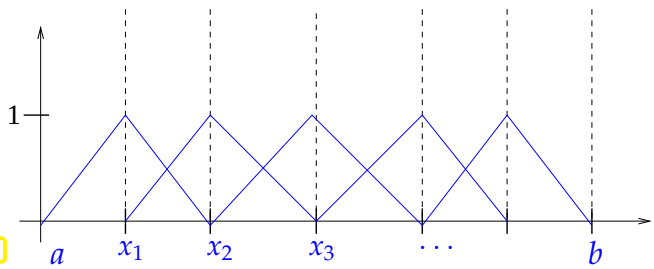


Fig. 89

The "nodal (value) property" condition (1.5.71) already *defines* a tent function in the space $\mathcal{S}_1^0(\mathcal{M})$. This approach carries over to 2D.

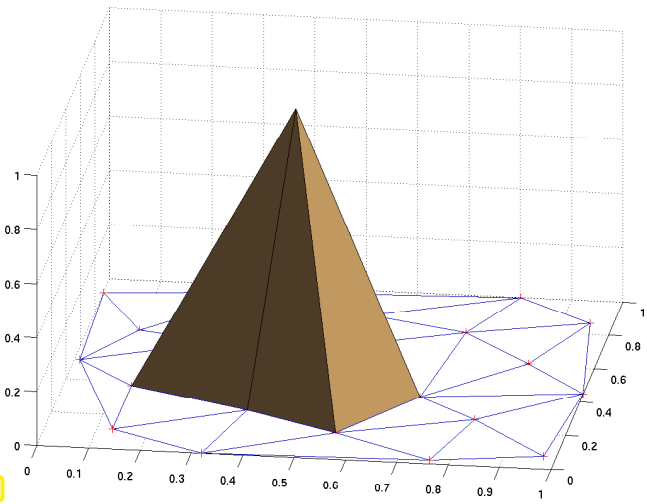
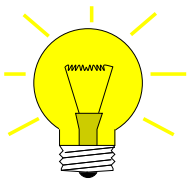


Fig. 90



Idea: *define* (**?**) basis function $b_N^{\boldsymbol{x}}$, $\boldsymbol{x} \in \mathcal{V}(\mathcal{M})$, by "nodal conditions"

$$b_N^{\boldsymbol{x}}(\boldsymbol{y}) = \begin{cases} 1 & \text{, if } \boldsymbol{y} = \boldsymbol{x}\,, \\ 0 & \text{, if } \boldsymbol{y} \in \mathcal{V}(\mathcal{M}) \setminus \{\boldsymbol{x}\}\,. \end{cases} \tag{3.3.9}$$

Is this possible ?

(3.3.10) Fixing a piecewise affine linear function

Heuristic reasoning: there is exactly one plane through three non-collinear points in $\mathbb{R}^3$. Moreover, the graph of a linear function $\mathbb{R}^2 \mapsto \mathbb{R}$ is a plane.

This can be made rigorous by a little linear algebra. Let $\boldsymbol{x} \to \alpha + \boldsymbol{\beta} \cdot \boldsymbol{x}$ describe the plane through $(\boldsymbol{a}^1, v_1)$, $(\boldsymbol{a}^2, v_2)$, $(\boldsymbol{a}^3, v_3)$, $v_i \in \mathbb{R}$, $\boldsymbol{a}^i \in \mathbb{R}^2$ not collinear. Then α, β_1, β_2 satisfy the linear system of equations

$$\begin{bmatrix} 1 & a_1^1 & a_2^1 \\ 1 & a_1^2 & a_2^2 \\ 1 & a_1^3 & a_2^3 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}\,, \tag{3.3.11}$$

where $\boldsymbol{a}^i = \begin{bmatrix} a_1^i \\ a_2^i \end{bmatrix}$. Since the points $\boldsymbol{a}^i$ do not lie on a line, the vectors $\boldsymbol{a}^2 - \boldsymbol{a}^1$ and $\boldsymbol{a}^3 - \boldsymbol{a}^1$ are linearly

independent, which ensures that (3.3.11) always has a unique solution.

➢ On a triangle K with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$: an (affine) linear scalar function $q : K \mapsto \mathbb{R}$ is uniquely determined by the values $q(\boldsymbol{a}^i)$.

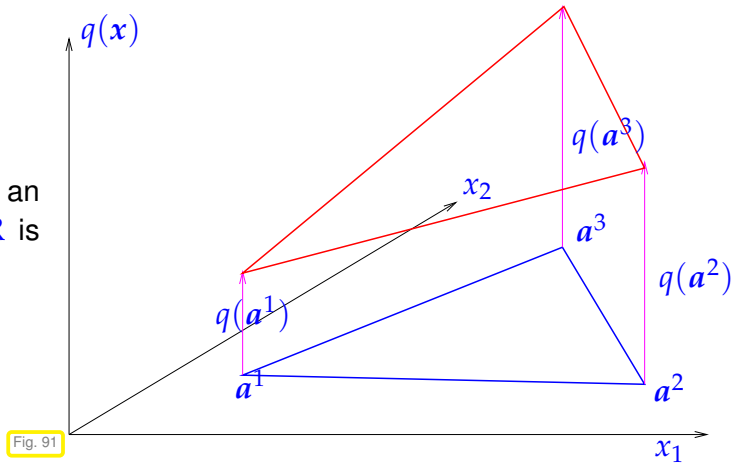


Fig. 91

Issue: Is a $\mathcal{M}$-piecewise affine linear function $v : \Omega \to \mathbb{R}$ continuous, when its vertex values are fixed?

Yes, because on each edge e of $\mathcal{M}$ $v_{|e}$ is linear and, thus, uniquely determined by its values in the endpoints of e, see Fig. 91 for an illustration. As a consequence, v has the same value on e no matter from which side it is approached.

▶ $v_N \in \mathcal{S}_1^0(\mathcal{M})$ uniquely determined by $\{v_N(\boldsymbol{x}), \boldsymbol{x}$ node of $\mathcal{M}\}$!

▶ $\boxed{\dim \mathcal{S}_1^0(\mathcal{M}) = \sharp\mathcal{V}(\mathcal{M})}$ ($\mathcal{V}(\mathcal{M})$ = set of nodes (= vertices of triangles) of $\mathcal{M}$)

Note: it is the *condition (iv)* on a valid triangulation that has made possible the construction of the basis function $b_N^{\boldsymbol{x}}$ for each $\boldsymbol{x} \in \mathcal{V}(\mathcal{M})$; no simple basis functions could be associated with the red vertices in Fig. 84.

Now we have found the perfect 2D counterpart of the tent function basis ($\to$ Fig. 37, (1.5.71)) of the linear finite element space in 1D:

Nodal basis of $\mathcal{S}_1^0(\mathcal{M})$: "tent functions"

Writing $\mathcal{V}(\mathcal{M}) = \{\boldsymbol{x}^1, \dots, \boldsymbol{x}^N\}$, the nodal basis $\mathfrak{B}_N := \{b_N^1, \dots, b_N^N\}$ of $\mathcal{S}_1^0(\mathcal{M})$ is defined by the conditions

$$b_N^i(\boldsymbol{x}^j) = \begin{cases} 1 & \text{, if } i = j\,, \\ 0 & \text{else,} \end{cases} \qquad i,j \in \{1,\dots,N\}\,. \tag{3.3.13}$$

Ordering ($\leftrightarrow$ numbering) of nodes assumed !

Piecewise linear nodal basis function
("hat function"/ "tent function")

$$u_N = \sum_{i=1}^{N} \mu_i b_N^i \in \mathcal{S}_1^0(\mathcal{M})$$

coefficient μ_j = "nodal value" of u_N at j-th node of $\mathcal{M}$

$$\boxed{u_N(\boldsymbol{x}^j) = \mu_j}$$

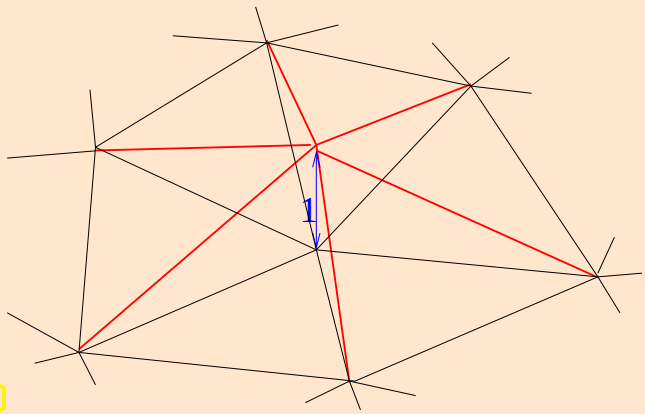


Fig. 92

(3.3.14) Linear finite element space for homogeneous Dirichlet problem

Recall that the Dirichlet problem with homogeneous boundary conditions $u_{|\partial\Omega} = 0$ is posed on the Sobolev space $H_0^1(\Omega)$ ($\rightarrow$ Def. 2.3.23), see (2.4.5), Ex. 2.9.2.

This leads to a "formal" characterization:

Galerkin space for homogeneous Dirichlet b.c.:	$V_{0,N} = \mathcal{S}_{1,0}^0(\mathcal{M}) := \mathcal{S}_1^0(\mathcal{M}) \cap H_0^1(\Omega)$

Notation: $\mathcal{S}_{1,0}^0(\mathcal{M})$ zero on $\partial\Omega$, *cf.* $H_0^1(\Omega)$

Fortunately, this space can immediately be obtained from $\mathcal{S}_1^0(\mathcal{M})$ by dropping basis functions on the boundary:

$$\boxed{\mathcal{S}_{1,0}^0(\mathcal{M}) = \mathrm{Span}\{b_N^j \colon \boldsymbol{x}^j \in \Omega \text{ (interior node !)}\}}$$

$$\dim \mathcal{S}_{1,0}^0(\mathcal{M}) = \sharp\{\boldsymbol{x} \in \mathcal{V}(\mathcal{M}) \colon \boldsymbol{x} \notin \partial\Omega\}$$

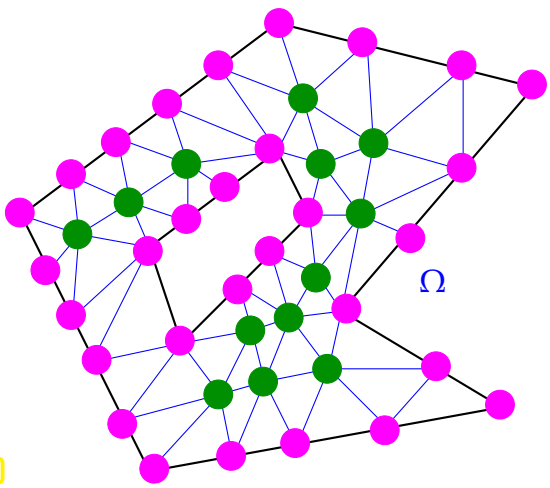


Fig. 93

◁ "Location" of nodal basis functions:
(mesh $\mathcal{M}$ $\rightarrow$ Fig. 83)

●, ● $\rightarrow$ nodal basis functions of $\mathcal{S}_1^0(\mathcal{M})$

● $\rightarrow$ nodal basis functions of $\mathcal{S}_{1,0}^0(\mathcal{M})$

Bottom line: the linear finite element trial/test space contained in $H_0^1(\Omega)$ is obtained by dropping all "tent functions" that do not vanish on $\partial\Omega$ from the basis.

3.3.4 Sparse Galerkin matrix

Already for linear finite element Galerkin discretization in one dimension in Section 1.5.2.2 the *tridiagonal* structure of the Galerkin matrices (1.5.88) caught our attention. Will Galerkin matrices in 2D also turn out to be banded?

Thus, now we study the filling pattern of the Galerkin matrix arising from the discretization of a 2nd-order scalar linear elliptic variational problem with linear finite elements. By filling pattern we mean the number and location of non-zero entries of that matrix. It will turn out that Galerkin matrices are always sparse, that is, most of their entries vanish.

(3.3.15) Model variational problem

Now: $\mathsf{a} \mathrel{\hat{=}}$ any (symmetric) bilinear form occurring in a linear 2nd-order variational problem, most general form

$$\mathsf{a}(u,v) := \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v + c(\boldsymbol{x})u\,v\,\mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} h\,v\,\mathrm{d}S\,, \quad u,v \in H^1(\Omega)\,. \tag{3.3.16}$$

$b_N^j \mathrel{\hat{=}}$ nodal basis function assciated with vertex $\boldsymbol{x}^j$ of triangulation $\mathcal{M}$ of Ω, see Section 3.3.3.

Note: a symmetric $\Rightarrow$ symmetric Galerkin matrix

Now, for the "tent" basis functions b_N^i of $\mathcal{S}_1^0(\mathcal{M})$ from (3.3.13), we study the sparsity ($\rightarrow$ [14, **??**]) of the Galerkin matrix

$$\mathbf{A} := \left(\mathsf{a}(b_N^j, b_N^i)\right)_{i,j=1}^N \in \mathbb{R}^{N,N}\,, \quad N := \dim \mathcal{S}_1^0(\mathcal{M}) = \sharp\mathcal{V}(\mathcal{M})\,,$$

as introduced in Section 3.2.

The consideration are fairly parallel to those that made us understand that the Galerkin matrix for the 1D case was *tridiagonal*, see (1.5.77). Again, a key concept is that of the support of a function as defined in Def. 1.5.76. We first examine the possible relative locations of the supports of two nodal basis functions.

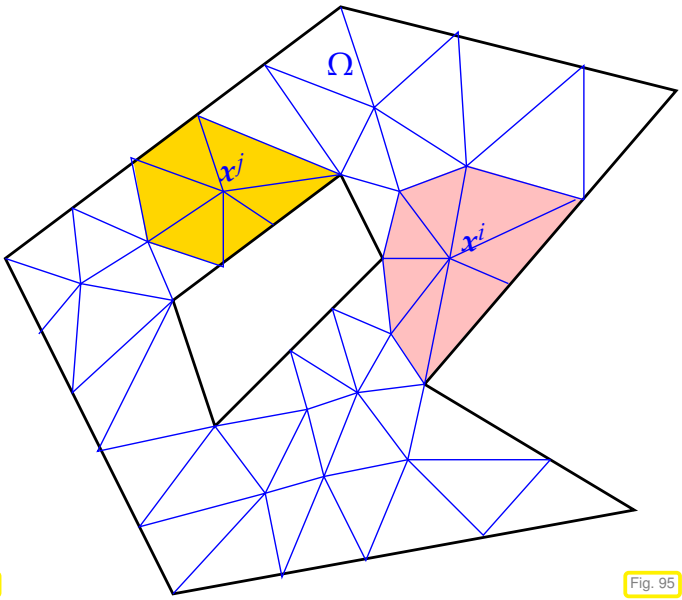


Fig. 94

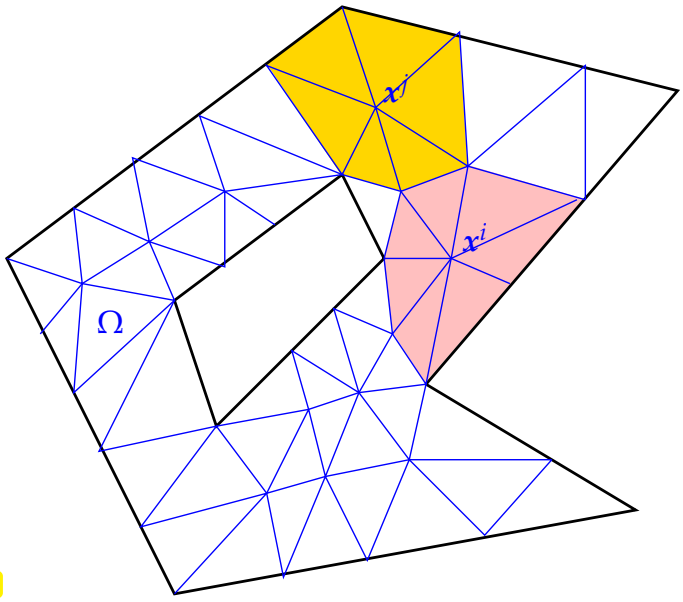


Fig. 95

$$\left\{ \begin{array}{c} \text{Nodes } \boldsymbol{x}^i, \boldsymbol{x}^j \in \mathcal{V}(\mathcal{M}) \\ \text{not connected by an edge} \end{array} \quad \Leftrightarrow \quad \mathrm{Vol}(\mathrm{supp}(b_N^i) \cap \mathrm{supp}(b_N^j)) = 0 \right\} \quad \Rightarrow \quad (\mathbf{A})_{ij} = 0 \,.$$

Lemma 3.3.17. Sparsity of Galerkin matrix

There is constant $C > 0$ depending only on the topology of Ω, that is, the number of "holes" in it, such that for any triangular mesh $\mathcal{M}$ of Ω ($N := \sharp\mathcal{V}(\mathcal{M})$ = number of vertices)

$$\sharp\{(i,j) \in \{1,\dots,N\}^2 \colon (\mathbf{A})_{ij} \neq 0\} \leq 7 \cdot N + C \,,$$

where $\mathbf{A}$ is any Galerkin matrix arising from a discretization of a 2nd-order linear scalar elliptic variational problem with linear finite elements.

Proof. We rely on Euler's formula for triangulations.

$$\sharp\mathcal{M} - \sharp\mathcal{E}(\mathcal{M}) + \sharp\mathcal{V}(\mathcal{M}) = \chi_\Omega \,, \quad \chi_\Omega = \text{Euler characteristic of } \Omega \,.$$

Note that χ_Ω is a topological invariant (alternating sum of Betti numbers).

By combinatorial considerations (traverse edges and count triangles):

$$2 \cdot \sharp\mathcal{E}_I(\mathcal{M}) + \sharp\mathcal{E}_B(\mathcal{M}) = 3 \cdot \sharp\mathcal{M} \,,$$

where $\mathcal{E}_I(\mathcal{M})$, $\mathcal{E}_B(\mathcal{M})$ stand for the sets of interior and boundary edges of $\mathcal{M}$, respectively.

$$\blacktriangleright \quad \sharp\mathcal{E}_I(\mathcal{M}) + 2\sharp\mathcal{E}_B(\mathcal{M}) = 3(\sharp\mathcal{V}(\mathcal{M}) - \chi_\Omega) \,.$$

Then use

$$N = \sharp\mathcal{V}(\mathcal{M}) \quad , \quad \mathrm{nnz}(\mathbf{A}) \leq N + 2 \cdot \sharp\mathcal{E}(\mathcal{M}) \leq 7 \cdot \sharp\mathcal{V}(\mathcal{M}) - 6\chi_\Omega \,,$$

which yields the assertion for **any** triangulation. □

Recall from [14, Notion 1.7.1] (not a definition in a rigorous mathematical sense):

Notion 3.3.18. Sparse matrix

$\mathbf{A} \in \mathbb{K}^{m,n}$, $m, n \in \mathbb{N}$, is sparse, if

$$\mathrm{nnz}(\mathbf{A}) := \#\{(i,j) \in \{1,\dots,m\} \times \{1,\dots,n\} \colon a_{ij} \neq 0\} \ll mn \,.$$

Sloppy parlance: matrix sparse :⇔ "almost all" entries $= 0$ /"only a few percent of" entries $\neq 0$

Galerkin discretization of a 2nd-order linear variational problems
utilizing the *nodal basis* of $\mathcal{S}_1^0(\mathcal{M})$/$\mathcal{S}_{1,0}^0(\mathcal{M})$
leads to *sparse* linear systems of equations.

Example 3.3.19 (Sparse Galerkin matrices)

$\mathcal{M}$ = triangular mesh, $V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M})$, homogeneous Dirichlet boundary conditions, linear 2nd-order scalar elliptic differential operator.

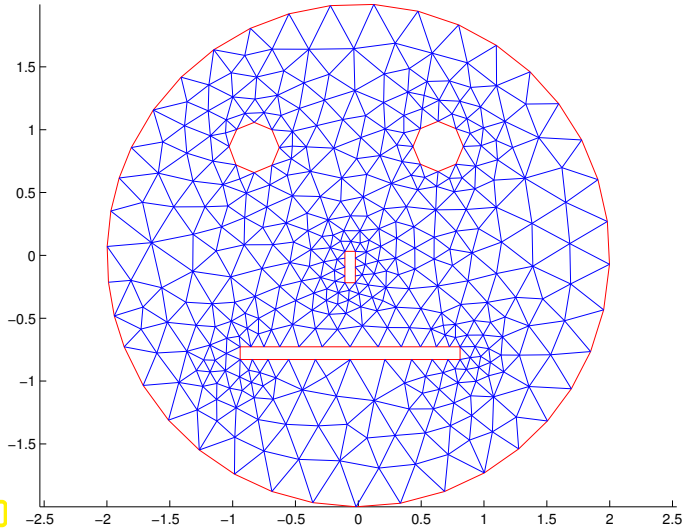


Fig. 96

Triangular mesh $\mathcal{M}$

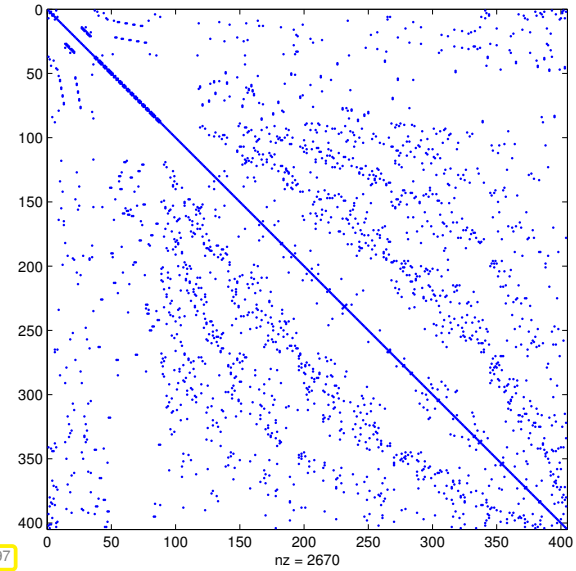


Fig. 97

Resulting sparsity pattern of Galerkin matrix

Recall: visualization of sparsity pattern by means of MATLAB's `spy`-command, by which Fig. 97 was created.

3.3.5 Computation of Galerkin matrix

Now we learn efficient algorithm for computing the non-zero entries of the sparse finite element Galerkin matrix.

(3.3.20) Model variational prolem

For sake of simplicity consider

$$\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H^1_0(\Omega)\,.$$

and Galerkin discretization based on

- triangular mesh, see Section 3.3.1, set of vertices $\{\boldsymbol{x}^i\} = \mathcal{V}(\mathcal{M})$,
- discrete trial/test space $\mathcal{S}^0_{1,0}(\mathcal{M}) \subset H^1_0(\Omega)$,
- *nodal basis* $\mathfrak{B}_N = \left\{b^j_N\right\}$ according to (3.3.9).

Entries of the Galerkin matrix $\mathbf{A}$:

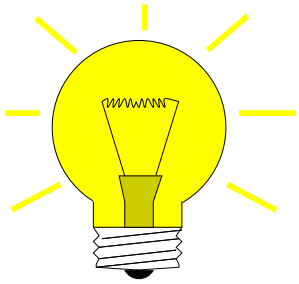
$$(\mathbf{A})_{i,j} = \mathsf{a}(b^j_N, b^i_N) = \int_\Omega \mathbf{grad}\, b^j_N \cdot \mathbf{grad}\, b^i_N \,\mathrm{d}\boldsymbol{x}$$

Section 3.3.4: when computing $(\mathbf{A})_{i,j}$ we need deal only with the situations, where $\boldsymbol{x}^i, \boldsymbol{x}^j \in \mathcal{V}(\mathcal{M})$

(i) are connected by an edge of the triangulation,

(ii) coincide,

because in all other cases the matrix entries are known to vanish a priori. We first elaborate the case (i):



Idea:

"Assembly"
(add up *cell contributions*)

$$(\mathbf{A})_{ij} = \int_{K_1} \mathbf{grad}\, b^j_{N|K_1} \cdot \mathbf{grad}\, b^i_{N|K_1}\, \mathrm{d}\boldsymbol{x} + \int_{K_2} \mathbf{grad}\, b^j_{N|K_2} \cdot \mathbf{grad}\, b^i_{N|K_2}\, \mathrm{d}\boldsymbol{x}$$

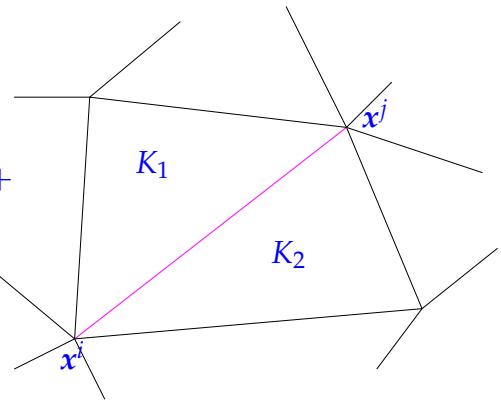


Fig. 98

Zero in on single triangle $K \in \mathcal{M}$:

$$\mathsf{a}_K(b^j_N, b^i_N) := \int_K \mathbf{grad}\, b^j_{N|K} \cdot \mathbf{grad}\, b^i_{N|K}\, \mathrm{d}\boldsymbol{x} \quad , \quad \boldsymbol{x}^i, \boldsymbol{x}^j \text{ vertices of } K\,. \tag{3.3.21}$$

Use analytic representation for $b^i_{N|K}$: if $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$ vertices of K, $\lambda_i := b^j_{N|K}$, $\boldsymbol{a}^i = \boldsymbol{x}^j$
($i \leftrightarrow$ local vertex number, $j \leftrightarrow$ global node number)

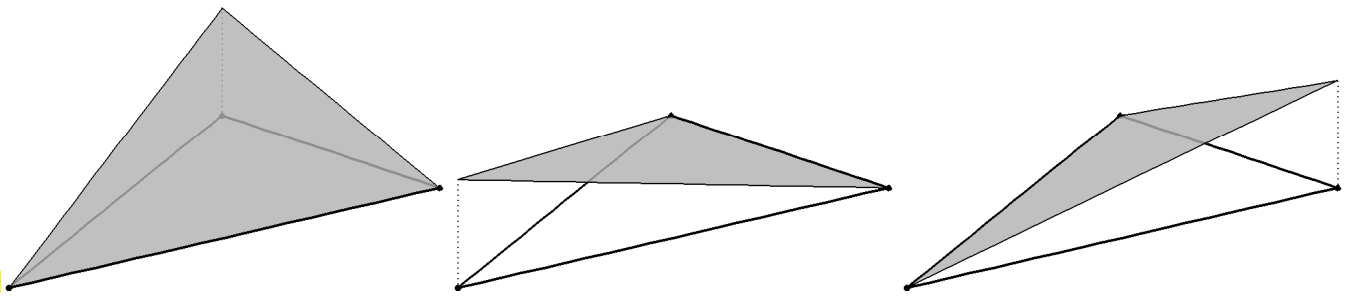


Fig. 99

Restrictions $\lambda_1, \lambda_2, \lambda_3$ of p.w. linear nodal basis functions of $\mathcal{S}^0_1(\mathcal{M})$ to triangle K

The functions $\lambda_1, \lambda_2, \lambda_3$ on the triangle K are also known as barycentric coordinate functions. They owe their name to the fact that they can be regarded as "coordinates of a point with respect to the vertices of a triangle" in the sense that

$$\boldsymbol{x} = \lambda_1(\boldsymbol{x})\boldsymbol{a}^1 + \lambda_2(\boldsymbol{x})\boldsymbol{a}^2 + \lambda_3(\boldsymbol{x})\boldsymbol{a}^3\,.$$

The attribute "barycentric" is related to barycenter = center of gravity, which has barycentric coordinates $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$.

They provide the nonzero restrictions of 2D tent functions to triangles, see Fig. 92.

$$\lambda_1(\mathbf{x}) = \frac{1}{2|K|}\left(\mathbf{x}-\boldsymbol{a}^2\right)\cdot\begin{pmatrix} a_2^2-a_2^3 \\ a_1^3-a_1^2\end{pmatrix} = -\frac{|e_1|}{2|K|}(\boldsymbol{x}-\boldsymbol{a}^2)\cdot\boldsymbol{n}^1\,,$$
$$\lambda_2(\mathbf{x}) = \frac{1}{2|K|}\left(\mathbf{x}-\boldsymbol{a}^3\right)\cdot\begin{pmatrix} a_2^3-a_2^1 \\ a_1^1-a_1^3\end{pmatrix} = -\frac{|e_2|}{2|K|}(\boldsymbol{x}-\boldsymbol{a}^3)\cdot\boldsymbol{n}^2\,,$$
$$\lambda_3(\mathbf{x}) = \frac{1}{2|K|}\left(\mathbf{x}-\boldsymbol{a}^1\right)\cdot\begin{pmatrix} a_2^1-a_2^2 \\ a_1^2-a_1^1\end{pmatrix} = -\frac{|e_3|}{2|K|}(\boldsymbol{x}-\boldsymbol{a}^1)\cdot\boldsymbol{n}^3\,.$$

(e_i = edge opposite vertex $\boldsymbol{a}^i$, see Figure for numbering scheme ▷)

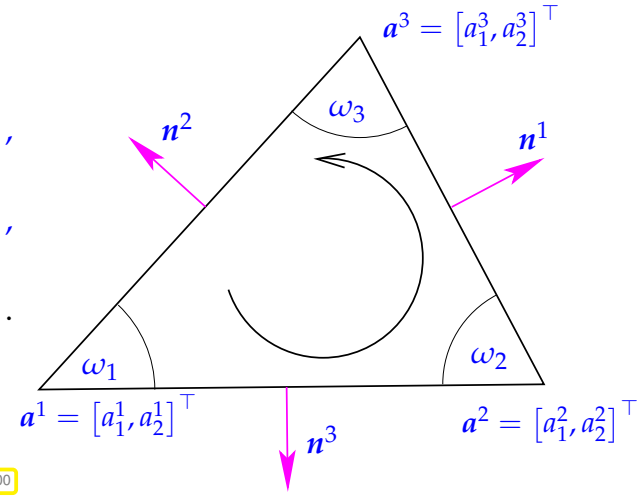


Fig. 100

From the distance formula for a point w.r.t. to a line given in Hesse normal form:

$(\boldsymbol{a}^i - \boldsymbol{a}^j)\cdot\boldsymbol{n}_i = \operatorname{dist}(\boldsymbol{a}^i; e_i) = h_i$ ($h_i \mathrel{\hat{=}}$ height) and $2|K| = |e_i|h_i \quad\Rightarrow\quad \lambda_i(\boldsymbol{a}^i) = 1$.

This shows that the λ_i really provide the restrictions of p.w. linear nodal basis functions (tent functions) of $\mathcal{S}_1^0(\mathcal{M})$ to triangle K, because they are clearly (affine) linear and comply with (3.3.9).

$$\mathbf{grad}\,\lambda_1 = -\frac{|e_1|}{2|K|}\boldsymbol{n}^1 = \frac{1}{2|K|}(\boldsymbol{a}^2-\boldsymbol{a}^3)^\perp = \frac{1}{2|K|}\begin{bmatrix} a_2^2-a_2^3 \\ a_1^3-a_1^2\end{bmatrix}\,,$$
$$\mathbf{grad}\,\lambda_2 = -\frac{|e_2|}{2|K|}\boldsymbol{n}^2 = \frac{1}{2|K|}(\boldsymbol{a}^3-\boldsymbol{a}^1)^\perp = \frac{1}{2|K|}\begin{bmatrix} a_2^3-a_2^1 \\ a_1^1-a_1^3\end{bmatrix}\,,$$
$$\mathbf{grad}\,\lambda_3 = -\frac{|e_3|}{2|K|}\boldsymbol{n}^3 = \frac{1}{2|K|}(\boldsymbol{a}^1-\boldsymbol{a}^2)^\perp = \frac{1}{2|K|}\begin{bmatrix} a_2^1-a_2^2 \\ a_1^2-a_1^1\end{bmatrix}\,.$$

Here $\boldsymbol{x}^\perp$ for $\boldsymbol{x} \in \begin{bmatrix} x_1 \\ x_2\end{bmatrix}$ indicates rotation by $\pi/2$ clockwise: $\boldsymbol{x}^\perp := \begin{bmatrix} x_2 \\ -x_1\end{bmatrix}$.

$$\mathbf{grad}\,\lambda_1\cdot\mathbf{grad}\,\lambda_2 = \frac{1}{4|K|^2}(\boldsymbol{a}^3-\boldsymbol{a}^2)\cdot(\boldsymbol{a}^1-\boldsymbol{a}^3)\,,\quad \mathbf{grad}\,\lambda_1\cdot\mathbf{grad}\,\lambda_1 = \frac{1}{4|K|^2}(\boldsymbol{a}^3-\boldsymbol{a}^2)\cdot(\boldsymbol{a}^3-\boldsymbol{a}^2)\,,$$
$$\mathbf{grad}\,\lambda_1\cdot\mathbf{grad}\,\lambda_3 = \frac{1}{4|K|^2}(\boldsymbol{a}^3-\boldsymbol{a}^2)\cdot(\boldsymbol{a}^2-\boldsymbol{a}^1)\,,\quad \mathbf{grad}\,\lambda_2\cdot\mathbf{grad}\,\lambda_2 = \frac{1}{4|K|^2}(\boldsymbol{a}^1-\boldsymbol{a}^3)\cdot(\boldsymbol{a}^1-\boldsymbol{a}^3)\,,$$
$$\mathbf{grad}\,\lambda_2\cdot\mathbf{grad}\,\lambda_3 = \frac{1}{4|K|^2}(\boldsymbol{a}^1-\boldsymbol{a}^3)\cdot(\boldsymbol{a}^2-\boldsymbol{a}^1)\,,\quad \mathbf{grad}\,\lambda_3\cdot\mathbf{grad}\,\lambda_3 = \frac{1}{4|K|^2}(\boldsymbol{a}^2-\boldsymbol{a}^1)\cdot(\boldsymbol{a}^2-\boldsymbol{a}^1)\,. \tag{3.3.22}$$

Use area formula $|K| = \frac{1}{2}|e_2||e_3|\sin\omega_1 = \frac{1}{2}|e_1||e_3|\sin\omega_2 = \frac{1}{2}|e_1||e_2|\sin\omega_3$:

element (stiffness) matrix $\mathbf{A}_K$

$$\left(\int_K \mathbf{grad}\,\lambda_i\cdot\mathbf{grad}\,\lambda_j\,\mathrm{d}\boldsymbol{x}\right)_{i,j=1}^3 = \frac{1}{2}\begin{pmatrix} \cot\omega_3+\cot\omega_2 & -\cot\omega_3 & -\cot\omega_2 \\ -\cot\omega_3 & \cot\omega_3+\cot\omega_1 & -\cot\omega_1 \\ -\cot\omega_2 & -\cot\omega_1 & \cot\omega_2+\cot\omega_1\end{pmatrix}\,. \tag{3.3.23}$$

The local numbering and naming conventions are displayed in Fig. 100.

Derivation of (3.3.23), see also [15, Lemma 3.47]: obviously, because the gradients $\mathbf{grad}\,\lambda_i$ are constant on K,

$$\mathsf{a}(\lambda_i,\lambda_j) = \int_K \mathbf{grad}\,\lambda_i \cdot \mathbf{grad}\,\lambda_j \,\mathrm{d}\boldsymbol{x} = \frac{1}{4|K|}|e_i||e_j|\,\boldsymbol{n}_i \cdot \boldsymbol{n}_j \,.$$

Then use:

- $\boldsymbol{n}_i \cdot \boldsymbol{n}_j = \cos(\pi - \omega_k) = -\cos\omega_k, \quad (i \neq j)$
- $|K| = \frac{1}{2}|e_i||e_j|\sin\omega_k, \quad (i \neq j)$.

Case $i = j$ employs a trick:

$$\boxed{\sum_{i=1}^{3} \lambda_i = 1 \quad \Rightarrow \quad \sum_{i=1}^{3} \mathsf{a}_K(\lambda_i,\lambda_j) = 0} \quad \Rightarrow \quad \mathsf{a}(\lambda_i,\lambda_i) = -\sum_{j\neq i} \mathsf{a}(\lambda_i,\lambda_j)\,.$$

Remark 3.3.24 (Alternative computation of element matrix for $-\Delta$)

From (3.3.11) we conclude that the coefficients in the representation $\lambda_i(\boldsymbol{x}) = \alpha_i + \boldsymbol{\beta}^i \cdot \boldsymbol{x}$ of the barycentric coordinate functions $\lambda_1, \lambda_2, \lambda_3$ on a triangle with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$ satisfy

$$\begin{bmatrix} 1 & a_1^1 & a_2^1 \\ 1 & a_1^2 & a_2^2 \\ 1 & a_1^3 & a_2^3 \end{bmatrix} \begin{bmatrix} \alpha_1 & \alpha_2 & \alpha_3 \\ \beta_1^1 & \beta_1^2 & \beta_1^3 \\ \beta_2^1 & \beta_2^2 & \beta_2^3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} . \tag{3.3.25}$$

Observe that $\mathbf{grad}\,\lambda_i = \boldsymbol{\beta}^i$, which explains, why Code 3.3.26 computes the gradients of the barycentric coordinate functions:

C++ code 3.3.26: Computation of gradients of barycentric coordinate functions on a triangle
➔ GITLAB

```
Eigen::Matrix<double, 2, 3> gradbarycoordinates(const t_TriGeo&
    Vertices)
{
  // Argument Vertices passes the vertex positions of the triangle
  // as the rows of a 3×2-matrix, , see Code 3.3.6..
  // The function returns the components of the gradients as the
  // columns of a 2×3-matrix

  // Computation based on (3.3.25), solving for the
  // coefficients of the barycentric coordinate functions.
  Eigen::Matrix<double, 3, 3> X; // Temporary matrix
  X.block<3, 1>(0, 0) = Eigen::Vector3d::Ones();
  X.block<3, 2>(0, 1) = Vertices;
  return X.inverse().block<2, 3>(1, 0);
}
```

This suggests an efficient way to compute the element matrix $\mathbf{A}_K$ given in (3.3.23). That formula should not be implemented, because computing the angles and, subsequently, their $\cot$ is very expensive.

C++ code 3.3.27: Computation of element matrix for $-\Delta$ on a triangle and for linear Lagrangian finite elements ➙ GITLAB

```
Eigen::Matrix3d ElementMatrix_Lapl_LFE(const t_TriGeo& V)
{
  // Argument V same as Vertices in Code 3.3.26.
  // The function returns the 3×3 element matrix as a fixed size
  // EIGEN matrix.

  // Evaluate (3.3.21), exploiting that the gradients are constant.
  // First compute the area of triangle by determinant formula
  const double area = 0.5*std::abs(
    (V(1,0)-V(0,0))*(V(2,1)-V(1,1))-(V(2,0)-V(1,0))*(V(1,1)-V(0,1)));
  // Get gradients of barycentric coordinate functions, see Code 3.3.26
  const Eigen::Matrix<double,2,3> X = gradbarycoordinates(V);
  // Compute inner products of gradients through matrix multiplication
  return area*X.transpose()*X;
}
```

Remark 3.3.28 (Scaling of entries of element matrix for $-\Delta$)

When we scale a mesh, we subject all cells to a uniform dilation. Let us elaborate, how entries of the Galerkin matrix change in the process.

An observation:

(3.3.23) ➢ $\mathbf{A}_K$ does not depend on the "size" of triangle K!
(more precisely, element matrices are equal for *similar* triangles)

This can be seen by the following reasoning:

- Obviously translation and rotation of K does not change. $\mathbf{A}_K$
- *Scaling* of K by a factor $\rho > 0$ has the following effect that
  - the area $|K|$ is scaled by ρ^2,
  - the gradients $\mathbf{grad}\,\lambda_i$ are scaled by ρ^{-1} (the barycentric coordinate functions λ_i become steeper when the triangle shrinks in size.).

  Both effects just offset in a_K from (3.3.21) such that $\mathbf{A}_K$ remains invariant under scaling.

Note, however: it is different in 3D (what is the scaling there?)

Now we tackle the computation of the big Galerkin matrix. This so-called "assembly" of $(\mathbf{A})_{ij}$ starts from the sum

$$(\mathbf{A})_{ij} = \int_{K_1} \mathbf{grad}\, b^j_{N|K_1} \cdot \mathbf{grad}\, b^i_{N|K_1}\,\mathrm{d}\boldsymbol{x} + \int_{K_2} \mathbf{grad}\, b^j_{N|K_2} \cdot \mathbf{grad}\, b^i_{N|K_2}\,\mathrm{d}\boldsymbol{x}\,.$$

➢ $(\mathbf{A})_{ij}$ can be obtained by summing respective$^{(*)}$ entries of the elements matrices of the elements adjacent to the edge connecting $\boldsymbol{x}^i$ and $\boldsymbol{x}^j$

$(*)$: watch correspondence of local and global vertex numbers !

When we use (3.3.23), we origin of the matrix entry $(\mathbf{A})_{ij}$, $i \neq j$, can be visualized as follows:

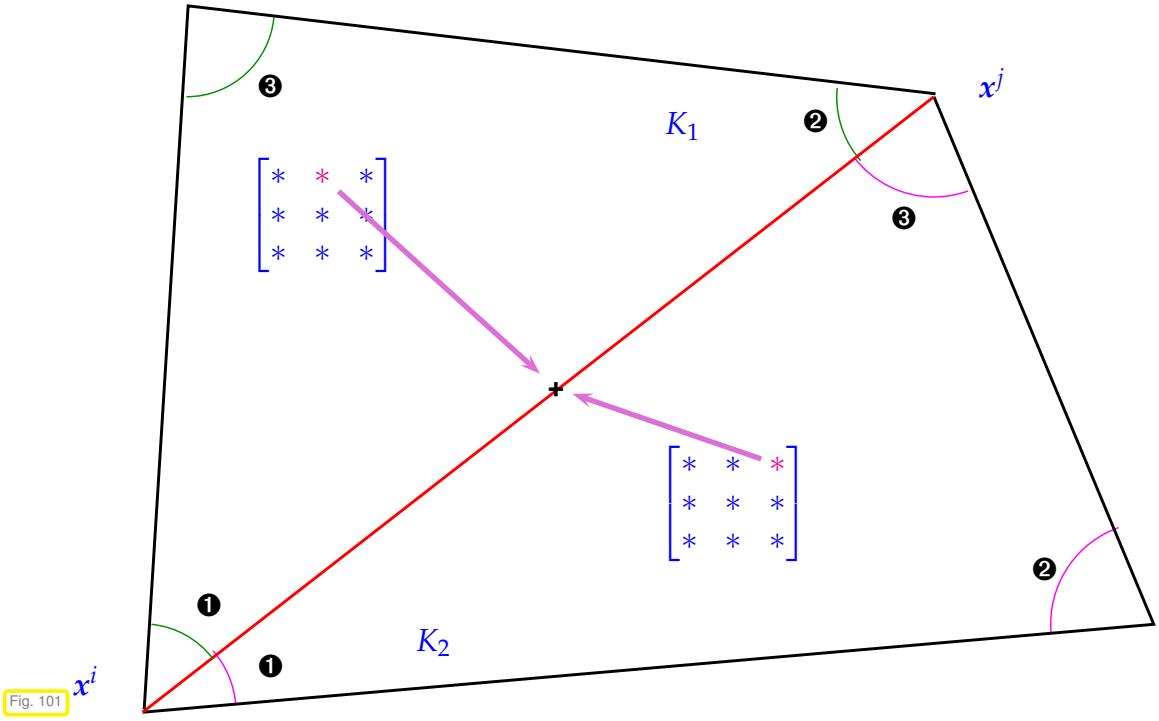


Fig. 101

$(\mathbf{A})_{ij}$ by summing entries of two element matrices

Next we look at the "assembly" of the diagonal entry $(\mathbf{A})_{ii}$ of the Galerkin matrix $\mathbf{A}$. It can be obtained by summing corresponding diagonal entries of element matrices belonging to triangles adjacent to node $\boldsymbol{x}^i$.

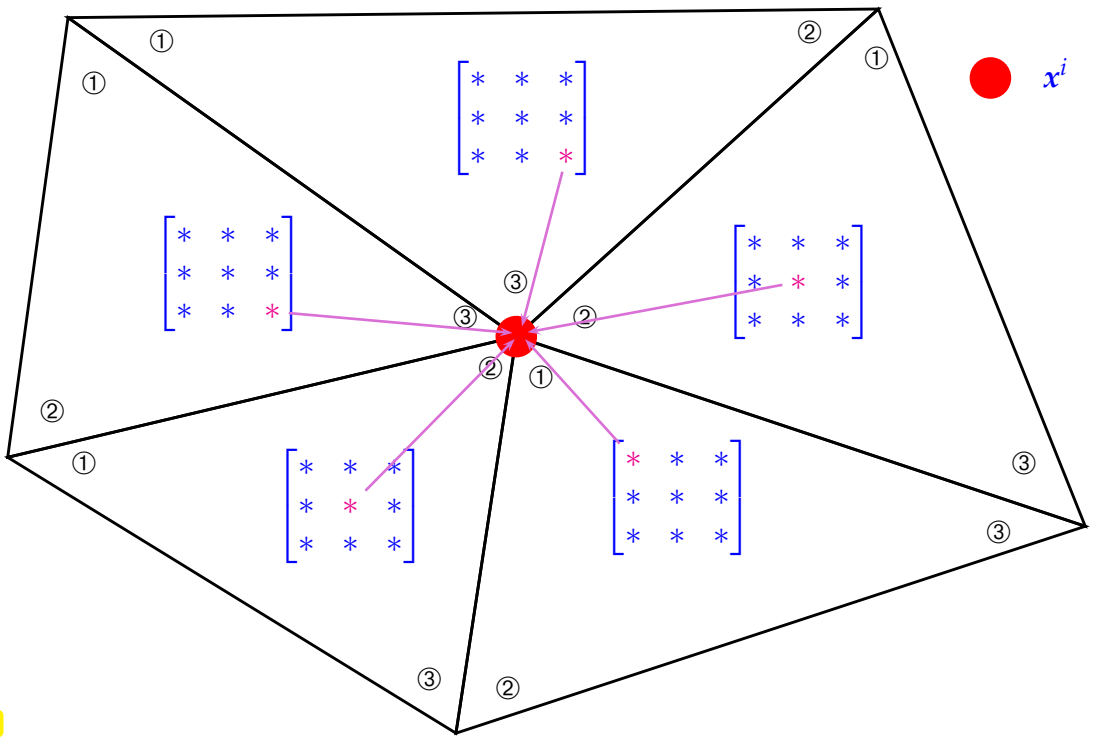


Fig. 102

$(\mathbf{A})_{ii}$ by summing diagonal entries of element matrices of adjacent triangles

(3.3.29) Assembly algorithm for linear Lagrangian finite elements

Assume:
- numbering of nodal basis functions $\leftrightarrow$ numbering of mesh vertices $\in \mathcal{V}(\mathcal{M})$
- numbering of triangles (cells) of mesh $\mathcal{M} = \{K_1, \dots, K_M\}$, $M := \sharp\mathcal{M}$
- (local) numbering of the vertices of every triangle $K \in \mathcal{M}$

The adding up of entries of element matrices illustrated in Fig. 101 and Fig. 102 might suggest the following implementation (pseudo-code) of the "*collect approach*" visualized in Fig. 101 and Fig. 102.

Pseudocode 3.3.30: Vertex-centered assembly of Galerkin matrix for linear finite elements

foreach $e \in \mathcal{E}(\mathcal{M})$ (✎ notation: $\mathcal{E}(\mathcal{M}) \hat{=}$ set of edges of $\mathcal{M}$)
  $(i,j) \hat{=}$ vertex numbers of endpoints of e
  $(\mathbf{A})_{i,j} \leftarrow 0$, $(\mathbf{A})_{j,i} \leftarrow 0$,
  foreach triangle K adjacent to e
    find local numbers $l, m \in \{1,2,3\}$ of endpoints of e
    $(\mathbf{A})_{i,j} \leftarrow (\mathbf{A})_{i,j} + (\mathbf{A}_K)_{l,m}$ $\rightarrow$ Fig. 101, $\mathbf{A}_K$ from (3.3.23)
    $(\mathbf{A})_{j,i} \leftarrow (\mathbf{A})_{j,i} + (\mathbf{A}_K)_{m,l}$ $\rightarrow$ Fig. 101, $\mathbf{A}_K$ from (3.3.23)
  endfor
endfor
foreach $v \in \mathcal{V}(\mathcal{M})$
  $j \hat{=}$ number of vertex v
  $(\mathbf{A})_{j,j} \leftarrow 0$
  foreach triangle K adjacent to v
    $l \hat{=}$ local number of v in K
    $(\mathbf{A})_{j,j} \leftarrow (\mathbf{A})_{j,j} + (\mathbf{A}_K)_{l,l}$ $\rightarrow$ Fig. 102, $\mathbf{A}_K$ from (3.3.23)
  endfor
endfor

This algorithm will strain the capabilities of the simple data structures available in a mesh object of type **TriaMesh2D**, because it requires information about the edges of the mesh. There is a dual way of organizing assembly, which needs only the basic topology and geometry information stored in **TriaMesh2D**, see Code 3.3.4.

Cell oriented assembly

However, a much simpler implementation can be achieved by adopting the perspective of *cell oriented assembly* ("collect scheme"): instead of traversing edges and vertices as in the above algorithm and collecting entries of element matrices of adjacent triangles, we loop over all triangles and *distribute* entries of their element matrices to their vertices and edges.

(3.3.32) Index mapping for linear finite elements on triangular mesh

Invariably, cell oriented assembly entails knowing the global number of the basis functions associated with the vertices of each triangle. This information must be provided in an easily accessible form:

Data structure: $\texttt{dofh} \in \mathbb{N}^{\sharp\mathcal{M},3}$: local$\rightarrow$global index mapping array: "d.o.f. mapper"

$$\texttt{dofh}(k,l) = \text{global number of vertex } l \text{ of } k\text{-th cell}$$
$$\boldsymbol{x}^{\texttt{dofh}(k,l)} = \boldsymbol{a}^l \quad \text{when} \quad \boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3 \text{ are the vertices of } K_k\,, \tag{3.3.33}$$

for $l \in \{1,2,3\}$, $k \in \{1,\dots,N\}$, $N = \sharp\mathcal{V}(\mathcal{M})$ ("mathematical indexing"!).

We assume that the triangulation is encoded in the data members `Coordinates` $\in \mathbb{R}^{N,2}$ and `Elements` $\in \mathbb{N}^{M,3}$ of an object `Mesh` of type **TriaMesh2D** as explained in § 3.3.3.

➢ simple realization of index mapping: `dofh(k,l) := Mesh.Elements(k-1,l-1)`

(C++ indexing used for accessing entries of EIGEN matrices!)

The use of index mapping in the context of assembly of a finite element Galerkin matrix will be discussed in more generality and detail in Section 3.6.4.

Example 3.3.34 (Index mapping by d.o.f. mapper)

Fig. 103 displays a small planar triangulation, complete with all *local and global* index numbers of the vertices and the index numbers of the triangles. On the right the corresponding `dofh`-array complying with (3.3.33) is displayed ("mathematical indexing").

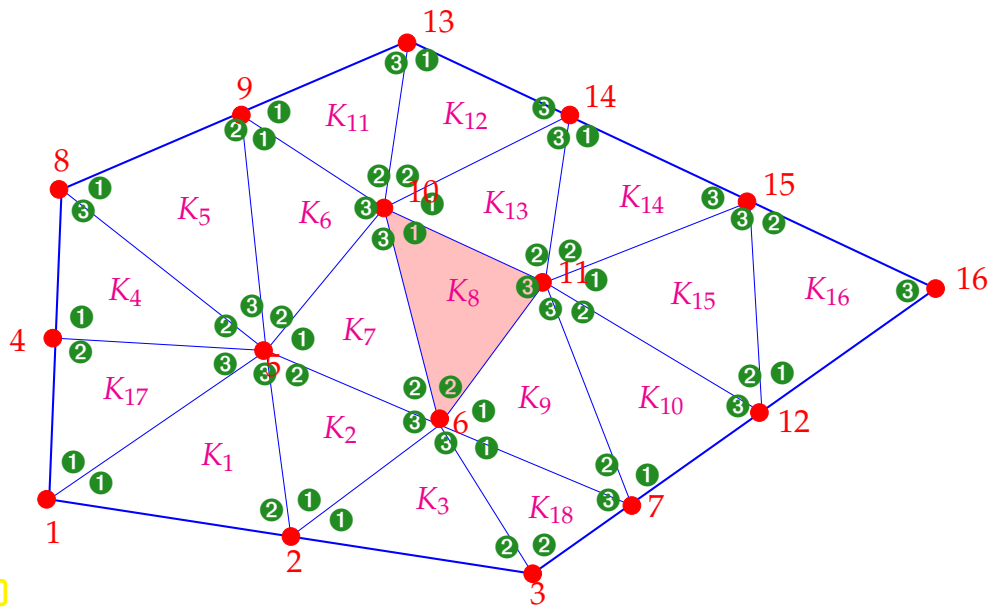


Fig. 103

`dofh`:

K_1	1	2	5
K_2	2	5	6
K_3	2	3	6
K_4	4	5	8
K_5	8	9	5
K_6	9	5	10
K_7	5	6	10
K_8	10	6	11
K_9	6	7	11
K_{10}	7	11	12
K_{11}	9	10	13
K_{12}	13	10	14
K_{13}	10	11	14
K_{14}	14	11	15
K_{15}	11	12	15
K_{16}	12	15	16
K_{17}	1	4	5
K_{18}	6	3	7

In Fig. 103, for cell K_8: element matrix $\mathbf{A}_K$ contributes to `A([10 6 11],[10 6 11])`

The algorithmic details of cell-oriented assembly are remarkably simple and illustrated by the following pseudocode. It takes for granted information about a triangular mesh to be available in an object named `mesh` of a type similar to **TriaMesh2D**, see § 3.3.3 and, in particular, Code 3.3.4.

- $\leftrightarrow$ edges, to which we can formally associate off-diagonal entires of the Galerkin matrix. $\triangleright$
- $\leftrightarrow$ vertices, carrying diagonal entries of the Galerkin matrix, local numbering given. $\triangleright$

$\rightarrow$ $\hat{=}$ "contributes to"

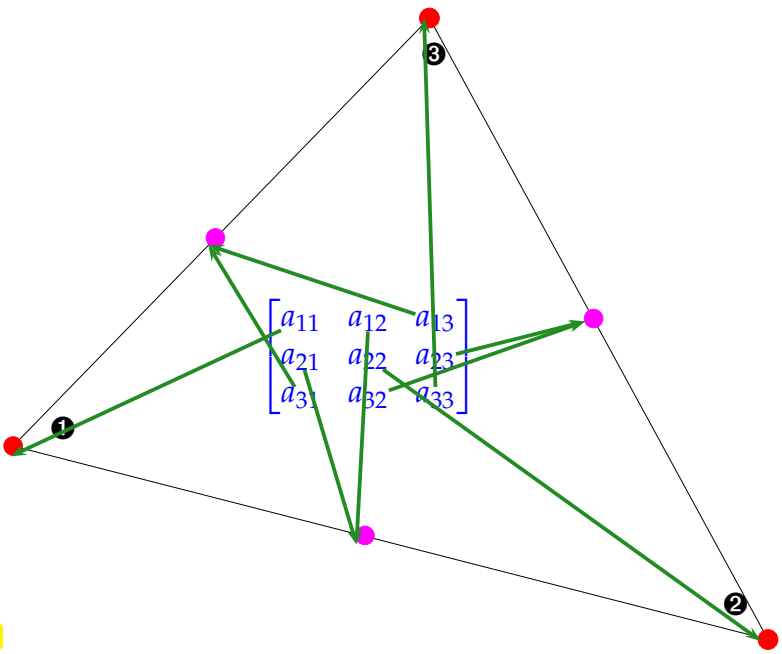


Fig. 104

Pseudocode 3.3.35: Assembly of finite element Galerkin matrix for linear finite elements

SparseMatrix $\mathbf{A} \in \mathbb{R}^{N,N}$, $N := \sharp\mathcal{V}(\mathcal{M})$ (number of vertices)
$\mathbf{A} := \mathbf{O}$;
for $i = 1$ `to` N **do**
  $K \leftarrow$ `mesh.`**getVtCoords**(i)
  $\mathbf{A}_K \leftarrow$ **getElementMatrix**(K);
  $\mathbf{A}(\texttt{dofh(i,:)},\texttt{dofh(i.:)}) += \mathbf{A}_K$;
endfor

We have resorted to MATLAB syntax to express the filling of the matrix $\mathbf{A}$ in a compact manner: $\mathbf{A}(\texttt{dofh(i,:)},\texttt{dofh(i.:)})$ is the 3×3 submatrix of $\mathbf{A}$ corresponding to the vertices of triangle $\sharp i$.

Note: Homogeneous Dirichlet boundary conditions are not taken into account in Code 3.3.35

Code 3.3.35 demonstrates a fundamental paradigm in the implementation of finite element Galerkin schemes for variational problems connected with partial differential equations: loops generally run over the mesh cells and, if possible, computations are carried out on the level of the mesh cells, which usually, results in *optimal ($*$) computational effort*. For Code 3.3.35 this means the following.

$$\boxed{\text{Computational effort} \quad = O(\sharp\mathcal{M})}$$

($*$): computational cost for assembly that is linearly proportional to the number of nonzero entries of the Galerkin matrix is considered optimal.

A concrete C++ implementation of Code 3.3.35 is given next. The function argument `Mesh` refers to an object of **TriaMesh2D** describing the triangulation in the form of the `Coordinates` and `Elements` matrices according to § 3.3.3. The parameter `getElementMatrix` must contain a function that expects a 3×2-matrix of vertex coordinates and returns a 3×3 element matrix. The function returns a sparse $N \times N$-matrix, where $N = \sharp\mathcal{V}(\mathcal{M})$ is the number of vertices of the mesh.

C++ code 3.3.36: Cell-oriented assembly of Galerkin matrix for linear finite elements on a triangular mesh ↪ GITLAB

```
1  // Functor referencing a function for the computation of the element
2  // matrices like ElementMatrix_Lapl_LFE from Code 3.3.27.
3  typedef function<Eigen::Matrix3d(const t_TriGeo &)>
       LocalMatrixHandle_t;
4
5  Eigen::SparseMatrix<double> assembleGalMatLFE(
6    const TriaMesh2D& Mesh,
7    const LocalMatrixHandle_t& getElementMatrix) {
8      // Fetch the number of vertices
9      int N = Mesh.Coordinates.rows();
10     // Fetch the number of elements/cells, see § 3.3.3
11     int M = Mesh.Elements.rows();
12     //create empty sparse Galerkin matrix A
13     Eigen::SparseMatrix<double> A(N,N);
14     // Loop over elements and "distribute" local contributions
15     for (int i = 0; i < M; i++) {
16         //get local→global index mapping for current element, cf.
               (3.3.33)
17         Eigen::Vector3i dofhk = Mesh.Elements.row(i);
18         t_TriGeo Vertices;
19         //extract vertices of current element, see § 3.3.3
20         for (int j = 0; j < 3; j++)
21           Vertices.row(j) = Mesh.Coordinates.row(dofhk(j));
22         // Compute 3×3 element matrix A_K
23         Eigen::Matrix3d Ak = getElementMatrix(Vertices);
24         // Add local contribution to Galerkin matrix
25         for (int j = 0; j < 3; j++)
26           for (int k = 0; k < 3; k++)
27             A.coeffRef(dofhk(j),dofhk(k)) += Ak(j, k);
28     }
29     A.makeCompressed();
30     return A;
31 }
```

! Regard Code 3.3.36 as "C++ pseudo-code": in actual implementation `A` must be initialized differently (→ Rem. 3.3.37), because random Lvalue access to entries of a sparse matrix in CRS format in Line 27 might be inefficient.

Remark 3.3.37 (Efficient assembly of sparse Galerkin matrices (in MATLAB))

Entry-by-entry initialization of a sparse matrix as in Code 3.3.35 involves huge hidden effort for moving data in memory, because sparse matrices are usually stored in CRS/CCS format, which exploits knowledge about vanishing matrix entries. An more detailed presentation is given in [14, Section 1.7.3] and [12].

More efficient initialization can be achieved by using an intermediate triplet/coordinate list (COO) format, see [14, § 1.7.6]. first store the $N \times N$ matrix as a vector of triplets (i, j, a_{ij}), $i, j \in \{1, \dots, N\}$, which allows adding entries with little effort, and finally compute the more economical CRS/CCS format. How to

do it in EIGEN is explained in [14, Section 1.7.3]. Triplet initialization is used in the following assembly code Code 3.3.38, which is an algebraically equivalent implementation of the function **assembleGalMatLFE** from Code 3.3.36.

C++ code 3.3.38: Efficient assembly of Galerkin matrix for linear finite elements on a triangular mesh ➙ GITLAB

```
Eigen::SparseMatrix<double> assembleGalMatLFE(
    const TriaMesh2D& Mesh,
    const LocalMatrixHandle_t& getElementMatrix) {
     //obtain the number of vertices
     int N = Mesh.Coordinates.rows();
     //obtain the number of elements/cells
     int M = Mesh.Elements.rows();
     vector<Eigen::Triplet<double> > triplets;
     //loop over elements and add local contributions

         for (int i = 0; i < M; i++) {
         //get local→global index mapping for current element, cf. (3.3.33)
         Eigen::Vector3i element = Mesh.Elements.row(i);
         t_TriGeo Vertices;
         //extract vertices of current element, see § 3.3.3
         for (int j = 0; j < 3; j++) {
             Vertices.row(j) = Mesh.Coordinates.row(element(j));
         }
         //compute element contributions
         Eigen::Matrix3d Ak = getElementMatrix(Vertices);
         //build triplets from contributions
         for (int j = 0; j < 3; j++) {
             for (int k = 0; k < 3; k++) {
                 triplets.push_back({element(j), element(k), Ak(j, k)});
             }
         }
     }
     //build sparse matrix from triplets
     Eigen::SparseMatrix<double> A(N, N);
     A.setFromTriplets(triplets.begin(), triplets.end());
     A.makeCompressed();
     return A;
}
```

As demonstrated in [11], even an utterly loop-free implementation is possible!

Example 3.3.39 (Impact of efficient initialization of sparse Galerkin matrix)

Code 3.3.35 is algebraically equivalent to Code 3.3.38, but much slower.

Comparison of runtimes of assembly of Galerkin matrices for $-\Delta$ (bilinear form from § 3.3.20) on triangular meshes with different numbers of elements.

Computation of element matrices by Code 3.3.38 and Code 3.3.36, timing by C++ `sys/time.h` routines, minimal time over 10 runs,

(OS: Linux Fedora 22, CPU: AMD Opteron 6174, Compiler: c++, optimization flag -O3.)

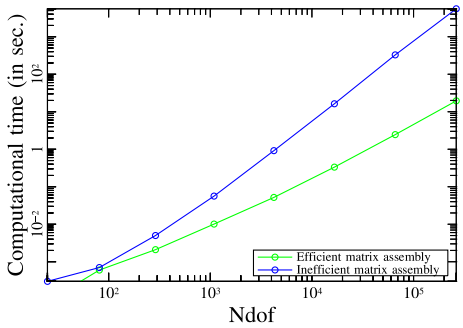


Fig. 105

We observe that for large matrices the triplet based initialization is significantly faster.

3.3.6 Computation of right hand side vector

(3.3.40) Model right hand side linear form

We consider the linear form (right hand side of linear variational problem), see (2.4.5), (3.1.4):

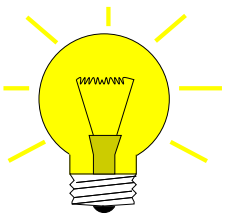
$$\ell(v) := \int_\Omega f(\boldsymbol{x})\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad v \in H^1(\Omega)\,, \quad f \in L^2(\Omega)\,.$$

Recall formula for right hand side vector, $N = \dim V_{N,0}$, $\mathfrak{B} = \{b_N^1, \dots, b_N^N\} \mathrel{\hat{=}}$ tent function basis, see (3.3.13),

$$(\vec{\boldsymbol{\varphi}})_j = \ell(b_N^j) = \int_\Omega f(\boldsymbol{x})\, b_N^j(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,, \quad j = 1, \dots, N\,. \tag{3.3.41}$$

Considerations parallel to Section 3.3.5: splitting of right hand side linear form into cell contributions, *cf.* (3.3.21), page 199, for similar approach to the bilinear form a.

Idea: "Assembly"



$$(\vec{\boldsymbol{\varphi}})_j = \sum_{l=1}^{N_j} \int_{K_l} f(\boldsymbol{x})\, b_{N\,|K_l}^j(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}\,,$$

where $K_1, \dots, K_{N_j} \mathrel{\hat{=}}$ triangles adjacent to node $\boldsymbol{x}^j$.

(Integration confined to $\operatorname{supp}(b_N^j)$!)

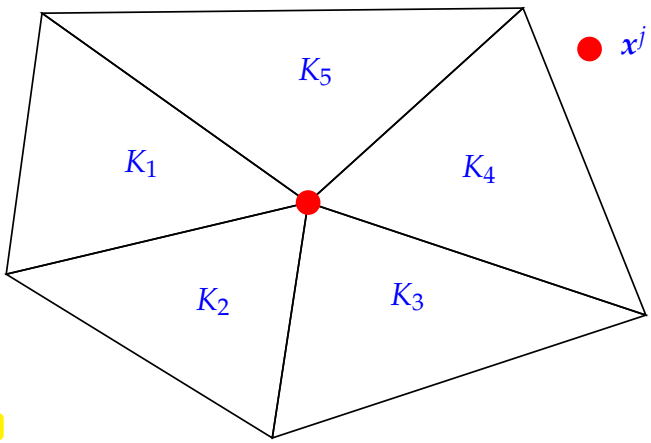


Fig. 106

Zero in on single triangle $K \in \mathcal{M}$:

$$\ell_K(b_N^j) := \int_K f(\boldsymbol{x})\, b_{N\,|K}^j(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \ell_K(\lambda_i)\,, \quad \boldsymbol{x}^j \text{ vertex of } K\,, \tag{3.3.42}$$

where λ_i is the barycentric coordinate function associated with (local) vertex i of the triangle and $j = \texttt{dofh}(k,i)$, with k the (global) number of the triangle K and $\texttt{dofh}$ defined in (3.3.33) on page 204. Recall that in this case $b^j_{N\,|K} = \lambda_i$.

As above in Fig. 102: Entries of the right hand side vector can be obtained by summing up the values that the localized right hand side functionals ℓ_K return for barycentric coordinate functions: This can be expressed through the vertex-oriented formula ("collect scheme")

$$(\vec{\boldsymbol{\varphi}})_j = \sum_{K,i:\texttt{dofh}(k,i)=j} \ell_K(\lambda_i)\,. \tag{3.3.43}$$

Here: $k \leftrightarrow$ global index of triangle K

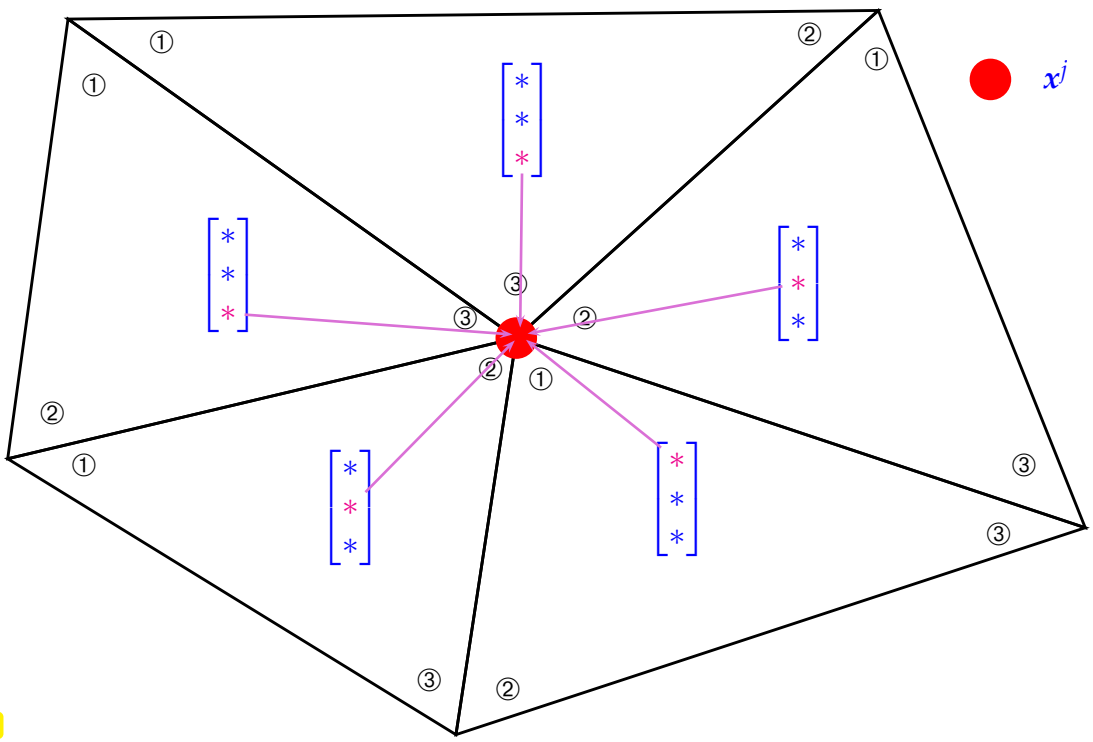


Fig. 107

However, implementation according to this formula would emulate the cumbersome algorithm on page 204 for the computation of the Galerkin matrix.

As in Section 3.3.5, § 3.3.29, we aim to compute $\vec{\boldsymbol{\varphi}}$ in a cell-oriented fashion ("distribute scheme"), as in Code 3.3.35.

To that end we need a counterpart of the element (stiffness) matrix from (3.3.23), the

$$\text{element (load) vector}: \qquad \vec{\boldsymbol{\varphi}}_K := (\ell_K(\lambda_i))_{i=1}^3 \in \mathbb{R}^3\,, \tag{3.3.44}$$

which is obtained by plugging the restrictions of basis functions to an element into that part of the right hand side linear form belonging to the element.

Cell-oriented "assembly" of $(\vec{\varphi})_j$ by summing up contributions from element vectors of triangles adjacent to $\boldsymbol{x}^j$ ($N_j \hat{=}$ no. of triangles abutting $\boldsymbol{x}^j$)

$$(\vec{\varphi})_j = \sum_{l=1}^{N_j} \ell_{K_l}(b^j_{N\,|K_l}) = \sum_{l=1}^{N_j} (\vec{\varphi}_K)_{i(l,j)}\,, \qquad (3.3.45)$$

where $i(l,j)$ is the local vertex index of the node $\boldsymbol{x}^j$ (global index j) in the triangle K_l.
Note: with index array `dofh` from § 3.3.29:

$$\texttt{dofh}(l, i(l,j)) = j$$

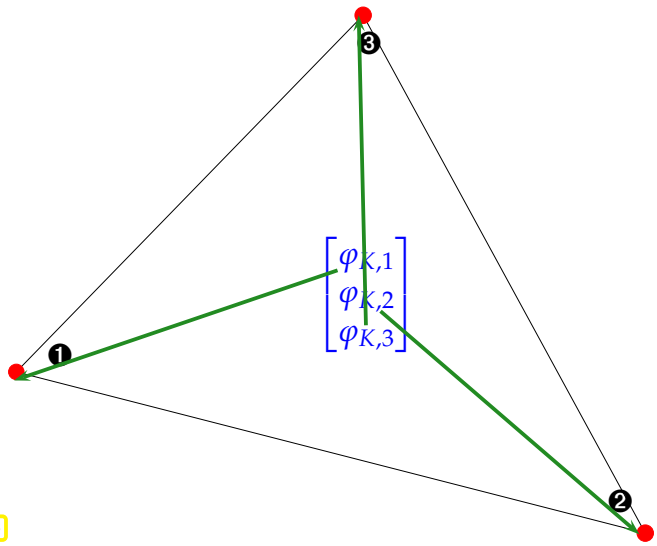


Fig. 108

Remark 3.3.46 (Assembly of right hand side vector for linear finite elements → § 3.3.29)

Entries of element load vectors from triangles sharing a vertex are summed up, see Fig. 107 for illustration and Code 3.3.47 for implementation of this cell-oriented assembly.

The argument `Mesh` passes a reference to an object of type **TriaMesh2D**, see Code 3.3.4 for the class definition, the argument `getElementVector` is a functor whose evaluation operator

(i) takes the geometry of a triangle in the form of a 3×2 coordinate matrix and returns the element load vector as defined in (3.3.45),

(ii) accepts a handle to a function $\mathbb{R}^2 \to \mathbb{R}$, which provides the source function f.

C++ code 3.3.47: Cell-oriented assembly of right hand side vector for linear finite elements, see (3.3.45) ➙ **GITLAB**

```
typedef function<double(const Eigen::Vector2d&)> FHandle_t;
typedef function<Eigen::Vector3d(const t_TriGeo &,FHandle_t)>
    LocalVectorHandle_t;

Eigen::VectorXd assemLoad_LFE(const TriaMesh2D &Mesh,
  const LocalVectorHandle_t &getElementVector,
  const FHandle_t &FHandle)
{
 // Obtain the number of vertices and cells (elements)
 int N = Mesh.Coordinates.rows();
 int M = Mesh.Elements.rows();
 // Initialize right hand side vector with zero.
 Eigen::VectorXd phi = Eigen::VectorXd::Zero(N);

 // Loop over elements and "distribute" local contributions
 for (int i = 0; i < M; i++) {
     // get local→global index mapping for current element,
     // cf. (3.3.33)
     Eigen::Vector3i dofhk = Mesh.Elements.row(i);
     t_TriGeo Vertices;
     // Extract geometry of current element, see § 3.3.3
```

```
      for (int j = 0; j < 3; j++)
        Vertices.row(j) = Mesh.Coordinates.row(dofhk(j));
      //compute element right hand side vector
      Eigen::Vector3d philoc = getElementVector(Vertices, FHandle);
      //add contributions to global load vector
      for (int j = 0; j < 3; j++)
          phi(dofhk(j)) += philoc(j);
  }
  return phi;
}
```

Same as in Code 3.3.35, also Code 3.3.47 employs only a loop over all cells of the mesh (*cell oriented assembly*), again resulting in *optimal computational effort* $O(\sharp\mathcal{M})$.

(3.3.48) Numerical quadrature for assembly of right hand side vector

Recall Rem. 1.5.5: $f : \Omega \mapsto \mathbb{R}$ given in procedural form

```
typedef function<double(const Eigen::Vector2d &)> FHandle_t;
```

Mandatory: use of numerical quadrature for approximate evaluation of $\ell_K(b_N^j)$, *cf.* (1.5.80).

In the 1D setting of Section 1.5.2.2 we used composite quadrature rules based on low order Gauss/Newton-Cotes quadrature formulas on the cells $[x_{j-1}, x_j]$ of the grid, e.g. the composite trapezoidal rule (1.5.80).

What is the 2D counterpart of the composite trapezoidal rule **?**

Recall:

trapezoidal rule [14, Eq. (5.2.5)] integrates linear interpolant of integrand based on endpoint values

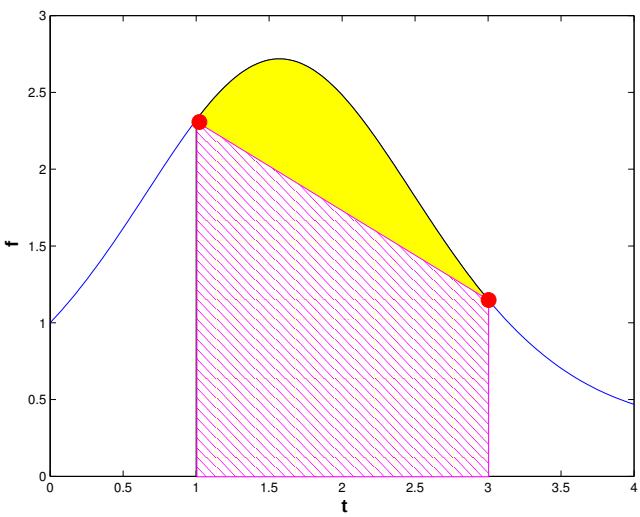
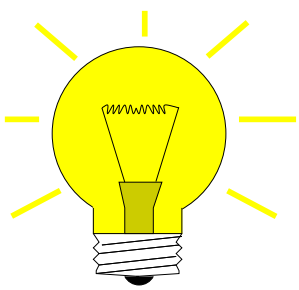


Fig. 109



Idea: 2D trapezoidal rule

for triangle K with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3}\left(f(\boldsymbol{a}^1) + f(\boldsymbol{a}^2) + f(\boldsymbol{a}^3)\right) . \tag{3.3.49}$$

$\hat{=}$ integration of linear interpolant $\sum_{i=1}^{3} f(\boldsymbol{a}^i)\lambda_i$ of f.

element (load) vector: $$\vec{\varphi}_K := \left(\ell_K(b_N^{j(i)}\big|_K)\right)_{i=1}^3 = (\ell_K(\lambda_i))_{i=1}^3 \approx \frac{|K|}{3}\begin{bmatrix} f(\boldsymbol{a}^1) \\ f(\boldsymbol{a}^2) \\ f(\boldsymbol{a}^3) \end{bmatrix}, \tag{3.3.50}$$

where $\boldsymbol{x}^{j(i)} = \boldsymbol{a}^i$, $i = 1,2,3$ (global node number $\leftrightarrow$ local vertex number).

The following code relies on (**??**) to compute the element load vector for an arbitrary triangle, whose vertex coordinates are passed as rows of a 3×2-matrix, *cf.* **??**. The source function f is made available through a functor object.

C++ code 3.3.51: (Approximate) computation of element load vector by means of 2D trapezoidal local quadrature rule (3.3.50) ➙ **GITLAB**

```
// Functor type for right hand side source function
typedef function<double(const Eigen::Vector2d &)> FHandle_t;

Eigen::Vector3d localLoadLFE(const t_TriGeo& V, const FHandle_t&
    FHandle)
{
  // Compute area of triangle, cf. ??
  double area =
      0.5*((V(1,0)-V(0,0))*(V(2,1)-V(1,1))-(V(2,0)-V(1,0))*(V(1,1)-V(0,1)));
  // Evaluate source function for vertex location
  Eigen::Vector3d philoc = Eigen::Vector3d::Zero();
  // Implements (3.3.50)
  for (int i = 0; i < 3; i++) philoc(i) = FHandle(V.row(i));
  // Scale with 1/3·area of triangle
  philoc *= area/3.0;
  return philoc;
}
```

?! *Review question(s) 3.3.52.* (Linear finite elements in 2D)

1. Chop up a square $\Omega \subset \mathbb{R}^2$ into n^2 congruent small squares and create a triangular mesh $\mathcal{M}$ of Ω by splitting each small square along parallel diagonals. What is $\dim \mathcal{S}_1^0(\mathcal{M})$ and $\dim \mathcal{S}_{1,0}^0(\mathcal{M})$ in terms of n?

2. For the domain and mesh from Item 1 determine the maximal number of non-zerp entries of the Galerkin matrix obtained when discretizing (3.1.4) with trial and test space $\mathcal{S}_{1,0}^0(\mathcal{M})$ (sharp bound).

3. We are provided with an **TriaMesh2D** object describing a planar triangulation $\mathcal{M}$ of a polygon Ω with N nodes and vector<bool> bdflags; where bdflags[k] == true, if the node with number k is located on $\partial\Omega$. Modify Code 3.3.36 so that it assembles a Galerkin matrix w.r.t. the trial/test space $\mathcal{S}_{1,0}^0(\mathcal{M})$.

4. Write $\mathbf{A}_K$ for the element matrix for linear finite elements, the bilinear form $\mathsf{a}(u,v) := \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$, and a triangle K. We use numerical quadrature base on the 2D trapezoidal rule to compute the element matrix $\mathbf{B}_K$ for the bilinear form $\widetilde{b}(u,v) := \int_\Omega \sigma(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$, $\sigma \in C^0(\overline{\Omega})$. How can $\mathbf{B}_K$ be computed from $\mathbf{A}_K$?

3.4 Building Blocks of General Finite Element Methods

(3.4.1) Overview

The previous section explored the details of a simple finite element discretization of 2nd-order elliptic variational problems. Yet, it already introduced *key features and components* that distinguish the finite element approach to the discretization of linear boundary value problems for partial differential equations:

- a focus on the variational formulation of a boundary value problem $\rightarrow$ Section 2.9,
- a partitioning of the computational domain Ω by means of a mesh $\mathcal{M}$ ($\rightarrow$ Section 3.3.1)
- the use of Galerin trial and test spaces based on piecewise polynomials w.r.t. $\mathcal{M}$ ($\rightarrow$ Section 3.3.2),
- the use of locally supported basis functions for the assembly of the resulting linear system of equations ($\rightarrow$ Section 3.3.3).

In this section a more abstract point of view is adopted and the components of a finite element method for scalar 2nd-order elliptic boundary value problems will be discussed in greater generality. However, prior perusal of Section 3.3 is strongly recommended.

3.4.1 Meshes

First main ingredient of FEM: triangulation/mesh of Ω $\rightarrow$ Section 3.3.1

Definition 3.4.2. Finite element mesh/triangulation

A mesh (or triangulation) of $\Omega \subset \mathbb{R}^d$ is a finite collection $\{K_i\}_{i=1}^M$, $M \in \mathbb{N}$, of *open* non-degenerate (curvilinear) polygons ($d = 2$)/polyhedra ($d = 3$) such that

(A) $\overline{\Omega} = \bigcup \{\overline{K}_i, i = 1, \dots, M\}$,

(B) $K_i \cap K_j = \emptyset \quad \Leftrightarrow \quad i \neq j$,

(C) for all $i, j \in \{1, \dots, M\}$, $i \neq j$, the intersection $\overline{K}_i \cap \overline{K}_j$ is either empty or a vertex, edge, or face of both K_i and K_j.

Requirement (C) rules out "hanging nodes", *cf.* condition (iv) on the triangulation introduced in Section 3.3.1, page 188. Fig. 84 depicts the "hanging node" situation.

(3.4.3) Finite element meshes: customary terminology

- Entities = geometric entities "vertex", "edge", "face" of polygon/polyhedron: meaning of these terms corresponds to geometric intuition.

  Entities can be classified by their dimension or co-dimension, which add up to the world dimension d:

geometric entity	dimension	codimension
2D, $d = 2$:		
triangles	2	0
edges	1	1
vertices	0	2
3D, $d = 3$:		
tetrahedra	3	0
faces	2	1
edges	1	2
vertices	0	3

- Given a mesh $\mathcal{M} := \{K_i\}_{i=1}^{M}$: K_i called cell or element = entities of co-dimension 0
- Vertices of a mesh are often called nodes (✎ notation for set of nodes: $\mathcal{V}(\mathcal{M})$)

(3.4.4) Types of meshes

Meshes according to Def. 3.4.2 can be classified further:

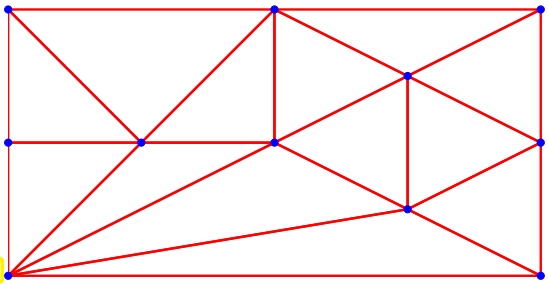


Fig. 110

Triangular mesh in 2D

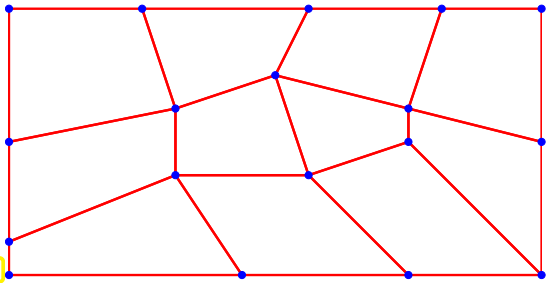


Fig. 111

Quadrilateral mesh in 2D

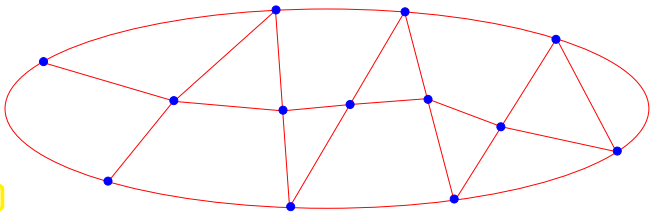


Fig. 112

◁ 2D hybrid mesh comprising
- triangles
- quadrilaterals
- curvilinear cells (at $\partial\Omega$)

(Curved) tetrahedral meshes in 3D (created with NETGEN):

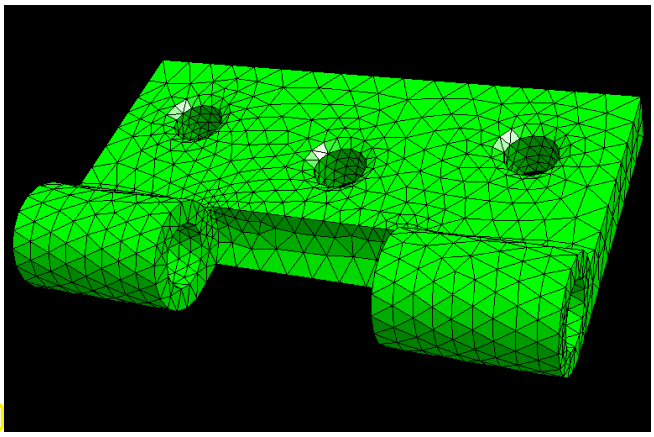


Fig. 113

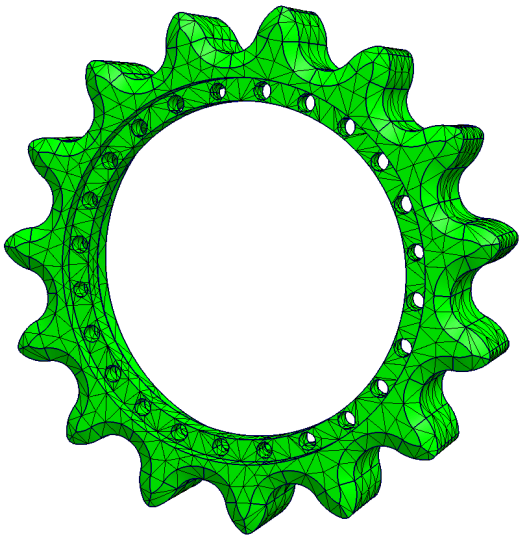


Fig. 114

Tensor product mesh = grid ▷

in 2D: $a = x_0 < x_1 < \ldots < x_n = b$,
$c = y_0 < y_1 < \ldots < y_m = d$.

$$\mathcal{M} = \{]x_{i-1}, x_i[\times]y_{j-1}, y_j[: \quad 1 \le i \le n, 1 \le j \le m\} . \tag{3.4.5}$$

☞ Restricted to tensor product domains

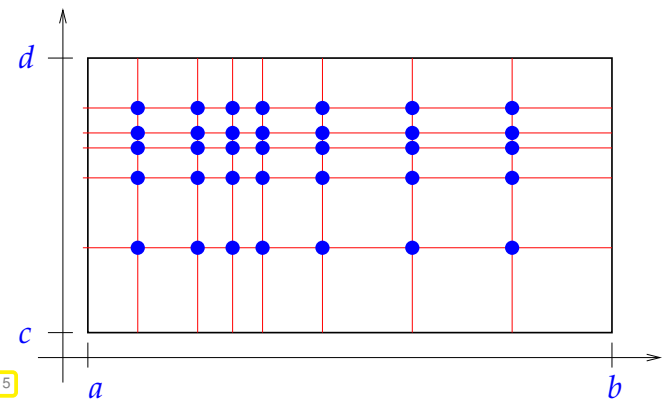


Fig. 115

Terminology: Simplicial mesh = triangular mesh in 2D / tetrahedral mesh in 3D

Remark 3.4.6 (Finite element meshes with hanging nodes)

If (C) does not hold

Triangular non-conforming mesh (with hanging nodes)

$\overline{K}_i \cap \overline{K}_j$ is only part of an edge/face for at most one of the adjacent cells.

(However, this mesh is conforming, if degenerate quadrilaterals are admitted.)

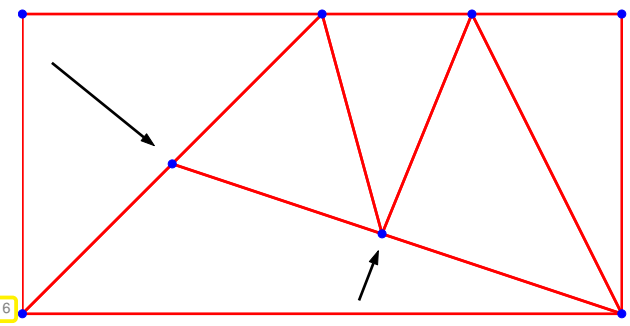


Fig. 116

3.4.2 Polynomials

Second main ingredient of FEM:

In FEM: Galerkin trial/test space comprise $\mathcal{M}$*-locally polynomial* functions on Ω

Polynomials are attractive, because

- they allow fast and easy evaluation [14, **??**]and straightforward analytic differentiation and integration,
- (smooth) functions can be approximated efficiently by means of polynomials [14, Section 4.1].

Known: polynomials of degree $\le p$, $p \in \mathbb{N}_0$, in 1D (univariate polyomials), see (1.5.28)

$$\mathcal{P}_p(\mathbb{R}) := \{x \mapsto c_0 + c_1 x + c_2 x^2 + \ldots c_p x^p\} .$$

In higher dimensions this concept allows various generalizations, one given in the following definition, one given in Def. 3.4.13.

(3.4.7) Multivariate Polynomials

Definition 3.4.8. Multivariate polynomials

Space of multivariate (d-variate) polynomials of (total) degree $p \in \mathbb{N}_0$:

$$\mathcal{P}_p(\mathbb{R}^d) := \{\boldsymbol{x} \in \mathbb{R}^d \mapsto \sum_{\boldsymbol{\alpha}\in\mathbb{N}_0^d,\,|\boldsymbol{\alpha}|\leq p} c_{\boldsymbol{\alpha}}\boldsymbol{x}^{\boldsymbol{\alpha}}\,,\, c_{\boldsymbol{\alpha}} \in \mathbb{R}\}\,.$$

Def. 3.4.8 relies on multi-index notation:

$$\boldsymbol{\alpha} = (\alpha_1,\dots,\alpha_d):\quad \boldsymbol{x}^{\boldsymbol{\alpha}} := x_1^{\alpha_1}\cdot\cdots\cdot x_d^{\alpha_d}\,, \tag{3.4.9}$$

$$|\boldsymbol{\alpha}| = \alpha_1+\alpha_2+\cdots+\alpha_d\,. \tag{3.4.10}$$

Special case:

$$d=2:\quad \mathcal{P}_p(\mathbb{R}^2) = \left\{ \sum_{\substack{\alpha_1,\alpha_2\geq 0\\ \alpha_1+\alpha_2\leq p}} c_{\alpha_1,\alpha_2} x_1^{\alpha_1} x_2^{\alpha_2},\, c_{\alpha_1,\alpha_2}\in\mathbb{R} \right\}.$$

Examples: $\mathcal{P}_2(\mathbb{R}^2) = \mathrm{Span}\{1, x_1, x_2, x_1^2, x_2^2, x_1x_2\}$,
$\mathcal{P}_1(\mathbb{R}^2)$ = affine linear functions $\mathbb{R}^2 \mapsto \mathbb{R}$, see Section 3.3.2

Lemma 3.4.11. Dimension of spaces of polynomials

$$\dim \mathcal{P}_p(\mathbb{R}^d) = \binom{d+p}{p} \quad \textit{for all } p\in\mathbb{N}_0,\, d\in\mathbb{N}$$

Proof. Distribute p "powers" to the d independent variables or discard them $\triangleright$ $d+1$ bins.

Combinatorial model: number of different linear arrangements of p identical items and d separators $= \binom{d+p}{p}$. □

Leading order for $p\to\infty$: $\dim \mathcal{P}_p(\mathbb{R}^d) = O(p^d)$

(3.4.12) Tensor product polynomials

Definition 3.4.13. Tensor product polynomials

Space of tensor product polynomials of degree $p\in\mathbb{N}$ in each coordinate direction

$$\mathcal{Q}_p(\mathbb{R}^d) := \{\mathbf{x} \mapsto p_1(x_1)\cdot\cdots\cdot p_d(x_d),\, p_i\in\mathcal{P}_p(\mathbb{R}), i=1,\dots,d\}\,.$$

Example: $\mathcal{Q}_2(\mathbb{R}^2) = \mathrm{Span}\{1, x_1, x_2, x_1x_2, x_1^2, x_1^2x_2, x_1^2x_2^2, x_1x_2^2, x_2^2\}$

Lemma 3.4.14. Dimension of spaces of tensor product polynomials

$$\dim \mathcal{Q}_p(\mathbb{R}^d) = (p+1)^d \quad \textit{for all} \quad p\in\mathbb{N}_0,\, d\in\mathbb{N}$$

Terminology: $\mathcal{P}_p(\mathbb{R}^d)/\mathcal{Q}_p(\mathbb{R}^d)$ = *complete* spaces of polynomials/tensor product polynomials

3.4.3 Basis functions

Third main ingredient of FEM: locally supported basis functions
(see Section 3.2 for role of bases in Galerkin discretization)

Basis functions $b_N^1, \dots, b_N^N$ for a finite element trial/test space $V_{0,N}$ built on a mesh $\mathcal{M}$ *must* satisfy:

(a) $\mathfrak{B}_N := \{b_N^1, \dots, b_N^N\}$ is basis of $V_{0,N}$ ➢ $N = \dim V_{0,N}$,

(b) each b_N^i is associated with a single geometric entity (cell/edge/face/vertex) of $\mathcal{M}$,

(c) $\operatorname{supp}(b_N^i) = \bigcup\{\overline{K}\colon K \in \mathcal{M}, \boldsymbol{p} \subset \overline{K}\}$, if b_N^i associated with cell/edge/face/vertex $\boldsymbol{p}$.

Finite element terminology: b_N^i = global shape functions/global basis functions

Mesh $\mathcal{M}$ + global shape functions ➡ complete description of finite element space

The specification of the global shape functions is considered an integral part of the description of a finite element method. However, remember from Thm. 1.5.25 and Section 3.2 that it is the sheer finite element space, that is, the span of the global shape functions, that determines the Galerkin solution.

Example 3.4.15 (Supports of global shape functions in 1D → Section 1.5.2.2)

✦ $\Omega =]a,b[\mathrel{\hat{=}}$ interval

✦ Equidistant mesh

$$\mathcal{M} := \{]x_{j-1}, x_j[,\ j = 1, \dots, M\}\ ,$$

$$x_j := a + hj,\ h := (b-a)/M,\ M \in \mathbb{N}.$$

Support (→ Def. 1.5.76) of global shape function (tent function) associated with x_7

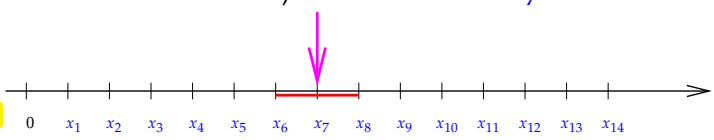


Fig. 117

Example 3.4.16 (Supports of global shape functions on triangular mesh)

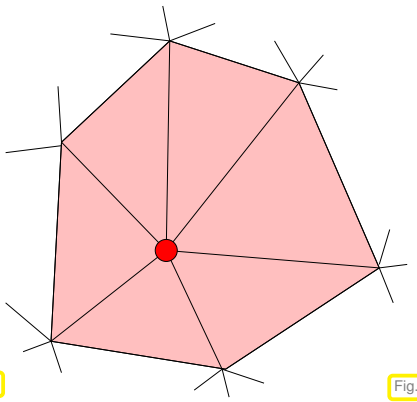


Fig. 118

Support of node-associated basis function, *cf.* Fig. 92

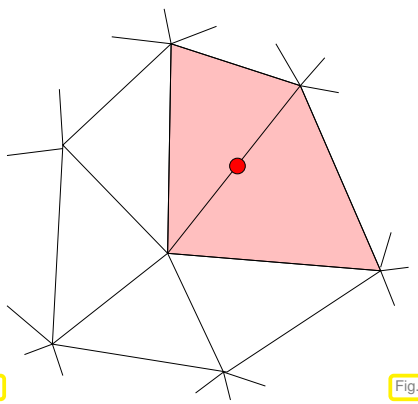


Fig. 119

Support of edge-associated basis function

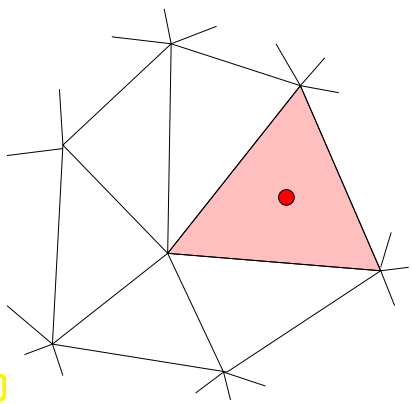


Fig. 120

Support of cell-associated basis function

(3.4.17) Importance of local supports

Requirement **(c)** implies that

global finite element basis functions are locally supported.

What is the rationale for this requirement **?**

Consider a generic bilinear form a arising from a linear scalar 2nd-order elliptic BVP, see (3.3.16): it involves integration over $\Omega/\partial\Omega$ of products of (derivatives of) basis functions. Thus the integrand for $\mathsf{a}(b_N^j, b_N^i)$ vanishes outside the overlap of the supports of b_N^j and b_N^i.

Galerkin matrix $\mathbf{A} \in \mathbb{R}^{N,N}$ with $(\mathbf{A})_{ij} := \mathsf{a}(b_N^j, b_N^i)$, $i,j = 1,\dots,N$ satisfies

$a_{ij} \neq 0$ only if b_N^i and b_N^j associated with vertices/faces/edges(cells) adjacent to common cell

Finite element stiffness matrices are sparse (→ Notion 3.3.18)

In turns, sparsity of the coefficient matrix is crucial for

- the ability to store the Galerkin matrix with $O(N)$ memory requirements, where N is the dimension of the finite element space,
- for the fast direct or iterative solution of the linear system of equations arising from finite element Galerkin discretization.

Now we introduce an important notion that will be crucial for understanding the efficient assembly of finite element Galerkin matrices. Recall that "assembly" $\hat{=}$ initialization of finite element Galerkin matrix from element contributions, *cf.* § 3.3.29.

$$\text{Global shape functions} \xrightarrow{\text{Restriction to element}} \text{local shape functions} \tag{3.4.18}$$

Definition 3.4.19. Local shape functions

Given a finite element function space on a mesh $\mathcal{M}$ with global shape functions b_N^i, $i = 1,\dots,N$:

$$\{b_{N|K}^j,\ K \subset \operatorname{supp}(b_N^j)\} = \text{set of local shape functions on } K \in \mathcal{M}\ .$$

A consequence of property **(b)** of global shape functions:

(b) Also local shape functions $b_K^1,\dots,b_K^Q$, $Q = Q(K) \in \mathbb{N}$ are associated with geometric entities (vertices/edges/faces/interior) of K.

Example 3.4.20 (Local shape functions for $\mathcal{S}_1^0(\mathcal{M})$ in 2D → Section 3.3.3)

Global basis function for $\mathcal{S}_1^0(\mathcal{M})$ ▷

On "unit triangle" K with vertices

$$\mathbf{a}^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \ \mathbf{a}^2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \ \mathbf{a}^3 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Local shape functions:

$$b_K^1(\mathbf{x}) = 1 - x_1 - x_2 \ ,$$
$$b_K^2(\mathbf{x}) = x_1 \ ,$$
$$b_K^3(\mathbf{x}) = x_2 \ .$$

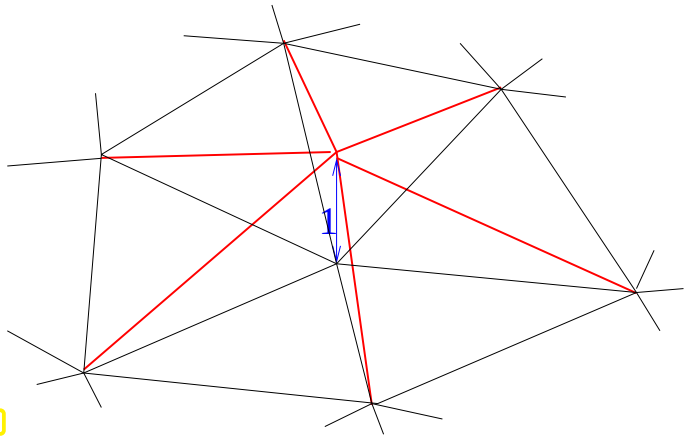


Fig. 121

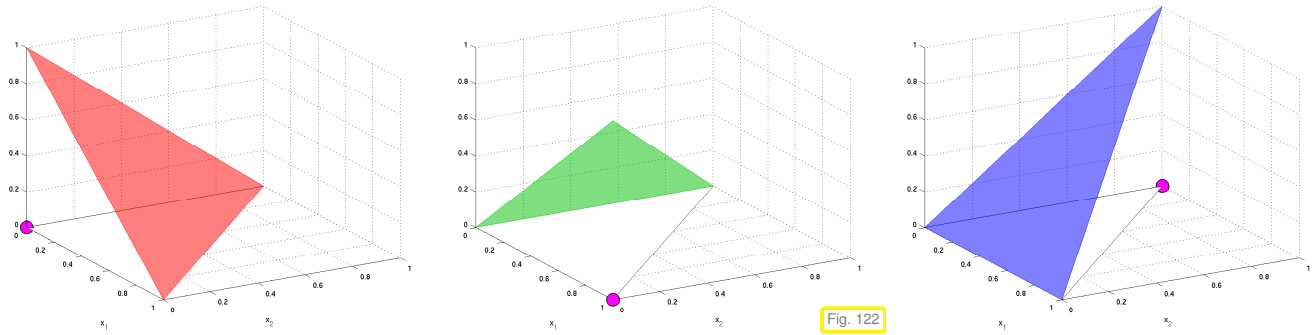


Fig. 122

These are the barycentric coordinate functions λ_1, λ_2, λ_3 introduced in Section 3.3.5

?! *Review question(s) 3.4.21.* (Principles of finite element discretization)

1. What 2D triangular meshes with hanging nodes can be regarded as valid hybrid meshes comprising triangular and quadrilateral cells.
2. Devise a class **QuadMesh2D** for general planar quadrilateral meshes in analogy to the class **TriaMesh2D**.
3. How can a quadrilateral mesh be converted into a triangular mesh? Based on the data structures developed in Item 2 outline an algorithm.
4. Consider the Galerkin discretization of (3.1.4) on a planar triangular mesh $\mathcal{M}$ using global shape functions associated with the edges of the mesh. Give a sharp bound for the number of non-zero entries of the Galerkin matrix in terms of the number of vertices $\sharp\mathcal{V}(\mathcal{M})$, number of edges $\sharp\mathcal{E}(\mathcal{M})$, and number of cells $\sharp\mathcal{M}$ of the mesh.

3.5 Lagrangian Finite Element Spaces

Taken for granted in this section: finite element mesh $\mathcal{M}$ according to Def. 3.4.2.

(3.5.1) H^1-conforming finite element spaces

Goal: construction of finite element spaces and global shape functions of higher polynomials degrees, generalizing the space $\mathcal{S}_1^0(\mathcal{M})$ introduced in Section 3.3.3.

Lagrangian finite element spaces provide spaces $V_{0,N}$ of $\mathcal{M}$-piecewise polynomials that fulfill

$$V_{N,0} \subset C^0(\overline{\Omega}) \overset{\text{Thm. 2.3.35}}{\Longrightarrow} \boxed{V_{N,0} \subset H^1(\Omega)} \; .$$

Parlance: finite element spaces that are contained in $H^1(\Omega)$ are often called "H^1-conforming".

Notation:
(Lagrangian FE spaces)

$$\mathcal{S}_p^0(\mathcal{M})$$

continuous functions, *cf.* $C^0(\Omega)$

locally polynomials of degree p , *e.g.* $\mathcal{P}_p(\mathbb{R}^d)$

3.5.1 Simplicial Lagrangian FEM

Now $\mathcal{M}$ = simplicial mesh, consisting of triangles in 2D, tetrahedra in 3D.

Now we generalize $\mathcal{S}_1^0(\mathcal{M})/\mathcal{S}_{1,0}^0(\mathcal{M})$ from Section 3.3 to higher polynomial degree $p \in \mathbb{N}_0$.

Definition 3.5.2. Simplicial Lagrangian finite element spaces

Space of p-th degree Lagrangian finite element functions on simplicial mesh $\mathcal{M}$

$$\mathcal{S}_p^0(\mathcal{M}) := \{v \in C^0(\overline{\Omega}) \colon v_{|K} \in \mathcal{P}_p(K) \quad \forall K \in \mathcal{M}\} \; .$$

Def. 3.5.2 merely describes the space of trial/test functions used in a Lagrangian finite element method on a simplicial mesh. A crucial ingredient is still missing ($\rightarrow$ Section 3.4.3): the global shape functions still need to be specified. This is done by generalizing (3.3.9) based on sets of special *interpolation nodes*.

Example 3.5.3 (Triangular quadratic ($p = 2$) Lagrangian finite elements)

Suitable set of interpolation nodes

$$\mathcal{N} := \mathcal{V}(\mathcal{M}) \cup \{\text{midpoints of edges}\} \; ,$$
$$\mathcal{N} = \{\boldsymbol{p}_1, \dots, \boldsymbol{p}_N\} \quad \text{(ordered)} \; .$$

Nodal basis functions b_N^j, $j = 1, \dots, N$, defined by, *cf.* (3.3.9)

$$b_N^j(\boldsymbol{p}_i) = \begin{cases} 1 & \text{, if } i = j \; , \\ 0 & \text{else.} \end{cases} \tag{3.5.4}$$

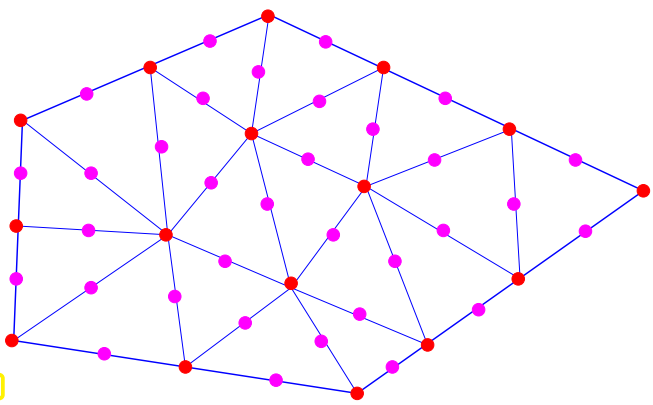


Fig. 123

A "definition" like (3.5.4) is cheap, but it may be pointless, in case no such functions b_N^j exist. To establish their existence, we first study the case of a single triangle K.

We have to show that there is a basis of $\mathcal{P}_2(\mathbb{R}^2)$ that satisfies (3.5.4) in the case of a mesh consisting of a single triangle $\mathcal{M} = \{K\}$.

Interpolation nodes on triangle K with vertices $\boldsymbol{a}^1$, $\boldsymbol{a}^2$, and $\boldsymbol{a}^3$, see Fig. 124:

$$\begin{aligned} &\boldsymbol{p}_1 = \boldsymbol{a}^1 \,, && \boldsymbol{p}_2 = \boldsymbol{a}^2, && \boldsymbol{p}_3 = \boldsymbol{a}^3 \,, \\ &\boldsymbol{p}_4 = \tfrac{1}{2}(\boldsymbol{a}^1 + \boldsymbol{a}^2) \,, && \boldsymbol{p}_5 = \tfrac{1}{2}(\boldsymbol{a}^2 + \boldsymbol{a}^3), && \boldsymbol{p}_6 = \tfrac{1}{2}(\boldsymbol{a}^1 + \boldsymbol{a}^3) \,. \end{aligned} \tag{3.5.5}$$

What is the rationale for this numbering? There is absolutely **none**, because the numbering of the local interpolation nodes can be chosen *arbitrarily*. Once it is decided, however, one has to adhere to this choice consistently throughout a finite element code.

A first simple *consistency check*: does the number of interpolation nodes $\sharp\mathcal{N}$ for $\mathcal{M} = \{K\}$ agree with $\dim \mathcal{P}_2(\mathbb{R}^2) = 6$? Yes, it does!

Next step: "Proof by constuction" ; give formulas for local shape functions.

Local shape functions

$$\begin{aligned} b_K^1 &= (2\lambda_1 - 1)\lambda_1 \,, \\ b_K^2 &= (2\lambda_2 - 1)\lambda_2 \,, \\ b_K^3 &= (2\lambda_3 - 1)\lambda_3 \,, \\ b_K^4 &= 4\lambda_1\lambda_2 \,, \\ b_K^5 &= 4\lambda_2\lambda_3 \,, \\ b_K^6 &= 4\lambda_1\lambda_3 \,. \end{aligned} \tag{3.5.6}$$

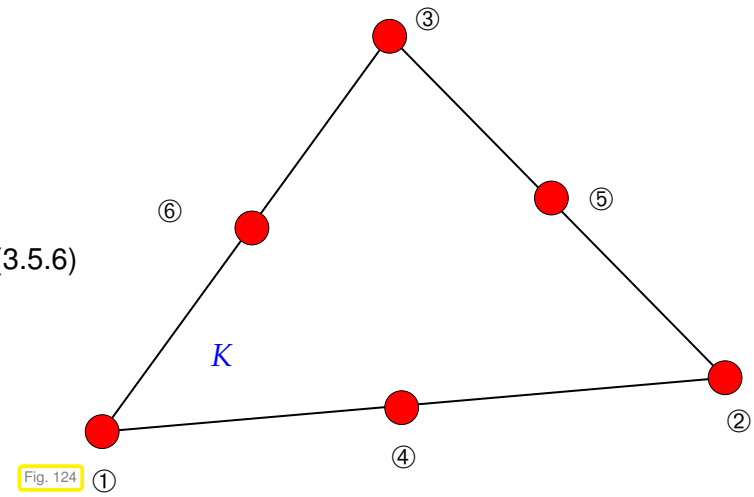


Fig. 124

It is generally true for Lagrangian finite elements that local shape functions are linear combinations of (products of) barycentric coordinate functions.

To confirm the validity of the formulas (3.5.6), that is, the compliance with (3.5.4), note that

- $\lambda_i(\boldsymbol{a}^i) = 1$ and $\lambda_i(\boldsymbol{a}^j) = 0$, if $i \neq j$, where $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3$ are the vertices of the triangle K,
- $\lambda_1(\mathbf{m}^{12}) = \lambda_1(\mathbf{m}^{13}) = \frac{1}{2}$, $\lambda_2(\mathbf{m}^{12}) = \lambda_2(\mathbf{m}^{23}) = \frac{1}{2}$, $\lambda_3(\mathbf{m}^{13}) = \lambda_3(\mathbf{m}^{23}) = \frac{1}{2}$, $\lambda_1(\mathbf{m}^{23}) = \lambda_2(\mathbf{m}^{13}) = \lambda_3(\mathbf{m}^{12}) = 0$, where $\mathbf{m}^{ij} = \frac{1}{2}(\mathbf{a}^i + \mathbf{a}^j)$ denotes the midpoint of the edge connecting $\mathbf{a}^i$ and $\mathbf{a}^j$,
- each barycentric coordinate function λ_i is affine linear such that $\lambda_i\lambda_j \in \mathcal{P}_2(\mathbb{R}^2)$.

Graphs of selected local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ over a triangle:

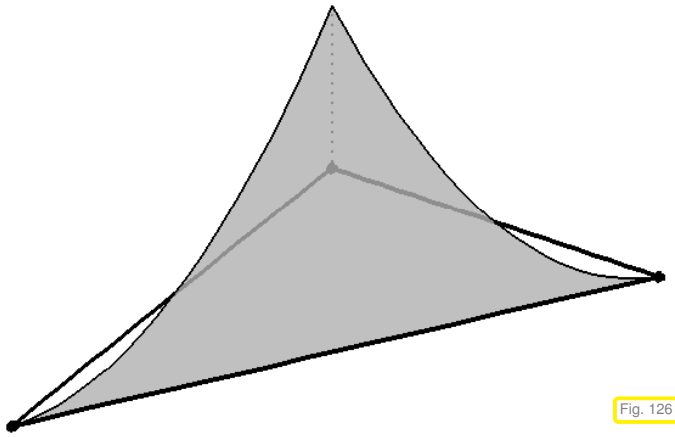


Fig. 125

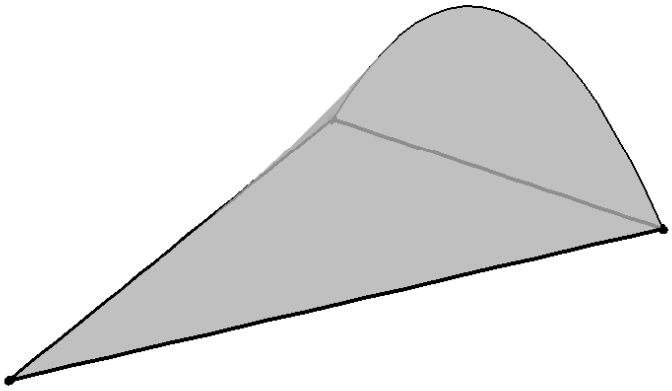


Fig. 126

So far we have seen that *local shape functions* can be found that satisfy (3.5.4).

Issue: can the local shape functions from (3.5.6) be "stiched together" across interelement edges such that they yield a *continuous* gobal basis function? (Remember that Thm. 2.3.35 demands global continuity in order to obtain a subspace of $H^1(\Omega)$.)

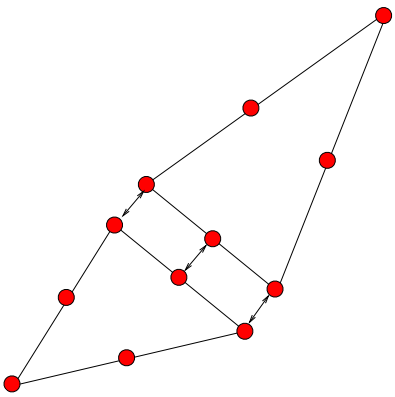


Fig. 127

The restriction of a quadratic polynomial to an edge is an *univariate* quadratic polynomial.

Fixing its value in three points, the midpoint of the edge and the endpoints, *uniquely* fixes this polynomial.

The local shape functions associated with the same interpolation node "from left and right" agree on the edge.

➢ continuity !

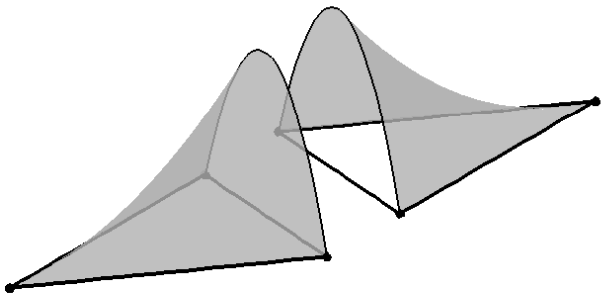


Fig. 128

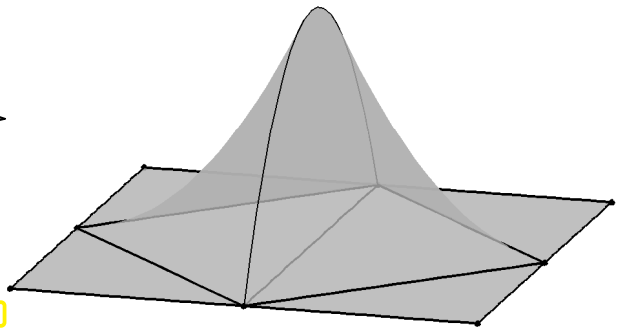


Fig. 129

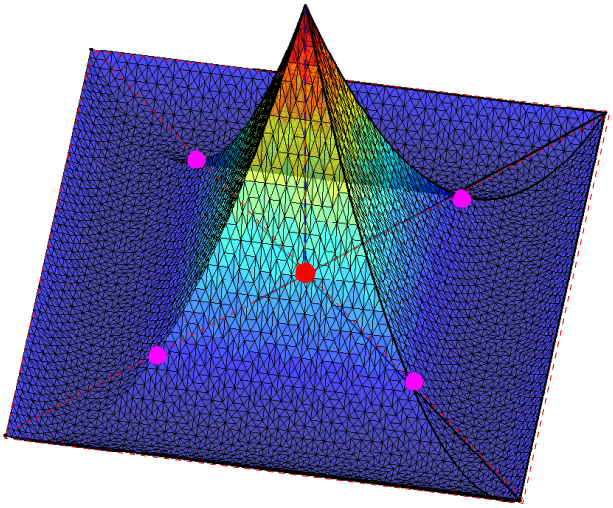


Fig. 130

◁ Global basis function for $\mathcal{S}_2^0(\mathcal{M})$ associated with a vertex

(3.5.4): this function attains value $= 1$ at a vertex (•) and vanishes at the midpoints (•) of the edges of adjacent triangles, as well as at any other vertex.

Example 3.5.7 (Local interpolation nodes for cubic ($p = 3$) and quartic ($p = 4$) Lagrangian FE in 2D)

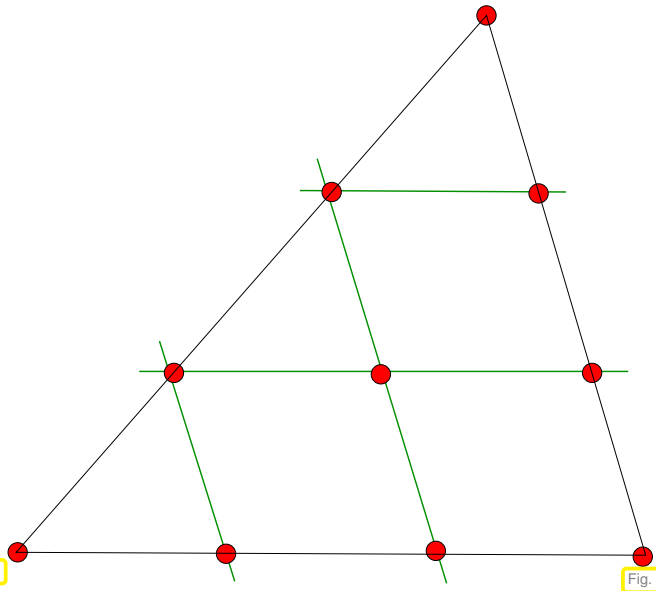


Fig. 131

(local) interpolation nodes for $\mathcal{S}_3^0(\mathcal{M})$

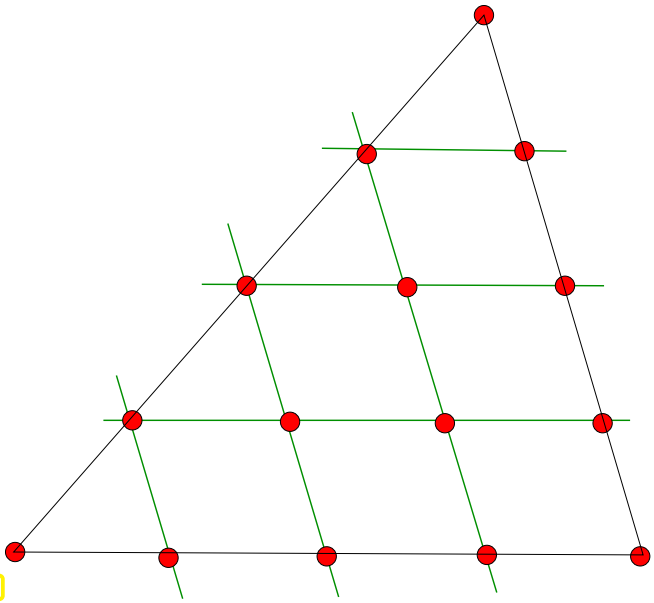


Fig. 132

(local) interpolation nodes for $\mathcal{S}_4^0(\mathcal{M})$

Can you already guess a general pattern underlying the location of local interpolation nodes for degree p Lagrangian finite elements on triangles? They are the points whose barycentric coordinates satisfy $\lambda_i(\boldsymbol{p}_j) \in \{\frac{0}{p}, \frac{1}{p}, \dots, \frac{p-1}{p}, \frac{p}{p}\}$.

3.5.2 Tensor-product Lagrangian FEM

Now we consider tensor product meshes (grids), see (3.4.5), Fig. 115, for a 2D example.

Example 3.5.8 (Bilinear Lagrangian finite elements)

Sought: generalization of 1D piecewise linear finite element functions from Section 1.5.2.2, see § 1.5.69, to 2D tensor product grid $\mathcal{M}$.

Tensor product structure of $\mathcal{M}$ ➤ tensor product construction of FE space

This is best elucidated by a tensor product construction of basis functions:

$$\begin{array}{lcl} b^j_{N,x}(x) & : & \text{1D tent function on } \mathcal{M}_x = \{[x_{j-1},x_j],\ j=1,\dots,n\} \\ b^l_{N,y}(y) & : & \text{1D tent function on } \mathcal{M}_y = \{[y_{j-1},y_j],\ j=1,\dots,n\} \end{array}$$

2D tensor product "tent function" associated with node $\boldsymbol{p}$:

$$b^{\boldsymbol{p}}_N(\boldsymbol{x}) = b^j_{N,x}(x_1)\cdot b^l_{N,y}(x_2)\,, \quad \text{where} \quad \boldsymbol{p} = (x_j,y_l)^T\,. \tag{3.5.9}$$

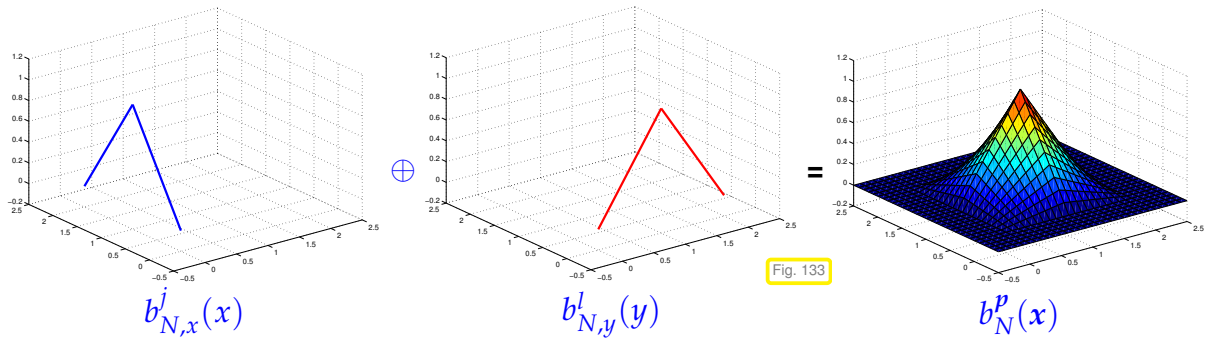


Fig. 133

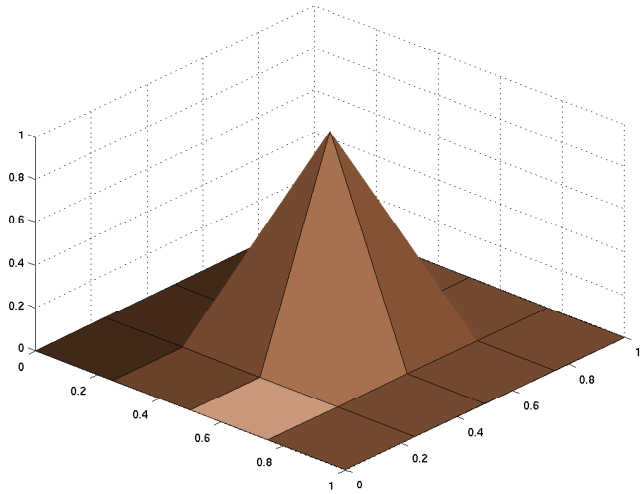


Fig. 134

◁ 2D tensor product tent function

No pyramid !

Basis functions *associated* (→ Section 3.4.3, condition (c)) with nodes of $\mathcal{M}$,

Tensor product construction ➤ bilinear local shape functions, e.g. on $K =]0,1[^2$

$$\begin{aligned} b^1_K(\boldsymbol{x}) &= (1-x_1)(1-x_2)\,, \\ b^2_K(\boldsymbol{x}) &= x_1(1-x_2)\,, \\ b^3_K(\boldsymbol{x}) &= x_1x_2\,, \\ b^4_K(\boldsymbol{x}) &= (1-x_1)x_2\,. \end{aligned} \tag{3.5.10}$$

$$\blacktriangleright \quad b^i_K(\boldsymbol{a}^j) = \delta_{ij}\,, \quad 1 \le i,j \le 4\,,$$

that is, these basis functions satisfy a local version of (3.5.4).

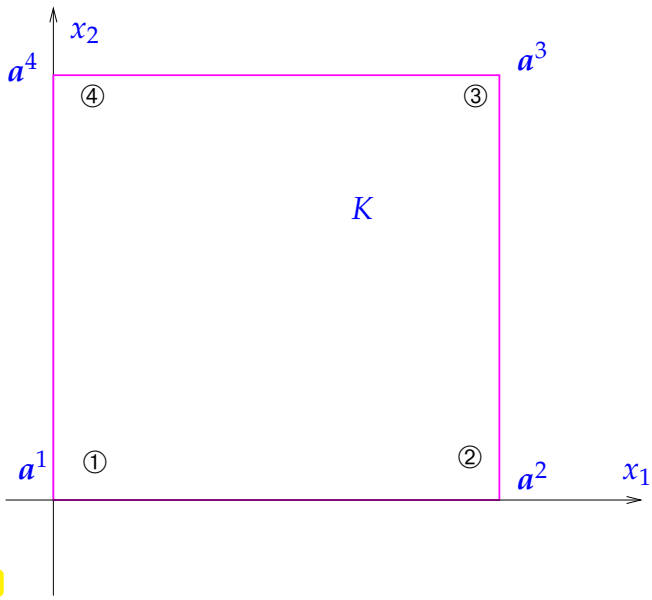


Fig. 135

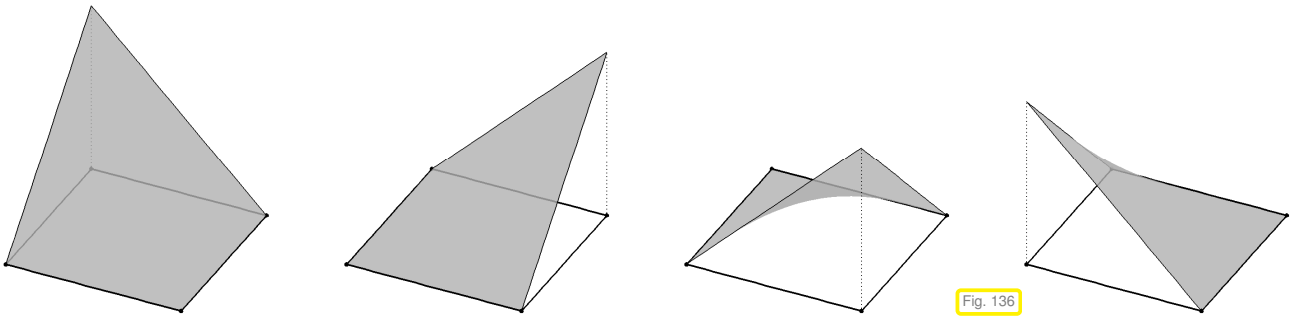


Fig. 136

Bilinear local shape functions on unit square K

$$\text{Span}\{b_K^1, b_K^2, b_K^3, b_K^4\} = \mathcal{Q}_1(\mathbb{R}^2) \ .$$

Bilinear Lagrangian finite element space on 2D tensor product mesh $\mathcal{M}$:

$$\mathcal{S}_1^0(\mathcal{M}) := \{v \in C^0(\Omega) \colon v_{|K} \in \mathcal{Q}_1(\mathbb{R}^2) \ \forall K \in \mathcal{M}\} \ . \tag{3.5.11}$$

The following is a natural generalization of (3.5.11) to higher degree local tensor product polynomials, see Def. 3.4.13:

Definition 3.5.12. Tensor product Lagrangian finite element spaces

Space of p-th degree Lagrangian finite element functions on tensor product mesh $\mathcal{M}$

$$\mathcal{S}_p^0(\mathcal{M}) := \{v \in C^0(\overline{\Omega}) \colon v_{|K} \in \mathcal{Q}_p(K) \ \forall K \in \mathcal{M}\} \ .$$

Terminology: $\mathcal{S}_1^0(\mathcal{M})$ = multilinear finite elements ($p = 1, d = 2$ = bilinear finite elements)

Remaining issue: definition of global basis functions (global shape functions)

Policy: use of interpolation nodes as in Section 3.5.1, see Ex. 3.5.3.

Example 3.5.13 (Quadratic tensor product Lagrangian finite elements)

Consider the case $p = 2, d = 2$ for Def. 3.5.12:

Interpolation nodes for $\mathcal{S}_2^0(\mathcal{M})$

$$\mathcal{N} = \mathcal{V}(\mathcal{M}) \cup \{\text{midpoints of edges}\} \ .$$

Note: number of interpolation nodes belonging to one cell is

$$9 = \dim \mathcal{Q}_2(\mathbb{R}^2) \ .$$

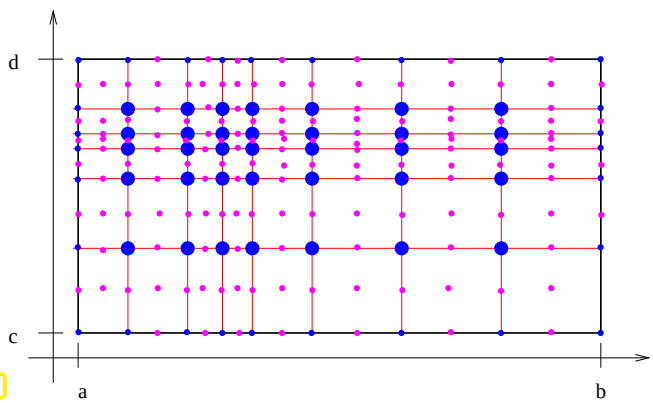
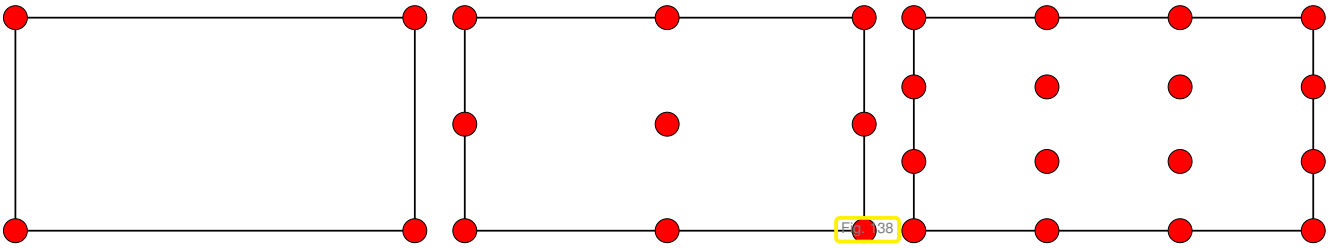


Fig. 137

Global basis functions defined analoguously to (3.5.4).

$$\mathcal{N} = \{\boldsymbol{p}_1, \dots, \boldsymbol{p}_N\}: \qquad b_N^j \in \mathcal{S}_2^0(\mathcal{M}), \quad b_N^j(\boldsymbol{p}_l) = \begin{cases} 1 & \text{, if } j = l\,, \\ 0 & \text{else.} \end{cases}$$

Choice of interpolation nodes for tensor product Lagrangian finite elements:



$p = 1$ $p = 2$ $p = 3$

(3.5.14) Imposing homogeneous Dirichlet boundary conditions

What is a global basis for $\mathcal{S}_p^0(\mathcal{M}) \cap H_0^1(\Omega)$, where $\mathcal{M}$ is either a simplicial mesh or a tensor product mesh?

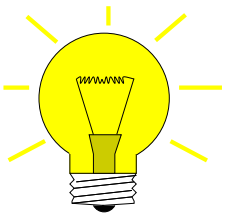
We proceed analoguous to § 3.3.14: recall that global basis functions are defined via interpolation nodes $\boldsymbol{p}^j$, $j = 1, \dots, N$, see (3.5.4).

$$\boxed{\mathcal{S}_{p,0}^0(\mathcal{M}) := \mathcal{S}_p^0(\mathcal{M}) \cap H_0^1(\Omega) = \mathrm{Span}\{b_N^j \colon \boldsymbol{p}^j \in \Omega \text{ (interior node)}\}} \,. \tag{3.5.15}$$

In words: the subspace $\mathcal{S}_{p,0}^0(\mathcal{M})$ of functions in $\mathcal{S}_p^0(\mathcal{M})$ that vanish on $\partial\Omega$ can be obtained by dropping all global shape functions associated with interpolation nodes on $\partial\Omega$.

Remark 3.5.16 ((Bi)-linear Lagrangian finite elements on hybrid meshes)

$\mathcal{M}$: 2D hybrid mesh comprising triangles & rectangles ▷



Idea: use
- ✦ linear functions (→ Def. 3.4.8, $p = 1$) on triangular cells,
- ✦ bi-linear functions (→ Def. 3.5.12, $p = 1$) on rectangles.

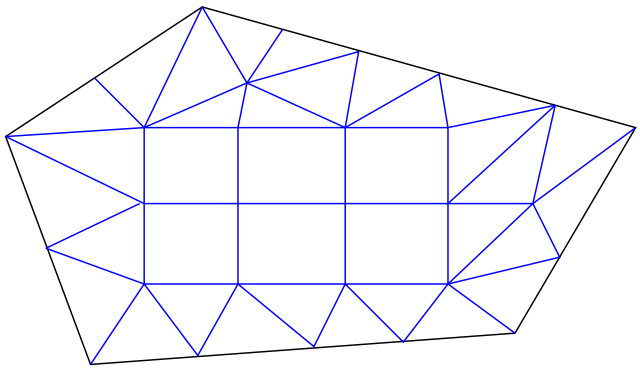


Fig. 139

$$\mathcal{S}_1^0(\mathcal{M}) = \left\{ v \in H^1(\Omega) \colon v_{|K} \in \begin{cases} \mathcal{P}_1(\mathbb{R}^2) & \text{, if } K \in \mathcal{M} \text{ is triangle,} \\ \mathcal{Q}_1(\mathbb{R}^2) & \text{, if } K \in \mathcal{M} \text{ is rectangle} \end{cases} \right\}. \tag{3.5.17}$$

Two issues arise:

1. Does the prescription (3.5.17) yield a large enough space? (Note that $v \in H^1(\Omega) \Rightarrow \mathcal{S}_1^0(\mathcal{M}) \subset C^0(\Omega)$, see Thm. 2.3.35, but continuity might enforce too many constraints.)
2. Does the space from (3.5.17) allow for locally supported basis functions associated with nodes of the mesh?

We wil give a positive answer to both question by constructing the basis functions:

Define global shape functions b_N^j according to (3.3.13)

This makes sense, because

- ✦ linear/bi-linear functions on K are uniquely determined by their values in the vertices,
- ✦ the restrictions to an edge of K of the local linear and bi-linear shape functions are both *linear* univariate functions, see Fig. 99 and Fig. 136.

▶ Fixing vertex values for $v_N \in \mathcal{S}_1^0(\mathcal{M})$ uniquely determines v on all edges of $\mathcal{M}$ already, thus, *ensuring global continuity*, which is necessary due to Thm. 2.3.35.

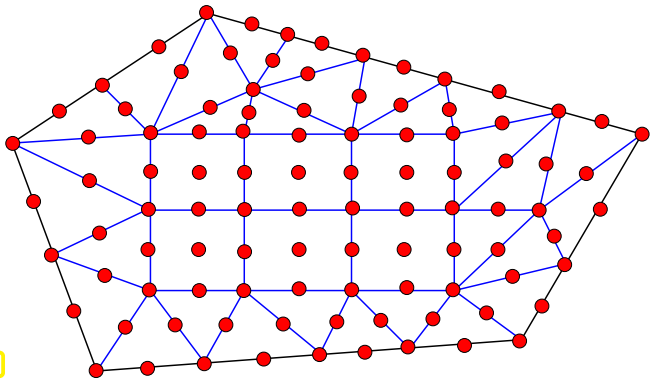
Remark 3.5.18 (Lagrangian finite elements on hybrid meshes)

$\mathcal{M}$: 2D hybrid mesh comprising triangles & rectangles

☞ Matching interpolation nodes on edges of triangles and rectangles

▶ Glueing of local shape functions on triangles and rectangles possible

gobal interpolation nodes for $p = 2$ ▷ Fig. 140



?! *Review question(s) 3.5.19.* (Lagrangian finite elements)

1. Explain why the local shape functions according to **??** and (**??**) remain valid local shape functions for the lowest degree Lagrangian finite element space on the hybrid mesh shown in Fig. 139.
2. Let $\mathcal{M}$ be a triangular mesh with $\sharp\mathcal{V}(\mathcal{M})$ vertices, $\sharp\mathcal{E}(\mathcal{M})$ edges, and $\sharp\mathcal{M}$ cells. What is $\dim \mathcal{S}_p^0(\mathcal{M})$ for $p = 1, 2, 3$?
3. For a triangular mesh $\mathcal{M}$ with $\sharp\mathcal{V}(\mathcal{M})$ vertices, $\sharp\mathcal{E}(\mathcal{M})$ edges, and $\sharp\mathcal{M}$ cells give sharp upper bounds for the number of non-zero entries of the Galerkin matrix arising from the finite element discretization of (3.1.4) with trial and test space $\mathcal{S}_p^0(\mathcal{M})$, $p = 1, 2$.
4. Consider a tensor product mesh $\mathcal{M}$ of $\Omega :=]0,1[^2$ and the space
$$V_{0,N} := \left\{ v \in H_0^1(\Omega) \colon v_{|K} \in \mathcal{P}_1(\mathbb{R}^2)\, \forall K \in \mathcal{M} \right\} .$$
What is the dimension of this space?
5. Let $\mathcal{M}$ be a tensor product mesh and $\widetilde{\mathcal{M}}$ a triangular mesh arising from $\mathcal{M}$ by splitting each rectangular cell into two congruent triangles. Show that $\mathcal{S}_1^0(\mathcal{M}) \neq \mathcal{S}_1^0(\widetilde{\mathcal{M}})$.
6. Express the local shape functions for linear Lagrangian finite elements on a triangle as linear combinations of the quadratic local shape functions as given in (3.5.6).

7. Characterize the space of gradients of $\mathcal{P}_p(\mathbb{R}^2)$ and $\mathcal{Q}_p(\mathbb{R}^2)$.

3.6 Implementation of Finite Element Methods

This section discusses algorithmic details of Galerkin finite element discretization of 2nd-order elliptic variational problems for spatial dimension $d = 2, 3$ on bounded polygonal/polyhedral domains $\Omega \subset \mathbb{R}^d$.

The guiding principle behind the implementation of finite element codes is

to rely on *local* computations as much as possible!

We witnessed this principle in action already in Section 3.3.5 ($\rightarrow$ Code 3.3.35) and Section 3.3.6 ($\rightarrow$ Code 3.3.47). Local computations are enough thanks to *local supports* of the global basis functions, see Section 3.4.3, Ex. 3.4.16.

Remark 3.6.1 (DUNE – Distributed and Unified Numerics Environment)

The finite element implementation part of this course is based on the concepts devised and realized in the context of DUNE:

The core idea behind DUNE is to use generic programming techniques available through the template facilities of C++ to specify interfaces between core modules of finite elememt codes.

From the DUNE website: "The underlying idea of DUNE is to create slim interfaces allowing an efficient use of legacy and/or new libraries. Modern C++ programming techniques enable very different implementations of the same concept (i.e. grids, solvers, ...) using a common interface at a very low overhead. Thus DUNE ensures efficiency in scientific computations and supports high-performance computing applications."

DUNE based codes are widely used for scientific simulations of complex PDE based models, see these examples. The DUNE interfaces have been designed with a focus on

- parallel implementation of finite element meshes on distributed memory multi-processor architectures,
- handling of hierarchical meshes created by local mesh refinement.

These aspects will not play a role in this course.

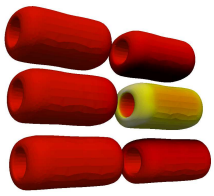
Basic DUNE implementations are open source and available under a GPL library license. It is possible to use the DUNE interface for any code.

Fig. 141

Background information and applications of DUNE are covered in [2, 3, 9] and many more references can be accessed through the DUNE publications page.

Remark 3.6.2 (BETL – a DUNE based finite element and boundary element code)

The implementation of finite element methods in this course will rely on the code suite **BETL** (**B**oundary **E**lement **T**emplate **L**ibrary) a DUNE compliant software package offering a framework for the implementation of finite element methods on a variety of 2D and 3D meshes.



BETL is an open source software and can be freely used for academic research and education. Some industrial companies rely on BETL for their in-house simulation code development.

The current version of this document often covers both the strictly DUNE-compliant implementation and its adaptation to the use of the BETL library. The former should be treated as legacy codes and the reader is advised to focus on the latter!

Remark 3.6.3 (Installation of BETL)

The BETL source distribution consists of two parts, the basis DUNE style interface definitions called ethGenericGrid and BETL proper. The installation of both employs the tool **cmake** in order to achieve portability between various operating systems (Linux, Mac OS X, Windows). The operation of **cmake** is controlled by CMakeLists.txt-files providing a bare minimum of information about the code to be built. **cmake** supplements suitable compiler and linker flags fitting the current operating system and build environment. Essentially it creates a Makefile containing all that information, which passes it to the UNIX make utility.

For the sake of simplicity, the installation of both libraries (ethGenericGrid and Betl2) are managed by the **cmake** file in the lectures Gitlab repository ➙ GITLAB

In order to make it work, please perform the following steps:

1. Check out the BETL submodule

   ⚠ You need to have a Gitlab account in https://gitlab.math.ethz.ch and to be registered in the lecture in order to access this submodule!
   You will need to type in your NETHZ username and password.

- If you are cloning the lecture repository for the first time, please type the following shell commands

```
git clone --recursive https://gitlab.math.ethz.ch/NumPDE/NumPDE.git
cd NumPDE
mkdir build
```

- If you have already cloned the lecture repository, you need to update it such that it includes the submodules. Asumming you are in the root folder in the folder NumPDE, this is done by typing

```
cd third_party/Betl2
git submodule init
git submodule update --init --recursive
```

2. Make sure the library boost is installed in your system
3. Compile the Gitlab repository as usual, i.e., type

```
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

This will handle the installation of ethGenericGrid and BETL. You may find some troubleshooting instructions in the repository readme.

Remark 3.6.4 (Learning BETL)

As of now August 16, 2016, BETL's documentation is still rudimentary, a shortcoming it has in common with many simulation codes. These course notes attempt to fill this gap for some parts of BETL. To learn the finite element functionality of BETL you can rely on three resources:

- commented listings in this lecture document and the accompanying explanations,
- example codes in the lecture's Gitlab repository ➙ GITLAB, many containing code snippets discussed in these lecture notes,
- BETL based implementation tasks submitted as homework problems and coming with detailed solutions.

A DOXYGENdocumentation of BETL is work in progress.

In order to learn the finite element capabilities of BETL you may follow these steps:

(I) Start from a small code that reads a mesh file and creates a BETL internal mesh data structure, see Code 3.6.18.

(II) Learn from Ex. 3.6.27 about ways how to traverse the geometric entities of a mesh (→ Code 3.6.29). You should know about the `GridView` concept and indexing beforehand, see § 3.6.24.

(III) Become familiar with how to access geometric entities locally (→ Code 3.6.46) and retrieve geometric information (→ Ex. 3.6.52). Read Ex. 3.6.31, Rem. 3.6.48, and § 3.6.49 before, in order to understand fundamental notions and conventions.

(IV) After you have completely grasped the algorithmic ideas underlying *cell-oriented assembly* (→ Section 3.6.4.1, § 3.6.71, Section 3.6.4.2), study the use `FESpace` objects: look at Ex. 3.6.86, Code 3.6.87 first and also consult the explanations about the facilities provided by `FESpace` given in § 3.6.83.

(V) From Code 3.6.84 and the explanations in § 3.6.75 get an idea how to use BETL's built-in Lagrangian finite element spaces, see also the first part of Code 3.7.14.

(VI) Study Ex. 3.6.94 and § 3.6.107 that will tell you how to build the sparse linear system of equations arising from Galerkin finite element discretization.

(VII) When you have understood transformation based *local quadrature* you may look at its use in BETL, see § 3.6.164.

(VIII) Once you have grasped § 3.6.177 examine Ex. 3.6.181 for the treatment of *essential boundary conditions* in BETL. Familiarity with `FESpace` is essential.

(IX) § 3.6.139 will tell you how to access information crucial for the use of *transformation techniques* in BETL.

(X) Learn how to specify local shape functions relying on the `FEBasis` concept as introduced in § 3.6.75, Ex. 3.7.13 and demonstrated in Ex. 3.6.76, Code 3.7.14. Before you look at these paragraphs, you have to understand the paradigm of *parametric finite elements* (→ Section 3.7).

(XI) From Ex. 3.7.33 learn the BETL implementation of transformation techniques for local computations. This requires familiarity with local quadrature and `FEBasis`.

Remark 3.6.5 (LehrFEM – a MATLAB finite element code)

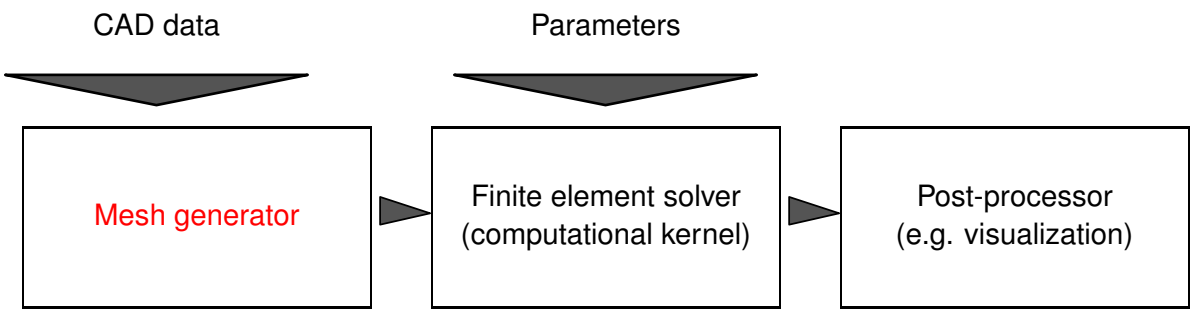
Earlier version of this course relied on the LehrFEM finite element MATLAB library implementing data structures and algorithms for 2D finite elements on triangular meshes. A detailed documentation is available from [6].

Other MATLAB based finite element programming environments are iFEM and the codes presented in [1, 11],

---

3.6.1 Mesh generation and mesh file format

In Section 3.4.1 we identified triangulations (→ Def. 3.4.2) as one of the main building blocks of finite element methods. Their algorithmic generation turns out to be a separate issue, because the data flow in (most) finite element software packages look like this:



Here "▶" designates passing of information, which is usually done by writing and reading files to and from hard disk. This requires particular file formats.

Algorithms for generating a finite element mesh from some description of the geometry of the computational domain are *beyond the scope of this course*. Sophisticated methods have been developed over

many years and they are implemented in powerful commercial software packages. The problem of generating "suitable" finite element meshes without user interference is a persistent research topic, because complex geometries (slender domain, multiple length scales, layered media, etc.) entail immense challenges.

Remark 3.6.6 (Gmsh – geometric modeling and mesh generation tool)

We use the open source public domain geometric modeler and mesh generator **Gmsh** (pronounced "G-mesh"), which is employed in many projects in academic and industrial research.

Gmsh has been and is being developed by Prof. Ch. Geuzaine and Prof. J.-F. Remacle at the University of Liége in Belgium. Code and documentation are available through

`http://geuz.org/gmsh/`.

Gmsh graphical user interface (GUI) ▷

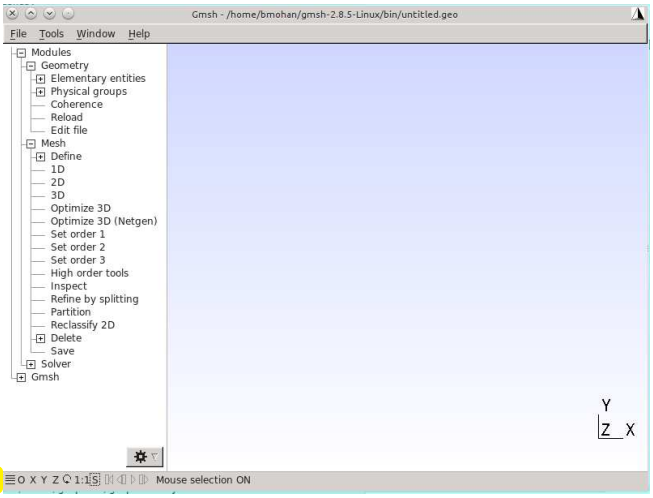


Fig. 142

Example 3.6.7 (Geometric modeling with Gmsh)

In this example we define a simple geometry interactively using the **Gmsh** geometric modeling interface. We specify *Points, Lines* and *Surfaces* that define our computational domain, the unit square $\Omega =]0,1[^2$.

❶ **Setting points.** Select the menu item

`Modules -> Geometry -> Elementary entities -> Add -> Point.`

You can start adding points by interactively clicking, holding the position of the mouse, and pressing 'e'. The coordinates of your mouse pointer are reflected on the `Contextual Geometry Defintions` window that appears. It is, however, advisable to use this window and manually enter the coordinates of the points you want to create in the X, Y and Z coordinate text boxes.

Start by adding your first point with coordinates $(0,0,0)$ in the coordinate field. After you press return, one point should appear on your canvas. Similarly, add points $(1,0,0)$, $(1,1,0)$, and $(0,1,0)$. This sets all four corners of the square.

❷ **Defining lines.** To add the lines that form the edges of our square, use the menu

`Modules -> Geometry -> Elementary entities -> Add -> Straight Line.`

Now, select the point $(0,0)$ as starting point. The selected point will be show in red. Then, complete the line by selecting $(1,0)$ as the end point. Similarly, create three other lines, forming a square. See the figure beside.

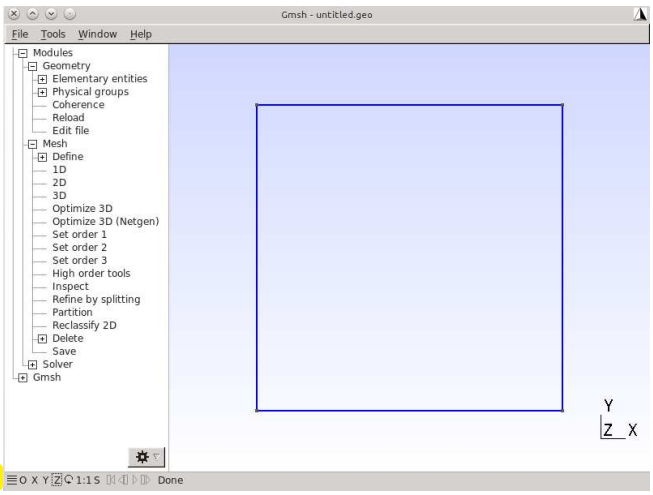


Fig. 143

❸ **Creating surfaces (domains).** To finish the definition of a computational domain, we have to tell **Gmsh** which of closed line loops form a surface. This can be easily done by using the menu

`Modules -> Geometry -> Elementary entities -> Add -> Plane Surface.`

Then click on any part of the square. After the boundary has been selected, press `'e'`, to create a surface. See the figure beside for a screenshot.

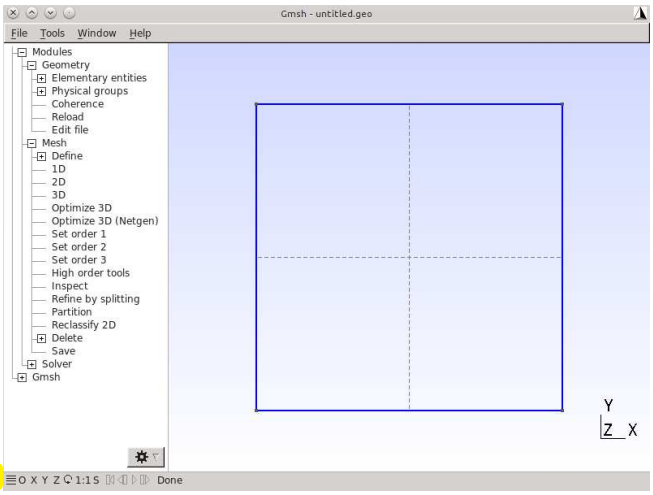


Fig. 144

Example 3.6.8 (Gmsh geometry description file)

When geometric elements are created interactively using the GUI, **gmsh** stores the data in a `.geo` file, with its own scripting language. This file can be opened, and edited by the menu

`Modules -> Geometry -> Edit File`

The .geo file for the square mesh reads:

```
Point(1) = {0, 0, 0, 1};
Point(2) = {1, 0, 0, 1};
Point(3) = {1, 1, 0, 1};
Point(4) = {0, 1, 0, 1};
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};
Line Loop(5) = {1, 2, 3, 4};
Plane Surface(6) = {5};
```

The line numbers 1-4 in the above code define `Points` 1-4, with the following syntax,

```
Point(id) = {x, y, z, mesh-size};
```

Similarly, line numbers 5-8 specify the four edges of the square, as

```
Line(id) = {id-of-start-point, id-of-end-point};
```

Defining a closed polygon (line loop) also follows a similar syntax, but you have to make sure that lines are in proper cyclic order. To invert the direction of a line, use a minus sign before line id.

```
Line Loop(id) = {id-of-line -1, id-of-line -2, id-of-line -3, ...};
```

The final line of the code defines surface that is created with the line loop as the boundary, and is defined as follows.

```
Plane Surface(id) = {id-of-line-loop, id-of-holes-loop};
```

(3.6.9) Generating a mesh with Gmsh

After the geometry has been specified or a `.geo` file has been read in, a mesh can be generated for currently active domain (surface). Click the menu

`Modules -> Mesh -> 2D.`

Then **Gmsh** should display an unstructured mesh for the square surface, see the figure beside for a mesh covering the square domain created in Ex. 3.6.7.

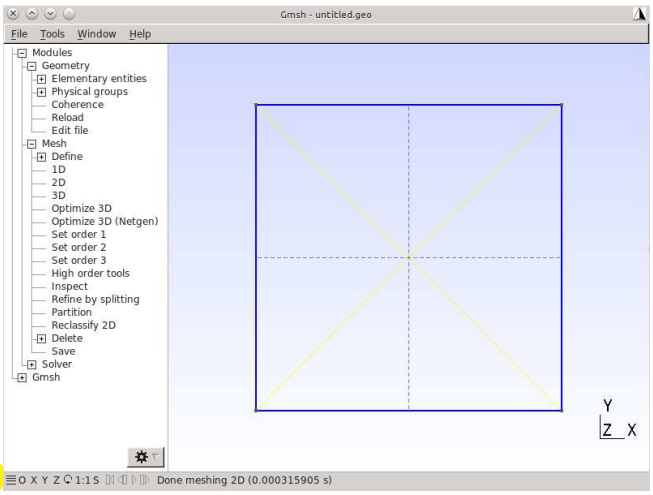


Fig. 145

When point are added (*to* Ex. 3.6.7) in the dialogue there is text box for `Prescribed mesh element size at point`. This can be used to define the size of the meshes around a particular point. In this example this was set to 1. In general, this parameter can be used to control the local resolution of the mesh.

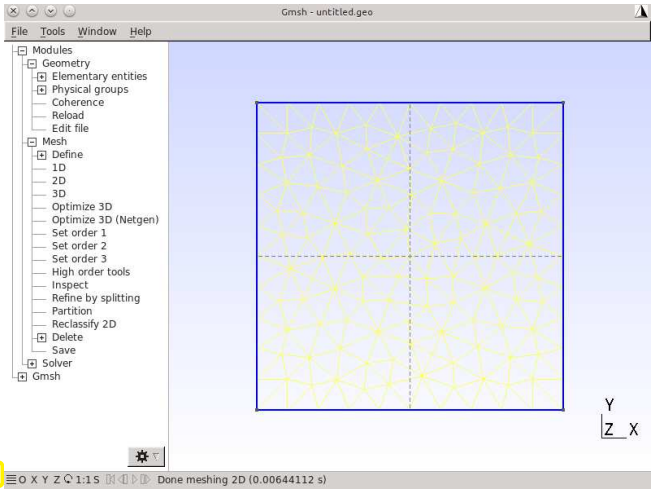


Fig. 146

To create a finer mesh, edit the `.geo`-file and specify a smaller local mesh size for the points (→ Ex. 3.6.8). Do not forget a subsequent click on `Modules -> Geometry -> Reload`.

The figure beside displays a mesh for the square generated with local mesh size 0.1.

Example 3.6.10 (Gmsh file format for storing meshes)

Gmsh stores mesh data in plain ASCI `.msh`-files. The file corresponding to the mesh from Fig. 145 is as follows ▷

```
1  $MeshFormat
2  2.2 0 8
3  $EndMeshFormat
4  $Nodes
5  5
6  1 0 0 0
7  2 1 0 0
8  3 1 1 0
9  4 0 1 0
10 5 0.5 0.5 0
11 $EndNodes
12 $Elements
13 12
14 1 15 2 0 1 1
15 2 15 2 0 2 2
16 3 15 2 0 3 3
17 4 15 2 0 4 4
18 5 1 2 0 1 1 2
19 6 1 2 0 2 2 3
20 7 1 2 0 3 3 4
21 8 1 2 0 4 4 1
22 9 2 2 0 6 1 2 5
23 10 2 2 0 6 1 5 4
24 11 2 2 0 6 2 3 5
25 12 2 2 0 6 3 4 5
26 $EndElements
```

Lines 1-3: Version number, file format, and floating point format used.

Line 4-11 (between `$Nodes` and `$EndNodes`): List of nodes
The first line in the nodes section gives the number of nodes in the mesh, followed by each of the nodes. In our case, the mesh comprises 5 nodes. In each node line, the first integer describes the id(entifier) of the entity in the `.msh`-file, followed by the x-, y- and z- coordinates of the node (floating point numbers). In the example, the four points we created are part of the mesh, and **Gmsh** has created a new fifth node in the center of the mesh at coordinates $(0.5, 0.5, 0)$.

Line 12-26 (between `$Elements` and `$EndElements`): List of elements and boundary entities

Some entities, such as points, lines, triangles, quadrangles, etc., are coded in this section. The first line (line number 13) gives the number of entities listed. In each entity line, the integer denotes the entity id(entifier), followed by an identifier for a *type* of the entity. The third integer denotes the number of tags for this entity, followed by that many integers (tags). The meaning of the tags will be covered in Rem. 3.6.12. The remainder of the line lists the id(entifier)s of nodes which are contained in the boundary of this particular entity.

(3.6.11) Gmsh Element Types

A selection of entity types used by **Gmsh** for 2D meshes

The meaning of "3-node line" and "6-node triangle will be explained in 3.7.41.

Number	Element Type
15	1-node point
1	2-node line
2	3-node triangle
3	4-node quadrilateral
8	3-node line
9	6-node triangle

For example, line number 14 describes an entity with identifier 1, and the element type of 15, which is a 1-node point. This entity has 2 tags (for now ignore the following 2 integers). The last integer, 1, corresponds to the node identifier which is a part of the entity. The node identifier 1 is the point that is located at $(0, 0, 0)$. Hence the entity 1, correspondes to the node 1. The entities 2, 3, and 4 are also of type "1-node point",

and represent the points $(1,0,0)$, $(1,1,0)$ and $(0,1,0)$, respectively.

The line numbers 18-21 code for entities with identifiers 5-8, and represent entities of type 1, which is a 2-node line. Ignoring the tags, the last two integers represent the nodes corresponding to the endpoints of the lines. Their order endows the line with a direction. For instance, entity number 7 represents a line between the nodes 3 and 4. Hence, this is a line between the Points $(1,1,0)$ and $(0,1,0)$.

The element numbers 9-12 represent, the element of type 2, which are 3-node triangles. The last three integers give the node numbers of the vertices. For example, the element number 9, is a triangle with vertices $(0,0,0)$, $(1,0,0)$ and $(0.5,0.5,0)$.

! Note that only the points and lines (edges) which are a part of the boundary are included as separate entities; interior edges and points are not.

Remark 3.6.12 (Gmsh – marking parts of a mesh by tags)

Often one wants to distinguish parts of the computational domain (sub-domains), where special coefficient functions or source functions should be used. Moreover, parts of the boundary have to be marked, if they carry different boundary conditions as in Ex. 2.7.8. In **Gmsh** this can be achieved by assigning mesh entities to different physical groups. Those can be created using the menu item

`Modules -> Geometry -> Physical Groups -> Add`

Physical groups are distinguished by their name and the `.geo`-file for the square extended by physical groups may look as follows. ▷

```
Point(1) = {0, 0, 0, 1};
Point(2) = {1, 0, 0, 1};
Point(3) = {1, 1, 0, 1};
Point(4) = {0, 1, 0, 1};
Physical Point("bottom-pts") = {1, 2};
Line(1) = {1, 2};
Line(2) = {2, 3};
Line(3) = {3, 4};
Line(4) = {4, 1};
Physical Line("top-bottom") = {1, 3};
Physical Line("left-right") = {2, 4};
Line Loop(5) = {1, 2, 3, 4};
Plane Surface(6) = {5};
Physical Surface("thesurface") = {6};
```

Gmsh manages physical groups using the tags, whose discussion we skipped in Ex. 3.6.10. For instance, the `.msh`-file generated from the above `.geo`-file is printed beside. ▷

```
1  $MeshFormat
2  2.2 0 8
3  $EndMeshFormat
4  $PhysicalNames
5  4
6  0 1 "bottom-pts"
7  1 2 "top-bottom"
8  1 3 "left-right"
9  2 4 "thesurface"
10 $EndPhysicalNames
11 $Nodes
12 5
13 1 0 0 0
14 2 1 0 0
15 3 1 1 0
16 4 0 1 0
17 5 0.5 0.5 0
18 $EndNodes
19 $Elements
20 10
21 1 15 2 1 1 1
22 2 15 2 1 2 2
23 3 1 2 2 1 1 2
24 4 1 2 3 2 2 3
25 5 1 2 2 3 3 4
26 6 1 2 3 4 4 1
27 7 2 2 4 6 1 2 5
28 8 2 2 4 6 1 5 4
29 9 2 2 4 6 2 3 5
30 10 2 2 4 6 3 4 5
31 $EndElements
```

The `.msh` file has changed; a new section `$PhysicalNames` has been created which tabulates the keys for looking up the physical groups the entities belong to. Line 5 gives the number of physical groups. Each line of this section has the following format: the first integer defines the dimension of the entities of the group (0 for points, 1 for lines and 2 for surfaces). The second integer gives the tag number for the physical group, and third its corresponding string identifier.

Comparing with Ex. 3.6.10, we can see that the tag numbers of elements have changed. The first tag denotes the physical group an entity belongs to, and the second tag represents the geometric entity it belongs to. The groupings can be read off from the tags of the entities and the corresponding name of the physical group. For example, the elements 1 and 2 belong to the physical group 1, which is "bottom-pts". The elements 3 and 5 that are lines have the physical tag 2, which corresponds to the group "top-bottom". Similarly, entities 4 and 6 belong to "left-right". The remaining entities that are triangles belong to the physical group "thesurface".

Example 3.6.13 (Gmsh – meshing more complex geometries)

Curved boundaries can also be modelled in **Gmsh**. Refer to the documentation for details.

```
Point(1) = {0, 0, 0, 0.25};
Point(2) = {1, 0, 0, 0.25};
Point(3) = {0, 2, 0, 0.25};
Point(4) = {1, 2, 0, 0.25};
Point(5) = {2, 2, 0, 0.25};
Point(6) = {2, 3, 0, 0.25};
Point(7) = {2, 4, 0, 0.25};
Point(8) = {4, 4, 0, 0.25};
Point(9) = {4, 3, 0, 0.25};
Line(1) = {3, 1};
Line(2) = {1, 2};
Line(3) = {2, 4};
Line(4) = {6, 9};
Line(5) = {9, 8};
Line(6) = {8, 7};
Circle(7) = {4, 5, 6};
Circle(8) = {3, 5, 7};
Line Loop(9) = {1, 2, 3, 4, 5, 6, 7, -8};
Plane Surface(10) = {9};
```

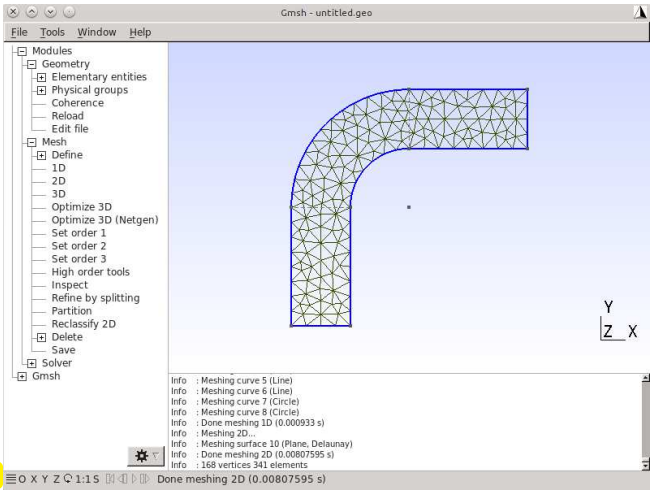


Fig. 147

Remark 3.6.14 (Other tools for mesh generation)

Freely available mesh generators:

- DistMesh (MATLAB, used in "LehrFEM", see [6, Sect. 1.2])
- Triangle (easy to use 2D mesh generator)
- TETGEN (Tetrahedral mesh generation)
- NETGEN (industrial strength open source mesh generator)

Example 3.6.15 (DUNE - building mesh from Gmsh mesh file)

When using the DUNE library, one has to make a decision which of the available seven grid implementations to use. The following code makes use of the ALUGrid DUNE module [8] to handle conforming two-dimensional simplicial meshes.

C++11 code 3.6.16: DUNE code: reading a `.msh`-file and building a mesh from it

```
  // Includes skipped ....
int main(int argc, char *argv[]){
  try {
    // Get mesh file name from command line arguments
    const string FileName = argv[1];

    // Initialize triangular mesh reading a Gmsh mesh file
    using GridType = Dune::ALUSimplexGrid<2,2>;
    Dune::GridFactory<GridType> gridFactory;
```

```
10      Dune::GmshReader<GridType>::read(gridFactory,FileName.c_str(),
11                                       false,true);
12      GridType *workingGrid = gridFactory.createGrid();
13      workingGrid->loadBalance(); // undocumented internal setup
14
15      // Get the grid view
16      using GridView = GridType::LeafGridView;
17      GridView gv    = workingGrid->leafGridView();
18    }
19  catch (Dune::Exception &e) {
20    cerr << "Dune reported error: " << e << endl;
21    }}
```

This example uses the DUNE mesh implementation ALUGrid, selected by instantiating the corresponding template in Line 8 of Code 3.6.16. The actual mesh data structure is initialized by a `gridFactory` object, which reads the data from a `.msh`-file. Eventually a reference to the mesh of type GridView is created and stored in the variable `gv` in Line 17.

Example 3.6.17 (BETL - building mesh from Gmsh mesh file)

BETL offers rather advanced facilities for parsing Gmsh mesh files (suffix .msh) and building mesh data structures from the information contained in them. The following code reads the data for a 2D hybrid mesh from file.

C++11 code 3.6.18: BETL code reading 2D hybrid mesh from a `.msh`-file ➙ GITLAB

```
// wrapper for the input stream amenable to the mesh file parser
const string basename( "meshfile" ); // omit suffix .msh!
using input_t        = betl2::input::gmsh::Input;
using inpInterface_t = betl2::input::InputInterface<input_t>;
input_t input(basename); inpInterface_t inpInterface(input);

// Define the grid type:  we chose a hybrid 2D grid → § 3.4.4.
using grid_t = betl2::volume2dGrid::hybrid::Grid;

// Focus on a single refinement level provided by leafView.
const eth::grid::GridViewTypes view =
    eth::grid::GridViewTypes::LeafView;
using gridView_t = typename eth::grid::GridView<
      grid_t::gridTraits_t::template viewTraits_t<view> >;
// Basic information about the grid can be accessed through grid traits
using gridTraits_t = gridView_t::gridTraits_t;

// Dynamically allocate an instance of Grid, accessible through
// a pointer.  We use a shared pointer that will trigger an automatic
// delete when it reaches the end of its lifetime, see documentation.
using grid_ptr_t = shared_ptr<
                   eth::grid::Grid<grid_t::gridTraits_t> > ;
```

```
22 grid_ptr_t grid_ptr(new grid_t(inpInterface));
23
24 // A grid factory object actually builds the internal
25 // mesh data structure.
26 using gridFactory_t = eth::grids::utils::GridViewFactory<grid_t,view>;
27 const gridFactory_t gridFactory(grid_ptr);
28
29 // Access to a grid is channelled through a gridView object.
30 // Note that this is a const data type, which rules
31 // out altering the mesh
32 const gridView_t gridView = gridFactory.getView();
33
34 // Fetch dimension of ambient space and grid, which may differ for
     triangulated
35 // surfaces for instance
36 const int worlddim = gridTraits_t::dimWorld;
37 const int griddim = gridTraits_t::dimMesh;
38
39 // get the size of the entity container of a specified co-dimension
40 const int numNodes    = gridView.size(2);  // vertices ↔ co-dim.  2
41 const int numEdges    = gridView.size(1);  // edges ↔ co-dim.  1
42 const int numElements = gridView.size(0);  // cells ↔ co-dim.  0
```

The grid allocation (Line 20-Line 27) can be simplified by calling the instance `betl2::GridCreator`. For this, Line 15-Line 27 can be replaced by

```
using gridCreator_t = betl2::GridCreator<grid_t,view>;
using gridFactory_t = gridCreator_t::gridFactory_t;
const gridFactory_t gridFactory = gridCreator_t()(input);
```

- Line 8: We select the type of the mesh to be read, here a 2D hybrid mesh, see § 3.4.4.
- Line 15: Through traits static properties of a **GridView** object can be accessed. For instance, in Line 27 and Line 31 we fetch relevant dimensions, which should both be 2 in this case.
- Line 22: Here an empty grid object is created dynamically and a pointer to it is stored.
- Line 27: A BETL internal mesh data structure is initialized according to the data in the input buffer.
- Line 32 The constant reference to a **GridView** object allows access to all data connected with the grid, see § 3.6.24.

(The **GridView** interface is defined in `grid_view.hpp` in namespace `eth::grid`.)

Example 3.6.19 (Processing extra information in Gmsh mesh file with BETL)

In Rem. 3.6.12 we saw that geometric mesh entities can be endowed with special "physical groups" tags in Gmsh. These are automatically extracted by the BETL Gmsh reader and is managed via the **InputInterface** interface in `betl2::input` that was created in Ex. 3.6.17. After the creation of the grid object (see Ex. 3.6.17), we additionally introduce a structure that manages the "physical groups" of Gmsh elements. It is given through the class **GridElementsIdentifier** that is defined in `grid_elements_identifier.hpp` in `betl2::input::gmsh`.

C++11 code 3.6.20: Reading Gmsh's physical groups with BETL ➔ GITLAB

```
// define type shorthands and get the index set
const eth::grid::GridViewTypes view =
    eth::grid::GridViewTypes::LeafView;
using gridView_t = typename eth::grid::GridView<
    grid_t::gridTraits_t::template viewTraits_t<view> >;
const gridView_t& gridView = gridFactory.getView();
auto& set = gridView.indexSet(); // See § 3.6.41.

using idx_t = unsigned int; // type for indices
typedef set< idx_t > tagSet;   // type for collection of tags

// First check what physical tags are associated with the mesh,
// and what Gmsh element co-dimension they are associated to
// initialize a set object containing all tags for Gmsh elements
// of a given co-dimension, which is used as array index.
array<tagSet,gridFactory_t::gridTraits_t::dimMesh+1> codim2tags;
for( const auto elType : inpInterface.getElementTypes() ){
  // Retrieve the co-dimension of the Gmsh element type
  const auto refElType  = ElementTypeInfo::getRefElType( elType );
  const idx_t elCodim = gridFactory_t::gridTraits_t::dimMesh -
      eth::base::ReferenceElements::getDimension(refElType);
  // Loop over the Gmsh elements of that particular type
  for( auto ellter = inpInterface.begin( elType );
       ellter != inpInterface.end( elType ); ellter++ ){
      // Collect and store the Gmsh element physical tag
      const idx_t tagID =
          (inpInterface.getElementTags(*ellter)).at(0);
      // insert in set, which automatically weeds out duplicates
      codim2tags[elCodim].insert(tagID);
    }
}

// instantiate class to recover tagged subsets of entities
betl2::input::gmsh::GridElementsIdentifier< inpInterface_t ,
    gridFactory_t >
      gridMarker( inpInterface, gridFactory );

// Print information on physical tags of co-dimension 0 entities
    (cells)
// Meaning and use of entity indices are explained in § 3.6.41.
for( const idx_t tag : codim2tags[0]) {
  cout<< "Grid idx of physical entities of CODIM 0 with tag " << tag
      << ": " << endl;
  const auto &taggedEntities = gridMarker.template
      retrieveEntities<0>( tag );
  for( const auto &entity : taggedEntities )
    cout<< set.index(*entity) << "(" << entity->refElType() << ") ";
  cout << endl;
}
```

```
42
43 // Scan entities of co-dimension 1 (edges)
44 // Those are stored as intersections, not entities, see § 3.6.56!
45 for( const idx_t tag : codim2tags[1]) {
46   cout<< "Grid idx of physical entity of CODIM 1 with tag " << tag <<
       ": " << endl;
47   const auto& taggedInt = gridMarker.template
       retrieveEntities<1>(tag);
48   for( const auto& inters : taggedInt ){
49     // get pointer to element (el)
50     const auto& in = inters->inside();
51     // get local index of this side (wrt el)
52     const auto inLclIdx = inters->indexInInside();
53     // get global index of this side (a bit convoluted,
54     // since set maps entities and not intersections)
55     const auto glbIdx = set.index( *( in->template subEntity<1>(
         inLclIdx ) ) );
56     cout<< glbIdx << " (" << inters->geometry().refElType() << ")  ";
57   }
58   cout<< endl;
59 }
60
61 // scan entities of codim 2 (vertices)
62 for( const idx_t tag : codim2tags[2]){
63   cout<< "Grid idx of physical entities of CODIM 2 with tag " << tag
       << ": " << endl;
64   const auto& taggedEntities = gridMarker.template
       retrieveEntities<2>(tag);
65   for( const auto& entity : taggedEntities )
66     cout << set.index(*entity) << " (" << entity->refElType() << ")
         ";
67   cout<< endl;
68 }
```

- Line 8: To store the occuring tags, we use the `set` data structure.
- Line 14: The tags that occur for Gmsh elements of a given codimension are stored in an `array`.
- Line 14-Line 27: We fill the array by iterating over all Gmsh elements that are present in the mesh (note that a Gmsh element can also be a entity of codimension bigger than 0, see § 3.6.11), storing its "physical groups" tag in the array at the Gmsh elements codimension.
- Line 31: Here the object that finally manages the "physical groups" of Gmsh elements is instantiated.
- Line 37, Line 47, Line 64: Show how to access the Gmsh elements of a given codimension that are part of the "physical group" associated with the tag `tag`.

The above code is also included in the header file `lecture_codes/FEMwithBETL/topology/NPDE_topology_functions.hpp`, in function `printPh`
It is accessed via the following function call.

C++11 code 3.6.21: Function call for output of physical tags ➙ GITLAB

```
// printing the information about Gmsh physical sets:
betl2::NPDE::printPhysicalSetsInfo(inpInterface,gridFactory);
```

A minimal working example related to Code 3.6.20 is available in ➙ GITLAB.

3.6.2 Mesh data structures

Topic: internal *representation* of mesh (→ Def. 3.4.2) in computer code and definition of suitable programming *interface*.

Purposes of mesh data structures

mesh data structures must

1. offer unique identification of cells/(faces)/(edges)/vertices (for instance, by an integer index)
2. make possible traversal of cells of the mesh (→ global numbering)
3. represent mesh topology (= incidence relationships of cells/faces/edges/vertices)
4. allow sequential access to edges/faces of a cell (→ traversal of local shape functions/degrees of freedom)
5. describe mesh geometry (= location/shape of cells/faces/edges/vertices)

Two kinds of objects can be distinguished:

Global objects	↔	*Local objects*
• single instance exists • container for geometric entities		• many instances exist • store geometry/topology

(3.6.23) Importance of global numbering of geometric entities

Remember from Section 3.2: we need on *ordered* basis $\mathfrak{B}$ of the finite element space, that is, we have to establish a consecutive numbering of the finite element basis functions/global shape functions, $\mathfrak{B} = \{b_N^1, \dots, b_N^N\}$.

For assembly as explained in Section 3.3.5 we also assumed that the local shape functions carried numbers, there corresponding to the (local) numbers of the vertices of each triangle of the mesh. Thus, in a code using linear Lagrangian finite elements, we have to number the vertices of a mesh.

More generally, in Section 3.4.3 we saw that global shape functions are associated with geometric entities.

➢ Numbering geometric entities paves the way for numbering global shape functions.

(3.6.24) BETL grid view concept

In DUNE the basic device to handle global aspects of a mesh is the GridView concept, see documentation. It has been modified slightly in BETL. The basic facilities offered by a **GridView** object are

- traits data types (**GridTraits** & **ViewTraits**) for accessing static data and data types for (pointers to) geometric entities (distinguished by co-dimension → § 3.4.3),
- access to sequential containers for geometric entities of the mesh through the `entities<codim>()` method.
- `size(int codim)` method returning the total number of geometric entities of a particular co-dimension,
- indexing (numbering) of geometric entities through consecutive integers provides by an **IndexSet** sub-object, which can be obtained through the `indexSet()` method.

Refer to Code 3.6.18 (Line 32)to learn how to obtain a reference to an object of `GridView` type.

`GridView` objects are *constant*; they do not allow modification of the mesh!

Example 3.6.25 (Using entity iterators of a DUNE GridView)

C++11 code 3.6.26: Looping over entities of a DUNE grid of a particular co-dimension

```
1  // Function template parameter k specifies co-dimension
2  template<class GridView,int k>
3  int traverseEntities(const GridView &gv) {
4    using EntityIterator=typename GridView::template Codim<k>::Iterator;
5    using IndexSetRef=typename GridView::IndexSet const &;
6    // The indexSet manages unique and consecutive integer entity indices
7    IndexSetRef set(gv.indexSet());
8
9    int cnt = 0;
10   cout << "Grid dimension = " << GridView::dimension << ", ";
11   cout << "iterating over entities of codimension " << k << ", ";
12   cout << gv.size(k) << " exist" << endl;
13   for(EntityIterator it=gv.template begin<k>();
14       it != gv.template end<k>(); ++it) {
15     cout << "object " << cnt;
16     cout << ": id = " << set.index(*it) << endl;
17     cnt++;
18   }
19 }
```

In Code 3.6.26 the `EntityIterator` behaves like a pointer to an object of type Entity. In Line 16 an iterator is dereferenced, producing an `Entity` object that is passed to the `index` method of the `IndexSet`, which returns the global index of that object, see § 3.6.41 below.

Example 3.6.27 (Using entity iterators in BETL)

In BETL an instance of **GridView** owns a method `entities<k>()`, with `k` a cardinal template parameter that passes the co-dimension, that returns a reference to an **EntityCollection**, which serves as a sequential container for the entities of a particular co-dimension.

The following Code 3.6.28 demonstrates sequential access to all entities of a **GridView** object of a particular co-dimension, which is passed as a template parameter

C++11 code 3.6.28: Looping over entities of a particular co-dimension in BETL grid ➙ GITLAB

```
template<int k,class VIEW_TRAITS>
int traverseEntities(const eth::grid::GridView<VIEW_TRAITS> &gv) {
  // type of current GridView object contained in gv
  using gridView_t = eth::grid::GridView<VIEW_TRAITS>;
  // GridTraits and ViewTraits of current GridView
  using gridTraits_t = typename gridView_t::gridTraits_t;
  using viewTraits_t = typename gridView_t::viewTraits_t;
  // iterator type for entities of co-dimension k
  using entityIteratorImpl_t = typename viewTraits_t::template
      entityIterator_t<k>;
  using entityIterator_t = eth::grid::EntityIterator<
      gridTraits_t,entityIteratorImpl_t >;
  // An EntityCollection is a sequential container of entities
  using entityCollection_t =
      eth::grid::EntityCollection<entityIterator_t>;
  // An IndexSet manages unique consecutive indices for the entities
  // of a GridView object
  using indexSet_t = eth::grid::IndexSet<gridTraits_t,typename
      viewTraits_t::indexSet_t>;
  // References to IndexSet objects are not mutable
  using indexSetRef_t = indexSet_t const &;

  // The indexSet method of a GridView object returns a reference
  // to its associated index set, see § 3.6.41.
  indexSetRef_t set(gv.indexSet());

  int cnt = 0;
  cout << "Grid dimension = " << gridTraits_t::dimMesh << ", ";
  cout << "iterating over entities of codimension " << k << ", ";
  cout << gv.size(k) << " exist" << endl;
  // Obtain reference to a sequential container for entities of
      codimension k
  // by calling the entities<k> method.
  entityCollection_t entityCollection = gv.template entities<k>();
  // Standard loop for visiting elements of a sequential container.
  for(entityIterator_t it=entityCollection.begin();
      it != entityCollection.end(); ++it) {
    cout << "object " << cnt;
    // Retrieve global index of current entity through index
    // method of IndexSet.
    cout << ": id = " << set.index(*it) << endl;
    cnt++;
  }
```

```
39    return cnt;
40  }
```

Obtaining the right types for entities and iterators is cumbersome and error prone. Thus the `auto` facility of C++11 comes very hand, which lets the compiler determine the type of most objects. The next codes relies on `auto` as does the same as Code 3.6.28.

C++11 code 3.6.29: Looping over entities of a particular co-dimension in BETL grid, version with automatic type deduction ➙ GITLAB

```
1  template<int k,class VIEW_TRAITS>
2  int traverseEntities_auto(
3      const eth::grid::GridView<VIEW_TRAITS> &gv) {
4    // Fetch reference to the index set for current GridView object
5    // referenced by gv.
6    auto& set = gv.indexSet();
7    int cnt = 0;
8    cout << "BETL: No of entities of codimension " << k << " = "
9           << gv.size(k) << endl;
10   // range based loop running through all entities of co-dimension k
11   for(auto& it : gv.template entities<k>()) {
12     cout << "object " << cnt << ": id = " << set.index(it) << endl;
13     cnt++;
14   }
15   return cnt;
16 }
```

Note the difference between the loop variables in the loops implemented in Code 3.6.28 (Line 32) and Code 3.6.28 (Line 11), respectively. In the former code, `it` is a pointer to an entity, whereas in he latter it provides a constant reference. This reference can simply be passed to the `index` method in Line 12 of Code 3.6.29.

(3.6.30) Representation of mesh topology

When we talk about the "topology" of a mesh, we ignore the location and shape of entities and focus on how those are connected, that is we are interested in the

incidence relations:
- ✦ "boundary contains"-relation: boundary of an entity of a higher dimension contains entity of a lower dimension
- ✦ "is part of"-relation: entity of lower dimension is part of the boundary of an entity of a higher dimension

► The incidence relations yield an abstract (generalized) graph description of a finite element mesh

Possible internal realization of incidence relations:

- for some $(j,k) \in \{0,\dots,d\}^2$, $0 \le j < k \le d$, entities of dimension k hold ordered lists (vectors) of references (pointers) to those entities of dimension j contained in their boundary,

- for some $(m,n) \in \{0,\dots,d\}^2$, $0 \le m < n \le d$, entities of dimension m hold ordered lists (vectors) of references (pointers) to those entities of dimension n that contain them.

Example 3.6.31 (Storing topology of triangular mesh in 2D)

Let $\mathcal{M}$ be a triangular mesh according to Def. 3.4.2 as in Section 3.3.1. Various schemes for storing topological information are conceivable. In the figures below: black $\leftrightarrow$ triangles, blue $\leftrightarrow$ edges, red $\leftrightarrow$ vertices.

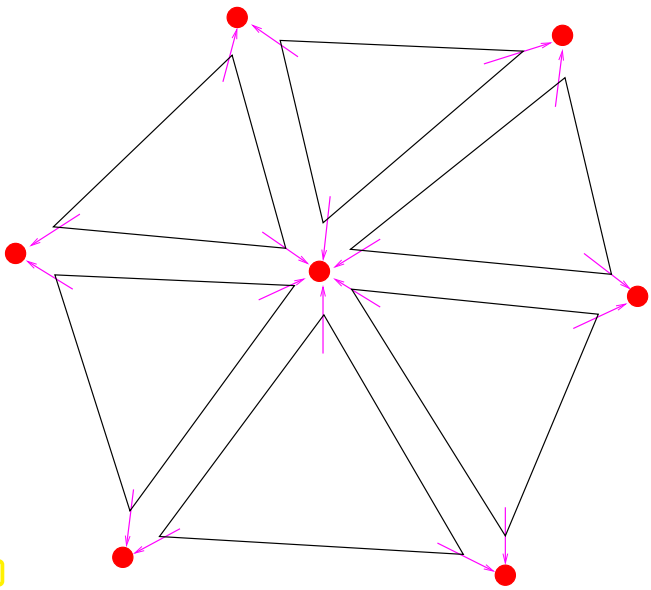


Fig. 148

(A) Minimal scheme: triangles hold lists/vectors of references to their vertices. Edges not stored.

Realized in the MATLAB mesh data structure discussed in § 3.3.3

Sufficient for linear Lagrangian finite elements, if no special boundary conditions have to be dealt with, see Code 3.3.38

This scheme already provides complete topological information (edges can be reconstructed)!

(B) Element centered scheme with edges:

- Elements, edges and vertices stored as (virtual) "objects"
- Elements have lists/vectors of references to their vertices and edges.

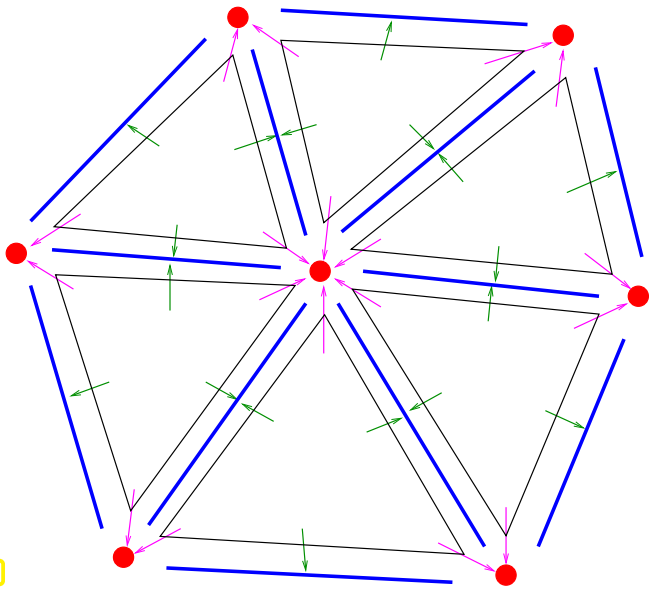


Fig. 149

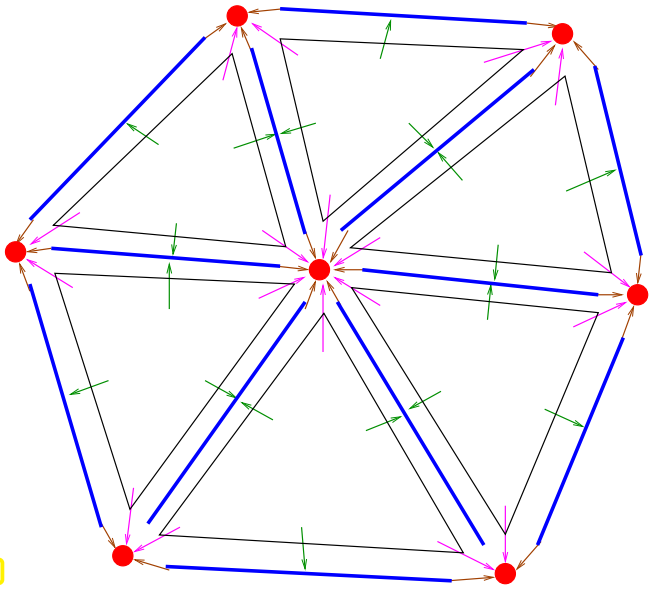


Fig. 150

(C) Full unidirectional topology representation:

- All geometric entities are stored as (virtual) "objects".
- Elements hold lists/vectors of references to their vertices and edges.
- Edges have references to their endpoints.

(D) Restricted bidirectional topology representation:

- All geometric entities are stored as (virtual) "objects".
- Elements hold vectors of references to their vertices and edges.
- Elements also possess a vector of references to their neighbors.

Topology representation in DUNE/BETL

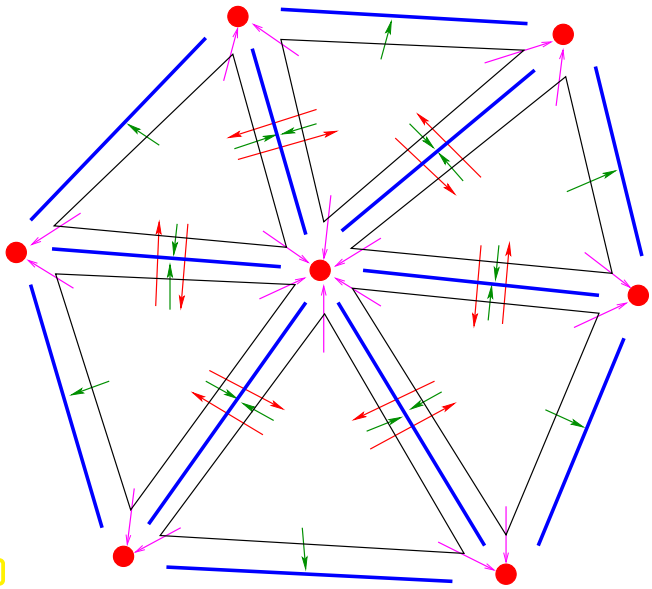


Fig. 151

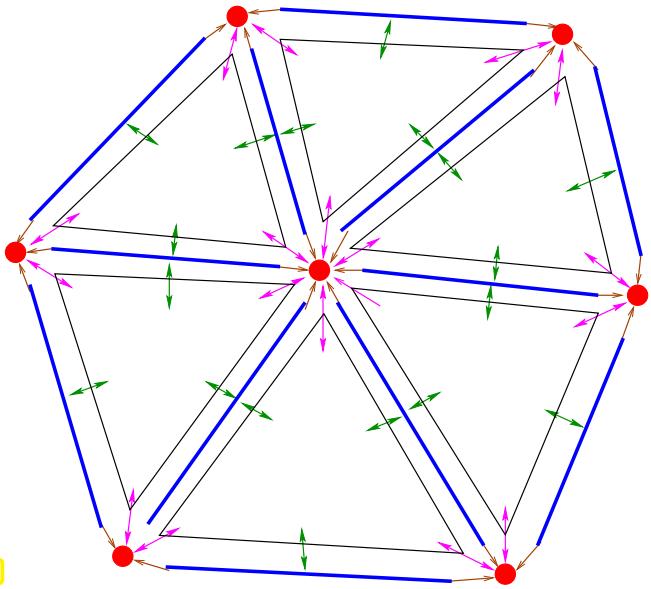


Fig. 152

(E) Full bidirectional topology representation:

- Elements hold vectors of references to their vertices and edges.
- Edges have references to their endpoints and their adjacent triangles
- Vertices have references to their adjacent triangles.

Notation: $\mathcal{M}$ = mesh (set of elements = set of geometric entities of co-dimension 0)
$\mathcal{V}(\mathcal{M})$ = set of nodes (vertices) in $\mathcal{M}$ (geometric entities of co-dimension 2)
$\mathcal{E}(\mathcal{M})$ = set of edges in $\mathcal{M}$ (geometric entities of co-dimension 1)

(3.6.32) DUNE concept for geometric entities

In DUNE objects describing geometric entities fit the Entity concept, see documentation. The corresponding types can be obtained via

```
using Entity=typename GridView::template Codim<k>::Entity;
using EntityPtr=typename GridView::template Codim<k>::EntityPointer;
```

co-dimension is a template parameter!

`Entity` offers the following (few) features and facilities:

- `type()` tells the geometric type of an entity, is `Simplex2D`, `Simplex1D`, or `Simplex0D` for a triangular mesh in two dimensions.
- `geometry()` provides a reference to the "geometry" of an entity, see § 3.6.49.

(Only!) *entities of co-dimension* 0 (= cells/elements of the mesh) provide an extended functionality in the form of the following methods:

- `int count<codim>()` returns the number of entities of co-dimension `codim` contained in the boundary of the cell.
- `EntityPtr subEntity<codim>(int locidx)` returns a pointer to the entity with local number `locidx` and co-dimension `codim` contained in the boundary of the cell.

  This function gives complete information about the mesh topology (access arrows $\rightarrow$ and $\rightarrow$ in Fig. 151).

(3.6.33) BETL concept for geometric entity

The entity concept underlying BETL represents a small modification of its DUNE counterpart explained in § 3.6.32. In BETL the types can be obtained by

```
using gridTraits_t = typename GRID_VIEW::gridTraits_t;
template<int k>
  using entity_t = eth::grid::Entity<gridTraits_t,k>;
template<int k>
  using entityPtrImpl_t =
    typename gridTraits_t::template entityPointer_t<k>;
template<int k>
  using entityPtr_t =
    eth::grid::EntityPointer<gridTraits_t,entityPtrImpl_t<k> >;
```

The **Entity** interface is defined in `entity.hpp` and resides in namespace `eth::grid`.

Entities are always immutable and cannot be copied or assigned to; if pointers are needed, the use of the above **EntityPointer** types is mandatory.

The following methods are provided by an entity:

- `refElType()` provides information about the geometric type of the mesh entity.

  More precisely, `refElType()` tells about the underlying reference element. ➢ The meaning and use of reference elements will be explained in Section 3.7.

- **POINT**, **SEGMENT** (LineSegment $[-1,1]$),
- **TRIA** (reference triangle with ordered vertices $\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}$.
- **QUAD** (unit square with ordered vertices $\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}$)
- **TETRA** (reference tetrahedron with ordered vertices $\begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}$
- **HEXA** (unit cube),
- **PRISM** (reference prism),
- **PYRAMID** (reference pyramid)

These are defined in `enum_ref_el_types.hpp`, namespace `eth::base`, e.g.

```
using triangle_t = eth::base::RefElType::TRIA;
```

✦ `geometry()` returns a (constant) reference to the geometric information attached to the entity, see § 3.6.49 for explanation concerning the **Geometry** concept.

Entities of co-dimension 0 (= cells/elements of the mesh) *alone* provide an extended functionality in the form of the following methods:

✦ `int countSubEntities<codim>()` returns the number of entities of co-dimension `codim` contained in the boundary of the cell.

✦ `EntityPtr subEntity<codim>(int locidx)` returns a pointer to the entity with local number `locidx` and co-dimension `codim` contained in the boundary of the cell.

This function gives complete information about the mesh topology (access arrows $\rightarrow$ and $\rightarrow$ in Fig. 151).

Supplement 3.6.34 (Definition of reference elements in BETL).

The geometry of the reference elements is specified in `ref_el_types_i.hpp`, namespace `eth::base`. For instance the definition of the reference triangle by means of a template specialisation of the *static* class **ReferenceElement** is given in the following code (partial listing from the file `EthGenericGrid/Libs/eth_bas` It makes use of template specialization.

C++11 code 3.6.35: Part of the definition of the reference triangle in BETL

```
// subEntityTypes_[i][j] = type of subentity with codim i, index j
template<>
  const itVec_t ReferenceElement<RefElType::TRIA>::subEntityTypes_{
    {RefElType::TRIA}, // co-dimension = 0
    {RefElType::SEGMENT, RefElType::SEGMENT, RefElType::SEGMENT}, // edges
    {RefElType::POINT, RefElType::POINT, RefElType::POINT}        // vertices
};
// subEntityCorners_[i][j][k] = Index of k-th vertex of subentity
```

```
  with codim i, index j
template<>
  const vector<iiVec_t>
      ReferenceElement<RefElType::TRIA>::subEntityCorners_{
    {{0,1,2}},                // vertex numbers for triangle
    { {0,1},{1,2},{2,0} },    // endpoint indices of edges
    { {0},{1},{2} }           // indices of vertices
};
// cornerCoord_[i][k]} = k-th coordinate of i-th node.
template<>
  const fixedMatrix_t<2,3>
      ReferenceElement<RefElType::TRIA>::cornerCoord_(
    (fixedMatrix_t<2,3>() << 0,1,1,
                             0,0,1).finished());
```

Note the use of initializer lists to set the values of the static class variables `subEntityCorners_` and `subEntityCorners_`. The static variable `cornerCoord_` is initialized by means of the contructor of a fixed size matrix type provided by EIGEN.

△

Remark 3.6.36 (Internal mesh data structures of BETL)

This supplementary information can utterly be ignored by a user of the BETL library.

The **GRID_TRAITS** concept collects basic types and cardinals characterising an instance of a mesh. An excerpt from file `volume2d_traits.hpp` is displayed next.

C++11 code 3.6.37: GRID_TRAITS for a 2D hybrid mesh

```
struct GridTraits
  {
    typedef eth::base::unsigned_t unsigned_t;
    typedef eth::base::signed_t   signed_t;
    typedef unsigned_t            size_type;

    // fix dimension of triangulated manifold and ambient space
    static const int dimMesh  = 2;
    static const int dimWorld = 2;

    typedef Grid gridImpl_t;

    template<int CODIM>
      using entity_t =
          betl2::grid2d::hybrid::Entity<CODIM, GridTraits>;
    template<int CODIM>
      using entityPointer_t = betl2::grid2d::hybrid::EntityPointer<
          CODIM, GridTraits >;
    template<int CODIM>
      using entityIterator_t = betl2::grid2d::hybrid::EntityIterator<
          CODIM, GridTraits >;
```

```
    template<int CODIM>
      using geometry_t = betl2::Geometry< CODIM, dimMesh-CODIM,
          dimWorld >;
    template<int DIM_FROM, int DIM_TO>
      using localGeometry_t = betl2::Geometry< 0, DIM_FROM, DIM_TO >;
    template<signed_t ROWS, signed_t COLS>
      using fixedSizeMatrix_t = betl2::real_matrix_t< ROWS, COLS >;

    typedef double ctype_t;
    typedef size_type idType_t;
    ...
  };
```

The next code lists essential parts of a template specialization the forms the basis for a "Vertex type" for a 2D hybrid mesh (partial listing from file `Library/grid/grid2d_entity_node`).

C++11 code 3.6.38: BETL class representing a vertex in a 2D hybrid mesh

```
// special instance of a geometric entity with co-dimension 2
template< typename GRID_TRAITS >
 class Entity<2,GRID_TRAITS>:
   public ::eth::grid::Entity< GRID_TRAITS,2>
  {
   private:
   // A vertex is an entity of codimension 2, see § 3.4.3
    static const int codim = 2;
    // type of geometric object, see § 3.6.33
    static const eth::base::RefElType ret_ =
        eth::base::RefElType::POINT;
   public:
    typedef eth::grid::Entity<GRID_TRAITS, codim>
        interface_t;
    typedef typename GRID_TRAITS::size_type                    index_t;
    typedef typename GRID_TRAITS::template geometry_t<codim>
        geometry_impl_t;
    typedef eth::grid::Geometry<GRID_TRAITS,geometry_impl_t>
        geometry_t;
   private:
     // EIGEN fixed size matrix type for storing coordinate vectors
     typedef typename GRID_TRAITS::
       template fixedSizeMatrix_t<GRID_TRAITS::dimWorld,1> matrix_t;
    private:
     // Global index number
     index_t          index_;
     // Information about the geometry.  This data member actually
     // stores the (world) coordinates of the vertex.
     geometry_impl_t geometry_impl_;
    public:
     Entity(): interface_t(),index_(),geometry_impl_(matrix_t()) {}
```

```
    Entity( index_t index, const matrix_t& coords )
      :interface_t(),index_(index),geometry_impl_(coords) {}

    index_t index() const { return index_; }
    eth::base::RefElType refElType() const { return ret_; }
    geometry_t geometry() const { return geometry_t(geometry_impl_);
      }
  };
```

Now we look at the type for an edge entity in 2D (partial listing from file `Library/grid/grid2d_entity_edge`
We see that the edge data structure contains a pair of pointers to the vertices representing its endpoints.

C++11 code 3.6.39: BETL class representing an edge of a 2D hybrid mesh

```
// specialisation of entity with co-dimension 1
template< typename GRID_TRAITS >
class Entity< 1, GRID_TRAITS >:
public ::eth::grid::Entity< GRID_TRAITS,1 >
{
private:
// An edge is an entity of codimension 1, see § 3.4.3
static const int myCodim_ = 1;
// type of geometric object, see § 3.6.33
static const eth::base::RefElType ret_ =
   eth::base::RefElType::SEGMENT;

public:
typedef ::eth::grid::Entity< GRID_TRAITS,myCodim_ > interface_t;
typedef eth::base::unsigned_t index_t;

private:
template< int CODIM >
using entityPtr_t =
    typename GRID_TRAITS::template entityPointer_t<CODIM>;
typedef typename GRID_TRAITS::template entity_t< 2 > node_t;
typedef array< node_t*, 2 > pair_t;

private:
// Global index of the edge
index_t       index_;
// An edge is stored as an array of two pointers to nodes.
const pair_t  edge_nodes_;

public:
// An edge is built from a pair of nodes
Entity( node_t* first, node_t* last ):
  interface_t(),index_( ), edge_nodes_() {
    edge_nodes_[0] = first; edge_nodes_[1] = last; }

```

```
// Return index of edge
index_t index( ) const { return index_; }
// Return the entity's reference type, which will be SEGMENT
eth::base::RefElType refElType( ) const { return ret_; }
// Get number of sub-entities
template< int CODIM > int count( ) const {
   BOOST_STATIC_ASSERT_MSG( CODIM == 2,
     "Entity<1>::Codimension other than 2 is meaningless" );
   return 2; // An edge always consists of 2 nodes
 }
 // Get pointer to i-th subentity
 template< int CODIM > entityPtr_t<CODIM> subEntity( int i ) {
      BOOST_STATIC_ASSERT_MSG( CODIM == 2,
        "Entity<1>::Codimension other than 2 is meaningless" );
      BOOST_ASSERT_MSG( i < this -> count<CODIM>(),
        "Entity<1>::Wrong index!" );
      return ( i == 0 ?
               entityPtr_t<CODIM>( edge_nodes_[0] ) :
               entityPtr_t<CODIM>( edge_nodes_[1] ) );
  }
};
```

More complex is the type for a cell of a 2D hybrid mesh, whose definition through template specialization is introduced next (partial listing from file `Library/grid/grid2d_entity_element.hpp`). The implementations of the methods have been omitted.

C++11 code 3.6.40: BETL class representing a cell of a 2D hybrid mesh

```
template< typename GRID_TRAITS >
class Entity<0,GRID_TRAITS >: public ::eth::grid::Entity<
   GRID_TRAITS,0 >
 {
   private:
   // A cell is an entity of codimension 2, see § 3.4.3
     static const int codim = 0;
     typedef Entity<0, GRID_TRAITS> self_t;
   public:
     typedef eth::base::unsigned_t index_t;
     typedef betl2::ElementType element_t;
     typedef typename GRID_TRAITS::template geometry_t<codim>
        geometry_impl_t;
     typedef typename GRID_TRAITS::entityOrientation_t
        entityOrientation_t;
     typedef eth::grid::Geometry<GRID_TRAITS,geometry_impl_t>
        geometry_t;
     template<int CODIM>
       using entity_pointer_impl_t = typename GRID_TRAITS::template
          entityPointer_t<CODIM>;
     template<int CODIM>
```

```
      using entity_pointer_t = eth::grid::EntityPointer<
          GRID_TRAITS, entity_pointer_impl_t<CODIM> >;
  private:
    typedef eth::base::ReferenceElements Rets;
    typedef typename GRID_TRAITS::template entity_t< 1 > edge_t;
    typedef typename GRID_TRAITS::template entity_t< 2 > node_t;
    typedef vector<edge_t*> edge_vec_t;
    typedef vector<node_t*> node_vec_t;
    typedef vector<const self_t*> element_vec_t;
    typedef tuple<edge_vec_t,node_vec_t> entity_collection_t;
  private:
    // Internal code for storage class, defined in file
       element_types.hpp
    const element_t element_type_;
    // Global index
    const index_t index_;
    // Array of pointers to edges and vertices
    entity_collection_t entity_collection_;
    // Matrix of node coordinates
    geometry_impl_t geometry_impl_;
    // Orientation of current element
    entityOrientation_t entityOrientation_;
  public:
    // Main constructor building a cell from pieces of information
    template< typename NODE_ITERATOR >
    Entity(element_t      element_type,
           index_t        index,
           NODE_ITERATOR  node_begin,
           NODE_ITERATOR  node_end);
    // Fetch the element index
    inline index_t index( ) const;
    // Get the underlying reference element's type
    eth::base::RefElType refElType( ) const;
    // Get number of subentities
    template< int CODIM > inline int countSubEntities( ) const;
    template< int CODIM >
    entity_pointer_t<CODIM> subEntity( int i ) const;
    // Return the entity's geometry representation
    geometry_t geometry( ) const;
    /// Get the orientation of the i-th sub-entity of codim CODIM
    template< int CODIM >
    bool orientationSign( size_type i ) const
    // Determine relative orientation of the edges of the cell
    void initializeOrientation(void);
  };
```

The method `orientationSign` returns the *relative orientation* of a sub-entity, usually of an edge. The orientation tells its direction specified by an ordering of the endpoints. The method gives $+1$ if the intrinsic orientation of an edge matches its local orientation, -1 otherwise. Local orientations are explained in Rem. 3.6.48.

(3.6.41) Numbering of geometric entities in DUNE/BETL

All geometric entities of a fixed co-dimension and geometric type are numbered by consecutive integer indices starting from 0. These indices are *not accessible* from the entity itself, but are stored in an `IndexSet` object, see documentation: If `GRIDVIEW` is the type of the current grid, a reference to which is stored in the variable `gv`, then in a standard **DUNE** interface a reference to its index set can be fetched as follows:

```
using IndexSetRef=typename GridView::IndexSet const &;
using index_t =typename GridView::IndexSet::IndexType;
IndexSetRef set(gv.indexSet());
```

In **BETL** access to the type and the instance of the index set associated with a grid is slightly different:

```
using grid_t = betl2::volume2dGrid::hybrid::Grid;
// we use the leaf view of a grid only
const eth::grid::GridViewTypes view =
   eth::grid::GridViewTypes::LeafView;
using indexSet_betl2_t = typename grid_t::gridTraits_t::template
   viewTraits_t<view> ::indexSet_t;
using indexSet_eth_t = typename eth::grid::IndexSet<gridTraits_t,
   indexSet_betl2_t>;
using indexSetRef_t = indexSet_eth_t const &;
using index_t = typename indexSet_t::size_type;
indexSetRef_t set(gv.indexSet());
```

The BETL way looks more complicated, but everything becomes very simple with `auto` type deduction, see Code 3.6.29.

```
auto & set = gv.indexSet();
```

`IndexSet` has the method

```
index_t index(const Entity &) const;
```

which returns the unique index (actually an integer) of any entity passed to it.

For *entities of co-dimension* 0 (= cells, elements) there is a shortcut access to the indices of sub-entities:

- In the standard **DUNE** specification:

  ```
  index_t subIndex(const Entity &element,size_type
     locidx,size_type codim)
  ```

  whose function is equivalent to calling

  ```
  set.index(T.template subEntity<codim>(locidx));
  ```

  for the entity object `T` of co-dimension 0.

- A slightly altered variant in **BETL**, templated by the co-dimension:

  ```
  template <CODIM> index_t subIndex(
     const Entity<GRID_TRAITS,0> &element,size_type locidx)
  ```

Example 3.6.42 (Scanning mesh topology in standard DUNE interface)

Code 3.6.43 demonstrates the access of lower-dimensional entities in a DUNE compliant code; the use of `subEntity` methods and the retrieval of global index numbers, also by `subIndex()`.

C++11 code 3.6.43: Accessing geometric entities of a mesh in DUNE

```
1  template<class GridView>
2  void scanTopology(const GridView &gv) {
3    using IndexSetRef=typename GridView::IndexSet const &;
4    using Index=typename GridView::IndexSet::IndexType;
5    // Types for geoemtric entities of the mesh
6    using Triangle=typename GridView::template Codim<0>::Entity;
7    using TrianglePtr=typename GridView::template
       Codim<0>::EntityPointer;
8    using Edge=typename GridView::template Codim<1>::Entity;
9    using EdgePtr=typename GridView::template Codim<1>::EntityPointer;
10   using Node=typename GridView::template Codim<2>::Entity;
11   using NodePtr=typename GridView::template Codim<2>::EntityPointer;
12
13   IndexSetRef set(gv.indexSet());
14   // loop over all cells of the mesh
15   for(auto it=gv.template begin<0>();it != gv.template end<0>();
       ++it) {
16     const Triangle &T = *it;
17     cout << it->type() << ", id = " << set.index(T);
18     const int Ned = T.template count<1>(); // No.  of edges @@
19     // loop over edges and print their global indices
20     cout << ", edges = ";
21     for(int j=0;j<Ned;j++) {
22       EdgePtr edptr = T.template subEntity<1>(j);
23       const Edge &ed = *edptr;
24       cout << set.index(ed);
25       cout << " (subindex = " << set.subIndex(T,j,1) << "), ";
26       }
27     // loop over vertices and print their global index numbers
28     cout << " vertices = ";
29     const int Nvt = T.template count<2>();
30     for(int j=0;j<Nvt;j++) {
31       NodePtr vtptr = T.template subEntity<2>(j);
32       const Node &vt = *vtptr;
33       cout << set.index(vt);
34       cout << "(subindex = " << set.subIndex(T,j,2) << "), ";
35       }
36     cout << endl;
37   }
38 }
```

Line 18, Line 16: the `count` function gives the number of sub-entities.

Line 11, Line 19: the `index` function of the `IndexSet` contained in the `GridView` allows access to unique global index numbers starting from 0.
Line 25, Line 20: indices can also be requested by calling `subIndex()`.

Example 3.6.44 (Inspecting mesh topology in BETL)

The following two codes are meant to demonstrate access to entities and sub-entities using the DUNE-style mesh interface provided by BETL. Two versions are given; a long version with explicit statement of all types and a short version relying on automatic type deduction.

C++11 code 3.6.45: Accessing sub-entities and their index numbers in BETL ➙ GITLAB

```
1  template<class VIEW_TRAITS>
2  void scanTopology(const eth::grid::GridView<VIEW_TRAITS> &gv) {
3    using gridView_t = eth::grid::GridView<VIEW_TRAITS>;
4    using gridTraits_t = typename gridView_t::gridTraits_t;
5    using viewTraits_t = typename gridView_t::viewTraits_t;
6    using indexSet_t = eth::grid::IndexSet<gridTraits_t,typename
       viewTraits_t::indexSet_t>;
7    using indexSet_t = typename viewTraits_t::indexSet_t;
8    using indexSetRef_t = indexSet_t const &;
9    using index_t = typename indexSet_t::size_type;
10
11   // Types for geometric entities of the mesh and pointers to them
12   using triangle_t = eth::grid::Entity<gridTraits_t,0>;
13   using trianglePtrImpl_t = typename gridTraits_t::template
       entityPointer_t<0>;
14   using trianglePtr_t =
       eth::grid::EntityPointer<gridTraits_t,trianglePtrImpl_t>;
15   using edge_t = eth::grid::Entity<gridTraits_t,1>;
16   using edgePtrImpl_t = typename gridTraits_t::template
       entityPointer_t<1>;
17   using edgePtr_t =
       eth::grid::EntityPointer<gridTraits_t,edgePtrImpl_t>;
18   using node_t = eth::grid::Entity<gridTraits_t,2>;
19   using nodePtrImpl_t = typename gridTraits_t::template
       entityPointer_t<2>;
20   using nodePtr_t =
       eth::grid::EntityPointer<gridTraits_t,nodePtrImpl_t>;
21
22   // Obtain handle to index set for current grid
23   indexSetRef_t set(gv.indexSet());
24   // loop over all cells of the mesh
25   for (auto& t : gv.template entities<0>()) {
26     // Fetch index of current element
27     const index_t idxT = set.index(t);
28     cout << t.refElType() << ", index = " << idxT << endl;
29     const int Ned = t.template countSubEntities<1>(); // No. of edges
         @@
```

```
    // loop over edges and print their global indices
    for(int j=0;j<Ned;j++) {
      cout << "Edge " << j << ": ";
      edgePtr_t edptr = t.template subEntity<1>(j);
      const edge_t &ed = *edptr;
      cout << set.index(ed);
      const index_t edIdx = set.template subIndex<1>(t,j);
      cout << " (subindex = " << edIdx << "), ";
      }
    cout << endl;
    // loop over vertices and print their global index numbers
    const int Nvt = t.template countSubEntities<2>();
    for(int j=0;j<Nvt;j++) {
      cout << "Vertex " << j << ": ";
      nodePtr_t vtptr = t.template subEntity<2>(j);
      const node_t &vt = *vtptr;
      cout << set.index(vt);
      const index_t vtIdx = set.template subIndex<2>(t,j);
      cout << "(subindex = " << vtIdx << "), ";
      }
    cout << endl;
  }
}
```

The cumbersome extraction of types can be avoided when relying on the C++11 `auto` type deduction mechanism:

C++11 code 3.6.46: Accessing sub-entities and their index numbers in BETL ➔ GITLAB

```
template<class VIEW_TRAITS>
void scanTopology_auto(const eth::grid::GridView<VIEW_TRAITS> &gv) {
  auto &set = gv.indexSet(); // See § 3.6.41
  // loop over all cells of the mesh (entities of co-dimension 0)
  for (auto& t : gv.template entities<0>()) {
    cout << t.refElType() << ", id = " << set.index(t) << endl;
    auto Ned = t.template countSubEntities<1>(); // No. of edges
    // loop over edges and print their global indices
    for(int j=0;j<Ned;j++) {
      auto edptr = t.template subEntity<1>(j);
      cout << "Edge " << j << ": " << set.index(*edptr);
      cout << " (subidx = " << set.template subIndex<1>(t,j) << "),
          ";
      }
    cout << endl;
    // loop over vertices and print their global index numbers
    const int Nvt = t.template countSubEntities<2>();
    for(int j=0;j<Nvt;j++) {
      auto vtptr = t.template subEntity<2>(j);
      cout << "Vertex " << j << ": " << set.index(*vtptr);
      cout << "(subidx = " << set.template subIndex<2>(t,j) << "), ";
```

```
21          }
22      cout << endl;
23    }
24  }
```

Line 6: via the index set request unique global index number of current cell referenced by iterator `t`.

Line 7, Line 16: the method `countSubEntities` yields the number of lower-dimensional sub-entities of a particular co-dimension.

Line 10, Line 18: by means of `subEntity()` one can obtain a pointer to a particular sub-entity.

Line 12, Line 20: a call to the `subindex` method of an **indexSet_t** object also gives the global index of a sub-entity.

A working example calling `scanTopology` is given in ➙ GITLAB.

Example 3.6.47 (Global indices of entities of a hybrid mesh in BETL)

This example highlights the fact that consecutive indexing is done *separately* for geometric entities of different type, though they may be of the same co-dimension.

```
$MeshFormat
2.2 0 8
$EndMeshFormat
$Nodes
8
1 0 0 0
2 1 0 0
3 2 0 0
4 2 2 0
5 1 2 0
6 0 2 0
7 0 1 0
8 1 1 0
$EndNodes
$Elements
17
1 15 2 5 1 1
2 15 2 5 3 3
3 15 2 5 4 4
4 15 2 5 6 6
5 1 2 2 1 1 2
6 1 2 2 2 2 3
7 1 2 2 3 3 4
8 1 2 4 3 3 4
9 1 2 2 4 4 5
10 1 2 2 5 5 6
11 1 2 3 6 6 7
12 1 2 3 6 7 1
13 2 2 1 11 3 4 8
14 2 2 1 11 3 8 2
15 2 2 1 11 8 4 5
16 3 2 1 12 1 2 8 7
17 3 2 1 12 7 8 5 6
$EndElements
```

◁ `.msh`-file (→ Ex. 3.6.10) created by **Gmsh** describing the mesh drawn in Fig. 153.

Simple planar hybrid mesh comprising triangular and rectangular cells:

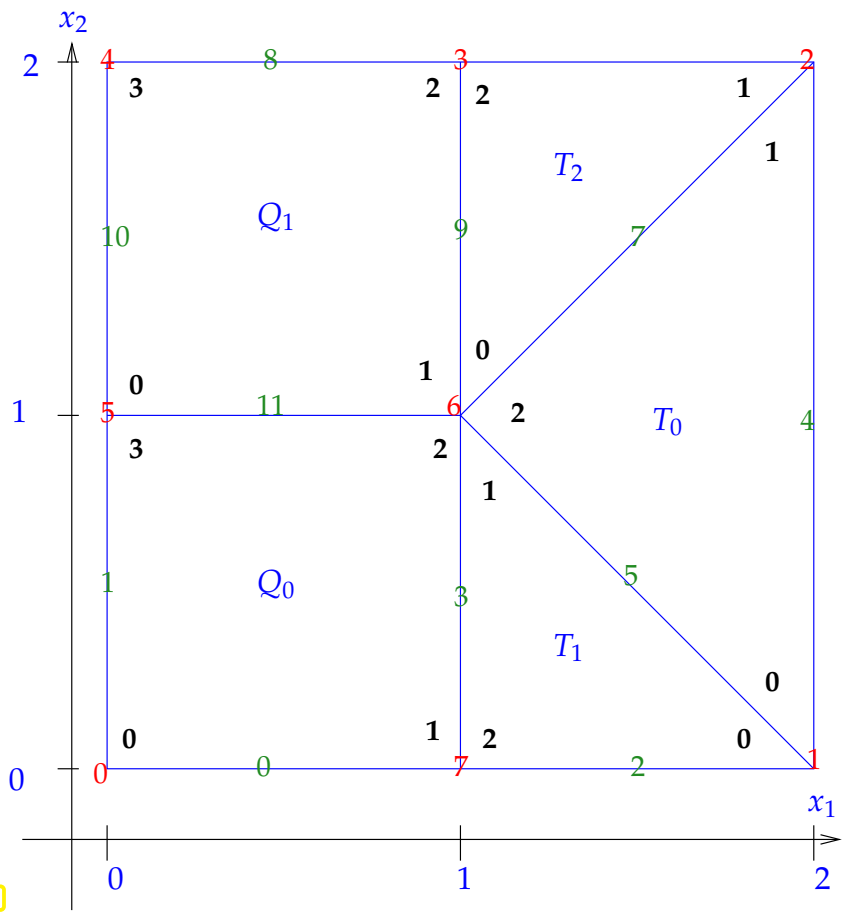


Fig. 153

0,... ≙ local index of vertices
1,... ≙ global indices of vertices
1,... ≙ global indices of edges
T_i, Q_i ≙ cells with global index

The internal index numbers of all mesh entities are displayed as deduced from the information output by diagnostic functions and listed below.

Output of executable `NPDE_topolgy` ➜ GITLAB, when supplied with the name of the above `.msh`-file as argument:

```
BETL demo: ———————— INPUT MESH FROM Gmsh MESH FILE ————————
BETL demo: input from: hybrid_mesh_5.msh
BETL demo: input data   Version                = 2.2
    Format                  = ASCII
    No. of phys. names      = 0
    No. of nodes            = 8
    No. of elements         = 5
        No. of elements of type 'QUAD_4' = 2
        No. of elements of type 'TRIA_3' = 3

BETL demo: numNodes     = 8
BETL demo: numEdges     = 12
BETL demo: numElements  = 5

```

```
TRIA, id = 0
Edge 0   -> Id: 4 with vertices [2 0], [2 2]
Edge 1   -> Id: 7 with vertices [2 2], [1 1]
Edge 2   -> Id: 5 with vertices [1 1], [2 0]
Vertex 0 -> Id: 2 : [2 0]
Vertex 1 -> Id: 3 : [2 2]
Vertex 2 -> Id: 7 : [1 1]

TRIA, id = 1
Edge 0   -> Id: 5 with vertices [1 1], [2 0]
Edge 1   -> Id: 3 with vertices [1 1], [1 0]
Edge 2   -> Id: 2 with vertices [1 0], [2 0]
Vertex 0 -> Id: 2 : [2 0]
Vertex 1 -> Id: 7 : [1 1]
Vertex 2 -> Id: 1 : [1 0]

TRIA, id = 2
Edge 0   -> Id: 7 with vertices [2 2], [1 1]
Edge 1   -> Id: 6 with vertices [2 2], [1 2]
Edge 2   -> Id: 9 with vertices [1 2], [1 1]
Vertex 0 -> Id: 7 : [1 1]
Vertex 1 -> Id: 3 : [2 2]
Vertex 2 -> Id: 4 : [1 2]

QUAD, id = 0
Edge 0   -> Id: 0 with vertices [0 0], [1 0]
Edge 1   -> Id: 3 with vertices [1 1], [1 0]
Edge 2   -> Id: 11 with vertices [1 1], [0 1]
Edge 3   -> Id: 1 with vertices [0 1], [0 0]
Vertex 0 -> Id: 0 : [0 0]
Vertex 1 -> Id: 1 : [1 0]
Vertex 2 -> Id: 7 : [1 1]
Vertex 3 -> Id: 6 : [0 1]

QUAD, id = 1
Edge 0   -> Id: 11 with vertices [1 1], [0 1]
Edge 1   -> Id: 9 with vertices [1 2], [1 1]
Edge 2   -> Id: 8 with vertices [1 2], [0 2]
Edge 3   -> Id: 10 with vertices [0 2], [0 1]
Vertex 0 -> Id: 6 : [0 1]
Vertex 1 -> Id: 7 : [1 1]
Vertex 2 -> Id: 4 : [1 2]
Vertex 3 -> Id: 5 : [0 2]
```

We summarize the main insight gleaned from this example:

> Only the pair of `index` and `RefElType` uniquely identifies a mesh entity in BETL.

Remark 3.6.48 (Local numbering of sub-entities of a triangle in DUNE and BETL)

The `subEntity()` access method of an element accepts a local index number, returns a pointer to a

sub-entity and thus defines a local numbering of the sub-entities of an element (see also Code 3.6.35).

This local numbering must be fixed by *convention*. The conventions adopted by DUNE and BETL for setting the local indices of the edges of a triangle once the vertices are numbered are *different*. They are illustrated below (red ↔ vertex numbers, green ↔ edge numbers). BETL's local numbering scheme can also be deduced from the program output listed in Ex. 3.6.47.

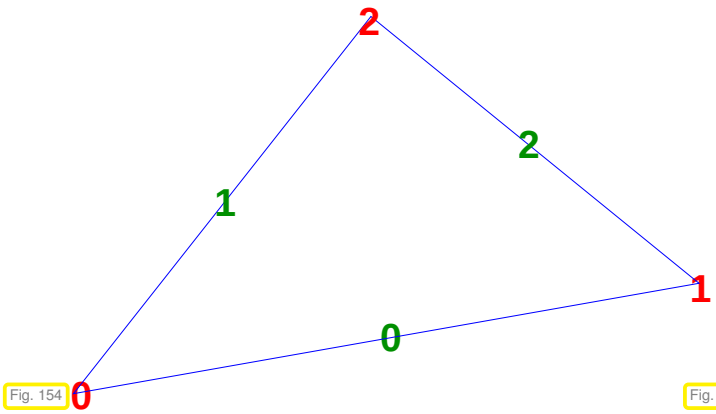


Fig. 154

DUNE convention

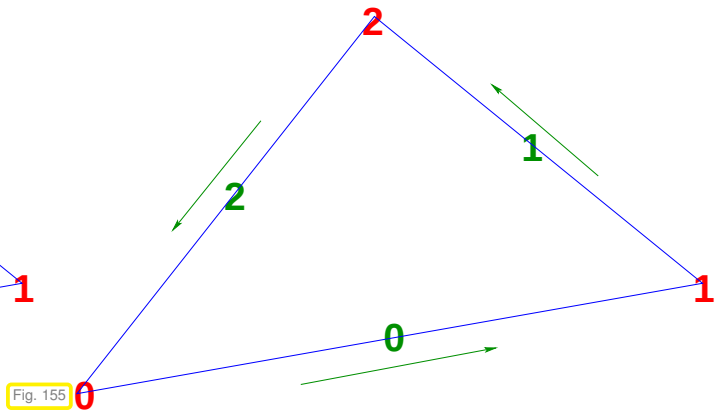


Fig. 155

BETL convention

BETL's numbering convention for quadrilaterals (reference element type QUAD) is given beside

In the figures the green arrows indicate the convention about the *local* orientation of the edges of a cell, which are always supposed to point from the vertex with the smaller index to the vertex with the larger index. Distinguis from *global* orientation, which is intrinsic to an edge and fixed arbitrarily during the construction of a mesh.

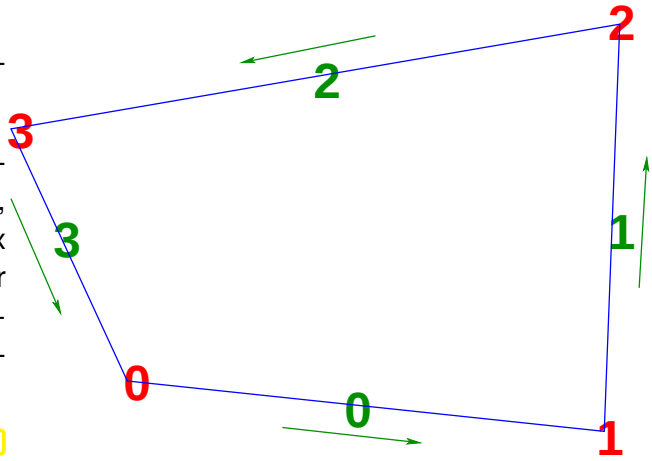


Fig. 156

(3.6.49) Geometric information in DUNE & BETL

In the **DUNE** standard interface the `geometry` method available for any entity (→ § 3.6.32 & § 3.6.33) provides access to a `Geometry` structure with the following types and methods, see documentation:

- Type `ctype` ≙ type for the components of coordinate vectors
- Type `GlobalCoordinate` ≙ vector of global coordinates
- Method `int corners()` ≙ number of vertices/endpoints
- Method `corner(int i)` ≙ global location of corner with local index `i`
- Method `volume()` ≙ returns volume/length of geometric entity

In DUNE coordinate vectors are of type `Dune::FieldVector`, which offers elementary linear algebra, see DUNE Doxygen documentation.

In **BETL** this interface has been modified slightly and the class **Geometry** is defined in `geometry.hpp` in namespace `eth::grid`. The following types and methods can be accessed:

- Constant dimFrom telling the dimension of the reference element

- Constant dimTo, the dimension of ambient space
- Type `gridTraits_t` of the **GridTraits** of the mesh of which the entity is part of.
- Vector type `globalCoord_t` for absolute coordinates of points in ambient space.
- Vector type `localCoord_t` for relative coordinates in a reference element.
- Integer type `size_type` for indices.
- Method `size_type numCorners()` telling the number of vertices of the entity.
- Method `globalCoord_t mapCorner(int i)` returning the global coordinates of the vertices of the geometric entity.
- Method `gridTraits_t::ctype_t volume()` telling the volume/area of the geometric entity.
- Method `globalCoord_t center()` obtaining the global coordinates of the center of gravity of the geometric entity.

In BETL coordinate vectors are small *fixed size* EIGEN *vector types*, see [14, § 1.2.11] and Section 3.6.3. Thus, all of EIGEN's linear algebra operations and functions are available for them.

Example 3.6.50 (Accessing locations in DUNE)

Code 3.6.51 demonstrates the use of the `corner()` method to request to location of vertices of geometric entities.

C++11 code 3.6.51: Printing the locations of vertices associated with geometric entities

```
template<class GridView,int k>
void printMeshGeo(const GridView &gv) {
  enum { domdim = GridView::dimension };
  using EntityIterator=typename GridView::template Codim<k>::Iterator;
  using Entity = typename GridView::template Codim<k>::Entity;
  using GeometryRef=typename Entity::Geometry const &;
  using PointCoords=typename Entity::Geometry::GlobalCoordinate;

  cout << "Dim. = " << GridView::dimension << endl;
 for(EntityIterator it=gv.template begin<k>();it != gv.template
    end<k>(); ++it) {
    GeometryRef geo = it->geometry();
    int Nvt = geo.corners();
    cout << Nvt << " vertices at ";
    for(int j=0;j<Nvt;j++) {
      const PointCoords &vpos(geo.corner(j));
      cout << "(" << vpos << ") ";
    }
    cout << endl;
  }
}
```

Example 3.6.52 (Geometry related queries in BETL)

The following codes demonstrate geomtric queries in BETL through the **Geometry** interface.

C++11 code 3.6.53: Output of information on the geometry of an entity ➙ GITLAB

```
template<class GEOMETRY>
 void printGeometryInfo(const GEOMETRY &geoEnt) {
  // Type for length of vectorss
  using size_t = typename GEOMETRY::size_type;
  // Traits for underlying grid, see Code 3.6.37
  using gridTraits_t = typename GEOMETRY::gridTraits_t;
  // Type for components of coordinate vectors (double)
  using ctype_t = typename gridTraits_t::ctype_t;
  // Type for coordinate vectors, small fixed size vector from EIGEN
  using globalCoord_t = typename GEOMETRY::globalCoord_t;
  // Dimension of the mesh entity
  static const int dimFrom = GEOMETRY::dimFrom;
  // Dimension of the ambient space (world dimension)
  static const int dimTo = GEOMETRY::dimTo;

  // Fetch the dimFrom-dimensional volume of the entity
  const ctype_t volEnt = geoEnt.volume();
  // Inquire about coordinates of the center
  // (The center is the image of the barycenter of the reference
     element)
  const globalCoord_t cntrEnt = geoEnt.center();
  // Find out whether mapping from reference element is affine
  const bool affine = geoEnt.isAffine();
  // Print obtained information
  cout << "(type = " << geoEnt.refElType()
         << ",dim_from = " << dimFrom << ", dim_to = " << dimTo
         << "), Volume = " << volEnt << ", center = ["
            << cntrEnt.transpose() << "], ";
  if (affine)  cout << "[affine] " << endl;
  else         cout << "[not affine] " << endl;

  // For demonstration purposes:  direct computation of affine
  // barycenter of vertices of the mesh entity
  const size_t nCorners = geoEnt.numCorners();
  // Sum position vectors of vertices in vector s
  globalCoord_t s; s.setZero();
  for(int j=0; j < nCorners; j++) {
    globalCoord_t cornCoords = geoEnt.mapCorner(j);
    s += cornCoords;
  }
  s /= nCorners;
  cout << endl << "Affine barycenter at "
            << s.transpose() << endl;
```

```
43 }
```

Four important query options are available through **Geometry**. We can find out about the volume (→ Line 17), the center of gravity (→ Line 20), the number of corners (vertices, → Line 33), and their location in the world coordinate system (→ Line 37).

C++11 code 3.6.54: Output of information on the geometry of an entity ➙ GITLAB

```
1  template<class GEOMETRY>
2  void printGeometryInfo(const GEOMETRY &geoEnt) {
3    cout << "(type = "      << geoEnt.refElType()
4           << ",dim_from = " << GEOMETRY::dimFrom
5           << ", dim_to = "  << GEOMETRY::dimTo
6           << "), Volume = " << geoEnt.volume()
7           << ", center = [" << geoEnt.center().transpose() << "], ";
8    if (geoEnt.isAffine())  cout << "[affine] "      << endl;
9    else                    cout << "[not affine] " << endl;
10
11   typename GEOMETRY::globalCoord_t s; s.setZero();
12   int j=0; for(; j < geoEnt.numCorners(); j++)
13     s += geoEnt.mapCorner(j);
14   s /= j;
15   cout << endl << "Affine barycenter at [" << s.transpose() << ']' <<
       endl;
16 }
```

The meaning of the `isAffine()` in Line 8 query will be explained in Section 3.7. The type `globalCoord_t` defined in Line 11 is that of a fixed size EIGEN vector.

The output functions are invoked from an loop over mesh entities of a particular co-dimension (passed as template parameter) as in Code 3.6.29.

C++11 code 3.6.55: Output of information on the geometry of an entity ➙ GITLAB

```
1 template<int k,class VIEW_TRAITS>
2 void printMeshGeo(const eth::grid::GridView<VIEW_TRAITS> &gv) {
3   // loop over entities of co-dimension k
4   for(auto& it : gv.template entities<k>())
5     printGeometryInfo(it.geometry());
6 }
```

An executable code using `printMeshGeo` can be accessed through ➙ GITLAB.

(3.6.56) DUNE/BETL – Intersections

The representation of mesh topology implemented in DUNE/BETL, see Fig. 151, allows direct access to adjacent elements through the device of intersections, see DUNE documentation:

Intersection object	$=$	part of the boundary of an element
	$\neq$	geometric entity of co-dimension 1 (edge)

- ☞ An intersection object always belongs to a cell.
- ☞ Intersection objects do not have any global indices.

In the standard DUNE interface sequential access to the intersection of a cell `e` is possible through the iterator pair

```
GridView::ibegin(e) ... GridView::iend(e)
```

In **BETL** an intermediate sequential container of type **EntityCollection** serves the same purpose. It can be accesses through the following member function of a **GridView** object:

```
const EntityCollection< IntersectionIterator<VIEW_TRAITS> >
  intersections (const Entity<GRID_TRAITS,0> &e) const
```

An intersection object is equipped with the following methods:

- `bool boundary()`: if false, no other neighbor exists.
- `bool neighbor()`: true, if a neighbor cell exists.
- `geometry()`: geometry of intersection object, see § 3.6.49
- `inside()`: return pointer to "master element"
- `outside()`: returns pointer to neighbor element; well defined return value only if this exists.
- `indexInInside()`: local number of edge corresponding to intersection object in "master element"
- `indexInOutside()`: local number of edge corresponding to intersection object in neighboring element; well defined return value only if this exists.

Example 3.6.57 (Using DUNE intersections to query local topology of mesh)

The following code uses DUNE's intersection facility to access the cells adjacent to a current cell.

C++11 code 3.6.58: Use of intersection objects in DUNE

```
template<class GridView>
void visitIntersections(const GridView &gv) {
  using IndexSetRef=typename GridView::IndexSet const &;
  using Index=typename GridView::IndexSet::IndexType;
  // Types for geoemtric entities of the mesh
  using TrianglePtr=typename GridView::template
      Codim<0>::EntityPointer;
  // Types connected with intersections
  using Side=typename GridView::Intersection;
  using SideIterator=typename GridView::IntersectionIterator;

```

```
11    IndexSetRef set(gv.indexSet());
12    // loop over all cells of the mesh
13    for(auto it=gv.template begin<0>();it != gv.template end<0>();
        ++it) {
14      const Triangle &T = *it;
15      // loop over the intersections (sides) of the current cell
16      for (auto iit = gv.ibegin(T); iit != gv.iend(T); ++iit){
17        const Side& side = *iit;
18        TrianglePtr nb1ptr = side.inside(); // the triangle itself
19        int locidx = side.indexInInside();  // local number of side
20        // obtain global index number of current side
21        Index glbidx = set.index(*T.template subEntity<1>(locidx));
22        cout << "side " << locidx << " (idx = " << glbidx << ") ";
23        // If the side is on the boundary there is no other neighbor
24        if (side.boundary()) cout << "on boundary, ";
25        else {
26       // Get point to triangle on the other side
27       TrianglePtr nb2ptr = side.outside();
28       cout << "-> neighbor = " << set.index(*nb2ptr) << ", ";
29        }
30      }
31      cout << endl;
32    }
33  }
```

Testing `boundary()` in Line 24 is essential; otherwise the result of `outside()` in Line 27 is an invalid pointer.

Example 3.6.59 (Using BETL intersections to query local topology of mesh)

The following code uses BETL's intersection facility to access the cells adjacent to a current cell. We first give a version with full specification of the types

C++11 code 3.6.60: Use of intersection objects in BETL ➙ GITLAB

```
template< class VIEW_TRAITS>
void visitIntersections(const eth::grid::GridView<VIEW_TRAITS> &gv) {
  using gridView_t = eth::grid::GridView<VIEW_TRAITS>;
  using gridTraits_t = typename gridView_t::gridTraits_t;
  using viewTraits_t = typename gridView_t::viewTraits_t;
  using indexSet_t = eth::grid::IndexSet<gridTraits_t, typename
      viewTraits_t::indexSet_t>;
  using index_t = typename indexSet_t::size_type;
  // Types for geometric entities of the mesh and pointers to them
  using element_t = eth::grid::Entity<gridTraits_t, 0>;
  using elemPtrImpl_t = typename gridTraits_t::template
      entityPointer_t<0>;
  using elemPtr_t = eth::grid::EntityPointer<gridTraits_t,
```

```
    elemPtrImpl_t>;
using elemIteratorImpl_t = typename viewTraits_t::template entityIterator_t<0>;
using elemIterator_t = eth::grid::EntityIterator<gridTraits_t, elemIteratorImpl_t>;
using edge_t = eth::grid::Entity<gridTraits_t, 1>;
using edgePtrImpl_t = typename gridTraits_t::template entityPointer_t<1>;
using edgePtr_t = eth::grid::EntityPointer<gridTraits_t, edgePtrImpl_t>;
// Types for handling intersections
using itsct_t = eth::grid::Intersection<gridTraits_t>;
using itsctIterator_t = eth::grid::IntersectionIterator<viewTraits_t>;
using itsctCollection_t = eth::grid::EntityCollection<itsctIterator_t>;

const indexSet_t &set(gv.indexSet());
// Loop over all the elements in the mesh
for(elemIterator_t elit = gv.template entities<0>().begin();
    elit != gv.template entities<0>().end(); ++elit) {
  // Get reference to the element itself
  const element_t &el = *elit;
  // Global index of the element
  const index_t elGlbIdx = set.index(el);
  // Get reference to the intersections container
  const itsctCollection_t &itsctColl = gv.intersections(el);
  // Loop over intersections (side) of current element
  for(itsctIterator_t itsctit = itsctColl.begin(); itsctit != itsctColl.end(); ++itsctit) {
    const itsct_t &inters = *itsctit;
    // Get the pointer to the "inside" element, coinciding with el
    elemPtr_t inPtr = inters.inside();
    // Get local index of this side wrt el
    const index_t inLclIdx = inters.indexInInside();

    // Get global index of current side.  Since the IndexSet maps
    // entities, not intersections, we need to get the corresponding entity first.
    const edgePtr_t locEdPtr = inPtr->template subEntity<1>(inLclIdx);
    const index_t glbIdx = set.index( *locEdPtr );

    cout << "Intersection " << glbIdx << " is a side of element " << elGlbIdx << " (idx: " << inLclIdx << ")";

    // check if this intersection is shared or is a boundary
    if(itsctit->boundary()) cout << " and is on boundary";
    else {
   // Get pointer to element on the other side
```

```
      elemPtr_t outPtr = inters.outside();
      const index_t outLclIdx = inters.indexInOutside();
      const index_t elOutGlbIdx = set.index(*outPtr);
      cout << " and of element " << elOutGlbIdx << " (idx :" <<
         outLclIdx << ")";
       }
       cout << endl;
     }}}
```

Using automatic type deduction makes possible a more compact implementation and a more readable code.

C++11 code 3.6.61: Use of intersection objects in BETL (with auto typing) ➙ GITLAB

```
template< class VIEW_TRAITS >
void visitIntersections_auto( const eth::grid::GridView<VIEW_TRAITS>
   &gv ){
 auto& set = gv.indexSet();
 // Loop over mesh elements (codim=0)
 for ( const auto& el : gv.template entities<0>() ) {
   // Loop over element intersections (sides)
   for ( const auto& inters : gv.intersections(el) ) {
     const auto elInGlbIdx = set.index(*inters.inside());
     if (elInGlbIdx != set.index(el))
       cout << "WARNING: index mismatch between 'inside' and current
          elements!" << endl;
      // Get local index of this side (wrt inside element/ el)
     const auto inLclIdx = inters.indexInInside();
     // Get global index of this side
     const auto locEdPtr = el.template subEntity<1>(inLclIdx);
     const auto glbIdx = set.index( locEdPtr );
     cout<<"Intersection "<< glbIdx << " is a side of element " <<
        elInGlbIdx<<" (idx: "<< inLclIdx <<")";
     // Check if this intersection is shared or is a boundary
     if (inters.boundary()) cout<< "is on boundary";
     else {
       // Get local index of this side
       // (wrt neighbor/other element sharing this intersection)
       const auto outLclIdx = inters.indexInOutside();
       const auto elOutGlbIdx = set.index( *inters.outside() );
       cout<< " and of element " << elOutGlbIdx <<" (idx: "<<
          outLclIdx <<")";
     }
     cout << endl;
   }}}
```

A minimal working code is available from ➙ GITLAB.

3.6.3 Vectors and matrices

(3.6.62) Functions of vectors and matrices in a FE code

Data structures representing matrices and vectors serve different important purposes in a finite element code:

❶ They represent coordinate vectors of points and have to support geometric calculations, see Code 3.6.54.

❷ They store element matrices and element vectors, recall Section 3.3.5, Section 3.3.6, and see Def. 3.6.69 below.

❸ They are needed for handling the Galerkin matrices and and have to be used by the linear (direct or iterative) solver.

For ❶ small fixed size vectors and matrices are sufficient, and they may also be used for ❷, if the mesh consists of a single type of cells only. For ❷ we need data structures suitable for large variable size vectors and matrices, where the latter are sparse, moreover, see Section 3.3.4.

(3.6.63) EIGEN– A C++ template library for numerical linear algebra

BETL relies on the open source software EIGEN for its numerical linear algebra needs.

EIGEN is a C++ template library designed to enable easy, natural and efficient numerical linear algebra: it provides data structures and a wide range of operations for matrices and vectors, see below. EIGEN also implements (→ doc)

- all important matrix decompositions of dense numerical linear algebra (LU-, QR-, Cholesky-decompositions) and direct solvers based on them,
- "direct" eigensolvers for various types of dense eigenvalue problems,
- the singular value decomposition (SVD) of a matrix,
- ranks, determinants and inverses of matrices.

Eigen relies on expression templates to allow the efficient evaluation of complex expressions involving matrices and vectors. Refer to the example given in the EIGEN documentation for details.

The principal components and capabilities of the EIGEN library have been covered in the course "Numerical Methods for Computational Science and Engineering" [14, Section 1.2.3].

In **BETL** *all* matrices and vectors are objects of a suitable **Eigen::(Sparse)Matrix** type.

(3.6.64) EIGEN: some pointers to information

✦ Matrix and vector data types in EIGEN: see [14, § 1.2.11] and documentation.

✦ Initialization of *dense* matrices in EIGEN: see [14, § 1.2.12].

✦ Access to submatrices in EIGEN: see [14, § 1.2.13] and documentation.

- Componentwise operations in EIGEN: see [14, § 1.2.15] and documentation.
- Sparse matrices in EIGEN (CRS/CCS-format): see [14, Section 1.7.3] and documentation; already used in Code 3.3.38.

3.6.4 Assembly

"Assembly" = term used for computing entries of stiffness matrix/right hand side vector (load vector) in a finite element context, *cf.* § 3.3.29.

From the dictionary: "Assemble" = to fit together all the separate parts of something.

Aspects of assembly for linear Lagrangian finite elements ($V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M})$) were discussed in Section 3.3.5 and Section 3.3.6. (Refresh yourself on these sections in case you cannot remember the main ideas behind building the Galerkin matrix and right hand side vector.)

3.6.4.1 Assembly: Localization

Cell-local concepts and operations play a key role in the efficient initialization of finite element Galerkin matrices and right hand side vectors.

(3.6.65) Localized (bi-)linear forms in variational formulations

We consider a discrete variational problem ($V_{0,N}$ = FE space, $\dim V_{0,N} = N \in \mathbb{N}$, see (3.2.8))

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N} \,. \tag{3.2.8}$$

To be computed (see also Section 3.3.5 and Section 3.3.6):

- Galerkin matrix (stiffness matrix): $\mathbf{A} = \left(\mathsf{a}(b_N^j, b_N^i)\right)_{i,j=1}^N \in \mathbb{R}^{N,N}$
- r.h.s. vector (load vector): $\vec{\boldsymbol{\varphi}} := \left(\ell(b_N^i)\right)_{i=1}^N \in \mathbb{R}^N$

both can be written in terms of *local cell contributions*, since usually

$$\boxed{\mathsf{a}(u,v) = \sum_{K\in\mathcal{M}} \mathsf{a}_K(u_{|K}, v_{|K}) \quad , \quad \ell(v) = \sum_{K\in\mathcal{M}} \ell_K(v_{|K})} \,, \tag{3.6.66}$$

where $\cdot|_K$ designates the restriction of a function to cell K; $u_{|K}$ and $v_{|K}$ are not defined outside K.

Example: bilinear forms/linear forms arising from 2nd-order elliptic BVPs, e.g, (2.10.2), (2.10.3), (2.10.4), can be localized in straightforward fashion by restricting integration to mesh cells ($\rightarrow$ Rem. 3.3.8): for $u, v \in H^1(\Omega)$

$$\mathsf{a}(u,v) := \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} = \sum_{K\in\mathcal{M}} \underbrace{\int_K \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}_K(u_{|K}, v_{|K})} \,, \tag{3.6.67}$$

$$\ell(v) := \int_\Omega f v \,\mathrm{d}\mathbf{x} = \sum_{K\in\mathcal{M}} \underbrace{\int_K f v \,\mathrm{d}\mathbf{x}}_{=:\ell_K(v_{|K})} . \tag{3.6.68}$$

Recall (3.4.18): Restrictions of global shape functions to cells = local shape functions

Definition 3.6.69. Element (stiffness) matrix and element (load) vector

Given a cell $K \in \mathcal{M}$ and local shape functions $\{b_K^1, \dots, b_K^Q\}$, $Q = Q(K) \in \mathbb{N}$, we introduce

the element (stiffness) matrix $\mathbf{A}_K := \left(\mathsf{a}_K(b_K^j, b_K^i)\right)_{i,j=1}^Q \in \mathbb{R}^{Q,Q}$,

and the element (load) vector $\vec{\boldsymbol{\varphi}}_K := \left(\ell_K(b_K^i)\right)_{i=1}^Q \in \mathbb{R}^Q$.

(3.6.70) Numbers of local shape functions

In Def. 3.6.69: Q = the number of local shape functions on element $K \in \mathcal{M}$, may be different for different mesh cells K. For instance, this occurs

- in the case of hybrid meshes as discussed in Rem. 3.5.16: $Q(K) \in \{3,4\}$,
- in the case of enforcement of zero essential boundary conditions by dropping basis functions associated with interpolation nodes on $\partial\Omega$, as explained in § 3.5.14: according to the formal definition Def. 3.4.19 this will lead to a reduced number of local shape functions. However, in implementations zero essential boundary conditions are handled differently, see Section 3.6.6.

For standard Lagrangian finite element spaces $\mathcal{S}_p^0(\mathcal{M})$ the dimensions of the spaces spanned by local shape functions are the same for all mesh cells and given by the following formulas:

Type of FE space	Q
degree p Lagrangian FE on *triangular* mesh	$\to \dim \mathcal{P}_p(\mathbb{R}^2) = \frac{1}{2}(p+1)(p+2)$
degree p Lagrangian FE on *tetrahedral* mesh	$\to \dim \mathcal{P}_p(\mathbb{R}^3) = \frac{1}{6}(p+1)(p+2)(p+3)$
degree p Lagrangian FE on *tensor product* mesh in 2D	$\to \dim \mathcal{Q}_p(\mathbb{R}^2) = (p+1)^2$

We arrive at these formulas, by the following considerations:

- For Lagrangian finite element spaces the local shape functions span a polynomial space, either $\mathcal{P}_p(\mathbb{R}^d)$ (simplicial mesh) or $\mathcal{Q}_p(\mathbb{R}^d)$ (tensor product mesh).
- The dimensions of $\mathcal{P}_p(\mathbb{R}^d)/\mathcal{Q}_p(\mathbb{R}^d)$ are given in Lemma 3.4.11 and Lemma 3.4.14.

3.6.4.2 Assembly: Index Mappings

What we have discovered in the case of linear finite elements in Section 3.3.5 (conveyed in Fig. 101 and Fig. 102 and the accompanying remarks) and implemented in Code 3.3.35 is a general principle.

We find that in the (not so special) setting of this section, characterized by the possibility to localize the bilinear form a and right hand side linear form ℓ in the sense of (3.6.66),

- ✦ the entries of the finite element Galerkin matrix can be obtained by summing *corresponding* entries of *some* element matrices,
- ✦ this corresponding entry of an element matrices is determined by the unique association of a local basis function to a global basis function (through a "d.o.f. mapper").

(3.6.71) Abstract "d.o.f. mapper" facility

The correct assignment of local contributions to entries of the Galerkin matrix and the right hand side vector requires a

☞ Local→global index map ("d.o.f. mapper")

$$\begin{aligned} &\texttt{locglobmap}: \mathcal{M} \times \mathbb{N} \to \mathbb{N}\,, \\ &\texttt{locglobmap}(K,i) = j\,, \quad \text{if} \quad b^j_{N\,|K} = b^i_K\,, \quad i \in \{1,\dots,Q(K)\}\,. \end{aligned} \tag{3.6.72}$$

global shape function local shape function

Remark 3.6.73 (Local→global index mapping and index array)

The mapping `locglobmap` generalizes the device of the index mapping array `dofh` introduced in (3.3.33) on Page 204 for linear Lagrangian finite elements on 2D triangular meshes and also used in Code 3.3.35: Precisely, they are related by

$$\texttt{dofh}(k,l) = \texttt{locglobmap}(K,l)\,, \quad \text{if } K \text{ has index } k\,, \quad l \in \{1,2,3\}\,.$$

(Here, mathematical counting from 1 is used.)

Note that the representation of `locglobmap` through a matrix `dofh` assumes unique consecutive indexing of all cells of the mesh. In DUNE/BETL this is not the case, if the mesh comprises cells of different geometric type.

Example 3.6.74 (Local→global mapping for linear Lagrangian finite elements on triangular mesh)

This example refreshes § 3.3.32.

Using the local/global numbering indicated in the figure to the right the local→global index map for the marked cells yields

$$\texttt{locglobmap}(K^*,(1,2,3)) = (2,7,9)\,.$$

Here: "MATLAB-style" row vector argument makes `locglobmap` return row vector output.

See also Fig. 103 for similar considerations.

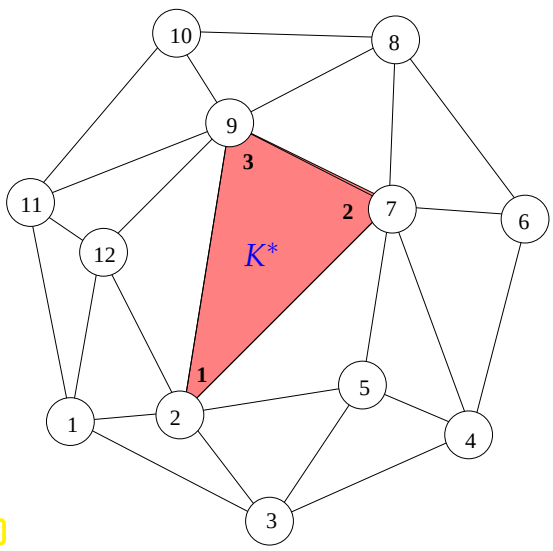


Fig. 157

(3.6.75) Specifying the location of global shape functions in BETL

By "location" of a global shape function we mean the unique geometric entity it is associated with, see Section 3.4.3. BETL adopts a *cell-oriented perspective*, to represent this linkage: for each cell of the mesh (element) the locations of the local shape functions are fixed.

More precisely, BETL imposes a substantial constraint: the locations of local shape functions have to be the **same** for all cells of the same geometric type (encoded in the `RefElType`, see § 3.6.33). The concept providing this information is **betl2::fe::FEBasis** (`Library/fe/febasis.hpp`). It has to provide the following types and methods:

- `size_type`: standard type for indexing
- `template< eth::base::RefElType RET >`
  `static constexpr size_type` **multiplicity**`()`
  `static size_type` **multiplicity**`(const eth::base::RefElType ret)`
  → tell number of local shape functions belonging to subentities of a particular type.
- `template< eth::base::RefElType RET > static size_type` **numDofs**`()`
  `static size_type` **numDofs**`(const eth::base::RefElType ret)`
  → gives *total number* of local shape functions associated with a subentities of a given type and its boundary. Usually this method is called only for an entity of co-dimension 0 and then it returns the total number of local shape functions.

In fact **betl2::fe::FEBasis** has more facilities, but the discussion of those will be postponed to Ex. 3.7.33.

Several standard finite element spaces are already built into BETL and accessible through

```
template<int APPROX_ORDER,enum FEBasisType FE_TYPE> class FEBasis;
```

where the template parameters serve as selectors, and are defined in `Library/fe/fe_enumerators.hpp` ↪ BETL, and can attain the following values

```
enum ApproxOrder { Constant=0, Linear=1, Quadratic=2, Cubic=3 };
enum class FEBasisType : unsigned
  { Lagrange, Div, Curl, LagrangeHierarchical };
```

The type `Lagrange` selects the $H^1(\Omega)$-conforming Lagrangian finite elements introduced in Section 3.5. For a discussion of the types `Div` and `curl` see [4, Sections 2.3, 2.4] and [13, Section 3].

Example 3.6.76 (Implementation of an FEBasis compatible type)

The following data type realizes an **FEBasis** compatible type defining a finite element space, which has 1 shape function associated with each vertex, 2 with each edge, and 2 more with each cell. The implementation is generic, however, and can easily be adapted to any finite scheme on 2D hybrid meshes that are uniform in the sense that the arrangement of local shape functions is the same for every cell.

The definition of the data type is as follows:

C++11 code 3.6.77: Definition of FEBasis compatible type ➙ GITLAB

```
struct MyFEBasis {
  typedef eth::base::unsigned_t size_type;

  /* static methods: return number of dof per entity */
  template< enum eth::base::RefElType RET >
  static constexpr size_type multiplicity( );
  static size_type multiplicity( const eth::base::RefElType
     ref_element );

  /* Approximation order: dummy implementation */
  static constexpr int approxOrder( ) { return 1; }

  /* returns number of dof associated with an entity type, that is
     the total number of dofs
     belonging to the entity and all sub-entities */
  template< eth::base::RefElType RET >
  static constexpr size_type numDofs();
  static size_type numDofs( const eth::base::RefElType ref_element );
};
```

The next listing shows the implementation of the `multiplicity()` methods:

C++11 code 3.6.78: Implementation of `multiplicity()` methods for MyFEBasis ➙ GITLAB

```
// Template specializations:  no.  of lsf per cell
template <> constexpr MyFEBasis::size_type
MyFEBasis::multiplicity<eth::base::RefElType::TRIA>() { return 2; }
template <> constexpr MyFEBasis::size_type
MyFEBasis::multiplicity<eth::base::RefElType::QUAD>() { return 2; }
// no.  of local shape functions per SEGMENT
template <> constexpr MyFEBasis::size_type
MyFEBasis::multiplicity<eth::base::RefElType::SEGMENT>() { return 2; }
// no.  of local shape functions per POINT
template <> constexpr MyFEBasis::size_type
MyFEBasis::multiplicity<eth::base::RefElType::POINT>() { return 1; }
// Non-template version, runtime binding to entity type
MyFEBasis::size_type
MyFEBasis::multiplicity(const eth::base::RefElType ref_element) {
  switch (ref_element) {
  case eth::base::RefElType::TRIA: { return
     multiplicity<eth::base::RefElType::TRIA>(); }
  case eth::base::RefElType::QUAD: { return
     multiplicity<eth::base::RefElType::QUAD>(); }
  case eth::base::RefElType::SEGMENT: { return
     multiplicity<eth::base::RefElType::SEGMENT>(); }
  case eth::base::RefElType::POINT: { return
     multiplicity<eth::base::RefElType::POINT>(); }
  default: { ETH_ASSERT(false); return 0; }
```

```
}}
```

The `numDofs` method has to add up the numbers of local shape functions associated with the geometric type of a (sub)entity. In fact, these numbers can be computed by calling `multiplicity()` in a generic fashion. However, the next listing gives a specific implementation.

C++11 code 3.6.79: Implementation of `numDofs()` methods for MyFEBasis ⇝ GITLAB

```
template <> constexpr MyFEBasis::size_type
MyFEBasis::numDofs<eth::base::RefElType::POINT>() { return 1; }
template <> constexpr MyFEBasis::size_type
MyFEBasis::numDofs<eth::base::RefElType::SEGMENT>() { return 4; }
template <> constexpr MyFEBasis::size_type
MyFEBasis::numDofs<eth::base::RefElType::TRIA>() { return 11; }
template <> constexpr MyFEBasis::size_type
MyFEBasis::numDofs<eth::base::RefElType::QUAD>() { return 14; }

MyFEBasis::size_type MyFEBasis::numDofs( const eth::base::RefElType
    ref_element ) {
  switch (ref_element) {
  case eth::base::RefElType::TRIA: { return
      numDofs<eth::base::RefElType::TRIA>(); }
  case eth::base::RefElType::QUAD: { return
      numDofs<eth::base::RefElType::QUAD>(); }
  case eth::base::RefElType::SEGMENT: { return
      numDofs<eth::base::RefElType::SEGMENT>(); }
  case eth::base::RefElType::POINT: { return
      numDofs<eth::base::RefElType::POINT>(); }
  default: { ETH_ASSERT(false); return 0; }
  }}
```

(3.6.80) Ordering of local shape functions in BETL

Using the the **FEBasis** concept to define the association of local shape functions and geometric (sub-)entities of a cell does not immediately imply a numbering of those local shape functions. This numbering is arbitrary and calls for another *convention* to organize the finite element code.

The following convention is universally applied in BETL for $d = 2$:

(I) The local shape functions are arranged according to increasing dimension of their associated geometric entities:

$$\textbf{POINT} \longrightarrow \textbf{SEGMENT} \longrightarrow \{\textbf{TRIA,QUAD}\}\,.$$

(II) Local shape functions belonging to geometric entities of the same dimension are ordered according to the intrinsic local ordering of those entities. See Rem. 3.6.48 for BETL's conventions.

(III) No ordering of local shape functions attached to the same geometric entity is implied.

For the finite element scheme defined through the type **MyFEBasis** in Ex. 3.6.76 (1/2/2 local shape functions assigned to vertices, edges, and cells, respectively) we find the following numbering scheme for the local shape functions: ▷

This output was generated by the code lines Code 3.6.82 listed below ➙ GITLAB.

```
# Local basis on triangle:
# 11 local shape functions
lsf no.  0 <-> POINT, (sub-)entity no. 0
lsf no.  1 <-> POINT, (sub-)entity no. 1
lsf no.  2 <-> POINT, (sub-)entity no. 2
lsf no.  3 <-> SEGMENT, (sub-)entity no. 0
lsf no.  4 <-> SEGMENT, (sub-)entity no. 0
lsf no.  5 <-> SEGMENT, (sub-)entity no. 1
lsf no.  6 <-> SEGMENT, (sub-)entity no. 1
lsf no.  7 <-> SEGMENT, (sub-)entity no. 2
lsf no.  8 <-> SEGMENT, (sub-)entity no. 2
lsf no.  9 <-> TRIA, (sub-)entity no. 0
lsf no.  10 <-> TRIA, (sub-)entity no. 0
# Local basis on quadrilateral:
# 14 local shape functions
lsf no.  0 <-> POINT, (sub-)entity no. 0
lsf no.  1 <-> POINT, (sub-)entity no. 1
lsf no.  2 <-> POINT, (sub-)entity no. 2
lsf no.  3 <-> POINT, (sub-)entity no. 3
lsf no.  4 <-> SEGMENT, (sub-)entity no. 0
lsf no.  5 <-> SEGMENT, (sub-)entity no. 0
lsf no.  6 <-> SEGMENT, (sub-)entity no. 1
lsf no.  7 <-> SEGMENT, (sub-)entity no. 1
lsf no.  8 <-> SEGMENT, (sub-)entity no. 2
lsf no.  9 <-> SEGMENT, (sub-)entity no. 2
lsf no.  10 <-> SEGMENT, (sub-)entity no. 3
lsf no.  11 <-> SEGMENT, (sub-)entity no. 3
lsf no.  12 <-> QUAD, (sub-)entity no. 0
lsf no.  13 <-> QUAD, (sub-)entity no. 0
```

The following helper class probes the assignment of local shape functions for standard 2D element type. It relies on the numbering convention for local shape functions outlines in Item (I). If, for a triangle n_p, n_e, n_c local shape functions are associated with `POINT`s, `SEGMENT`s, and `TRIA`, respectively, then the first $3n_p$ local shape functions will sit on vertices, the next $3n_e$ on edges, and the remaining are supported on the element. The local index of the corresponding geometric object can be determined by module arithmetic. This algorithm is implemented in the two methods of the class.

C++11 code 3.6.81: Class telling placement of local shape functions defined by an FEBasis ➙ GITLAB

```
template <typename FEBASIS>
struct ProbeFEBasis {
  using index_t = typename FEBASIS::size_type;
  using size_t  = typename FEBASIS::size_type;
  using refEl_t = eth::base::RefElType;

  // Return entity type for local shape function with number lidx
  static refEl_t getRETDof(refEl_t ret,index_t lidx) {
    switch (ret) {
    case eth::base::RefElType::TRIA: {
      if (lidx < 3*FEBASIS::template
          multiplicity<eth::base::RefElType::POINT>())
       return eth::base::RefElType::POINT;
```

```
      lidx -= 3*FEBASIS::template multiplicity<eth::base::RefElType::POINT>();
      if (lidx < 3*FEBASIS::template multiplicity<eth::base::RefElType::SEGMENT>())
     return eth::base::RefElType::SEGMENT;
      return eth::base::RefElType::TRIA;
    }
    case eth::base::RefElType::QUAD: {
      if (lidx < 4*FEBASIS::template multiplicity<eth::base::RefElType::POINT>())
     return eth::base::RefElType::POINT;
      lidx -= 4*FEBASIS::template multiplicity<eth::base::RefElType::POINT>();
      if (lidx < 4*FEBASIS::template multiplicity<eth::base::RefElType::SEGMENT>())
     return eth::base::RefElType::SEGMENT;
      return eth::base::RefElType::QUAD;
    }
    default: { ETH_ASSERT(false); return eth::base::RefElType::POINT; }
    }}

  // Return local index number of sub-entity associated with local shape functions
  // of index lidx
  static index_t getSubentIdx(refEl_t ret,index_t lidx) {
    switch (ret) {
    case eth::base::RefElType::TRIA: {
      const size_t ndp = FEBASIS::template multiplicity<eth::base::RefElType::POINT>();
      const size_t nde = FEBASIS::template multiplicity<eth::base::RefElType::SEGMENT>();
      const size_t ndt = FEBASIS::template multiplicity<eth::base::RefElType::QUAD>();
      if (ndp > 0) { if (lidx < 3*ndp) return lidx/ndp; else lidx -= 3*ndp; }
      if (nde > 0) { if (lidx < 3*nde) return lidx/nde; else lidx -= 3*nde; }
      if (lidx < ndt) return 0;
      ETH_ASSERT_MSG(false,"Illegal index"); return 0; }
    case eth::base::RefElType::QUAD: {
      const size_t ndp = FEBASIS::template multiplicity<eth::base::RefElType::POINT>();
      const size_t nde = FEBASIS::template multiplicity<eth::base::RefElType::SEGMENT>();
      const size_t ndt = FEBASIS::template multiplicity<eth::base::RefElType::QUAD>();
      if (ndp > 0) { if (lidx < 4*ndp) return lidx/ndp; else lidx -= 4*ndp; }
      if (nde > 0) { if (lidx < 4*nde) return lidx/nde; else lidx -=
```

```
            4*nde; }
        if (lidx < ndt) return 0;
        ETH_ASSERT_MSG(false,"Illegal index"); return 0; }
      default: { ETH_ASSERT(false); return 0; }
      }}
}; // end class definition ProbeFEBasis
```

C++11 code 3.6.82: Listing of local shape functions described by MyFEBasis defined in Ex. 3.6.76.

```
typedef MyFEBasis febasis_t; // see Code 3.6.77
typedef ProbeFEBasis<febasis_t> probe_t;
std::cout << "# Local basis on triangle:" << std::endl;
int Q = febasis_t::numDofs<eth::base::RefElType::TRIA>();
std::cout << "# " << Q << " local shape functions" << std::endl;
for(int j=0;j<Q;j++) {
 std::cout << "lsf no. " << j << " <-> "
           << probe_t::getRETDof(eth::base::RefElType::TRIA,j)
           << ", (sub-)entity #" <<
              probe_t::getSubentIdx(eth::base::RefElType::TRIA,j)
           << std::endl;
}
std::cout << "# Local basis on quadrilateral:" << std::endl;
Q = febasis_t::numDofs<eth::base::RefElType::QUAD>();
std::cout << "# " << Q << " local shape functions" << std::endl;
for(int j=0;j<Q;j++) {
 std::cout << "lsf no." << j << " <-> "
           << probe_t::getRETDof(eth::base::RefElType::QUAD,j)
           << ", (sub-)entity #" <<
              probe_t::getSubentIdx(eth::base::RefElType::QUAD,j)
           << std::endl;
}
```

(3.6.83) D.o.f. handler and d.o.f. mapper in BETL

In BETL local→global index mappings in the spirit of the d.o.f. mapper function `locglobmap` from (3.6.72) are managed by objects of type **fe::FESpace**. In BETL, the d.o.f. mapper has to be initialized, which is done through a so-called "d.o.f. handler" object, which is templated by the concrete kind of finite element underlying the discretization. The instantiation of a d.o.f. handler is done in the following code for quadratic Lagrangian finite elements.

C++11 code 3.6.84: Instantiation of a d.o.f. handler for $\mathcal{S}_2^0(\mathcal{M})$ in BETL ➙ GITLAB

```
// define the finite element space and global basis functions
// Here:  2nd-order Lagrangian finite elements, nodal basis
typedef fe::FEBasis<fe::Quadratic,fe::FEBasisType::Lagrange>
```

```
    febasis_t;
4 // define dofhandler type for the grid
5 typedef betl2::fe::DofHandler<febasis_t,
    fe::FESContinuity::Continuous, gridFactory_t > DH_t;
6 DH_t dh;  // instantiate dofhandler
7 // Initialize internal data structures for d.o.f.  mapper.
8 dh.distributeDofs( gridFactory );
9 cout << "Number of created dofs = " << dh.numDofs() << endl;
10 // The function basisFuncIndicesRET is defined in Code 3.6.87.
11 // It generates index mapping matrices for specific cell types, see
    § 3.6.33
12 static const eth::base::RefElType TRIA = eth::base::RefElType::TRIA;
13 static const eth::base::RefElType QUAD = eth::base::RefElType::QUAD;
14 // get index mapping matrix for reference element type TRIA
15 auto dofh_tria = basisFuncIndicesRET( dh, gridView, TRIA );
16 // get index mapping matrix for reference element type QUAD
17 auto dofh_quad = basisFuncIndicesRET( dh, gridView, QUAD );
```

Line 3: the d.o.f. handler needs the typedef of a **fe::FEBasis** object (see § 3.6.75). In the example we use the basis type **fe::FEBasisType::Lagrange**, hence we intend to use Lagrangian finite elements (→ Section 3.5). The polynomial degree is specified by **fe:Quadratic**, hence we use quadratic Lagrangian finite elements (→ Ex. 3.5.3, Ex. 3.5.13).

Line 5: The d.o.f. handler object itself is implemented by the class **fe::DofHandler**. It takes the specification of the **fe::FEBasisType**, the **fe::FESContinuity**, a legacy parameter, which should always be set to (**fe::FESContinuity::Continuous**, and the **eth::grids::utils::GridViewFactory** type.

Line 6–Line 8: The actual instantiation of the **fe::DofHandler** object. In order to initialize the degrees of freedom, the member function `distributeDofs(gridFactory)` is called, where **gridFactory** is an object of type `eth::grids::utils::GridViewFactory`.

Line 17: `basisFuncIndicesRET( dh,gridView,RET)` outputs the matrix `dofh` storing the local→global index mappings for each cell (entity of codimension 0) of reference element type RET. A listing and details are given in Code 3.6.87. The matrix `dofh` is built based on the convention from § 3.6.71. The object **gridView** of type `eth::grids::utils::GridViewFactory` is needed to obtain the index set, which stores the indices of the cells § 3.6.41.

The **fe::DofHandler** object in BETL only takes care of the *initialization of the degrees of freedom*. The object that provides the actual *local→global index map* afterwards is an instantiation of the class **fe::FESpace**. It is a data member of **fe::DofHandler** and can be obtained by calling the following method of **fe::DofHandler**

```
const auto& fe_space = dh.fespace();
```

The class **fe::FESpace** provides the following important member functions:

- **begin()** and **end()** (Code 3.6.85, Line 40) return the constant iterator to the beginning and end of the container of cells, i.e. entities of codimension zero. This enables `foreach` loops over **fe::FESpace**s, Code 3.6.87 Line 21.
- begin(e) and end(e) (Code 3.6.85, Line 46) take `e`, a constant reference to an entity of co-dimension zero (cell) and provide a constant iterator to the beginning and end of the vector of dofs for the

element/cell `e`.

- dofsOnElements() (Code 3.6.85, Line 50) returns a constant reference to the container of all dofs managed by the **fe::FESpace**.
- globalIndex(dIter) (Code 3.6.85, Line 54) takes a constant dof iterator `dIter` and returns its global index.
- localIndex(dIter,e) (Code 3.6.85, Line 56) takes a constant dof iterator `dIter` and a constant reference to an entity `e` of codimension zero and returns the local index of the dof from `dIter` with respect to the cell `e`.
- filter<CODIM>(e, intersectionIndex) (Code 3.6.85, Line 63) takes `e`, a constant reference to an entity of codimension zero (cell) and an `intersectionIndex` of the element (type `int`), referring to one of the elements intersections (sides). It returns a standard vector containing the *local indices* (w.r.t. the cell `e`) of all dofs that are associated with entities of codimension `CODIM` contained in the intersection corresponding to the `intersectionIndex`.
- filterAll(e, intersectionIndex) (Code 3.6.85, Line 67) takes `e`, a constant reference to an entity of codimension zero (cell) and an `intersectionIndex` of the cell `e` (type `int`), referring to one of the elements intersections (sides). It returns a standard vector containing *pointers* to all dofs that are associated with the side corresponding to the `intersectionIndex`.
- filterIndices(e, intersectionIndex) (Code 3.6.85, Line 72) takes `e`, a constant reference to an entity of codimension zero (cell) and an `intersectionIndex` of the cell `e` (type `int`), referring to one of the elements intersections (sides). It returns a standard vector containing the *local indices* (w.r.t. the cell `e`) of all dofs that are associated with the intersection corresponding to the `intersectionIndex`.
- **indices(e, intersectionIndex)** (Code 3.6.85, Line 76) takes `e`, a constant reference to an entity of codimension zero (cell) and an `intersectionIndex` of the cell `e` (type `int`), referring to one of the elements intersections (sides). It returns a standard vector containing the *local→global index mappings* (w.r.t. the intersection associated with the `intersectionIndex`) of all dofs that are associated with the intersection corresponding to the `intersectionIndex`.
- **indices(e)** (Code 3.6.85, Line 74) takes `e`, a constant reference to an entity of codimension zero (cell), and provides a standard vector filled with its *local→global index mappings* (as pairs of integer indices, see below). This method is used, e.g., in Code 3.6.96.
- numDofs() returns the global number of dofs.
- numElements() returns the total number of elements.

The local→global index mappings in BETL are implemented via the class **IndexPair<size_t>** which is a standard pair, where the first entry is accessed via the member function `local()`, while the second entry is accessed calling `global()`.

The class **FESpace** is implemented as follows:

C++11 code 3.6.85: FESpace implementation in BETL (partial listing) ➙ **BETL**

```
template<typename FE_BASIS_T,
         enum FESContinuity FES,
         typename GRID_VIEW_FACTORY_T>
class FESpace
{
public:
```

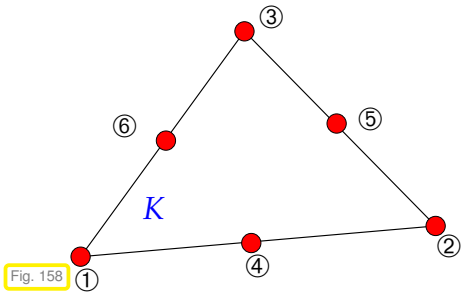
```
  typedef FE_BASIS_T fe_basis_t;
  typedef GRID_VIEW_FACTORY_T gridViewFactory_t;
  typedef typename GRID_VIEW_FACTORY_T::gridTraits_t gridTraits_t;
  typedef eth::grid::Entity< gridTraits_t,0> element_t;
  typedef typename gridTraits_t::size_type size_type;
  typedef typename DofDataSetFactory<
      gridTraits_t::dimMesh, FES,FE_BASIS_T,
        GRID_VIEW_FACTORY_T>::dofDataSet_t dofDataSet_t;
private:
  typedef typename gridTraits_t::template entityIterator_t<0>
      iteratorImpl_t;
public:
  typedef eth::grid::EntityIterator< gridTraits_t, iteratorImpl_t >
  const_element_iterator;
  typedef boost::indirect_iterator< Dof const* const*,
                                    const Dof > const_dof_iterator;
private:
  typedef eth::grid::EntityCollection< const_element_iterator >
  element_collection_t;

  /// the current grid view factory
  const GRID_VIEW_FACTORY_T& grid_view_factory_;
  /// the overall number of dofs
  size_type num_dofs_;
  /// container of element-wise degrees of freedom
  dofDataSet_t* dofs_on_elements_;
  /// store begin end end iterators to elements in an element
      collection
  element_collection_t* element_collection_;
  /// the overall number of elements which are associated to the dofs
  size_type num_elements_;
public:
  /// default constructor
  FESpace( const GRID_VIEW_FACTORY_T& grid_view_factory );

  /// Begin of element collection
  inline const_element_iterator begin( ) const
  { return element_collection_->begin(); }
  /// End of element collection
  inline const_element_iterator end( )   const
  { return element_collection_->end();   }
  /// Begin of dofs for element e
  const_dof_iterator begin( const element_t& e ) const;
  /// End of dofs for element e
  const_dof_iterator end( const element_t& e ) const;
  /// Return container of stored dofs
  const dofDataSet_t& dofsOnElements( ) const { return
      *dofs_on_elements_; }
  /// Return the grid factory
  const GRID_VIEW_FACTORY_T& gridFactory() const { return
```

```
    grid_view_factory_; }
  /// get the dof's global index
  size_type globalIndex( const const_dof_iterator& dIter ) const;
  /// Get the dof's local index w.r.t. element e
  size_type localIndex(const const_dof_iterator& dIter,
                       const element_t& e ) const;
  /// Returns the local indices w.r.t. to the cell e of the dofs
  /// associated with the entities of codimension CODIM contained
  /// in the intersection of e with index intersectionIndex.
  template< int CODIM >
  inline
  std::vector< int > filter( const element_t& e, int
    intersectionIndex ) const;
  /// returns pointers to the dofs that are associated with the
    intersection
  /// of e with index intersectionIndex
  inline
  std::vector< const Dof* > filterAll( const element_t& e, int
    intersectionIndex ) const;
  /// Returns the local indices (w.r.t. of the cell e) of the dofs
  /// that are associated with the intersection of e with index
  /// intersectionIndex
  inline
  std::vector< int > filterIndices( const element_t& e, int
    intersectionIndex ) const;
  /// get pairs of local->global indices for the cell e
  vector< IndexPair<size_type> > indices( const element_t& e ) const;
  /// get pairs of local->global indices for an intersection of the
    cell e with index intersectionIndex
  std::vector< IndexPair<size_type> > indices( const element_t& e,
    const int intersectionIndex ) const
  /// Get the total number of dofs
  size_type numDofs(void) const { return num_dofs_; }
  /// Get total number of elements
  size_type numElements(void) const { return num_elements_; }
}; // end class FESpace
```

Example 3.6.86 (Index mapping for quadratic Lagrangian FE in BETL)

Refer to Ex. 3.5.3 for details on local and global shape functions for quadratic Lagrangian finite element space $\mathcal{S}_2^0(\mathcal{M})$ on triangular mesh $\mathcal{M}$: a single global shape function is associated to each vertex and edge of $\mathcal{M}$, see Fig. 123.

BETL's numbering *convention* $(*)$
for local shape functions from (3.5.6)



Numbering *convention* $(*)$ for global shape functions of $\mathcal{S}_2^0(\mathcal{M})$ (→ § 3.6.88)

- First number global shape functions associated with vertices.
- Then number global shape functions belonging to edges.
- If there are global shape functions associated to faces (which is the case for quadrilaterals), we number these shape functions too.
- Use numbering provided by `IndexSet` for ordering both vertices and edges, see § 3.6.33.

$(*)$ "Convention" means that this choice can essentially be made in arbitrary ways, but has to be applied consistently throughout the code afterwards.

The above conventions are used for the implementation of the local → global index mapping managed by an **FESpace** object in BETL. The initialization of the d.o.f. handler for quadratic Lagrangian finite elements is done in Code 3.6.84.

The next code accesses the local→global index mapping using the d.o.f handler structure in BETL (compare with Example Ex. 3.6.74). This sample code shows how to use the **FESpace** member object of the **DofHandler** in order to retrieve the local→global index mapping in the form of a matrix of `int`s.

The function `basisFuncIndicesRET` takes **fe:DofHandler** and `GRIDVIEW` arguments together with a geomtric type `RET` and returns an $n \times Q$-EIGEN matrix `dofh`, where

- n is the number of cells of the type `RET`,
- Q is the (maximal) number of local shape functions on cells of type `RET`.

The entry `dofh(k,l)` provides the global index of the local d.o.f. `l` in the cell of type `RET` with index `k`. When invoked for all cell types in a mesh, the function builds an algebraic representation of `locglobmap` from (3.6.72).

The following code creates the index mapping matrix managing the local to global index mapping for a specific reference element type using **dofHandler**/**FESpace** in BETL.

C++11 code 3.6.87: Assmebly of "local→global mapping matrix" ➙ **GITLAB**

```
template< class DOF_HANDLER, class VIEW_TRAITS >
Eigen::Matrix<int, Eigen::Dynamic, Eigen::Dynamic>
  basisFuncIndicesRET(const DOF_HANDLER &dh,
    const eth::grid::GridView<VIEW_TRAITS> &gv,
    const eth::base::RefElType& RET) {
// Obtain a reference to an FESpace from the DofHandler
const auto& fe_space = dh.fespace();
// Fetch reference to global indices
auto& set = gv.indexSet();
// Get the number of local dofs for the reference element type RET
```

```
const size_t max_local_dofs =
    DOF_HANDLER::fespace_t::fe_basis_t::numDofs(RET);
// Get the number of elements of type RET
const size_t num_elements = set.size( RET );
// Define matrix type for matrix describing locglobmap
typedef Eigen::Matrix< int , Eigen::Dynamic , Eigen::Dynamic >
    dense_matrix_t;
// Instantiate dofh matrix with number of columns equal num_elements
dense_matrix_t dofh( num_elements, max_local_dofs );
cout << "Reference element type: " << RET << endl;
// Fill the matrix with the global indices
// Loop over all elements on which the fespace is based on
for( const auto& e : fe_space) {
  if (e.refElType() == RET){
    cout<<"Local to global index mapping for cell of index "<<
        set.index(e) << ": ";
    const auto& local_to_global_mapper_vector = fe_space.indices(e);
    // Loop over local shape functions
    for( const auto map : local_to_global_mapper_vector) {
        // local() and global() methods of map allow to retrieve
        // the actual mapping
        cout<< map.local( )<<"->"<< map.global( ) <<" ";
        // we use it to fill the dofh matrix
        dofh(set.index(e),map.local()) = map.global();
    } // end for
    cout << endl;
  }}
return dofh;
}
```

To demonstrate the numbering of global and local shape functions for quadratic Lagrangian finite elements in BETL, we list their indices for the hybrid mesh from Ex. 3.6.47 drawn in Fig. 153 using the function `printElementInfoFESpace` from ➔ GITLAB. The mainfile ➔ GITLAB calls the functions `printElementInfoFESpace` and `basisFuncIndicesRET` and returns the following output:

```
# Listing of elements as returned by iterator of FESpace
Output element information stored in the FESpace :
TRIA, index: 0, coordinates of corners:
a_1 = [2 0]^T, a_2 = [2 2]^T, a_3 = [1 1]^T,

TRIA, index: 1, coordinates of corners:
a_1 = [2 0]^T, a_2 = [1 1]^T, a_3 = [1 0]^T,

TRIA, index: 2, coordinates of corners:
a_1 = [1 1]^T, a_2 = [2 2]^T, a_3 = [1 2]^T,

QUAD, index: 0, coordinates of corners:
a_1 = [0 0]^T, a_2 = [1 0]^T, a_3 = [1 1]^T, a_4 = [0 1]^T,

QUAD, index: 1, coordinates of corners:
```

```
a_1 = [0 1]^T, a_2 = [1 1]^T, a_3 = [1 2]^T, a_4 = [0 2]^T,
```

By coincidence the cells are arranged in the same order as that given by their global indices. However, this is not guaranteed. The next output listing probes the d.o.f. mapper.

```
# Local→global index mapping matrices as returned from basisFuncIndicesRET
# Type TRIA
Reference element type: TRIA
local to global index mapping for cell of index 0: 0->0 1->1 2->2 3->12
    4->15 5->13
local to global index mapping for cell of index 1: 0->0 1->2 2->3 3->13
    4->11 5->10
local to global index mapping for cell of index 2: 0->2 1->1 2->4 3->15
    4->14 5->17
dofh =  0  1  2 12 15 13
        0  2  3 13 11 10
        2  1  4 15 14 17
# Type QUAD
Reference element type: QUAD
local to global index mapping for cell of index 0: 0->5 1->3 2->2 3->6
    4->8 5->11 6->19 7->9 8->20
local to global index mapping for cell of index 1: 0->6 1->2 2->4 3->7
    4->19 5->17 6->16 7->18 8->21
dofh =  5  3  2  6  8 11 19  9 20

    6  2  4  7 19 17 16 18 21
```

(3.6.88) Global ordering of degrees of freedom in BETL's FESpace

The dof mapper of BETL must number the global basis functions in order to relate basis expansion coefficients to components of vectors. This is required by the second step of Galerkin discretization, remember Section 3.2.

The ordering of global basis functions is as arbitrary as that of local shape functions (→ § 3.6.80) and must be governed by universal convention. BETL adopts the following rules for numbering global finite element basis functions in 2D:

(I) Basis functions are sorted according to the geometric type (roughly, by increasing dimension) of the associated entity as follows:

$$\textbf{POINT} \longrightarrow \textbf{SEGMENT} \longrightarrow \textbf{TRIA} \longrightarrow \textbf{QUAD}.$$

(II) If several global shape functions belong belong to a single geometric entity, their ordering is induced by that implicitly used in the **FEBasis** type contained in the **FESpace**.

Example 3.6.89 ("Location" of global shape functions in BETL)

In this example we examine the assignment of global shape functions to geometric entities as constructed by BETL's `FESpace` for the mesh from Ex. 3.6.47 and the set of local shape functions defined by the type **MyFEBasis** from Ex. 3.6.76 (1/2/2 local shape functions associated with vertices, edges, and elements).

The following function permits us to inspect the geometric type and index number (as provided by the mesh's **IndexSet**). It relies on the helper class **ProbleFEBasis** from Code 3.6.81.

C++11 code 3.6.90: Prints entity types and index numbers for global shape functions handled by an `FESpace` ↪ GITLAB.

```
template< class FESPACE, class VIEW_TRAITS >
void listDOFs(const FESPACE &fe_space,
              const eth::grid::GridView<VIEW_TRAITS> &gv ){
  // Define important types
  using fe_basis_t = typename FESPACE::fe_basis_t;
  using size_t     = typename fe_basis_t::size_type;
  using refEl_t    = eth::base::RefElType;
  using index_t    = typename fe_basis_t::size_type;
  // Vector for storing information about dofs
  std::vector<std::pair<refEl_t,index_t> >
     dofentvec(fe_space.numDofs());
  // Handler for global indices of geometric entities
  auto &set = gv.indexSet();

  // Loop over all elements relevant for the FESpace
  for (const auto& el : fe_space) {
    const refEl_t ret = el.refElType();
    const auto& idx_it = fe_space.indices(el);
    // Loop over all local shape functions
    for (const auto idx : idx_it) {
      // Retreive local index of shape function
      const index_t lsfidx = idx.local();
      // Request type of associated geometric entity
         (POINT,SEGMENT,QUAD,TRIA)
      const refEl_t lsfret =
         ProbeFEBasis<fe_basis_t>::getRETDof(ret,lsfidx);
      // Local number of associated geometric entity
      const index_t entidx =
         ProbeFEBasis<fe_basis_t>::getSubentIdx(ret,lsfidx);
      // Find out global index of geometric entity
      index_t gidx;
      switch (lsfret) {
      case eth::base::RefElType::POINT:
     { gidx = set.template subIndex<2>(el,entidx); break; }
      case eth::base::RefElType::SEGMENT:
     { gidx = set.template subIndex<1>(el,entidx); break; }
      case eth::base::RefElType::TRIA:
      case eth::base::RefElType::QUAD:
     { gidx = set.index(el); break; }
      default: { gidx = 0; ETH_ASSERT(false); }
      }
      dofentvec[idx.global()] = std::make_pair(lsfret,gidx);
    }}
  for (size_t l=0; l<dofentvec.size(); l++) {
    cout << "gsf no. " << l << "-> " << dofentvec[l].first << ",idx =
```

```
        "
        << dofentvec[l].second << std::endl;
    }
} // end listDOFs
```

This is the output produced by `listDOFs` ➙ GITLAB:

```
gsf no. 0-> POINT,idx = 2
gsf no. 1-> POINT,idx = 3
gsf no. 2-> POINT,idx = 7
gsf no. 3-> POINT,idx = 1
gsf no. 4-> POINT,idx = 4
gsf no. 5-> POINT,idx = 0
gsf no. 6-> POINT,idx = 6
gsf no. 7-> POINT,idx = 5
gsf no. 8-> SEGMENT,idx = 0
gsf no. 9-> SEGMENT,idx = 0
gsf no. 10-> SEGMENT,idx = 1
gsf no. 11-> SEGMENT,idx = 1
gsf no. 12-> SEGMENT,idx = 2
gsf no. 13-> SEGMENT,idx = 2
gsf no. 14-> SEGMENT,idx = 3
gsf no. 15-> SEGMENT,idx = 3
gsf no. 16-> SEGMENT,idx = 4
gsf no. 17-> SEGMENT,idx = 4
gsf no. 18-> SEGMENT,idx = 5
gsf no. 19-> SEGMENT,idx = 5
gsf no. 20-> SEGMENT,idx = 6
gsf no. 21-> SEGMENT,idx = 6
```

```
gsf no. 22-> SEGMENT,idx = 7
gsf no. 23-> SEGMENT,idx = 7
gsf no. 24-> SEGMENT,idx = 8
gsf no. 25-> SEGMENT,idx = 8
gsf no. 26-> SEGMENT,idx = 9
gsf no. 27-> SEGMENT,idx = 9
gsf no. 28-> SEGMENT,idx = 10
gsf no. 29-> SEGMENT,idx = 10
gsf no. 30-> SEGMENT,idx = 11
gsf no. 31-> SEGMENT,idx = 11
gsf no. 32-> TRIA,idx = 0
gsf no. 33-> TRIA,idx = 0
gsf no. 34-> TRIA,idx = 1
gsf no. 35-> TRIA,idx = 1
gsf no. 36-> TRIA,idx = 2
gsf no. 37-> TRIA,idx = 2
gsf no. 38-> QUAD,idx = 0
gsf no. 39-> QUAD,idx = 0
gsf no. 40-> QUAD,idx = 1
gsf no. 41-> QUAD,idx = 1
```

3.6.4.3 Assembly: Cell-oriented Algorithms

(3.6.91) Cell-oriented assembly of finite element Galerkin matrix and right hand side vector

Another fundamental design principle for the assembly realized already in Code 3.3.35 was to rely on

loops only over mesh cells combined with purely local operations.

Notion: local operations $\hat{=}$
- ✦ require data only from fixed "neighbourhood" of cell K
- ✦ computational effort "$O(1)$": independent of $\sharp\mathcal{M}$

This design principle is honored in the MATLAB-style "pseudo-code" Code 3.6.92, which extends Code 3.3.35, which was confined to linear Lagrangian finite elements, to general finite element methods. The local→global index mapping is realized through the `locglobmap`-function/matrix.

Pseudocode 3.6.92: Abstract assembly routine for finite element Galerkin matrices

```
A = sparse(N,N); % Allocated empty sparse matrix
for k = Mesh.Elements' % loop over all cells
  % Obtain number Q(K) of local shape functions, see Def. 3.6.69
  Qk  = no_loc_shape_functions(k);
  % Local operation: compute Q(K) × Q(K) element matrix → Def. 3.6.69,
  % usually incurs cost of only "O(1)"
  Ak = getElementMatrix(k);
  % Get vector of global indices (length Q(K));
  % Usage of locglobmap as in Ex. 3.6.74
  idx = locglobmap(k,(1:Qk));
  % Add local contributions to global matrix
  for i=1:Qk
    for j=1:Qk
       A(idx(i),idx(j)) = A(idx(i),idx(j)) + Ak(i,j);
    end
  end
end
```

Note that in Code 3.3.38 the local→global index mapping could be inferred from the mesh data directly through the `Elements`-vetor.

The very same ideas in a somewhat simpler version govern the initialization of the right hand side vector from element (load) vectors. The following MATLAB-style "pseudocode" Code 3.6.93 extends Code 3.3.47 and supplies a generic finite element assembly algorithm for right hand side vectors:

Pseudocode 3.6.93: Generic assembly algorithm for finite element right hand side vectors

```
f = zeros(N,1); % Allocated zero vector of appropriate length
for k = Mesh.Elements' % loop over all cells
  % Obtain number Q(K) of local shape functions, see Def. 3.6.69
  Qk  = no_loc_shape_functions(k);
  % Local operation: compute element vector, length Q(K) →
     Def. 3.6.69,
  % (usually incurs cost of only "O(1)")
  phi_k = getElementVector(k);
  % Get vector of global indices (length Q(K));
  % Usage of locglobmap as in Ex. 3.6.74
  idx = locglobmap(k,(1:Qk));
  % Add local contributions to global matrix
  for i=1:Qk
     f(idx(i)) = f(idx(i) + phi_k(i,j);
  end
end
```

Example 3.6.94 (An assembler class in BETL: Global assembly of Galerkin Matrices)

In this example, we present an implementation of the assembly of a Galerkin matrix according to Code 3.6.92. It is given by the class **NPDE::GalerkinMatrixAssembler** ➙ GITLAB.

The following ingredients are necessary to define the class **NPDE::GalerkinMatrixAssembler**:

- a reference to a (constant) `ELEM_MAT_BUILDER` object that computes the element matrix for a given element.

  An object of type `ELEM_MAT_BUILDER` must provide a *static* method

  ```
  template<class BUILDER_DATA_T, class ELEMENT>
  static result_t eval(const BUILDER_DATA_T& data, const ELEMENT&
     el);
  ```

  which computes the element matrix for the cell `el`. The arguments have the following meanings.

  - `el` is a reference to a cell of the mesh, the "element" for which the element matrix is to be computed.
  - `data` is a functor object meant to pass additional data like coefficients and source functions.
  - `result_t` is the type of the return value. It has to be a $N \times N$ EIGEN matrix type, where N corresponds to the number of local dofs on the element `el`. The return value is the local element (stiffness) matrix.

  This `eval()`-interface has been chosen for the sake of compatibility with BETL's built-in assembly. Note that being *static* the only way to pass data to the computation of the element matrix is through the `const BUILDER_DATA_T & data` argument.

  Types meeting the requirements of `ELEM_MAT_BUILDER` can be found in Code 3.6.125 and Code 3.7.37.

  A reference to a (constant) **fe::FESpace**-compatible object (template paramter `FESPACE_TEST_T`) that handles the test finite element space (and its dofs).

- A reference to a (constant) **fe::FESpace**-compatible object (template parameter `FESPACE_TRIAL_T`) that handles the trial finite element space (and its dofs).

- A reference to a (constant) functor object (template parameter `BUILDER_DATA_T`) to be passed to the `eval()` method of `ELEM_MAT_BUILDER`.

C++11 code 3.6.95: NPDE assembler in BETL: implementation of global assembly of Galerkin matrix ➙ GITLAB

```
template< typename ELEM_MAT_BUILDER >
class GalerkinMatrixAssembler{public:
 typedef double numeric_t;
 // Type for CRS matrix in EIGEN
 typedef Eigen::SparseMatrix< numeric_t > sparseMatrix_t;
 // Elementary triplet type, see [14, Section 1.7.3].
 typedef Eigen::Triplet< numeric_t > triplet_t;
 // Triplet container:  Matrix in triplet format
 typedef std::vector< triplet_t > tripletMatrix_t;
 // Type for small dense matrix, e.g, the element matrix
 typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, Eigen::Dynamic >
    matrix_t;
 // type of local matrix assembler providing static eval() method
```

```
typedef ELEM_MAT_BUILDER localAssembler_t;

GalerkinMatrixAssembler(void) { /* empty */ }
// Assemble global FE matrix as list of triplets
template<class FESPACE_TEST_T, class FESPACE_TRIAL_T, class
    BUILDER_DATA_T>
tripletMatrix_t assembleTripletMatrix(
  const FESPACE_TEST_T  &  fe_test,
  const FESPACE_TRIAL_T &  fe_trial,
  const BUILDER_DATA_T  & data );
// Assemble global FE Galerkin matrix as CRS matrix
template<class FESPACE_TEST_T, class FESPACE_TRIAL_T, class
    BUILDER_DATA_T>
sparseMatrix_t assembleMatrix(
  const FESPACE_TEST_T  &  fe_test,
  const FESPACE_TRIAL_T &  fe_trial,
  const BUILDER_DATA_T & data );
};// end class GalerkinMatrixAssembler
```

The next listings give the details of the implementation of the actual assembly routine.

C++11 code 3.6.96: NPDE assembler in BETL: implementation of method `assembleTripletMatrix` in Code 3.6.95 ➙ GITLAB

```
template< typename ELEM_MAT_BUILDER >
template< class FESPACE_TEST_T, class FESPACE_TRIAL_T,
        class BUILDER_DATA_T >
typename GalerkinMatrixAssembler< ELEM_MAT_BUILDER >::tripletMatrix_t
GalerkinMatrixAssembler< ELEM_MAT_BUILDER >::assembleTripletMatrix(
  const FESPACE_TEST_T& fe_test,const FESPACE_TRIAL_T &fe_trial,const
      BUILDER_DATA_T& data) {
  // initialize empty container of triplets:  this will store
  // contributions to the global matrix
  tripletMatrix_t contributions;
  // initialize the local aseembler's static data
  ELEM_MAT_BUILDER::initialize();
  // Loop over all the grid elements:  cell-oriented assembly
  for(const auto& el : fe_test){
    // compute (small local) element matrix
    const auto lclMat = localAssembler_t::eval( data, el );
    // store local contributions in vector of triplets
    // get the local->global indices for this element
    const auto id_test  = fe_test.indices(el);
    const auto id_trial = fe_trial.indices(el);
    // for all rows (test func)
    for( const auto test_idx : id_test ) {
      const auto row_loc = test_idx.local( );
      const auto row_glo = test_idx.global( );
      // and all columns (trial func)
```

```
25        for( const auto trial_idx : id_trial ) {
26          const auto col_loc = trial_idx.local( );
27          const auto col_glo = trial_idx.global( );
28          const auto value = lclMat( row_loc, col_loc );
29          // map the contribution from the local matrix and store them
30          contributions.push_back( triplet_t( row_glo, col_glo, value )
              );
31      }}}
32    // Move triplet list into return argument.  No copying of data!
33    return(contributions); //@@
34  }
```

Line 13 loops over all cells (elements) contributing to the finite element space, see the discussion of **fe:FESpace** in § 3.6.83. This realizes the outer loop in Line 2, Code 3.6.92.

Line 21 implements a loop over all local shape functions in the test space corresponding to the rows of the element matrix.

Line 25 loops over the local shape functions in the trial space, that is, the columns of the element matrix.

Line 13–Line 33 in Code 3.6.96 show that the assembly is performed over all global basis functions included in `fe_test` and `fe_trial`. Therefore, in the case of essential boundary conditions, as in Section 3.6.6, additional manipulations must be carried out outside of this class. This will be further discussed in Ex. 3.6.181.

A working example demonstrating `NPDE::GalerkinMatrixAssembler` in action can be found in ➙ GITLAB.

Remark 3.6.97 (Variational problems with different trial and test spaces)

So far we have always considered variational problems where trial and test space coincided both on the continuous and discrete level.

This need not be the case, because the natural Sobolev spaces for a bilinear form might differ, as for

$$\mathsf{a}(u,v) := \int_\Omega \boldsymbol{c}\cdot\mathbf{grad}\,u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad u\in H^1(\Omega),\ v\in L^2(\Omega)\,. \tag{3.6.98}$$

The simplest Galerkin finite element discretization of this bilinear form on a mesh $\mathcal{M}$ would employ $\mathcal{S}_1^0(\mathcal{M})\subset H^1(\Omega)$ for u and merely $\mathcal{M}$-piecewise constant functions as test space. The corresponding Galerkin matrix could be built with **GalerkinMatrixAssembler** from Code 3.6.95 by providing suitable type as `FESPACE_TEST_T` and `FESPACE_TRIAL_T`.

Another more exotic case is the deliberate use of different finite element subspaces even in the case of a variational problem for which test and function space are the same. This generalization of the Galerkin approach is called a Petrov-Galerkin discretization.

(3.6.99) Cell oriented assembly: "$O(N)$" computational effort

If we assume "constant cost" for the local operations then we conclude for the asymptotic computational effort as we use meshes with more and more elements:

$$\boxed{\text{Computational cost(Assembly of Galerkin matrix } \mathbf{A}) = O(\sharp\mathcal{M})}$$

This statement can be specialized:

For Lagrangian FEM of fixed degree p ($\rightarrow$ Section 3.5):

the total computational effort is of the order $O(\sharp\mathcal{M}) = O(N)$, $N := \dim \mathcal{S}_p^0(\mathcal{M})$.

(3.6.100) Global assembly of right hand side vector in BETL

The cell oriented assembly of the right hand side (load) vector according to Code 3.6.93 makes use of element (load) vectors, see Def. 3.6.69.

As in the case of the class **NPDE::GalerkinMatrixAssembler** from Ex. 3.6.94, its implementation in BETL relies on objects of type **fe::FESpace** (template parameters `FESPACE_TEST_T`, `FESPACE_TRIAL_T`), and a `ELEM_VEC_BUILDER` object. As in Ex. 3.6.94, the latter has to provide a *static* `eval()` method with the following signature:

```
template< class BUILDER_DATA_T, class ELEMENT>
static vector_t eval(const BUILDER_DATA_T &F,const ELEMENT &el)
```

The arguments are analogous to those of the `eval()` method of the `ELEM_MAT_BUILDER` type in Ex. 3.6.94, with the exception of the return type `vector_t`, which is supposed to represent an EIGEN column vector of length N_Q now, where N_Q is the number of local degrees of freedom. In addition, the `eval()` method of an `ELEM_VEC_BUILDER` wants an object of type `BUILDER_DATA_T`. This can be used to pass arbitrary data to the code computing the element load vector. This is a welcome possibility, because, as a static method, `eval()` itself cannot access any class data members.

An implementation of a class conforming with the concept of an `ELEM_VEC_BUILDER` is given in Code 3.6.168 below.

The following code implements a generic assembler class for right hand side vectors **NPDE::LoadVectorAssembler**.

C++11 code 3.6.101: NPDE assembler in BETL: code for global assembly of right hand side vector ➙ GITLAB

```
template< typename ELEM_VEC_BUILDER >
class LoadVectorAssembler{
public:
typedef double numeric_t;
// the right hand side vector
typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, 1 > vector_t;
// type for small dense matrix (just for convenience)
typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, Eigen::Dynamic >
    matrix_t;
// type of local matrix assembler (needs info on basis datas to use)
typedef ELEM_VEC_BUILDER localAssembler_t;

LoadVectorAssembler(void){   /* empty */  }
// Assemble global right hand side vector
```

```
template< class FESPACE_TEST_T, class BUILD_DATA_T >
vector_t assembleRhs( const FESPACE_TEST_T& fe_test,
          const BUILD_DATA_T&  data );
};// end class LoadVectorAssembler
```

Parallel to the method `assembleTripletMatrix` from **GalerkinMatrixAssembler**, see Code 3.6.96, the method `assembleRhs` performs cell-oriented assembly of the right hand side vector according to Code 3.6.93.

C++11 code 3.6.102: Implementation of LoadVectorAssembler ➙ GITLAB

```
template< typename ELEM_VEC_BUILDER >
template< class FESPACE_TEST_T, typename BUILDER_DATA_T >
typename LoadVectorAssembler<ELEM_VEC_BUILDER>::vector_t
LoadVectorAssembler<ELEM_VEC_BUILDER>::assembleRhs(
  const FESPACE_TEST_T&  fe_test, const BUILDER_DATA_T&  data ) {
  // Initialize the integrator's static data
  localAssembler_t::initialize( );
  // Initialize the right hand side vector with zero
  vector_t rhsvec(fe_test.numDofs()); rhsvec.setZero();
  // Loop over all the grid elements (cells)
  for( const auto& el : fe_test ) {
    // Compute element load vector
    const auto lclVec = localAssembler_t::eval( data, el );
    // Store local contributions to r.h.s.  vector
    // First get the local->global indices for this element
    const auto id  = fe_test.indices( el );
    // for all rows (test func)
    for( const auto test_idx : id ) {
      const auto row_loc = test_idx.local( );
      const auto row_glo = test_idx.global( );
      // add contribution to the correct index in the global vector
      rhsvec(row_glo) += lclVec(row_loc);
    }}
  return(rhsvec);
}
```

From the above listing, we see that **NPDE::LoadVectorAssembler** in Code 3.6.101 adds all global basis functions contained in `fespace`. Consequently, in the case of essential boundary additional manipulations must be carried out outside of this class (as we will see in Ex. 3.6.181).

The class `NPDE::LoadVectorAssembler` in Code 3.6.101 is a simplified version of the class `fem::LinearF` in `Library/fem_operator/linear_form.hpp`.

A working example demonstrating **NPDE::LoadVectorAssembler** in action can be found in ➙ GITLAB.

Example 3.6.103 (Global assembly of boundary contributions to Galerkin matrices in BETL)

In the case of a second-order elliptic boundary value problem with Robin boundary conditions (→ Ex. 2.7.5), we face a variational problem of the form (→ Ex. 2.9.6)

$$u \in H^1(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} q(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S(\boldsymbol{x}) = \int_\Omega fv\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{3.6.104}$$

with uniformly positive definite functions $\kappa \in C^0_{\mathrm{pw}}(\overline{\Omega})$, $q \in C^0_{\mathrm{pw}}(\partial\Omega)$. The implementation of a Lagrangian finite element Galerkin discretization requires the evaluation of integrals over (parts of) the boundary.

In this example, we present an implementation of the assembly of a Galerkin matrix according to Code 3.6.92 over **Intersection**-objects. It is given by the class **NPDE::IntersectionGalMatAsse**.

The following ingredients are necessary to define the class **NPDE::IntersectionGalMatAsse**:

- a reference to a (constant) `INTER_MAT_BUILDER` object that computes the element matrix for a given intersection. An object of type `INTER_MAT_BUILDER` must provide a *static* method `eval` similar to the one described in Ex. 3.6.94.

  ```
  template<class BUILDER_DATA_T, class INTERSECTION>
  static result_t eval(const BUILDER_DATA_T& data, const
     INTERSECTION& ic);
  ```

  A reference to a (constant) **fe::FESpace**-compatible object (template paramater `FESPACE_TEST_T`) that handles the test finite element space (and its dofs).
- A reference to a (constant) **fe::FESpace**-compatible object (template parameter `FESPACE_TRIAL_T`) that handles the trial finite element space (and its dofs).
- A reference to a (constant) functor object (template parameter `BUILDER_DATA_T`) to be passed to the `eval()` method of `INTER_MAT_BUILDER`.

C++11 code 3.6.105: NPDE assembler in BETL: Implementation of global assembly of Galerkin matrix over boundary ➟ GITLAB

```
template< typename INTER_MAT_BUILDER >
class IntersectionGalMatAsse{
public:
  typedef double numeric_t;
  // Type for CRS matrix in EIGEN
  typedef Eigen::SparseMatrix< numeric_t > sparseMatrix_t;
  // Elementary triplet type
  typedef Eigen::Triplet< numeric_t > triplet_t;
  // Matrix in triplet format
  typedef std::vector< triplet_t > tripletMatrix_t;
  // Type for small dense matrix
  typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, Eigen::Dynamic >
     matrix_t;

  // type of local matrix assembler (needs info on basis functions to
  // use, and the quadrature rule to apply for integration if needed)
  typedef INTER_MAT_BUILDER localAssembler_t;

```

```
  IntersectionGalMatAsse(void) { /* empty */ }
  // Assemble global FE matrix as list of triplets
  template<class FESPACE_TEST_T, class FESPACE_TRIAL_T, class
     BUILDER_DATA_T, class INTERSECTION >
  tripletMatrix_t assembleTripletMatrix(
     const FESPACE_TEST_T  &  fe_test,
     const FESPACE_TRIAL_T &  fe_trial,
     const BUILDER_DATA_T & data,
     const std::vector< const INTERSECTION* >& intersections );
  // Assemble global FE matrix as CRS matrix
  template<class FESPACE_TEST_T, class FESPACE_TRIAL_T, class
     BUILDER_DATA_T, class INTERSECTION >
  sparseMatrix_t assembleMatrix(
     const FESPACE_TEST_T  &  fe_test,
     const FESPACE_TRIAL_T &  fe_trial,
     const BUILDER_DATA_T & data,
     const std::vector< const INTERSECTION* >& intersections );
};// end class IntersectionGalMatAsse
```

The next listing gives the details of the implementation of the actual assembly routine `assembleTripletMatrix`. The main loop adds the triplets contributed by a particular element to the global triplet container.

C++11 code 3.6.106: NPDE assembly in BETL: implementation of method `assembleTripletMatrix` in Code 3.6.105 ➙ GITLAB

```
template< typename INTER_MAT_BUILDER >
template< class FESPACE_TEST_T, class FESPACE_TRIAL_T,
        class BUILDER_DATA_T, class INTERSECTION >
typename IntersectionGalMatAsse< INTER_MAT_BUILDER >::tripletMatrix_t
IntersectionGalMatAsse< INTER_MAT_BUILDER >::assembleTripletMatrix(
 const FESPACE_TEST_T& fe_test,
 const FESPACE_TRIAL_T& fe_trial,
 const BUILDER_DATA_T& data,
 const std::vector< const INTERSECTION* >& intersections ) {
  // Initialize empty container of triplets:  this will store
  // contributions to the global matrix
  tripletMatrix_t contributions;
  // Initialize the local assembler's static data
  INTER_MAT_BUILDER::initialize( );
  // loop over all the grid elements
  for( const auto& I : intersections ) {
    // Compute element matrix for restricted set of local shape
       functions
    const auto lclMat = localAssembler_t::eval( data, *I );
    // Store local contributions in vector of triplets
    // Fetch the local->global index mapping for this element
    const auto id_test  = fe_test.indices( *(I->inside()),
       I->indexInInside() );
    const auto id_trial = fe_trial.indices( *(I->inside()),
       I->indexInInside() );
```

```
    // for all rows (test local shape functions)
    for( const auto test_idx : id_test ) {
      const auto row_loc = test_idx.local( );
      const auto row_glo = test_idx.global( );
      // and all columns (trial local shape functions)
      for( const auto trial_idx : id_trial ) {
     const auto col_loc = trial_idx.local( );
     const auto col_glo = trial_idx.global( );
     const auto value = lclMat( row_loc, col_loc );
     // Store contribution from local matrix in global matrix
     contributions.push_back( triplet_t( row_glo, col_glo, value ) );
      }}}
  // Move triplet list into return argument.  No copying of data!
  return(contributions);
}
```

A working example demonstrating `NPDE::IntersectionGalMatAsse` in action can be found in ➙ GITLAB.

(3.6.107) Global assembly of boundary contributions to the right hand side vector in BETL

In the variational formulation of a second-order elliptic Neumann problem with non-zero Neumann data $h \in L^2(\partial\Omega)$ involves a right hand side linear functional of the form $v \mapsto \int_{\partial\Omega} hv \,\mathrm{d}S$, see Ex. 2.9.10.

In this example we present a BETL class providing the finite element discretization of such functionals. As in the case of the class **NPDE::IntersectionGalMatAsse** from Ex. 3.6.103, the implementation in BETL relies on objects of type **fe::FESpace** (template parameters `FESPACE_TEST_T`, `FESPACE_TRIAL_T`), and a `INTER_MAT_BUILDER` object. As in Ex. 3.6.103, the latter has to provide a *static* `eval()` method. The structure of the class is very similar to that of **NPDE::LoadVectorAssembler** in Code 3.6.101.

C++11 code 3.6.108: Global assembly of boundary contribution to right hand side ➙ GITLAB

```
template< typename INTER_VEC_BUILDER >
class IntersectionLoadVectAsse {
public:
  typedef double numeric_t;
  // the right hand side vector
  typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, 1 > vector_t;
  typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, Eigen::Dynamic >
      matrix_t;
  // type of local matrix assembler providing eval()) method
  typedef INTER_VEC_BUILDER localAssembler_t;

  IntersectionLoadVectAsse(void) {}
  // Assemble global finite element Galerkin matrix
  template< class FESPACE_TEST_T, class BUILDER_DATA_T, class
      INTERSECTION >
  vector_t assembleRhs(const FESPACE_TEST_T& fe_test,
```

```
    const BUILDER_DATA_T &data,
    const std::vector< const INTERSECTION* >& intersections);
}; // end class IntersectionLoadVectAsse
```

A listing of the counterpart of the method `assembleRhs` of **LoadVectorAssembler** from Code 3.6.101 is given now.

C++11 code 3.6.109: Implementation of `assembleRhs` of IntersectionLoadVectAsse from cn Code 3.6.108 ➙ GITLAB

```
1  template< class FESPACE_TEST_T, class BUILDER_DATA_T, class
     INTERSECTION >
2  vector_t assembleRhs(const FESPACE_TEST_T&   fe_test,
3    const BUILDER_DATA_T&  data,
4    const std::vector< const INTERSECTION* >& intersections){
5    // In case initialize the integrator's static data
6    localAssembler_t::initialize( );
7    // Initialize vector
8    vector_t rhsvec(fe_test.numDofs()); rhsvec.setZero();
9    // loop over all the grid elements
10   for( const auto I : intersections ){
11     // assemble local FE vector
12     const auto lclVec = localAssembler_t::eval( data, *I );
13     // Store local contributions in global right hand side vector
14     // Fetch the local->global indices for this element
15     const auto id  =
         fe_test.indices(*(I->inside()),I->indexInInside());
16     // loop over relevant components of element vector
17     for( const auto test_idx : id ) {
18       const auto row_loc = test_idx.local( );
19       const auto row_glo = test_idx.global( );
20       // add contribution to the correct index in the global vector
21       rhsvec( row_glo ) += lclVec( row_loc );
22     }}
23   return(rhsvec);
24 }
```

Note that in Line 15 a set of local shape functions is selected that is associated with the current intersection.

The class `NPDE::IntersectionLoadVectAsse` in Code 3.6.108 is a simplified version of the class `fem::IntersectionLinearForm` in `Library/fem_operator/intersection_linear_form.hp`

A working example demonstrating the use of **NPDE::IntersectionLoadVectAsse** can be found in ➙ GITLAB.

Example 3.6.110 (Driver code for global assembly in BETL)

The following code demonstrates the use of global assembly facilities introduced in **??**, and Code 3.6.101 together with the local assembler classes for local computations from Code 3.6.125 and Code 3.6.129. The

code relies on two classes supplying suitable static `eval()` functions: **NPDE::AnalyticStiffnessLocalAssembler** see Code 3.6.125, and **NPDE::LoadVectorAssembler**, see Code 3.6.129.

C++11 code 3.6.111: Building global matrix and global load vector using assembler classes in BETL ⇝ GITLAB

```
// ASSEMBLING FE GALERKIN MATRIX
typedef double numeric_t;
typedef Eigen::SparseMatrix< numeric_t > sparseMatrix_t;
// type of objects computing element matrix -> using Code 3.6.125
typedef NPDE::AnalyticStiffnessLocalAssembler
    aStiffnessMatAssembler_t;
// type taking care of assembly of Galerkin matrix
typedef NPDE::GalerkinMatrixAssembler< aStiffnessMatAssembler_t >
    AStiffGalMatA_t;
// instantiate corresponding object
AStiffGalMatA_t A;
// compute the (big) Galerkin (stiffness) matrix
const sparseMatrix_t& Ah =
  A.assembleMatrix( dh.fespace(), dh.fespace(), 1.0 );
// ASSEMBLING FE RIGHT HAND SIDE VECTORS
typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, 1 > vector_t;
// Define the source function f
using coord_t = typename gridTraits_t::template
fixedSizeMatrix_t<gridTraits_t::dimWorld,1 >;
// Source function for the right hand side functional
const auto f_double = [](const coord_t& x) {
  double res = x(0) * x(1); return res; };

// type computing local element vectors
typedef NPDE::LocalVectorAssembler  trapLocFunAssembler_t;
// type in charge of computing the right hand side vector using
    Code 3.6.129
typedef NPDE::LoadVectorAssembler< trapLocFunAssembler_t >
    traplinearForm_t;
// instantiate corresponding object
traplinearForm_t tF;
// compute the global functional vector
const vector_t& tfh = tF.assembleRhs( dh.fespace(), f_double );
```

3.6.4.4 Assembly: Linear algebra perspective

There is a formal "mathematical" way to express assembly in the language of linear algebra in terms of sums of matrix products. This is presented in the next theorem:

Theorem 3.6.112. Assembly through index mapping matrices

The stiffness matrix and load vector can be obtained from their cell counterparts, the element (stiffness) matrix $\mathbf{A}_K$ and element (load) vector $\vec{\boldsymbol{\varphi}}_K$ (→ Def. 3.6.69), by

$$\mathbf{A} = \sum_K \mathbf{T}_K^\top \mathbf{A}_K \mathbf{T}_K \quad , \quad \vec{\boldsymbol{\varphi}} = \sum_K \mathbf{T}_K^\top \vec{\boldsymbol{\varphi}}_K \, , \tag{3.6.113}$$

with the index mapping matrices ("T-matrices") $\mathbf{T}_K \in \mathbb{R}^{Q,N}$, defined by

$$(\mathbf{T}_K)_{ij} := \begin{cases} 1 & \text{, if } \quad (b_N^j)_{|K} = b_K^i \, , \\ 0 & \text{, otherwise.} \end{cases} \qquad 1 \le i \le Q, 1 \le j \le N \, . \tag{3.6.114}$$

☞ Every index mapping matrix has exactly one non-vanishing entry per row! (why?)

"MATLAB pseudo-code" for the initialization of a sparse index mapping matrix based on the local→global index map introduced in (3.6.72), `Qk` = number opf local shape functions,

```
TK = sparse(1:Qk,locglobmap(K,1:Qk),ones(Qk,1));
```

Proof. (of Thm. 3.6.112) Use the definition of the entries of the Galerkin matrix, of the element matrix (→ Def. 3.6.69), and of the local shape functions (→ Def. 3.4.19):

$$(\mathbf{A})_{ij} = \mathsf{a}(b_N^j, b_N^i) = \sum_{K \in \mathcal{M}} \mathsf{a}_K(b_{N|K}^j, b_{N|K}^i) =$$

$$\sum_{\substack{K \in \mathcal{M}, \operatorname{supp}(b_N^j) \cap K \neq \emptyset, \\ \operatorname{supp}(b_N^i) \cap K \neq \emptyset}} \mathsf{a}_K(b_K^{l(j)}, b_K^{l(i)}) = \sum_{\substack{K \in \mathcal{M}, \operatorname{supp}(b_N^j) \cap K \neq \emptyset, \\ \operatorname{supp}(b_N^i) \cap K \neq \emptyset}} (\mathbf{A}_K)_{l(i),l(j)} \, .$$

Here, $l(i) \in \{1, \dots, Q\}$, $1 \le i \le N \mathrel{\hat{=}}$ index of the local shape function corresponding to the global shape function b_N^i on K.

➢ By (3.6.114), the indices $l(i)$ encode the T-matrix according to

$$(\mathbf{T}_K)_{l(i),i} = 1 \, , \quad i = 1, \dots, N \, ,$$

where all other entries of $\mathbf{T}_K$ are understood to vanish.

$$\Rightarrow \qquad (\mathbf{A})_{ij} = \sum_{\substack{K \in \mathcal{M}, \operatorname{supp}(b_N^j) \cap K \neq \emptyset, \\ \operatorname{supp}(b_N^i) \cap K \neq \emptyset}} \sum_{l=1}^{Q} \sum_{n=1}^{Q} (\mathbf{T}_K)_{li} (\mathbf{A}_K)_{ln} (\mathbf{T}_K)_{nj} \, .$$

The rules for matrix multiplication give the assertion of the theorem. □

Example 3.6.115 (Index mapping matrix for linear Lagrangian finite elements on triangular mesh)

The local $\to$ global index mapping for linear finite elements with vertex associated global basis functions and three local basis functions was studied in Section 3.3.5, see also Rem. 3.6.73.

This example is connected to Ex. 3.6.74.

Using the local/global numbering indicated beside we find

$$\to \mathbf{T}_{K^*} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

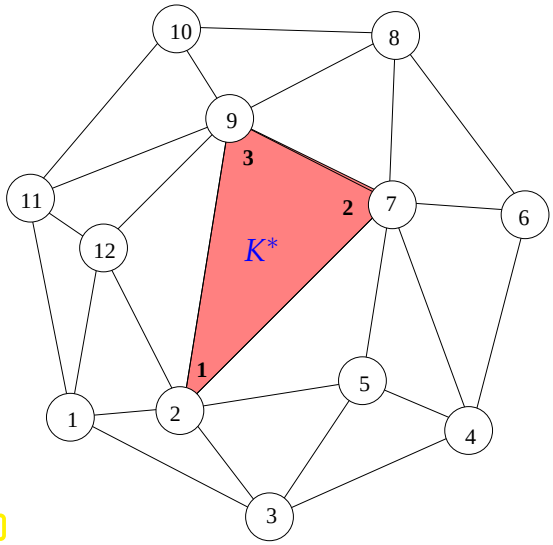


Fig. 159

Now we can rephrase the operation of Code 3.6.92 and Code 3.6.96 from a linear algebra point of view:

$$\boxed{\text{Cell oriented assembly} \quad \leftrightarrow \quad (3.6.113) \leftrightarrow \mathbf{A} = \sum_K \mathbf{T}_K^\top \mathbf{A}_K \mathbf{T}_K}$$

$$\Updownarrow$$

$$\mathbf{A} = \sum_K \mathbf{T}_K^\top \mathbf{A}_K \mathbf{T}_K := \left\{ \begin{array}{l} \textbf{foreach } K \in \mathcal{M} \textbf{ do} \\ \quad \textit{local} \text{ operations on } K \ (\to \mathbf{A}_K) \text{ and } \mathbf{A} = \mathbf{A} + \mathbf{T}_K^\top \mathbf{A}_K \mathbf{T}_K \\ \textbf{enddo} \end{array} \right\}$$

Obviously, this is little to do with the actual implementation and may just serve as a convenient notation.

3.6.5 Local computations

We have seen that the (global) Galerkin matrix and right hand side vector are conveniently generated by "assembling" entries of element (stiffness) matrices and element (load) vectors.

Now we study the computation of these local quantities for Lagrangian finite elements on 2nd-order scalar linear boundary value problems in weak form, see also Section 3.3.5 and Section 3.3.6.

3.6.5.1 Analytic formulas for entries of element matrices

First option: Direct analytic evaluations (➥ "closed form" expressions)

We discuss this for the bilinear form related to $-\Delta$, triangular Lagrangian finite elements of degree p, Section 3.5.1, Def. 3.5.2:

$$K \text{ triangle:} \quad \mathsf{a}_K(u,v) := \int_K \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} \quad \blacktriangleright \quad \text{element stiffness matrix .}$$

Use barycentric coordinate representations of local shape functions, in 2D

$$b_K^i = \sum_{\boldsymbol{\alpha} \in \mathbb{N}_0^3, |\boldsymbol{\alpha}| \le p} \kappa_{\boldsymbol{\alpha}} \, \lambda_1^{\alpha_1} \lambda_2^{\alpha_2} \lambda_3^{\alpha_3} \,, \quad \kappa_{\boldsymbol{\alpha}} \in \mathbb{R} \,, \tag{3.6.116}$$

where λ_i are the affine linear barycentric coordinate functions (linear shape functions), see Fig. 99.

For the barycentric coordinate representation of the quadratic local shape functions see (3.5.6), for a justification of (3.6.116) consult Rem. 3.7.10.

$$\Rightarrow \quad \mathbf{grad}\, b_K^i = \sum_{\boldsymbol{\alpha}\in\mathbb{N}_0^3,\,|\boldsymbol{\alpha}|\le p} \kappa_{\boldsymbol{\alpha}} \Big(\alpha_1 \lambda_1^{\alpha_1-1}\lambda_2^{\alpha_2}\lambda_3^{\alpha_3}\,\mathbf{grad}\,\lambda_1 + \alpha_2\lambda_1^{\alpha_1}\lambda_2^{\alpha_2-1}\lambda_3^{\alpha_3}\,\mathbf{grad}\,\lambda_2 + \alpha_3\lambda_1^{\alpha_1}\lambda_2^{\alpha_2}\lambda_3^{\alpha_3-1}\,\mathbf{grad}\,\lambda_3 \Big) . \tag{3.6.117}$$

To evaluate: $$\int_K \lambda_1^{\beta_1}\lambda_2^{\beta_2}\lambda_3^{\beta_3}\,\mathbf{grad}\,\lambda_i \cdot \mathbf{grad}\,\lambda_j \,\mathrm{d}\boldsymbol{x}\,, \quad i,j\in\{1,2,3\},\ \beta_k\in\mathbb{N}\,. \tag{3.6.118}$$

The (constant!) gradients of barycentric coordinate functions have already been computed in Section 3.3.5 on Page 200, see also Rem. 3.3.24.

If $\mathbf{a}^1, \mathbf{a}^2, \mathbf{a}^3$ vertices of K (counterclockwise ordering):

$$\lambda_1(\boldsymbol{x}) = \frac{1}{2|K|}\left(\boldsymbol{x} - \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix}\right) \cdot \begin{bmatrix} a_2^2 - a_2^3 \\ a_1^3 - a_1^2 \end{bmatrix},$$
$$\lambda_2(\boldsymbol{x}) = \frac{1}{2|K|}\left(\boldsymbol{x} - \begin{bmatrix} a_1^3 \\ a_2^3 \end{bmatrix}\right) \cdot \begin{bmatrix} a_2^3 - a_2^1 \\ a_1^1 - a_1^3 \end{bmatrix},$$
$$\lambda_3(\boldsymbol{x}) = \frac{1}{2|K|}\left(\boldsymbol{x} - \begin{bmatrix} a_1^1 \\ a_2^1 \end{bmatrix}\right) \cdot \begin{bmatrix} a_2^1 - a_2^2 \\ a_1^2 - a_1^1 \end{bmatrix}.$$

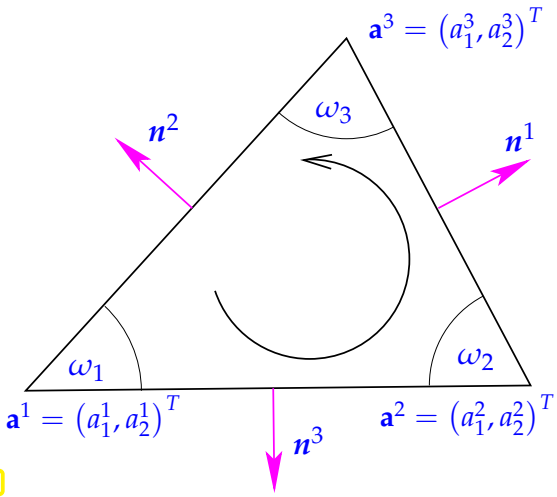


Fig. 160

$$\mathbf{grad}\,\lambda_1 = \frac{1}{2|K|}\begin{bmatrix} a_2^2 - a_2^3 \\ a_1^3 - a_1^2 \end{bmatrix}, \quad \mathbf{grad}\,\lambda_2 = \frac{1}{2|K|}\begin{bmatrix} a_2^3 - a_2^1 \\ a_1^1 - a_1^3 \end{bmatrix}, \quad \mathbf{grad}\,\lambda_3 = \frac{1}{2|K|}\begin{bmatrix} a_2^1 - a_2^2 \\ a_1^2 - a_1^1 \end{bmatrix}. \tag{3.6.119}$$

By (3.6.118), it remains to figure out the integral of products of powers of berycentric coordinate functions over a triangle.

Lemma 3.6.120. Integration of powers of barycentric coordinate functions

For any non-degenerate d-simplex K with barycentric coordinate functions $\lambda_1,\dots,\lambda_{d+1}$ and exponents $\alpha_j\in\mathbb{N}$, $j=1,\dots,d+1$,

$$\int_K \lambda_1^{\alpha_1}\cdot\dots\cdot\lambda_{d+1}^{\alpha_{d+1}}\,\mathrm{d}\boldsymbol{x} = d!|K|\,\frac{\alpha_1!\alpha_2!\cdot\dots\cdot\alpha_{d+1}!}{(\alpha_1+\alpha_2+\dots+\alpha_{d+1}+d)!} \quad \forall\boldsymbol{\alpha}\in\mathbb{N}_0^{d+1}\,. \tag{3.6.121}$$

Proof. (for $d = 2$) The idea is to transform K to the "unit triangle" $\widehat{K} := \operatorname{convex}\left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$:

$$\begin{aligned}
\Rightarrow \quad \int_K \lambda_1^{\beta_1} \lambda_2^{\beta_2} \lambda_3^{\beta_3} \,\mathrm{d}\boldsymbol{x} &= 2|K| \int_0^1 \int_0^{1-x_1} x_1^{\beta_1} x_2^{\beta_2} (1 - x_1 - x_2)^{\beta_3} \,\mathrm{d}x_2 \mathrm{d}x_1 \\
&\overset{(*)}{=} 2|K| \int_0^1 x_1^{\beta_1} \int_0^1 (1 - x_1)^{\beta_2+\beta_3+1} s^{\beta_2} (1 - s)^{\beta_3} \,\mathrm{d}s \,\mathrm{d}x_1 \\
&= 2|K| \int_0^1 x_1^{\beta_1} (1 - x_1)^{\beta_2+\beta_3+1} \,\mathrm{d}x_1 \cdot B(\beta_2 + 1, \beta_3 + 1) \\
&= 2|K| \; B(\beta_1 + 1, \beta_2 + \beta_3 + 2) \cdot B(\beta_2 + 1, \beta_3 + 1) \,,
\end{aligned}$$

At step $(*)$ we preformed the substitution $s(1 - x_1) = x_2$, $\quad B(\cdot,\cdot) \mathrel{\hat{=}}$ Euler's beta function, a well known special function defined as

$$B(\alpha, \beta) := \int_0^1 t^{\alpha-1} (1 - t)^{\beta-1} \,\mathrm{d}t \,, \quad 0 < \alpha, \beta < \infty \,.$$

It satisfies the important relation $\Gamma(\alpha + \beta)\, B(\alpha, \beta) = \Gamma(\alpha)\Gamma(\beta)$, where Γ denotes the Gamma function, which interpolates the factorials: $\Gamma(n) = (n - 1)!$,

$$\Rightarrow \quad \int_K \lambda_1^{\beta_1} \lambda_2^{\beta_2} \lambda_3^{\beta_3} \,\mathrm{d}\boldsymbol{x} = 2|K| \cdot \frac{\Gamma(\beta_1 + 1)\Gamma(\beta_2 + 1)\Gamma(\beta_3 + 1)}{\Gamma(\beta_1 + \beta_2 + \beta_3 + 3)} \,.$$

By the properties of the Gamma function, this amounts to the assertion of the lemma. □

Example 3.6.122 (Element matrix for quadratic Lagrangian finite elements)

In this example we, again, consider the local bilinear form related to $-\Delta$: $\mathsf{a}_K(u, v) = \int_K \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$. We state the element matrix for an arbitrary triangle K for the nodal local shape functions as given in (3.5.6):

$$\begin{aligned}
b_K^1 &= (2\lambda_1 - 1)\lambda_1 \,, & b_K^2 &= (2\lambda_2 - 1)\lambda_2 \,, & b_K^3 &= (2\lambda_3 - 1)\lambda_3 \,, \\
b_K^4 &= 4\lambda_1\lambda_2 \,, & b_K^5 &= 4\lambda_2\lambda_3 \,, & b_K^6 &= 4\lambda_1\lambda_3 \,,
\end{aligned}$$

where the λ_i are barycentric coordinate functions, see Section 3.3.5, Rem. 3.3.24.

We respect BETL's local numbering convention from Fig. 158. Then element matrix has the representation

$$\mathbf{A}_K = \frac{|K|}{3} \begin{bmatrix}
3\mathbf{g}_1 \cdot \mathbf{g}_1 & -\mathbf{g}_1 \cdot \mathbf{g}_2 & -\mathbf{g}_1 \cdot \mathbf{g}_3 & 4\mathbf{g}_1 \cdot \mathbf{g}_2 & 0 & 4\mathbf{g}_1 \cdot \mathbf{g}_3 \\
-\mathbf{g}_1 \cdot \mathbf{g}_2 & 3\mathbf{g}_2 \cdot \mathbf{g}_2 & -\mathbf{g}_2 \cdot \mathbf{g}_3 & 4\mathbf{g}_1 \cdot \mathbf{g}_2 & 4\mathbf{g}_2 \cdot \mathbf{g}_3 & 0 \\
-\mathbf{g}_1 \cdot \mathbf{g}_3 & -\mathbf{g}_2 \cdot \mathbf{g}_3 & 3\mathbf{g}_3 \cdot \mathbf{g}_3 & 0 & 4\mathbf{g}_3 \cdot \mathbf{g}_2 & 4\mathbf{g}_3 \cdot \mathbf{g}_1 \\
4\mathbf{g}_1 \cdot \mathbf{g}_2 & 4\mathbf{g}_1 \cdot \mathbf{g}_2 & 0 & d_4 & 8\mathbf{g}_1 \cdot \mathbf{g}_3 & 8\mathbf{g}_2 \cdot \mathbf{g}_3 \\
0 & 4\mathbf{g}_2 \cdot \mathbf{g}_3 & 4\mathbf{g}_3 \cdot \mathbf{g}_2 & 8\mathbf{g}_1 \cdot \mathbf{g}_3 & d_5 & 8\mathbf{g}_1 \cdot \mathbf{g}_2 \\
4\mathbf{g}_1 \cdot \mathbf{g}_3 & 0 & 4\mathbf{g}_3 \cdot \mathbf{g}_1 & 8\mathbf{g}_2 \cdot \mathbf{g}_3 & 8\mathbf{g}_1 \cdot \mathbf{g}_2 & d_6
\end{bmatrix}$$

with (constant!) vectors $\mathbf{g}_\ell := \mathbf{grad}\, \lambda_\ell$, $\ell = 1, 2, 3$, as given in (3.6.119), and

$$\begin{aligned}
d_4 &:= 8(\mathbf{g}_1 \cdot \mathbf{g}_1 + \mathbf{g}_1 \cdot \mathbf{g}_2 + \mathbf{g}_2 \cdot \mathbf{g}_2) \,, \\
d_5 &:= 8(\mathbf{g}_2 \cdot \mathbf{g}_2 + \mathbf{g}_2 \cdot \mathbf{g}_3 + \mathbf{g}_3 \cdot \mathbf{g}_3) \,, \\
d_6 &:= 8(\mathbf{g}_1 \cdot \mathbf{g}_1 + \mathbf{g}_1 \cdot \mathbf{g}_3 + \mathbf{g}_3 \cdot \mathbf{g}_3) \,.
\end{aligned}$$

Example 3.6.123 (Class providing analytically computed element matrix for $-\Delta$ and linear Lagrangian FE in BETL)

In this example we consider the very simple second-order linear variational problem

$$u \in H^1(\Omega): \quad \int_\Omega \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \tag{3.6.124}$$

on a polygonal domain $\Omega \subset \mathbb{R}^2$ for $f \in C^0(\overline{\Omega})$ given through a function handle. Note that (3.6.124) fails to have a unique solution and that existence of a solution hinges on a vanishing mean condition for f.

We perform Galerkin discretization of (3.6.124) by means of linear finite elements on a planar triangular mesh as introduced in Section 3.3.1. For the right hand side we use the 2D trapezoidal rule. This example together with Ex. 3.6.128 will guide you how to build the resulting linear system of equations using **NPDE::GalerkinMatrixAssembler** and **NPDE::LoadVectorAssembler**.

The following class can play the role of the `ELEM_MAT_BUILDER` template type argument for **NPDE::GalerkinMatrixAssembler** from Code 3.6.95 and BETL's built-in generic assembler **fem::BilinearForm**. The code uses the formulas already implemented in Code 3.3.27.

C++11 code 3.6.125: Class computing element matrix for $-\Delta$ analytically, compatible with GalerkinMatrixAssembler ➙ GITLAB

```
struct AnalyticStiffnessLocalAssembler{
private:
  static const int dim_ = 2;  // world dimension (2D)
public:
  typedef double numeric_t;
  typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, Eigen::Dynamic >
      result_t;
  // If FiniteElement class is being used, it needs to be initialized here
  static void initialize() {}

  template< typename MATERIAL, typename ELEMENT >
  static result_t eval(const MATERIAL& material, const ELEMENT& el);
};  //end class definition AnalyticStiffnessLocalAssembler
```

The actual computation of the element matrix is done in the `eval()` method. In this simple case no additional data but the positions of the vertices of the triangle are required.

C++11 code 3.6.126: Implementation of `eval()` for AnalyticStiffnessLocalAssembler ➙ GITLAB

```
template< typename BUILDER_DATA_T, typename ELEMENT >
 AnalyticStiffnessLocalAssembler::result_t
 AnalyticStiffnessLocalAssembler::eval( const BUILDER_DATA_T& data,
     const ELEMENT& el ) {
   result_t result;
```

```
  switch (el.refElType()) {
  case eth::base::RefElType::TRIA: {
    // Get element geometry and area
    const auto& geom = el.geometry();
    auto elem_area = geom.volume();
    // compute gradients
    Eigen::MatrixXd grads(2,3);
    grads << (geom.mapCorner(1)- geom.mapCorner(2)),
             (geom.mapCorner(2)- geom.mapCorner(0)),
             (geom.mapCorner(0)- geom.mapCorner(1));
    // compute local matrix
    result = grads.transpose()*grads/(4.*elem_area);
    break;
  }
  default: {
    ETH_ASSERT_MSG(false,"Implemented for TRIA 2D only!" );
    break;
  }}
  return(result);
}
```

Remark 3.6.127 (Symbolic computation)

Recommended for the direct computation of entries of element matrices for complicated finite element is the use of symbolic computing (MAPLE, Mathematica).

3.6.5.2 Local quadrature

At this point turn the pages back to (1.5.85) and remember the use of numerical quadrature for computing the Galerkin matrix for the linear finite element method in 1D. Also recall the rationale for using mesh based composite quadrature rules.

Also recall § 3.3.48, where a simple local quadrature rule was used for the computation of element vectors. Next we take a look at its implementation in BETL.

Example 3.6.128 (Class for the computation of element vectors for linear Lagrangian FE)

The following class can serve as type of `ELEM_VEC_BUILDER` for the assembler class **NPDE::LoadVectorAssembler** given as Code 3.6.101 and BETL's built-in generic right-hand-side vector assembler device **fem::LinearForm**. The local computations are based on the 2D trapezoidal rule and have already been explained in Section 3.3.6, see (3.3.50) and Code 3.3.51.

C++11 code 3.6.129: Class for computation of element (load) vector, compatible with LoadVectorAssembler ➙ GITLAB

```
struct LocalVectorAssembler{
```

```
private:
  static const int dim_ = 2;  // world dimension (2D)
public:
  typedef double numeric_t;
  typedef Eigen::Matrix< numeric_t, 3, 1 > result_t;
  // If FiniteElement class is being used, it need to be initialized here
  static void initialize() {}

  template< typename FUNCTION, typename ELEMENT >
  static result_t eval( const FUNCTION& f, const ELEMENT& el )
  {
    ETH_ASSERT_MSG( el.refElType() == eth::base::RefElType::TRIA ,
                "For this example, integration only works with 2D
                   triangles." );

    // get element geometry
    const auto& geom = el.geometry();
    auto elem_area = geom.volume();
    // 3-vector according to (3.3.50)
    result_t result; result.setZero();
    for (unsigned i=0;i<3;++i){
      result(i) = elem_area/3.0 * f(geom.mapCorner(i));
    }
    return(result);
  }
};  // end class LocalVectorAssembler
```

An important lesson can already be learned from this example:

> Since Lagrangian finite element functions are merely $\mathcal{M}$-piecewise smooth, numerical integration of expressions containing FE functions has to rely on composite quadrature rules on $\mathcal{M}$ ("cell based quadrature").

Reminder: numerical quadrature mandatory in the presence of coefficients/source terms in *procedural form* $\rightarrow$ Rem. 1.5.5.

(3.6.130) General local quadrature rules

A composite quadrature rule on a mesh $\mathcal{M}$ of a domain $\Omega \subset \mathbb{R}^d$ splits an integral over Ω into cell contributions and approximately evaluates those. This latter step is based on so-called local quadrature rules.

Definition 3.6.131. (Local) quadrature rule

A local quadrature rule on the element $K \in \mathcal{M}$ is an approximation

$$\int_K f(\mathbf{x})\,\mathrm{d}\boldsymbol{x} \approx \sum_{l=1}^{P_K} \omega_l^K f(\boldsymbol{\zeta}_l^K)\,, \quad \boldsymbol{\zeta}_l^K \in K\,, \omega_l^K \in \mathbb{R}\,, \quad P_K \in \mathbb{N}\,. \tag{3.6.132}$$

Terminology:

$\omega_l^K \to$ weights , $\boldsymbol{\zeta}_l^K \to$ quadrature nodes

(3.6.132) = P-point local quadrature rule

Def. 3.6.131 generalizes the quadrature rule (1.5.50) in 1D. The same terminology still applies.
An example for a local quadrature rule in 2D is the trapezoidal rule from (3.3.49).

Recall from § 1.5.79, § 1.5.84 that numerical quadrature is inevitable

- for computation of load vector, if f is complicated or only available in procedural form, Rem. 1.5.5,
- for computation of stiffness matrix, if the non-constant coefficient $\boldsymbol{\alpha} = \boldsymbol{\alpha}(\boldsymbol{x})$ in the bilinear form from (2.4.5), (2.9.16) does not permit analytic integration.

We recall a constraint on the weights of local quadrature rules:

Guideline [14, Section 5.2]: only quadrature rules with positive weights are numerically stable.

Once the local quadrature rules according to Def. 3.6.131, (3.6.132), are fixed, we formally use the approximation

$$\int_\Omega f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \sum_{K\in\mathcal{M}} \sum_{l=1}^{P_K} \omega_l^K f(\boldsymbol{\zeta}_l^K)\,. \tag{3.6.133}$$

For the variational problems (2.4.5) and (2.9.16) this means

$$\int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}))\cdot\mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \sum_{K\in\mathcal{M}} \sum_{l=1}^{P_K} \omega_l^K\,(\boldsymbol{\alpha}(\boldsymbol{\zeta}_l^K)(\mathbf{grad}\,u)(\boldsymbol{\zeta}_l^K))\cdot(\mathbf{grad}\,v)(\boldsymbol{\zeta}_l^K)\,,$$

$$\int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \sum_{K\in\mathcal{M}} \sum_{l=1}^{P_K} \omega_l^K f(\boldsymbol{\zeta}_l^K)v(\boldsymbol{\zeta}_l^K)\,.$$

Of course, in algorithms, in the spirit of local assembly as explained in Section 3.6.4, the focus is on local quadratures on the cells.

(3.6.134) Transformation of quadrature rules

Generically, the quadrature rule (3.6.132) is specific for the cell K. This begs the questions how local quadrature rules are handled on finite element meshes with millions of cells.

The policy is the same as in 1D in § 1.5.48: there the (local) quadrature rule was defined on a *reference interval*, e.g., $[-1,1]$ for Gaussian quadrature and mapped to a general interval by (affine) transformation, *cf.* [14, Rem. 5.1.4].

The local quadrature rules used in finite element methods are obtained by transformation from (a few) local quadrature rules defined on reference elements.

Reference elements and the associated transformations will be studied in the sequel with focus on the construction of local quadrature rules, and in a more general context in Section 3.7.

(3.6.135) Affine transformation of triangles

Now we examine the generalization of affine transformations from 1D to two dimensions:

Definition 3.6.136. Affine (linear) transformation

A mapping $\mathbf{\Phi} : \mathbb{R}^d \mapsto \mathbb{R}^d$ is affine (linear), if $\mathbf{\Phi}(\mathbf{x}) = \mathbf{F}\mathbf{x} + \boldsymbol{\tau}$ with some $\mathbf{F} \in \mathbb{R}^{d,d}$, $\boldsymbol{\tau} \in \mathbb{R}^d$.

✎ Reference triangle: 'unit triangle" $\widehat{K} := \operatorname{convex}\left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right\}$ (numbered vertices!)

Lemma 3.6.137. Affine transformation of triangles

For any non-degenerate triangle $K \subset \mathbb{R}^2$ ($|K| > 0$) with numbered vertices there is a unique affine transformation $\mathbf{\Phi}_K$, $\mathbf{\Phi}_K(\widehat{\boldsymbol{x}}) = \mathbf{F}_K\widehat{\boldsymbol{x}} + \boldsymbol{\tau}_K$ *($\rightarrow$ Def. 3.6.136), with $K = \mathbf{\Phi}_K(\widehat{K})$ and preserving the numbering of the vertices.*

Visualization of the affine mapping of the reference triangle onto K:

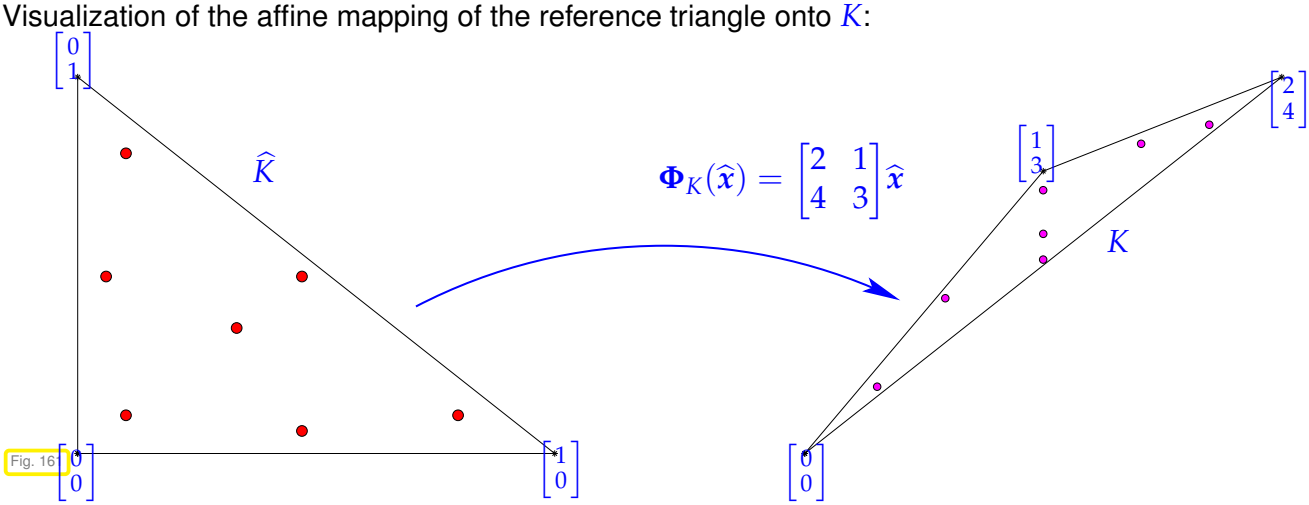


Fig. 16

The matrix $\mathbf{F}_K$ and translation vector $\boldsymbol{\tau}_K$ can be determined by solving a 6×6 linear system of equations, from which we obtain:

$$K = \operatorname{convex}\left\{ \begin{bmatrix} a_1^1 \\ a_2^1 \end{bmatrix}, \begin{bmatrix} a_1^2 \\ a_2^2 \end{bmatrix}, \begin{bmatrix} a_1^3 \\ a_2^3 \end{bmatrix} \right\} \quad \Rightarrow \quad \mathbf{\Phi}_K(\widehat{\boldsymbol{x}}) = \begin{bmatrix} a_1^2 - a_1^1 & a_1^3 - a_1^1 \\ a_2^2 - a_2^1 & a_2^3 - a_2^1 \end{bmatrix} \widehat{\boldsymbol{x}} + \begin{bmatrix} a_1^1 \\ a_2^1 \end{bmatrix} . \tag{3.6.138}$$

Note that $\boxed{|K| = |\widehat{K}|\,|\det \mathbf{F}_K|}$.

(3.6.139) Reference elements and transformations in BETL

Every entity of co-dimension 0 (= cell, element) of the mesh has a reference element associated with it, depending on the `RefElType` of the element, which is returned by the `refelType()` member function, see § 3.6.33.

$$\textbf{TRIA}: \quad \text{reference element} \quad \widehat{K} := \operatorname{convex}\left\{ \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix} \right\}, \tag{3.6.140}$$

$$\textbf{QUAD}: \quad \text{reference element} \quad \widehat{K} := \operatorname{convex}\left\{ \begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}1\\0\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix} \right\}. \tag{3.6.141}$$

For **TRIA** this convention is evident from Code 3.6.35. The reference triangle used in BETL is not the usual one!

Non-standard reference triangle in BETL

BETL uses the reference triangle $\widehat{K} := \left\{ \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix} \right\}$

The affine mapping from $\widehat{K}$ to the actual element K relies on the ordering of vertices from (3.6.140), (3.6.141) and the ordering implied by the `subEntity<dim>`-function, *cf.* Rem. 3.6.48.

By Lemma 3.6.137 there is a one-to-one correspondence between the geometric shape of a triangular element K and its associated afine mapping from/to the reference element $\widehat{K}$. ➢ Information about the mapping to/from the reference element is stored in the `Geometry`-object associated with every entity → § 3.6.49. The following data types and member functions of a `Geometry`-object are relevant:

- Type `localCoord_t` ≙ coordinate vectors for points in the reference element
- Type `globalCoord_t` ≙ coordinate vectors for points in the reference element
- `bool isAffine()` tells, whether the mapping from the reference element is affine.
- Method `global(const localCoord_t &)` returns the coordinates of the image of a point in the reference element under the mapping to the actual element.

The following code demonstrates the use of these facilities. It forgoes auto type detection to elucidate the relevant type, at the expense of significantly increased code length, of course.

C++11 code 3.6.143: Using coordinate transformation from reference element in BETL
➟ GITLAB

```
template<class ELEMENT>
void printTransformCoordinates(const ELEMENT &el) {
  using gridTraits_t = typename ELEMENT::gridTraits_t;
  using geometryElemImpl_t = typename
    gridTraits_t::template geometry_t<0>;
  using geometry_t =
    eth::grid::Geometry<gridTraits_t,geometryElemImpl_t>;
  using refElt_t = eth::base::RefElType;
  using globalCoord_t = typename geometry_t::globalCoord_t;
  using localCoord_t  = typename geometry_t::localCoord_t;

  using refEl_t = eth::base::RefElType;
  static const refEl_t triaType = refEl_t::TRIA;
```

```
  static const refEl_t quadType = refEl_t::QUAD;

  geometry_t geo = el.geometry();
  refElt_t refelType = geo.refElType();
  if(geo.isAffine())
    cout << "Affine element, type = " << refelType << " : ";

  // Determine location of barycenter in 2 different ways
  globalCoord_t center = geo.center();
  localCoord_t barycenter;
  // Barycenter of the reference triangle, BETL convention!
  if(refelType == triaType) barycenter << 2/3. , 1/3.;
  // Barycenter of the ref quad
  if(refelType == quadType) barycenter << 1/2. , 1/2.;

  // Print local and global coordinates of vertices.
  const size_t nCorners = geo.numCorners();
  cout << nCorners << " vertices at " << endl;
  for(int j=0; j < nCorners; j++) {
    globalCoord_t glbCoords = geo.mapCorner(j);
    localCoord_t locCoords =
        eth::base::ReferenceElements::getNodeCoord(refelType,j);
    globalCoord_t mappedlocCoords = geo.global(locCoords);
    cout << "global ( " << glbCoords.transpose() << " ) <-> "
         << "local ( " << locCoords.transpose() << " ) "
         << "mapped local ( " << mappedlocCoords.transpose() << " ) "
            << endl;
  }
  cout << "center = " << center.transpose() << " <-> "
       << geo.global(barycenter).transpose() << endl;
}
```

A partial listing of output when running the mainfile ➔ GITLAB for the hybrid mesh with five cells from Ex. 3.6.47. The listing shows the output of the function call `printMeshTransf(gridView)`, which internally calls `printTransformCoordinates` from Code 3.6.143 for all elements contained in the hybrid mesh with five cells from Ex. 3.6.47.

```
#Transformation of elements:
ENTITY TRIA(id = 0)
Affine element, type = TRIA : 3 vertices at
global ( 2 0 ) <-> local ( 0 0 )  mapped local ( 2 0 )
global ( 2 2 ) <-> local ( 1 0 )  mapped local ( 2 2 )
global ( 1 1 ) <-> local ( 1 1 )  mapped local ( 1 1 )
center = 1.66667        1 <-> 1.66667        1

ENTITY TRIA(id = 1)
Affine element, type = TRIA : 3 vertices at
global ( 2 0 ) <-> local ( 0 0 )  mapped local ( 2 0 )
global ( 1 1 ) <-> local ( 1 0 )  mapped local ( 1 1 )
global ( 1 0 ) <-> local ( 1 1 )  mapped local ( 1 0 )
```

```
center =  1.33333 0.333333 <-> 1.33333 0.333333

ENTITY TRIA(id = 2)
Affine element, type = TRIA : 3 vertices at
global ( 1 1 ) <-> local ( 0 0 )  mapped local ( 1 1 )
global ( 2 2 ) <-> local ( 1 0 )  mapped local ( 2 2 )
global ( 1 2 ) <-> local ( 1 1 )  mapped local ( 1 2 )
center = 1.33333 1.66667 <-> 1.33333 1.66667

ENTITY QUAD(id = 0)
4 vertices at
global ( 0 0 ) <-> local ( 0 0 )  mapped local ( 0 0 )
global ( 1 0 ) <-> local ( 1 0 )  mapped local ( 1 0 )
global ( 1 1 ) <-> local ( 1 1 )  mapped local ( 1 1 )
global ( 0 1 ) <-> local ( 0 1 )  mapped local ( 0 1 )
center = 0.5 0.5 <-> 0.5 0.5

ENTITY QUAD(id = 1)
4 vertices at
global ( 0 1 ) <-> local ( 0 0 )  mapped local ( 0 1 )
global ( 1 1 ) <-> local ( 1 0 )  mapped local ( 1 1 )
global ( 1 2 ) <-> local ( 1 1 )  mapped local ( 1 2 )
global ( 0 2 ) <-> local ( 0 1 )  mapped local ( 0 2 )
center = 0.5 1.5 <-> 0.5 1.5
```

Example 3.6.144 (Evaluation of local shape functions for triangular quadratic Lagrangian finite elements in BETL)

Recall from Ex. 3.5.3 the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ on triangle K in barycentric coordinate representation:

$$\begin{aligned} b_K^1 &= (2\lambda_1 - 1)\lambda_1 \,, \\ b_K^2 &= (2\lambda_2 - 1)\lambda_2 \,, \\ b_K^3 &= (2\lambda_3 - 1)\lambda_3 \,, \\ b_K^4 &= 4\lambda_1\lambda_2 \,, \\ b_K^5 &= 4\lambda_2\lambda_3 \,, \\ b_K^6 &= 4\lambda_1\lambda_3 \,, \end{aligned} \tag{3.5.6}$$

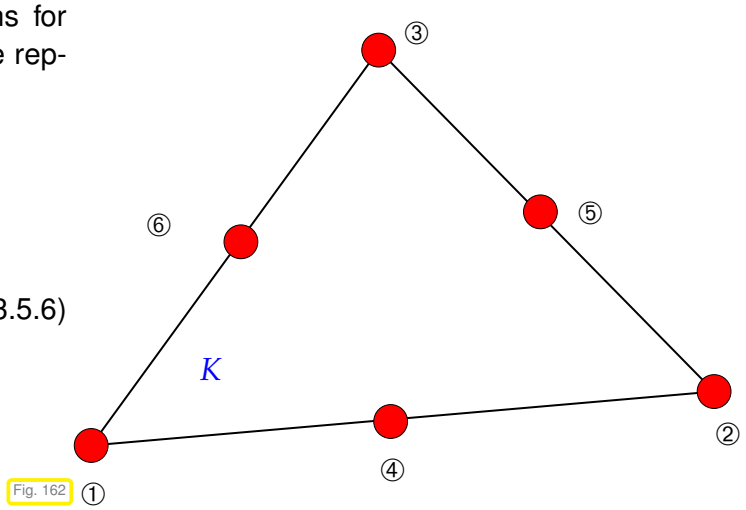


Fig. 162

Obviously, in order to compute $b_K^i(\boldsymbol{x})$, $\boldsymbol{x} \in K$, $i = 1,\dots,6$, all we need are the values $\lambda_\ell(\boldsymbol{x})$, $\ell = 1,2,3$, of the barycentric coordinate functions. As explained in § 3.3.10, since ($\boldsymbol{a}^k$, $k = 1,2,3$ are the position column vectors of the vertices of the triangle K)

$$\boldsymbol{x} = \lambda_1(\boldsymbol{x})\boldsymbol{a}^1 + \lambda_2(\boldsymbol{x})\boldsymbol{a}^2 + \lambda_3(\boldsymbol{x})\boldsymbol{a}^3 \quad , \quad \lambda_1(\boldsymbol{x}) + \lambda_2(\boldsymbol{x}) + \lambda_3(\boldsymbol{x}) = 1 \,, \tag{3.6.145}$$

these can be obtained from the linear system of equations

$$\begin{bmatrix} \boldsymbol{a}^1 & \boldsymbol{a}^2 & \boldsymbol{a}^3 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} \lambda_1(\boldsymbol{x}) \\ \lambda_2(\boldsymbol{x}) \\ \lambda_3(\boldsymbol{x}) \end{bmatrix} = \begin{bmatrix} \boldsymbol{x} \\ 1 \end{bmatrix}, \tag{3.6.146}$$

For the reference triangle on BETL with vertices $\boldsymbol{a}^1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, $\mathbf{a}^1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\mathbf{a}^3 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, we thus find

$$\widehat{\lambda}_1(\widehat{\boldsymbol{x}}) = 1 - \widehat{x}_1 \quad , \quad \widehat{\lambda}_2(\widehat{\boldsymbol{x}}) = \widehat{x}_1 - \widehat{x}_2 \quad , \quad \widehat{\lambda}_3(\widehat{\boldsymbol{x}}) = \widehat{x}_2 \,. \tag{3.6.147}$$

where we have used the hat tag for coordinates and barycentric coordinate functions on the BETL reference triangle $\widehat{K}$.

This formula and the numbering from (3.5.6) is used in the following code, which implements an `Eval()` function that returns the values of the local shape functions from (3.5.6) *on the* BETL *reference triangle*.

C++11 code 3.6.148: Computing values of local shape functions for quadratic Lagrangian FE in BETL ⇝ GITLAB

```
template< >
struct qfemLocalShape< eth::base::RefElType::TRIA >
{
  using vect_t = eth::base::fixedMatrix_t< 6,1 >;

  template< typename LOCAL_COORDS>
  static vect_t Eval(const LOCAL_COORDS& lclCoord) {
    const double lambda1 = 1 - lclCoord[0];
    const double lambda2 = lclCoord[0] - lclCoord[1];
    const double lambda3 = lclCoord[1];

    vect_t b(6);
    b << (2*lambda1-1)*lambda1,(2*lambda2-1)*lambda2,
         (2*lambda3-1)*lambda3,4 * lambda1 * lambda2,
         4 * lambda2 * lambda3,4 * lambda1 * lambda3;
    return(b);
  }
};
```

Note the use of template specialization in order to restrict the use to this class to cells with a **TRIA** reference element.

(3.6.149) Transformation of local quadrature rules on triangles

Now we resume the discussion started in § 3.6.134:

We write $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}) := \mathbf{F}_K\widehat{\boldsymbol{x}} + \boldsymbol{\tau}_K \mathrel{\widehat{=}}$ affine transformation ($\rightarrow$ Def. 3.6.136) mapping $\widehat{K}$ to triangle K, see Lemma 3.6.137.

By transformation formula for integrals [18, Satz 8.5.2]

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} f(\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}}))\,|\det \mathbf{F}_K|\,\mathrm{d}\widehat{\boldsymbol{x}}\,. \tag{3.6.150}$$

This enables the transition

$$P\text{-point quadrature formula on } \widehat{K} \quad \blacktriangleright \quad P\text{-point quadrature formula on } K,$$

and tells us how to adapt the quadrature weights ($|K| = \mathrm{area}(K)$):

$$\boxed{\int_{\widehat{K}} f(\widehat{\boldsymbol{x}})\,\mathrm{d}\widehat{\boldsymbol{x}} \approx \sum_{l=1}^{P} \widehat{\omega}_l f(\widehat{\boldsymbol{\zeta}}_l) \quad \blacktriangleright \quad \int_K f(\mathbf{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{|\widehat{K}|} \sum_{l=1}^{P} \omega_l^K f(\boldsymbol{\zeta}_l^K) \quad \text{with} \quad \omega_l^K = \widehat{\omega}_l\,, \quad \boldsymbol{\zeta}_l^K = \boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l)\,.} \tag{3.6.151}$$

➢ Only he quadrature formula (3.6.132) on the unit triangle $\widehat{K}$ needs to be specified!
(The same applies to tetrahedra, where affine mappings for $d = 3$ are used.)

(3.6.152) Order of local quadrature rule

How to gauge the quality of parametric local quadrature rules **?** We briefly review the discussion in [14, Section 5.4].

Gauging the quality of a quadrature formula

The quality of a parametric local quadrature rule on K is measured *maximal degree* of *polynomials* (multivariate $\to$ Def. 3.4.8, or tensor product $\to$ Def. 3.4.13) on K integrated exactly by the corresponding quadrature rule on K.

Definition 3.6.154. Order of a local quadrature rule

A local quadrature rule according to Def. 3.6.131 is said to be 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* $f \in \mathcal{Q}_{q-1}(\mathbb{R}^d)$.

Note: Quadrature rule exact for $\mathcal{P}_p(\mathbb{R}^d)$ $\Rightarrow$ quadrature rule of order $p+1$, degree of exactness p

How is the order of a local quadrature rule linked with the number of quadrature points?

Recall 1D: P-point Gaussian quadrature rule achieves maximal order $2P$, see [14, Section 5.3]

On triangles/tetrahedra there is no simple general formula has been found linking the order and the minimal number of quadrature nodes, but there is a simple overall relationship for "optimal" quadrature formulas:

The prize of higher order quadrature

For "optimal" local quadrature formulas:
the higher the order the more quadrature nodes are required.

(3.6.156) Preservation of order under affine mappings

An important observation is that the space $\mathcal{P}_p(\mathbb{R}^d)$ is *invariant* under *affine mappings*, that is

$$q \in \mathcal{P}_p(\mathbb{R}^d) \quad \Rightarrow \quad \widehat{\boldsymbol{x}} \mapsto q(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})) \in \mathcal{P}_p(\mathbb{R}^d) \quad \text{for any affine transformation } \boldsymbol{\Phi}\,. \tag{3.6.157}$$

This means, if a quadrature rule on the reference element integrates all polynomials up to degree p exactly, the same is achieved by the mapped quadrature rule on K, if the underlying mapping is affine.

- The orders of the quadrature rules on the left and right hand side of (3.6.151) agree!

- Its order is an intrinsic property of a quadrature rule on the reference triangle/tetrahedron $\widehat{K}$ and will be inherited by all derived quadrature rules on elements that are *affine* images of $\widehat{K}$.

Example 3.6.158 (Local quadrature rules on triangles)

By the transformation policy it is enough to specify the quadrature rule for the reference triangle ("unit triangle") $\widehat{K} := \operatorname{convex}\left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}\right\}$ (, which is different from BETL's $\widehat{K} := \operatorname{convex}\left\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\right\}$).

According to Def. 3.6.131 quadrature rules on $\widehat{K}$ can be described by pairs $(\widehat{\omega}_1, \widehat{\boldsymbol{\zeta}}_1), \dots, (\widehat{\omega}_P, \widehat{\boldsymbol{\zeta}}_P)$, $P \in \mathbb{N}$, of weights $\widehat{\omega}_P$ and nodes $\widehat{\boldsymbol{\zeta}}_P \in \widehat{K}$.

- P3O2: 3-point quadrature rule of order 2 (exact for $\mathcal{P}_1(\widehat{K})$)

$$\left\{\left(\frac{1}{3}, \begin{bmatrix}0\\0\end{bmatrix}\right), \left(\frac{1}{3}, \begin{bmatrix}0\\1\end{bmatrix}\right), \left(\frac{1}{3}, \begin{bmatrix}1\\0\end{bmatrix}\right)\right\}. \tag{3.6.159}$$

- P3O3: 3-point quadrature rule of order 3 (exact for $\mathcal{P}_2(\widehat{K})$)

$$\left\{\left(\frac{1}{3}, \begin{bmatrix}1/2\\0\end{bmatrix}\right), \left(\frac{1}{3}, \begin{bmatrix}0\\1/2\end{bmatrix}\right), \left(\frac{1}{3}, \begin{bmatrix}1/2\\1/2\end{bmatrix}\right)\right\}. \tag{3.6.160}$$

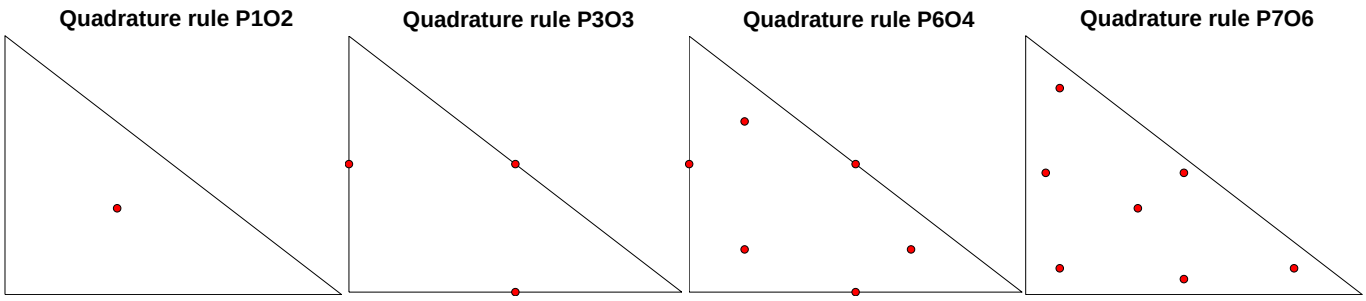
- P1O2: One-point quadrature rule of order 2 (exact for $\mathcal{P}_1(\widehat{K})$)

$$\left\{\left(1, \begin{bmatrix}1/3\\1/3\end{bmatrix}\right)\right\}. \tag{3.6.161}$$

- P7O6: 7-point quadrature rule of order 6 (exact for $\mathcal{P}_5(\widehat{K})$)

$$\begin{aligned}\Bigg\{&\left(\frac{9}{40}, \begin{bmatrix}1/3\\1/3\end{bmatrix}\right), \left(\frac{155+\sqrt{15}}{1200}, \begin{bmatrix}6+\sqrt{15}/21\\6+\sqrt{15}/21\end{bmatrix}\right), \left(\frac{155+\sqrt{15}}{1200}, \begin{bmatrix}9-2\sqrt{15}/21\\6+\sqrt{15}/21\end{bmatrix}\right),\\ &\left(\frac{155+\sqrt{15}}{1200}, \begin{bmatrix}6+\sqrt{15}/21\\9-2\sqrt{15}/21\end{bmatrix}\right), \left(\frac{155-\sqrt{15}}{1200}, \begin{bmatrix}6-\sqrt{15}/21\\9+2\sqrt{15}/21\end{bmatrix}\right),\\ &\left(\frac{155-\sqrt{15}}{1200}, \begin{bmatrix}9+2\sqrt{15}/21\\6-\sqrt{15}/21\end{bmatrix}\right), \left(\frac{155-\sqrt{15}}{1200}, \begin{bmatrix}6-\sqrt{15}/21\\6-\sqrt{15}/21\end{bmatrix}\right)\Bigg\}\end{aligned} \tag{3.6.162}$$

Location of quadrature nodes $\widehat{\boldsymbol{\zeta}}_l$ in the unit triangle $\widehat{K}$:



(3.6.161): $P = 1$, order 2 $P = 3$, order 3 $P = 6$, order 4 (3.6.162): $P = 7$, order 6

In the article [10] oen can find quadrature rules up to order $p = 21$ with $P \leq 1/6\,p(p+1) + 5$ points.

Example 3.6.163 (Local quadrature rules on quadrilaterals)

If K quadrilateral $\Rightarrow$ reference element $\widehat{K} := \operatorname{convex}\left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$ (unit square).

On $\widehat{K}$ use tensor product construction:

If $\{(\omega_1, \zeta_1), \dots, (\omega_P, \zeta_P)\}$, $P \in \mathbb{N}$, quadrature rule on the interval $]0,1[$, exact for $\mathcal{P}_p]0,1[$, then a quadrature rule on the unit square is given by the following sequence of P^2 weight–nodes pairs:

$$\left\{ \begin{array}{ccc} (\omega_1^2, \begin{bmatrix} \zeta_1 \\ \zeta_1 \end{bmatrix}) & \cdots & (\omega_1\omega_P, \begin{bmatrix} \zeta_1 \\ \zeta_P \end{bmatrix}) \\ \vdots & & \vdots \\ (\omega_1\omega_P, \begin{bmatrix} \zeta_P \\ \zeta_1 \end{bmatrix}) & \cdots & (\omega_P^2, \begin{bmatrix} \zeta_P \\ \zeta_P \end{bmatrix}) \end{array} \right\}$$

It provides a quadrature rule on the unit square $\widehat{K}$ that is exact for $\mathcal{Q}_p(\widehat{K})$. $\rightarrow$ order $p+1$!

Recall quadrature rules on $]0,1[$ ($\rightarrow$ [14, Chapter 5]):

- classical Newton-Cotes formulas (equidistant quadrature nodes).
- Gauss-Legendre quadrature rules, exact for $\mathcal{P}_{2P}(]0,1[)$ using only P nodes.
- Gauss-Lobatto quadrature rules: P nodes including $\{0,1\}$, exact for $\mathcal{P}_{2P-1}(]0,1[)$.

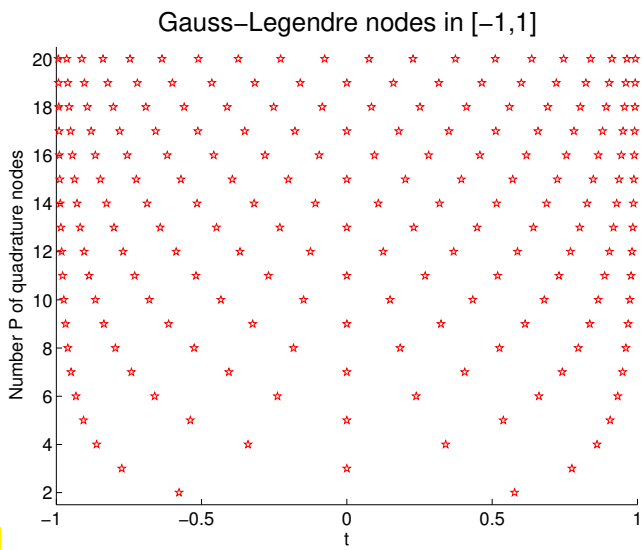


Fig. 163

(3.6.164) Numerical quadrature in BETL

In BETL quadrature rules on *reference elements* ($\rightarrow$ § 3.6.134) are provided by `Quadrature`-objects.

```
template <enum eth::base::RefElType RET,
          eth::base::signed_t NUM_POINTS> class Quadrature
```

`Quadrature` objects command the following member functions:

- `getNumPoints()` returns the number P of quadrature points (`NUM_POINTS`)
- `getRefEl()` returns the reference element type (`RET`), see § 3.6.33 for a list.
- `getPoints()` returns an object of type `Eigen::Matrix<refElDim,NUM_POINTS>` containing the *local* coordinates of the quadrature nodes as columns, that is, the coordinates of $\widehat{\zeta}_l$. `refElDim` is the dimension of the reference element.
- `getWeights()` returns an object of type `Eigen::Matrix< 1, NUM_POINTS >` containing the quadrature weights $\widehat{\omega}_l$.
- `getScale()` returns correction scaling factor σ, such that the sum of the quadrature weights multiplied with σ is equal to $|\widehat{K}|$, the area of the reference element. In other words, only after rescaling with σ we get a valid quadrature formula on $\widehat{K}$.
- `getRefDim()` returns the dimension of the reference element, for instance, 2 in the case of **TRIA** or **QUAD**.

One can verify, if a quadrature with `M` quadrature points is implemented for a given reference element type `RET` by using an object of type **betl2::quad::QuadratureTraits< RET >**, which provides the member function `isValid< M >()`. This method will return `true` if such quadrature is available.

In particular, we the following quadrature rules are available in BETL

RET	Implemented numbers of quadrature points
SEGMENT	$\{1,2,3,4,5\}$
TRIA	$\{1,3,6,7,12,16,19,25,33,37,42,61,73\}$
QUAD	$\{1,4,9,12,13,16,25,36,49,64\}$

As BETL allows hybrid meshes and each reference element type will require its own `Quadrature` object, the quadratures rules are handled via `QuadRuleList` objects. This object will store a list of `QuadRule` objects, that store a pair `{RET, NUM_POINTS}` at compile time.

An object of type `QuadRuleList` for an `Entity` object of co-dimension 0 in a 2D mesh can be initialized as follows (`NT` is the desired number of quadrature points for triangular elements, and `NQ` the desired number of quadrature points for quadrilateral elements):

```
// define quadrature rules for 2D-element types
typedef QuadRule< eth::base::RefElType::TRIA,NT > tria_t;
typedef QuadRule< eth::base::RefElType::QUAD,NQ > quad_t;
typedef QuadRuleList< tria_t, quad_t > quadrules_t;
```

The following function performs the integration of a function $f : \Omega \to \mathbb{R}^m$, $m \in \mathbb{N}$, over a triangulated domain Ω in BETL. The object of type **FUNCTION** must have a return type that supports elementary linear algebra operations and the method `setZero()`. This essential restricts the return type to EIGEN matrices.

C++11 code 3.6.165: Integration of a function on a 2D mesh using quadrature in BETL
➟ GITLAB

```
template<class QUAD_RULE_LIST,class VIEW_TRAITS,class FUNCTION>
auto integrateF(const eth::grid::GridView<VIEW_TRAITS> &gv,FUNCTION
    const &f)
  -> decltype(declval<FUNCTION>()(typename
      VIEW_TRAITS::gridTraits_t::template
```

```
      fixedSizeMatrix_t<VIEW_TRAITS::gridTraits_t::dimWorld,1>()))
{
  // Element types that can be handled by this function
  using refEl_t = eth::base::RefElType;
  static const refEl_t triaType = eth::base::RefElType::TRIA;
  static const refEl_t quadType = eth::base::RefElType::QUAD;

  // Type returned by the function
  using value_t = decltype(declval<FUNCTION>()(
    typename VIEW_TRAITS::gridTraits_t::template
      fixedSizeMatrix_t<VIEW_TRAITS::gridTraits_t::dimWorld,1>()));
  // The return type of the function must support the setZero() method
  // (in other words, it must be an Eigen::Matrix<>)
  value_t s; s.setZero();

  // Loop over cells of the mesh and apply quadrature rule on each
  for (auto& t : gv.template entities<0>()) {
    if (t.refElType() == triaType)      s +=
      loc_integrateF<QUAD_RULE_LIST,triaType>(t,f);
    else if (t.refElType() == quadType) s +=
      loc_integrateF<QUAD_RULE_LIST,quadType>(t,f);
    else ETH_ASSERT(false); // Not implemented!
  }
  return(s);
}
```

The actual implementation of the local quadrature comes next. It makes use of the method

```
template< int NUM_POINTS > matrix_t< 1, NUM_POINTS >
integrationElement( const matrix_t< dimFrom, NUM_POINTS >& local ) const;
```

of `GEOMETRY` objects, which were first introduced in Ex. 3.6.52. This method expects an argument of type **const_point_reference** supplied by a `QUADRATURE` objects. This argument is actually a list of point coordinates *in the reference element* $\widehat{K}$. What is returned is the value of $\det D\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})$ for each of the points $\widehat{\boldsymbol{x}}$. Of course, for an affine mapping $\boldsymbol{\Phi}_K$ ($\rightarrow$ Def. 3.6.136) these values are the same for every point, but in Section 3.7 we will learn about more general mappings with non-constant Jacobians.

C++11 code 3.6.166: Composite quadrature of a function on an element ➙ GITLAB

```
template<class QUAD_RULE_LIST, enum eth::base::RefElType RET,
         class GRID_TRAITS, class FUNCTION>
auto loc_integrateF(const eth::grid::Entity<GRID_TRAITS, 0> &e,
                    FUNCTION const &f)
  -> decltype(declval<FUNCTION>()(typename GRID_TRAITS::template
      fixedSizeMatrix_t<GRID_TRAITS::dimWorld,1>()))
{
  using value_t = decltype(declval<FUNCTION>()(
    typename GRID_TRAITS::template
      fixedSizeMatrix_t<GRID_TRAITS::dimWorld,1>()));
  static const int numQuadNodes = QUAD_RULE_LIST::template
```

```
      get<RET>();
10  using quadRule_t = betl2::quad::Quadrature<RET,numQuadNodes>;
11  using weights_t = typename quadRule_t::const_weight_reference;
12  using points_t = typename quadRule_t::const_point_reference;
13  weights_t weights = quadRule_t::getWeights();
14  points_t points = quadRule_t::getPoints();
15  const auto elemGeo = e.geometry();
16  // Transform all quadrature point simultaneously to save branching
      due to different element types.
17  const auto globPoints = elemGeo.global(points);
18  // Get local metric factors at quadrature points
19  const auto elemIE = elemGeo.integrationElement(points);
20
21  value_t ls; ls.setZero();
22  // Loop over quadrature nodes
23  // Here we use that globPoints contains an Eigen matrix
24  for(int j=0; j < points.cols(); j++) {
25    // Evaluate the function at the quadrature nodes
26    // Call to eval() enforces unraveling of expression templates
27    const value_t fVal = f(globPoints.col(j).eval());
28    ls += weights(j)*fVal*elemIE(j);
29  }
30  return (quadrule_t::getScale()*ls);
31 }
32
```

Line 10: The number of quadrature points for the `QuadratureRule` for the given reference element type `RET` is retrieved from the `QuadratureRuleList`.

Line 11: A `Quadrature` object for the reference element type `RET` of the current element `e` is created accordingly.

Line 18: Map quadrature points from reference element to current element `e`.

(3.6.167) Computation of element vector with local quadrature in BETL

In § 1.5.79 and § 1.5.84 we learned that local quadrature is the only option for evaluation the right hand side functional $v \mapsto \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$, if source function $f \in C^0(\overline{\Omega})$ is given in procedural form as a function that offers only point evaluation.

As pointed out in § 3.6.107, a `ELEM_VEC_BUILDER` object (for the right hand side vector) in BETL gets an argument of type `BUILDER_DATA_T`, which can be used to pass arbitrary information for local computations

As in Code 3.6.129 here we expect this type to provide a functor with an evaluation operator according to

```
inline result_t operator()(globalCoord_t x) const
```

with types as in Code 3.6.143).

The following code demonstrates the computation of the element load vector for the local right hand side linear form $\ell_K(v) = \int_K f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$ for quadratic Lagrangian finite elements using the local shape

functions as give in (3.5.6) (for triangles). The implementation relies on **qFEmLocalShape** for triangles given as Code 3.6.148.

C++11 code 3.6.168: Class performing local computation of element load vector ➔ GITLAB

```
template< typename QUADRULES >
struct MySimpleLocalVectorAssembler {
  typedef double numeric_t;
    typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, 1 > result_t;
  static void initialize() {}

  template< class FUNCTION, class ELEMENT>
  static result_t eval( const FUNCTION& F, const ELEMENT& el) {
    result_t result;
    switch( el.refElType() ){
    case eth::base::RefElType::TRIA: {
     eval_< eth::base::RefElType::TRIA > evaluator;
     result = evaluator.compute_( F, el ); break;  }
    case eth::base::RefElType::QUAD:  {
     eval_< eth::base::RefElType::QUAD > evaluator;
       result = evaluator.compute_( F, el ); break;  }
    default:
      ETH_ASSERT_MSG( false, "Implemented for TRIA only");
    }
    return(result);
  }
};  //end struct MySimpleLocalVectorAssembler
```

This class has a private member class whose `compute_` method carries out the actual quadrature:

C++11 code 3.6.169: Evaluator class for MySimpleLocalVectorAssembler ➔ GITLAB

```
// evaluation routine based on the reference element type as template parameter
template< eth::base::RefElType RET >
struct eval_ {
 typedef result_t returnType_t;

 template< class FUNCTION, class ELEMENT>
 static returnType_t compute_(const FUNCTION& F,
                              const ELEMENT& el){
   // fetch the quadrature rule that must be applied to this reference element type
   typedef betl2::quad::Quadrature< RET, QUADRULES::template get< RET >() > quadrule_t;
   // Utility function to find out number of dofs for Quadratic Lagrangian FE
   typedef betl2::fe::detail::FEBasisUtility <2,fe::FEBasisType::Lagrange > fe_utility_t;
   // number of test basis function with support on reference element type
   static const int numDofs_ = fe_utility_t::template
```

```
        Accumulate<RET>();
    // Initialize result vector to 0
    returnType_t result( numDofs_); result.setZero();
    // get element geometry
    const auto& geom = el.geometry();
    // get local integration points and weights
    const auto&  xi = quadrule_t::getPoints();
    const auto&  wi = quadrule_t::getWeights()*quadrule_t::getScale();
    // get metric factor (determinant of Jacobian of 'reference->actual'
        element transformation)
    const auto detJi = geom.template integrationElement<
        quadrule_t::getNumPoints()>(xi);
    // multiply everything together
    const auto coeff = detJi.cwiseProduct( wi );
    // for every integration point
    for( int i=0; i < xi.cols(); i++ ){
      // evaluate local shape functions, see Code 3.6.148
      const Eigen::Matrix< double, numDofs_ , 1  > phi_i
        = NPDE::qfemLocalShape<RET>::Eval(xi.col(i)) ;
      // evaluate integrand of linear form for each i test basis
          functions
      // multiply by integration weight and add contribution to final
          result
      result += coeff(i) * phi_i * F(geom.global(xi).col(i));
    }
    return result;
  }
};
```

Note the use of numerical quadrature based on `QuadRuleList` and `Quadrature` objects as explained in § 3.6.164.

3.6.6 Incorporation of Essential Boundary Conditions

According to the terminology introduced in Section 2.10, we call those boundary conditions essential that are imposed on the functions in the trial space of variational problems. For second order elliptic boundary value problems and the variational formulations discussed in Section 2.9, essential boundary conditions are synonymous to Dirichlet boundary conditions. Now we elaborate how to handle non-zero (non-homogeneous) Dirichlet boundary conditions within finite element Galerkin discretization.

Recall the variational formulation of a *non-homogeneous* Dirichlet boundary value problem from Ex. 2.9.2:

$$\begin{array}{c} u \in H^1(\Omega) \\ u = g \text{ on } \partial\Omega \end{array} : \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,. \tag{2.9.5}$$

$$\Downarrow$$

$$-\operatorname{div}(\kappa(\boldsymbol{x})\, \mathbf{grad}\, u) = f \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega \,,$$

with (admissible → § 2.10.6) Dirichlet data $g \in C^0(\partial\Omega)$.

Recall from Section 2.10: Dirichlet b.c. = essential boundary conditions (built into trial space)

Now we will learn, how discrete trial spaces and algorithms have to be modified in order to accommodate essential boundary conditions.

(3.6.170) Offset functions for Lagrangian finite element methods

Remember the offset function technique, see (1.3.30) and Section 2.2.3:

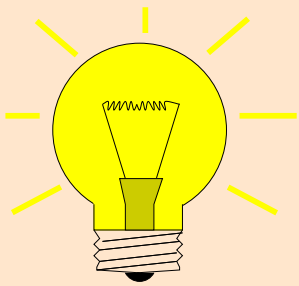
$$(2.9.5) \quad \Leftrightarrow \quad u = u_0 + w\,, \qquad \begin{aligned} & w \in H_0^1(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,w \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} \\ & = \int_\Omega -\kappa(\boldsymbol{x})\,\mathbf{grad}\,u_0 \cdot \mathbf{grad}\,v + fv\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,, \end{aligned} \tag{3.6.171}$$

with offset function $u_0 \in H^1(\Omega)$ satisfying $\boxed{u_0 = g \text{ on } \partial\Omega}$

We adapt the offset function policy to finite element Galerkin discretization by generalizing the 1D example from Rem. 1.5.89 to $d = 2, 3$:

Remember: we already know finite element subspaces $V_{0,N} := \mathcal{S}^0_{p,0}(\mathcal{M}) \subset H_0^1(\Omega)$, see § 3.5.14.

Finite element offset functions



Idea (from Rem. 1.5.89 in 1D):

use offset function $u_0 \in V_N := \mathcal{S}^0_p(\mathcal{M})$ *locally supported near the boundary*:

$\Updownarrow$

use offset function in the span of global basis functions associated with geometric entities on $\partial\Omega$

$$\text{supp}(u_0) \subset \bigcup\{K \in \mathcal{M}: \overline{K} \cap \partial\Omega \neq \emptyset\}\,. \tag{3.6.173}$$

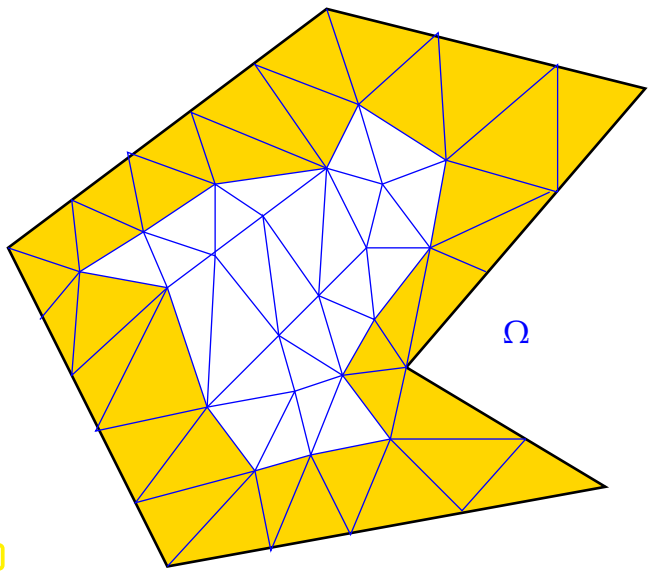


Fig. 164

(3.6.173) is a consequence of the local support property of finite element basis functions, see Ex. 3.4.16.

◁ Maximal support of u_0 on triangular mesh.

Example 3.6.174 (offset functions for linear Lagrangian FE)

Now we apply this idea to the special case of linear Lagrangian finite elements, which will yield a direct generalization of the choice of an offset function in 1D presented in Rem. 1.5.89.

For Dirichlet data $g \in C^0(\partial\Omega)$ use

$$u_0 = \sum_{\boldsymbol{x}\in\mathcal{V}(\mathcal{M})\cap\partial\Omega} g(\boldsymbol{x})\, b_N^{\boldsymbol{x}} \tag{3.6.175}$$

$b_N^{\boldsymbol{x}} \hat{=}$ tent function associated with node $\boldsymbol{x} \in \mathcal{V}(\mathcal{M})$, *cf.* Section 3.3.3. (3.6.175) generalizes (1.5.90) to 2D.

Note that this offset functions vanishes in all interior vertices: $u_0(\boldsymbol{x}) = 0$ for all $\boldsymbol{x} \in \mathcal{V}(\mathcal{M}) \cap \Omega$.

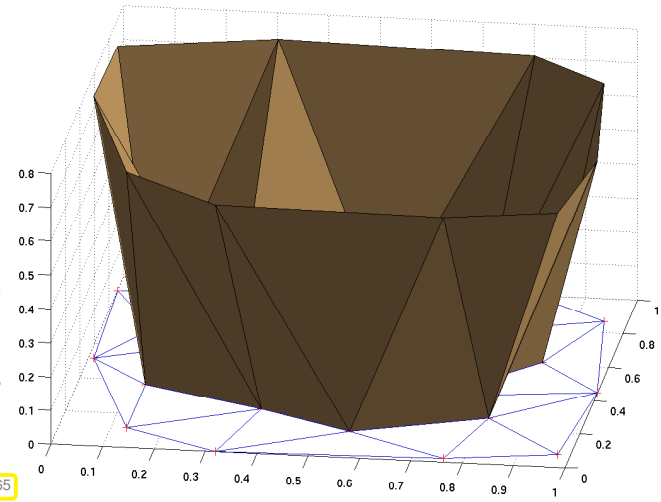


Fig. 165

Remark 3.6.176 (Approximate Dirichlet boundary conditions)

Be aware that the formula (3.6.175) actually violates the strict trace condition, because in general

$$u_0 \neq g \quad \text{on} \quad \partial\Omega\,.$$

Rather, u_0 is a *piecewise linear interpolant* of the Dirichlet data $g \in C^0(\partial\Omega)$. Therefore, another *approximation* comes into play when enforcing Dirichlet boundary conditions by means of piecewise polynomial offset functions.

(3.6.177) Implementation of non-homogeneous Dirichlet b.c. for linear FE: Elimination

Consider (2.9.5) and assume the following ordering of the nodal basis functions, see Fig. 93

$$\begin{aligned}
\mathfrak{B}_0 &:= \{b_N^1,\dots,b_N^N\} &&\hat{=}\ \text{nodal basis of } \mathcal{S}_{1,0}^0(\mathcal{M}),\\
&&&\quad\ \text{(tent functions associated with interior nodes)}\\
\mathfrak{B} &:= \mathfrak{B}_0 \cup \{b_N^{N+1},\dots,b_N^M\} &&\hat{=}\ \text{nodal basis of } \mathcal{S}_1^0(\mathcal{M})\\
&&&\quad\ \text{(extra basis functions associated with nodes} \in \partial\Omega).
\end{aligned}$$

Here: $M = \sharp\mathcal{V}(\mathcal{M}) = \dim \mathcal{S}_1^0(\mathcal{M})$, $N = \sharp\{\boldsymbol{x} \in \mathcal{V}(\mathcal{M}), \boldsymbol{x} \notin \partial\Omega\} = \dim \mathcal{S}_{1,0}^0(\mathcal{M})$ (no. of interior nodes)

$$\begin{aligned}
\mathbf{A}_0 \in \mathbb{R}^{N,N} &\ \hat{=}\ \text{Galerkin matrix for discrete trial/test space } \mathcal{S}_{1,0}^0(\mathcal{M}),\\
\mathbf{A} \in \mathbb{R}^{M,M} &\ \hat{=}\ \text{Galerkin matrix for discrete trial/test space } \mathcal{S}_1^0(\mathcal{M}).
\end{aligned}$$

This gives rise to a block-partitioning of the Galerkin matrix $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} \\ \mathbf{A}_{0\partial}^T & \mathbf{A}_{\partial\partial} \end{bmatrix}, \quad \begin{aligned} \mathbf{A}_{0\partial} &:= \left(\mathsf{a}(b_N^j, b_N^i)\right)_{\substack{i=1,\dots,N\\ j=N+1,\dots,M-N}} \in \mathbb{R}^{N,M-N}\,,\\ \mathbf{A}_{\partial\partial} &:= \left(\mathsf{a}(b_N^j, b_N^i)\right)_{\substack{i=N+1,\dots,M-N\\ j=N+1,\dots,M-N}} \in \mathbb{R}^{M-N,M-N}\,. \end{aligned} \tag{3.6.178}$$

If $u_0 \in \mathcal{S}_1^0(\mathcal{M})$ is chosen according to (3.6.175), then

$$u_0 \in \mathrm{Span}\{b_N^{N+1}, \dots, b_N^M\} \quad \Leftrightarrow \quad u_0 = \sum_{j=N+1}^{M} \gamma_{j-N} b_N^j \,,$$

with suitable coefficients γ_j, $j = 1, \dots, M-N$, defined, for instance, by (3.6.175). We can now plug this into the variational equation for the "correction" $w_N \in V_{0,N}$, which in abstract form reads

$$w_N \in V_{0,N}: \quad \mathsf{a}(w_N, v_N) = \ell(v_N) - \mathsf{a}(u_0, v_N) \quad \forall v_N \in V_{0,N} \,,$$

where we used the abbreviation a for the bilinear form in Eq. (2.9.5) and ℓ for the right hand side linear form. Thus, we get with $w_N = \sum_{j=1}^N \nu_j b_N^j$

$$\sum_{j=1}^{N} \nu_j \mathsf{a}(b_N^j, b_N^i) = \ell(b_N^i) - \sum_{k=N+1}^{M} \gamma_{k-N} \mathsf{a}(b_N^k, b_N^i) \,, \quad i = 1, \dots, N \,,$$

which means that the coefficient vector $\vec{\nu}$ of the finite element approximation $w_N \in \mathcal{S}_{1,0}^0(\mathcal{M})$ of $w \in H_0^1(\Omega)$ from (3.6.171) solves the linear system of equations

$$\boxed{\mathbf{A}_0 \vec{\nu} = \vec{\varphi} - \mathbf{A}_{0\partial} \vec{\gamma}} \,. \tag{3.6.179}$$

➢ Non-homogeneous Dirichlet boundary data are taken into account through a modified right hand side vector.

Alternative consideration leading to (3.6.179):

❶ First ignore essential boundary conditions and assemble the linear system of equations arising from the discretization of a on the (larger) FE space $\mathcal{S}_1^0(\mathcal{M})$:

$$\begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} \\ \mathbf{A}_{0\partial}^T & \mathbf{A}_{\partial\partial} \end{bmatrix} \begin{bmatrix} \vec{\mu}_0 \\ \vec{\mu}_\partial \end{bmatrix} = \begin{bmatrix} \vec{\varphi} \\ \vec{\varphi}_\partial \end{bmatrix} \,. \tag{3.6.180}$$

Here, $\vec{\mu}_0 \mathrel{\hat{=}}$ coefficients for *interior* basis functions $b_N^1, \dots, b_N^N$
$\vec{\mu}_\partial \mathrel{\hat{=}}$ coefficient for basis functions $b_N^{N+1}, \dots, b_N^M$ associated with nodes located on $\partial\Omega$.

❷ We realize that the coefficient vector of (3.6.180) is that of a FE approximation of u

► $\vec{\mu}_\partial$ known = values of g at boundary nodes: $\boxed{\vec{\mu}_\partial = \vec{\gamma}}$

❸ Moving known quantities in (3.6.180) to the right hand side yields (3.6.179).

Example 3.6.181 (Non-homogeneous Dirichlet boundary conditions in BETL)

BETL follows the approach from § 3.6.177. In order to build $\mathbf{A}_0$ and $\mathbf{A}_{0\partial}$, one first needs to partition the dofs (= finite element basis expansion coefficients) into *boundary dofs* and *interior dofs*. Based on this partition an *interior* finite element space and a *boundary* finite element space are created.

In BETL, the partitioning and the creation of the *interior* and *boundary* finite element space is handled by the class **fe::BoundaryDofMarker**. The *interior* and *boundary* finite element spaces are of type **fe::ConstrainedFESpace** and can be obtained by calling the following member functions of the instantiation `markerFull` of the class **fe::BoundaryDofMarker**:

```
// create the ConstrainedFESpaces via the BoundaryDofMarker
markerFull.mark( gridFactory );
// extract the ConstrainedFESpaces
const auto& constrained_FESpace_interior =
   markerFull.interiorFESpace();
const auto& constrained_FESpace_boundary =
   markerFull.boundaryFESpace();
```

See Code 3.6.183 for more details.

- The class **fe::BoundaryDofMarker** takes care of the *partitioning of the degrees of freedom* of an underlying **fe::FESpace**.
- The object that provides the actual *local→global index map* is an instantiation of the class **fe::ConstrainedFESpace**. It is a data member of **fe::BoundaryDofMarker**. Hence, the purpose of the **fe::BoundaryDofMarker** can be compared with that of the **fe::DofHandler** that is only a tool to create the **fe::FESpace**, see § 3.6.83.
- The class **fe::ConstrainedFESpace** has a member object of type **fe::FESpace** and hence has access to the dofs managed via the **fe::FESpace** object. **fe::ConstrainedFESpace** *makes it possible to work with a subset of the dofs* from the **fe::FESpace** (fulfilling a certain constraint). It provides almost all the member functions that we have already seen when discussing the class **fe::FESpace**, see Code 3.6.85.

The class **fe::ConstrainedFESpace** provides the following important member functions, similar to those of **fe::FESpace** presented in § 3.6.83.

- begin() and end() return the constant iterators to the beginning and end of the container of cells, i.e. entities of codimension zero. This enables `foreach` loops over **fe::ConstrainedFESpace**s.
- indices(e, intersectionIndex) takes `e`, a constant reference to an entity of codimension zero (cell) and an `intersectionIndex` of the cell `e` of type `int`, referring to one of the elements intersections (sides). It returns a standard vector containing the local→global index mappings (w.r.t. the intersection associated with the `intersectionIndex`) of all dofs that are associated with the intersection corresponding to the `intersectionIndex`.
- **indices(e)** takes `e`, a constant reference to an entity of codimension zero (cell), and provides a standard vector filled with its local→global index mappings.
- mapToGrid(data) takes a reference to the Eigen column vector `data`, representing a coefficient vector for the **fe::ConstrainedFESpace**. Hence, it needs to have a length that coincides with the global number of dofs of the **fe::ConstrainedFESpace**. It returns the Eigen column vector that represents that coefficient vector in the underlying **fe::FESpace** `fespace_`. It has length `fespace_.numDofs()`, which corresponds to the global number of dofs contained in `fespace_`.
- numDofs() returns the global number of dofs.
- numElements() returns the total number of elements.

First, we discuss the implementation of the **fe::ConstrainedFESpace** class and afterwards, we give an example how the **fe::BoundaryDofMarker** is used to obtain its member objects of class **fe::ConstrainedFESpace**. The template parameter `DOF_CONDITION` must provide a functor that takes a dof and returns a boolean value. It returns true if the inserted dof is part of the **fe::ConstrainedFESpace**.

C++11 code 3.6.182: Implementation of the class fe::ConstrainedFESpace in BETL (partial listing of `Library/fe/constrained_fespace.hpp`)

```
template< typename FESPACE_T, typename DOF_CONDITION>
class ConstrainedFESpace
{
public:
 // Types, essentially inherited from underlying FESpace
 using fespace_t            = FESPACE_T;
 using fe_basis_t           = typename fespace_t::fe_basis_t;
 using gridTraits_t         = typename fespace_t::gridTraits_t;
 using gridViewFactory_t    = typename fespace_t::gridViewFactory_t;
private:
 using index_t  = Dof::index_t;
 using vector_t = std::vector< index_t >;
 using size_type = typename gridTraits_t::size_type;
 // the element type
 using element_t = eth::grid::Entity< gridTraits_t , 0 >;
private:
 const fespace_t&     fespace_;
 // this is a vector that will be filled by the BoundaryDofMarker.
 // It will have length of the total number of dofs that are contained
 // in the underlying fespace_object.
 // It stores for each dof in the fespace_
 // the new index in the ConstrainedFESpace at position dof->index().
 vector_t             permutations_;
 // stores total number of dofs contained in the RestrictedFESpace
 size_type            size_;
public:
 /// constructor
 ConstrainedFESpace( const FESPACE_T& fespace ):
   fespace_(fespace),permutations_(),size_() {}
 /// return reference to the non-constrained fespace
 const FESPACE_T& feSpace( ) const { return fespace_; }
 /// return the number of dofs
 size_type numDofs( ) const { return size_; }
 /// return the permuatations
 const vector_t& permutations() const { return permutations_; }
 /// access the fespace's gridfactory
 decltype(fespace_.gridFactory()) gridFactory() const { return
    fespace_.gridFactory(); }
 /// return the continuity property
 constexpr static bool isContinuous( )
 { return FESPACE_T::isContinuous( ); }
 /// begin of entity collection of codimension zero (cells)
 inline decltype(fespace_.begin()) begin( ) const;
 /// end of entity collection of codimension zero (cells)
 inline decltype(fespace_.end()) end( ) const;
/// get the number of elements contained in the underlying grid
 inline decltype(fespace_.numElements()) numElements( ) const;
 /// get the local->global index map associated with the cell e
```

```
std::vector< IndexPair<size_type> > indices( const element_t& e ) const;
/// local→global index maps for the dofs associated with the intersection
/// of the cell e that is associated with the index IntersectionIndex.
std::vector< IndexPair<size_type> > indices( const element_t& e, const int intersectionIndex ) const;
/// provides the mapping of the coefficient vector data considered
/// in the ConstrainedFESpace of length size_ and maps it
/// to the corresponding coefficient vector in the underlying FESpace
/// of length fespace_.numDofs()
template< typename NUMERIC_T >
Eigen::Matrix<NUMERIC_T,Eigen::Dynamic,1> mapToGrid( const Eigen::Matrix<NUMERIC_T,Eigen::Dynamic,1>& data ) const;

/// member functions that are needed by the BoundaryDofMarker
/// to set up the ConstrainedFESpace
/// set the dimension of the constrained fespace
void setSize( size_type numDofs );
/// return the permutations (this vector has length is managing the indices of the dofs)
vector_t& permutations( ) { return permutations_; }
```

The following code shows how to set up a **fe::BoundaryDofMarker**, create the partitioning of the dofs of the underlying **fe::FESpace** into boundary dofs and interior dofs and access the **fe::ConstrainedFESpace**s containing the local→global index mappings of the boundary dofs and interior dofs, respectively.

C++11 code 3.6.183: partitioning of degrees of freedom ➙ GITLAB

```
// SETUP FE basis and dof handler
typedef fe::FEBasis< fe::Linear, fe::FEBasisType::Lagrange > febasis_t;
// define dofhandler type for the surface grid
typedef betl2::fe::DofHandler< febasis_t, fe::FESContinuity::Continuous, gridFactory_t > DH_t;
// instantiate dofhandler for grid, distribute the dofs
dh.distributeDofs ( gridFactory );
DH_t dh; dh.distributeDofs ( gridFactory );

fe::BoundaryDofMarker< DH_t::fespace_t > markerFull( dh.fespace() );
// this method call partitions the dofs into boundary dofs and interior
// dofs.  It also creates the respective ConstrainedFESpaces
markerFull.mark( gridFactory );
// extract the constrained space corresponding to the interior dofs
// of the underlying fespace
const auto& interiorSpaceFull = markerFull.interiorFESpace( );

// extract a finite element space containing the boundary dofs
// boundary of the underlying fespace
const auto& boundarySpaceFull = markerFull.boundaryFESpace( );
```

```
cout << "#(dofs on boundary): " << contrainedSpaceFull.numDofs() <<
    ", #(dofs interior): " << interiorSpaceFull.numDofs() <<endl;
```

The following code shows how to use the **fe::ConstrainedFESpaces** created previously in Code 3.6.183 to obtain the linear system as given in (3.6.179). A detailed documentation can be found right after the code listing.

C++11 code 3.6.184: Modification of Galerkin system according to § 3.6.177 ➙ GITLAB

```
1  typedef double numeric_t;
2  typedef Eigen::SparseMatrix< numeric_t > sparseMatrix_t;
3  // define element matrix assembler
4  typedef NPDE::AnalyticStiffnessLocalAssembler
       aStiffnessMatAssembler_t;
5  // define the associated bilinear form
6  typedef NPDE::GalerkinMatrixAssembler< aStiffnessMatAssembler_t >
7      GalerkinAssembler_t;
8
9  // compute bilinear form on interior space:
10 GalerkinAssembler_t A;
11 const auto Ah = A.assembleMatrix( interiorSpaceFull,
       interiorSpaceFull, 1.0 );
12 // Impose non-homogeneous Dirichlet boundary conditions:
13 // define functor providing Dirichlet data g
14 const auto dirFunctor = [](const coord_t& x) {
15   Eigen::Matrix<numeric_t, 1, 1> res;
16   res << x(0)*x(1); return res; };
17 // create AnalyticalGridFunction object
18 const auto dirFunc = fem::makeAnalyticalGridFunction( gridFactory,
       dirFunctor );
19 // interpolate the function into the boundary dofs
20 const auto uD = DofInterpolator()( dirFunc, boundarySpaceFull );
21 // compute the bilinear form for right-hand side; no contribution
22 // due to a source function f here!
23 GalerkinAssembler_t A_rhs;
24 const auto A_rhs_h = A_rhs.assembleMatrix( interiorSpaceFull,
       boundarySpaceFull, 1.0 );
25 // use it to get the right-hand side vector for the linear system
26 typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, 1 > vector_t;
27 const vector_t rhs = - A_rhs_h * uD;
```

Notice that the global assembly in BETL (as **NPDE::GalerkinMatrixAssembler** (→ Code 3.6.95) and **NPDE::LoadVectorAssembler** (→ Code 3.6.101)) construct the global matrix and vector, respectively, in the FE spaces received as arguments. In other words:

- Line 7: The global matrix assembly class **GalerkinMatrixAssembler** is set up for the bilinear form $(u,v) \mapsto \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}x$.
- Line 11: $\mathbf{A}_0$ is constructed using **interiorSpaceFull** as trial and test space.
- Line 24: $\mathbf{A}_{0\partial}$ is computed using **interiorSpaceFull** and **boundarySpaceFull** as test and trial

spaces, respectively.

- Line 27: The right hand side is modified as in (3.6.179).

Some additional comments on Code 3.6.184:

Line 18 constructs a `fem::AnalyticalGridFunction` given the Dirichlet data function `dirFunc`. An object of type **fem::AnalyticalGridFunction** provides the necessary interface that is needed from several BETL-routines working with so-called *grid functions*. They are named *grid functions*, since they provide *an evaluation on the grid*, i.e. an evaluation `f(q,e)`, where q represents a local quadrature point and `e` corresponds to a cell. (In addition to a `GRID_FACTORY` object, the class **fem::AnalyticalGridFunction** requires an object of type `FUNCTION_FUNCTOR` as argument. In this case, the method **fem::makeAnalyticalGridFunction** acts as a wrapper in order to receive a `lambda` function instead).

Line 20: Finally we get a vector with the values of the Dirichlet data function in the boundary dofs.

If the source function f is non-zero (for instance constant 1), the following code would have to be included before **??**, Code 3.6.184.

C++11 code 3.6.185: Right hand side assembly on interior nodes using BETL ➔ GITLAB

```
// Define a function on the domain
using coord_t = typename gridTraits_t::template fixedSizeMatrix_t<
    gridTraits_t::dimWorld,1 >;
const auto f = [](const coord_t& x){  double res = x(0) * x(1);
                                      return res; };

// define local element vector assembler
typedef NPDE::LocalVectorAssembler  trapLocFunAssembler_t;
// define the associated linear form
typedef NPDE::LoadVectorAssembler < trapLocFunAssembler_t >
    load_vector_assembler_t;
// define the associated linear form
 load_vector_assembler_t l;
// - compute the linear form with given function F
const auto f_vec = l.assembleRhs( interiorSpaceFull, f);
```

In addition Line 27 has to be replaced with

```
// -use it to get the problem right-hand side
 const vector_t rhs = f_vec - A_rhs_h * uD;
```

Example 3.6.186 (Non-homogeneous Dirichlet boundary conditions on parts of the boundary in BETL)

In this example we consider a simple planar triangular mesh and the *physical* tags for its boundary edges. The non-homogeneous Dirichlet b.c. are imposed on the right side of the square with tag 4.

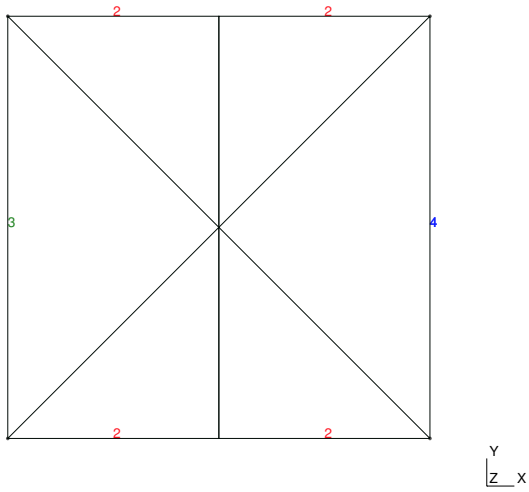


Fig. 166

Again, we follow the approach from § 3.6.177. But this time we consider *non-homogeneous Dirichlet boundary conditions only at a part of the boundary*, denoted by Γ_D. It is *characterized by a specific physical tag for edges*, i.e. for entities of codimension 1. For simplicity, we assume for this example that `tag = 4` (see Fig. 166). On all the other parts of the boundary $\partial\Omega \setminus \Gamma_D$, we impose *homogeneous Neumann boundary conditions*. Hence, the corresponding variational problem has the same form as the variational problem described in § 3.6.177. The only difference lies in the definition of the spaces. The *interior* finite element space is not only spanned by functions associated with interior nodes, it also contains all the nodal basis functions that are associated with nodes in $\partial\Omega \setminus \Gamma_D$. The *inactive* finite element space is spanned by the nodal basis functions associated with Γ_D.

In BETL, the implementation of the problem can done accomplished analogously to Ex. 3.6.181. We simply exchange the type of marker that we are using. We use the class **fe::IntersectionsDofMarker** instead of **fe::BoundaryDofMarker**. The marking and the extraction of the **fe::ConstrainedFESpace**s works in a similar way as in Ex. 3.6.181.

The *interior* and *inactive* finite element spaces are of type **fe::ConstrainedFESpace** (see Code 3.6.182) and can be obtained by calling the following member functions of the instantiation **intersection_marker** of the class **fe::IntersectionsDofMarker**:

```
// create the ConstrainedFESpaces via the IntersectionsDofMarker
intersection_marker.mark( gridFactory );
// extract the ConstrainedFESpaces
const auto& constrained_FESpace_interior =
   intersection_marker.interiorFESpace();
const auto& constrained_FESpace_inactive =
   intersection_marker.inactiveFESpace();
```

See Code 3.6.187 for more details.

C++11 code 3.6.187: partitioning of degrees of freedom ➙ GITLAB

```
// SETUP FE basis and dof handler
typedef fe::FEBasis<fe::Linear,fe::FEBasisType::Lagrange> febasis_t;
// define dofhandler type for the surface grid
typedef betl2::fe::DofHandler<febasis_t,
    fe::FESContinuity::Continuous,gridFactory_t> DH_t;
// instantiate and initialized dofhandler for the current mesh
```

```
DH_t dh; dh.distributeDofs (gridFactory);

// retrieve boundary entities carrying tag 4 → Code 3.6.20
const auto& taggedInt = gridMarker.template retrieveEntities<1>(4);
// instantiate the marker class
fe::IntersectionsDofMarker<DH_t::fespace_t> intersection_marker(
    dh.fespace() );
// partition the dofs of the underlying FESspace into
// dofs that are contained in taggedInt and all other dofs
intersection_marker.mark(taggedInt);
// extract the interior space,
// i.e.  all dofs except the ones that are associated with Γ_D
const auto& interiorSpace = intersection_marker.interiorFESpace();
// extract the ConstrainedFESpace that manages the dofs associated with
// Γ_D, i.e.  with the entities tagged with physical tag 4.
const auto& inactiveSpace = intersection_marker.inactiveFESpace();
```

The remainder of the code is the same as in Code 3.6.184, just replace **interiorSpaceFull** by **interiorSpace** and **boundarySpaceFull** by **inactiveSpace**.

(3.6.188) Implementation of non-homogeneous Dirichlet b.c. for linear FE: Augmentation

We use notations from § 3.6.177 and describe an alternative strategy for implementing (3.6.179).

Observe that the solution $\vec{\nu}$ of (3.6.179) can be obtained as one component of the solution of the block-partitioned linear system

$$\begin{bmatrix} \mathbf{A}_0 & \mathbf{A}_{0\partial} \\ \mathbf{0} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \vec{\nu} \\ \vec{\nu}_\partial \end{bmatrix} = \begin{bmatrix} \vec{\varphi} \\ \vec{\gamma} \end{bmatrix} . \tag{3.6.189}$$

The top block row of the coefficient matrix of (3.6.189) agrees with that of the Galerkin matrix $\mathbf{A}$ from (3.6.178) for the finite element space $\mathcal{S}_1^0(\mathcal{M})$ (without dropping basis functions on the boundary).

This leads to the following approach:

❶ Assemble the full Galerkin matrix $\mathbf{A} \in \mathbb{R}^{M,M}$ belonging to the (larger) FE space $\mathcal{S}_1^0(\mathcal{M})$.

❷ Set its left lower $(M-N) \times N$-block to zero: in MATLAB notation $\mathbf{A}(N+1:M, 1:N) = 0$.

❸ Replace its right lower $(M-N) \times (M-N)$-block on the diagonal with the identity matrix.

❹ Set the last $M-N$ components of the right hand side vector to the given values in the nodes on the boundary.

❺ Solve the resulting modified linear system: the solution provides the expansion coefficient vector $\vec{\mu}$ of the Galerkin finite element solution w.r.t. the nodal basis.

?! *Review question(s) 3.6.190.* (Finte element algorithms and implementation)

1. In BETL you have to deal with an exotic finite element scheme which assigns two local shape functions to each edge and two to each vertex. Which is the type of the geometric entity supporting the local shape function b_K^i, $i = 1, \dots, Q$, $Q \hat{=}$ the total number of local shape functions.
2. For the finite element scheme from Item 1, what is the dimension of the finite element space on a triangular space with $\sharp\mathcal{M}$ cells, $\sharp\mathcal{E}(\mathcal{M})$ edges, and $\sharp\mathcal{V}(\mathcal{M})$ vertices? Give a rule for telling the type of geometric entity associated with components of the vector of basis expansion coefficients.
3. Outline a way to create (in BETL) a vector of pairs of pointers to **POINT** objects with each pair corresponding to an edge of a 2D hybrid mesh.
4. Based on Def. 3.6.154 determine the minimal order of a quadrature rule on the *unit square* that is exact for all polynomials in $\mathcal{P}_p(\mathbb{R}^2)$.
5. Explain, why endowing edges of the mesh with an orientation, which means giving them a well-defined direction, is important for the implementation of cubic Lagrangian finite elements.
6. Outline the implementation of a function in BETL that takes a vector $\vec{\mu}$ of expansion coefficients of a finite element function $u_N \in \mathcal{S}_2^0(\mathcal{M})$ ($\mathcal{M}$ a *triangular* mesh), a suitable **FESpace** argument, and a coordinate vector $\boldsymbol{p} \in \mathbb{R}^2$ and returns the value $u_N(\boldsymbol{p})$.

3.7 Parametric Finite Elements

Already in Section 3.6.5 we exploited (affine) transformation ($\rightarrow$ Def. 3.6.136) to a reference cell in order to obtain numerical quadrature formulas (3.6.132) for all cells of a mesh in one fell swoop. In this section we will witness the full power of this idea of using transformations to reference cells. It will enable us to extend the range of Lagrangian finite element spaces significantly, and will also be a key element in algorithm design (The entire BETL finite element library relies on the construction of finite elements by transformation).

We need to enhance the flexibility of finite element spaces. For instance, the construction of Lagrangian finite element spaces done in Section 3.5 cannot cope with the following situation:

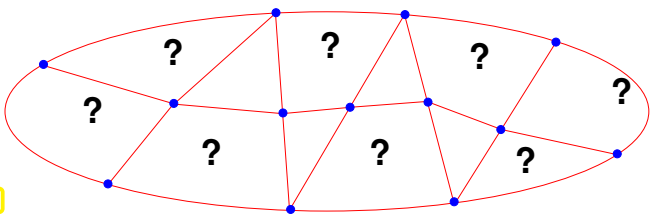


Fig. 167

◁ 2D hybrid mesh $\mathcal{M}$ with curvilinear triangles and general quadrilaterals

How to build $\mathcal{S}_1^0(\mathcal{M})$?

3.7.1 Affine equivalence

Recall Lemma 3.6.137: affine transformation of triangles (3.6.138)

▶ All cells of a triangular mesh are affine images of "unit triangle" $\widehat{K}$

"Unit triangle": $\widehat{K} = \left\langle \begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix} \right\rangle$

For $K = \operatorname{convex}\{\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3\}$:

$$\mathbf{F}_K = \begin{bmatrix} a_1^2 - a_1^1 & a_1^3 - a_1^1 \\ a_2^2 - a_2^1 & a_2^3 - a_2^1 \end{bmatrix}, \quad \boldsymbol{\tau}_K = \boldsymbol{a}^1 .$$

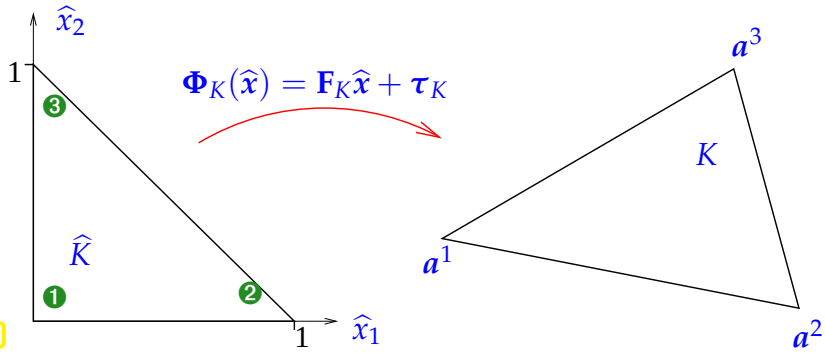


Fig. 168

(3.7.1) Pullback of functions

In a natural way, a transformation of domains induces a transformation of the functions defined on them:

Definition 3.7.2. Pullback

Given domains $\Omega, \widehat{\Omega} \subset \mathbb{R}^d$ and a bijective mapping $\boldsymbol{\Phi} : \widehat{\Omega} \mapsto \Omega$, the pullback $\boldsymbol{\Phi}^* u : \widehat{\Omega} \mapsto \mathbb{R}$ of a function $u : \Omega \mapsto \mathbb{R}$ is a function on $\widehat{\Omega}$ defined by

$$(\boldsymbol{\Phi}^* u)(\widehat{\boldsymbol{x}}) := u(\boldsymbol{\Phi}(\widehat{\boldsymbol{x}})) , \quad \widehat{\boldsymbol{x}} \in \widehat{\Omega} .$$

- Implicitly, we used the pullback of integrands when defining quadrature rules through transformation, see (3.6.150).
- Obviously, the pullback $\boldsymbol{\Phi}^*$ induces a *linear mapping* between spaces of functions on Ω and $\widehat{\Omega}$, respectively.

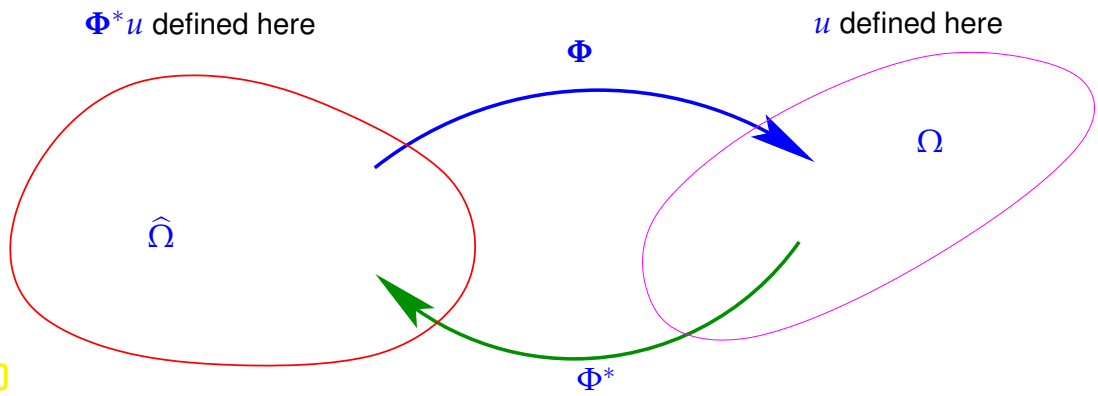


Fig. 169

In the context of numerical quadrature, when wondering whether transformation preserved the order of a quadrature rule, we made the following observation, *cf.* (3.6.157):

Lemma 3.7.3. Preservation of polynomials under affine pullback

If $\boldsymbol{\Phi} : \mathbb{R}^d \mapsto \mathbb{R}^d$ is an affine (linear) transformation ($\rightarrow$ Def. 3.6.136), then

$$\boldsymbol{\Phi}^*(\mathcal{P}_p(\mathbb{R}^d)) = \mathcal{P}_p(\mathbb{R}^d) \quad \text{and} \quad \boldsymbol{\Phi}^*(\mathcal{Q}_p(\mathbb{R}^d)) = \mathcal{Q}_p(\mathbb{R}^d) .$$

In fact, Lemma 3.6.137 reveals another reason for the preference for polynomials in building discrete Galerkin spaces.

Proof. (of Lemma 3.6.137) Since the pullback is linear, we only need to study its action on the (monomial) basis $\boldsymbol{x} \mapsto \boldsymbol{x}^{\boldsymbol{\alpha}}$, $\boldsymbol{\alpha} \in \mathbb{N}_0^d$ of $\mathcal{P}_p(\mathbb{R}^d)$, see Def. 3.4.8 and the explanations on multi-index notation (3.4.9).

Then resort to induction w.r.t. degree p.

$$\boldsymbol{\Phi}_K^*(\boldsymbol{x}^{\boldsymbol{\alpha}}) = \boldsymbol{\Phi}_K^*(x_1) \cdot \boldsymbol{\Phi}_K^*(\underbrace{\boldsymbol{x}^{\boldsymbol{\alpha}'}}_{\in \mathcal{P}_{p-1}(\mathbb{R}^d)}) = \underbrace{(\sum_{l=1}^{d} (\mathbf{F})_{1l} \widehat{x}_l + \tau_1)}_{\in \mathcal{P}_1(\mathbb{R}^d)} \cdot \underbrace{\boldsymbol{\Phi}_K^*(\boldsymbol{x}^{\boldsymbol{\alpha}'})}_{\in \mathcal{P}_{p-1}(\mathbb{R}^d)} \in \mathcal{P}_p(\mathbb{R}^d) \,,$$

with $\boldsymbol{\alpha}' := (\alpha_1 - 1, \alpha_2, \dots, \alpha_d)$, where we assumed $\alpha_1 > 0$. Here, we have used the induction hypothesis to conclude $\boldsymbol{\Phi}_K^*(\boldsymbol{x}^{\boldsymbol{\alpha}'}) \in \mathcal{P}_{p-1}(\mathbb{R}^d)$.

□

(3.7.4) Pullback of local shape functions for Lagrangian finite elements

A simple observation:

Consider $\mathcal{S}_1^0(\mathcal{M})$, triangle $K \in \mathcal{M}$, unit triangle $\widehat{K}$, affine mapping $\boldsymbol{\Phi}_K : \widehat{K} \mapsto K$

- b_K^1, b_K^2, b_K^3 (standard) local shape functions on K,
- $\widehat{b}^1, \widehat{b}^2, \widehat{b}^3$ (standard) local shape functions on $\widehat{K}$,

$\rightarrow$ Ex. 3.4.20

$$\boxed{\widehat{b}^i = \boldsymbol{\Phi}_K^* b_K^i \quad \Leftrightarrow \quad \widehat{b}^i(\widehat{\boldsymbol{x}}) = b_K^i(\boldsymbol{x}) \,, \quad \boldsymbol{x} = \boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})} \tag{3.7.5}$$

Of course, we assume that $\boldsymbol{\Phi}_K$ respects the local numbering of the vertices of $\widehat{K}$ and K: $\boldsymbol{\Phi}_K(\widehat{\boldsymbol{a}}^i) = \boldsymbol{a}^i$, $i = 1, 2, 3$.

The proof of (3.7.5) is straightforward: both $\boldsymbol{\Phi}_K^* b_K^i$ (by Lemma 3.7.3) and $\widehat{b}^i$ are (affine) linear functions that attain the same values at the vertices of $\widehat{K}$. Hence, they have to agree.

Note: (3.7.5) holds true for *all* simplicial Lagrangian finite element spaces

Proof. (of (3.7.5)*)* First, recall the definition of global shape functions and also local shape functions for $\mathcal{S}_p^0(\mathcal{M})$, $p \in \mathbb{N}$, by means of the conditions (3.5.4) at interpolation nodes, see Ex. 3.5.3 for $p = 2$.

Note: we already used the definition of basis functions through basis functions on the "reference cell" $[0, 1]$ and affine pullback in 1D, see § 1.5.45.

Now write $\boldsymbol{p}_K^i \mathrel{\hat{=}}$ (local) interpolation nodes on triangle K,
$\widehat{\boldsymbol{p}}^i \mathrel{\hat{=}}$ (local) interpolation nodes on unit triangle $\widehat{K}$.

Observe: Assuming a matching numbering $\boldsymbol{p}_K^i = \boldsymbol{\Phi}_K(\widehat{\boldsymbol{p}}^i)$. where $\boldsymbol{\Phi}_K : \widehat{K} \mapsto K$ is the unique affine transformation mapping $\widehat{K}$ onto K, see (3.6.138).

This is clear for $p = 2$, because affine transformations take midpoints of edges to midpoints of edges. The same applies to the interpolation nodes for higher degree Lagrangian finite elements defined in Ex. 3.5.7.

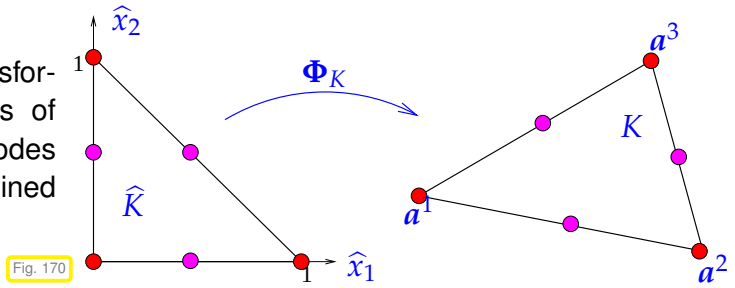


Fig. 170

For Lagrangian finite element spaces the local shape functions $b_K^i \in \mathcal{P}_p(\mathbb{R}^d)$, $\widehat{b}^i \in \mathcal{P}_p(\mathbb{R}^d)$, $i = 1, \dots, Q$, on K and $\widehat{K}$, respectively, are *uniquely defined* by the interpolation conditions

$$b_K^i(\boldsymbol{p}_K^j) = \delta_{ij} \quad , \quad \widehat{b}^i(\widehat{\boldsymbol{p}}^j) = \delta_{ij} \; . \tag{3.7.6}$$

Together with $\boldsymbol{p}_K^i = \boldsymbol{\Phi}_K(\widehat{\boldsymbol{p}}^i)$ this shows that $\boldsymbol{\Phi}_K^* b_K^i$ satisfies the interpolation conditions (3.7.6) on $\widehat{K}$ and, thus, has to agree with $\widehat{b}^i$.

□

The property (3.7.5) paves the way for profound algorithmic simplifications in finite element codes. Thus it is very desirable that global basis functions of finite element spaces comply with (3.7.5).

Terminology: Finite element spaces satisfying (3.7.5) with a affine mapping (→ Def. 3.6.136) $\boldsymbol{\Phi}_K : \widehat{K} \to K$ for every $K \in \mathcal{M}$ are called affine equivalent.

Remark 3.7.7 (Evaluation of local shape functions at quadrature points)

Affine equivalence can be exploited to achieve substantial reduction in computational effort for local computations: We consider Lagrangian finite element spaces on a simplicial mesh $\mathcal{M}$.

Recall from Section 3.6.5 the definition (3.6.151) of local quadrature formulas via transformation from a "unit simplex" (reference cell/element).

In particular the quadrature nodes on K are given by $\boldsymbol{\zeta}_l^K = \boldsymbol{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l)$. Hence, the values of local shape functions at quadrature points can be obtained by evaluating the local shape functions on $\widehat{K}$ in the quadrature points on $\widehat{K}$:

$$b_K^i(\boldsymbol{\zeta}_l^K) \overset{\text{Def. 3.7.2}}{=} \boldsymbol{\Phi}_K^*(b_K^i)(\widehat{\boldsymbol{\zeta}}^l) \overset{(3.7.5)}{=} \widehat{b}^i(\widehat{\boldsymbol{\zeta}}^l) \quad \text{independent of } K \; ! \; . \tag{3.7.8}$$

This can be exploited for the fast numerical quadrature of expressions depending on local shape functions only:

$$\int_K F(b_K^i(\boldsymbol{x}), b_K^j(\boldsymbol{x}))\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{|\widehat{K}|} \sum_{l=1}^{P} \widehat{\omega}_l F(\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_l), \widehat{b}^j(\widehat{\boldsymbol{\zeta}}_l)) \; , \tag{3.7.9}$$

for any integrable function $F : \mathbb{R}^2 \mapsto \mathbb{R}$.

➢ Precompute $\widehat{b}^i(\boldsymbol{\zeta}_l)$, $i = 1, \dots, Q$, $l = 1, \dots, P$, and store the values in a table!

Recall: (3.7.9) was applied in Code 3.6.169, with evaluation of local shape functions on $\widehat{K}$ farmed out to the function `qfemLocShape::Eval` listed as Code 3.6.148.

Remark 3.7.10 (Barycentric representation of local shape functions)

We consider Lagrangian finite element spaces on a simplicial mesh $\mathcal{M}$ in 2D, standard reference triangle used.

In (3.5.6) the formulas for local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ ($d=2$) were given in terms of barycentric coordinate fur λ_i, $i=1,2,3$. Is this coincidence? **NO**! Does

$$b_K^i = \sum_{\boldsymbol{\alpha}\in\mathbb{N}_0^3,|\boldsymbol{\alpha}|\leq p} \kappa_{\boldsymbol{\alpha}}\, \lambda_1^{\alpha_1}\lambda_2^{\alpha_2}\lambda_3^{\alpha_3}\,, \quad \kappa_{\boldsymbol{\alpha}}\in\mathbb{R}\,, \tag{3.6.116}$$

hold for any (simplicial) Lagrangian finite element space?

YES, because

$$\begin{aligned} b_K^i(\boldsymbol{x}) &\overset{(3.7.5)}{=} (\boldsymbol{\Phi}_K^{-1})^*\left(\widehat{\boldsymbol{x}}\mapsto \widehat{b}^i(\widehat{x}_1,\widehat{x}_2)\right) \\ &= \widehat{b}^i((\boldsymbol{\Phi}_K^{-1})^*(\widehat{\lambda}_2)(\boldsymbol{x}),(\boldsymbol{\Phi}_K^{-1})^*(\widehat{\lambda}_3)(\boldsymbol{x})) = \widehat{b}^i(\lambda_2(\boldsymbol{x}),\lambda_3(\boldsymbol{x}))\,, \end{aligned}$$

where $\lambda_2(\widehat{\boldsymbol{x}})=\widehat{x}_1$, $\lambda_3(\widehat{\boldsymbol{x}})=\widehat{x}_2$, $\lambda_1(\widehat{\boldsymbol{x}})=1-\widehat{x}_1-\widehat{x}_2$ $\hat{=}$ barycentric coordinate functions on $\widehat{K}$, see Ex. 3.4.20,

λ_i $\hat{=}$ barycentric coordinate functions on triangle K, see Fig. 99,

$\boldsymbol{\Phi}_K$ $\hat{=}$ affine transformation ($\rightarrow$ Def. 3.6.136), $\boldsymbol{\Phi}_K(\widehat{K})=K$, see (3.6.138).

The above formula is a consequence of the trivial fact that for an affine transformation $\boldsymbol{\Phi}_K:\widehat{K}\rightarrow K$ between simplices (triangles or tetrahedra) the corresponding pullback ($\rightarrow$ Def. 3.7.2) maps barycentric coordinate functions onto each other:

$$\boldsymbol{\Phi}_K^*(\lambda_k)=\widehat{\lambda}_k\,, \quad k=1,\dots,d+1\,. \tag{3.7.11}$$

➢ By the chain rule:

$$\begin{aligned} \mathbf{grad}\, b_K^i(\boldsymbol{x}) &= \frac{\partial\widehat{b}^i}{\partial\widehat{x}_1}(\widehat{\boldsymbol{x}})\,\mathbf{grad}\,\lambda_2 + \frac{\partial\widehat{b}^i}{\partial\widehat{x}_2}(\widehat{\boldsymbol{x}})\,\mathbf{grad}\,\lambda_3 \\ &= \begin{pmatrix}\mathbf{grad}\,\lambda_2 & \mathbf{grad}\,\lambda_3\end{pmatrix}\mathbf{grad}_{\widehat{\boldsymbol{x}}}\,\widehat{b}^i(\widehat{\boldsymbol{x}})\,, \quad \boldsymbol{x}=\boldsymbol{\Phi}_K(\widehat{\boldsymbol{x}})\,. \end{aligned} \tag{3.7.12}$$

This formula is convenient, because $\mathbf{grad}\,\lambda_i\equiv\mathrm{const}$, see (3.6.119).

This facilitates the computation of element (stiffness) matrices for 2nd-order elliptic problems in variational form with scalar valued coefficient $\alpha=\alpha(\boldsymbol{x})$: when using a quadrature formula according to (3.6.151)

$$\begin{aligned} &\int_K (\alpha(\boldsymbol{x})\,\mathbf{grad}\,b_K^i)\cdot\mathbf{grad}\,b_K^j\,\mathrm{d}\boldsymbol{x} \\ &\approx \frac{|K|}{|\widehat{K}|}\sum_{l=1}^{P_K}\widehat{\omega}_l\alpha(\boldsymbol{\zeta}_l)\left(\begin{pmatrix}\frac{\partial\widehat{b}^i}{\partial\widehat{x}_1}(\widehat{\boldsymbol{\zeta}}_l)\\ \frac{\partial\widehat{b}^i}{\partial\widehat{x}_2}(\widehat{\boldsymbol{\zeta}}_l)\end{pmatrix}^\top \begin{pmatrix}\mathbf{grad}\,\lambda_2\cdot\mathbf{grad}\,\lambda_2 & \mathbf{grad}\,\lambda_2\cdot\mathbf{grad}\,\lambda_3\\ \mathbf{grad}\,\lambda_2\cdot\mathbf{grad}\,\lambda_3 & \mathbf{grad}\,\lambda_3\cdot\mathbf{grad}\,\lambda_3\end{pmatrix}\begin{pmatrix}\frac{\partial\widehat{b}^j}{\partial\widehat{x}_1}(\widehat{\boldsymbol{\zeta}}_l)\\ \frac{\partial\widehat{b}^j}{\partial\widehat{x}_2}(\widehat{\boldsymbol{\zeta}}_l)\end{pmatrix}\right) \end{aligned}$$

This is attractive from an implementation point of view, because

- the values $\frac{\partial\widehat{b}^i}{\partial\widehat{x}_1}(\widehat{\boldsymbol{\zeta}}_l)$ can be *precomputed*,
- simple expressions for $\mathbf{grad}\,\lambda_i\cdot\mathbf{grad}\,\lambda_j$ are available, see Section 3.3.5.

More on the use of these transformation techniques ➢ Section 3.7.3

Example 3.7.13 (BETL style representation of local shape functions for Lagrangian finite elements)

BETL has built-in classes for (low order) Lagrangian finite elements. They are defined in `Library/fe/fe_lagrange_basis_functions.hpp` for each reference element type ➙ BETL. The interface to these pre-defined finite elements is **betl2::fe::FEBasis**, see § 3.6.75 for a first discussion.

However a **betl2::fe::FEBasis**-compatible object has to provide facilities beyond what was discussed in § 3.6.75, namely two types

```
typename FEBASIS::template basisFunction_t<RET>;
typename FEBASIS::template diffBasisFunction_t<RET>;
```

Both take an `enum eth::base::RefElType` as template argument and have to match the following specification:

```
// geometry type of element for which FEBasis was designed
static const eth::base::RefElType refElType = ;
// number of local shape functions
static const int numFunctions = ;
// Number of vector components of return value
static const int functionDim = ;
// Dimension of ambient space for reference element
static const int localDim = ;
using matrix_t = ; // fixed size EIGEN matrix
// Evaluation for multiple points passed a columns of a matrix
template< int NUM_POINTS >
static matrix_t< numFunctions, NUM_POINTS*functionDim >
Eval( const matrix_t< localDim, NUM_POINTS > & );
```

The type `basisFunction_t` does the evaluation for the local shape functions themselves, the `Eval()` method of `diffBasisFunction_t` returns the gradients of the local shape functions (for `fe::FEBasisType` `Eval()` takes a matrix with point coordinates with respect to the reference element in its columns. The number of columns has to be passed as a template parameter. It returns a matrix

- with a rows for each individual local shape functions,
- with the result (vectors) of the evaluations in the passed points horizontally concatenated in each row.

The following code snipped shows how to request values and gradients of the local shape functions for $\mathcal{S}_2^0(\mathcal{M})$ in a point on the BETL reference triangle.

C++11 code 3.7.14: Fundamental BETL types related to Lagrangian finite elements ➙ GITLAB

```
// definition of some static enumerators
static const betl2::fe::ApproxOrder order  = fe::Quadratic;
static const betl2::fe::FEBasisType type   =
    fe::FEBasisType::Lagrange;
static const eth::base::RefElType RET_TRIA =
```

```
    eth::base::RefElType::TRIA;
static const eth::base::RefElType RET_QUAD=
    eth::base::RefElType::QUAD;
// typedefs for local basis functions and its gradient for reference
    element type TRIA
using fe_basis_t = typename fe::FEBasis< order, type >;
using loc_fun_tria_t = typename fe_basis_t::template
    basisFunction_t<RET_TRIA>;
using grad_loc_fun_tria_t = typename fe_basis_t::template
    diffBasisFunction_t<RET_TRIA>;
// typedefs for local basis functions and its gradient for reference
    element type QUAD
using loc_fun_quad_t = typename fe_basis_t::template
    basisFunction_t<RET_QUAD>;
using grad_loc_fun_quad_t = typename fe_basis_t::template
    diffBasisFunction_t<RET_QUAD>;

// instantiation of local basis functions for quadratic Lagrangian
// FE for the reference element type TRIA
loc_fun_tria_t local_FE_tria;
// instantiation of the gradient of the local basis functions for
// quadratic Lagrangian FE for the reference element type TRIA
grad_loc_fun_tria_t local_FE_grad_tria;
// local point x = x̂= (1,1)^T for evaluation of local basis functions
grad_loc_fun_tria_t::matrix_t<2,1> x;
x << 1,1 ;
cout << "#(lsf) for type " << RET_TRIA << " = " <<
    grad_loc_fun_tria_t::numFunctions << endl;
cout << "lsf([" << x.transpose() << "]^T) = " << local_FE_tria.Eval(
    x ) << endl;
cout << "grad lsf([" << x.transpose() << "]^T = \n" <<
    local_FE_grad_tria.Eval( x ) << endl;

// instantiation of local basis functions for quadratic Lagrangian FES
// for the reference element type QUAD
loc_fun_quad_t local_FE_quad;
// instantiation of the gradient of the local basis functions for
// quadratic Lagrangian FES for the reference element type QUAD
grad_loc_fun_quad_t local_FE_grad_quad;
cout << "#(lsf) for type " << RET_QUAD << " = " <<
    grad_loc_fun_quad_t::numFunctions << endl;
cout << "lsf([" << x.transpose() << "]^T) = " << local_FE_quad.Eval(
    x ) << endl;
cout << "grad lsf([" << x.transpose() << "]^T) = %\n" <<
    local_FE_grad_quad.Eval(x) << endl;
```

The selector type `type = fe:FEBasisType::Lagrange` corresponds to Lagrangian finite elements. Here `order` is the polynomials degree. Lagrangian finite elements are available in BETL up to third order, i.e. **fe::Constant, fe::Linear, fe::Quadratic, fe::Cubic**.

As explained above an object of type `fe_basis_t::template basisFunction_t<RET>` or `fe_basis_t`

`diffBasisFunction_t<RET>` provides the following member function

```
template< int NUM_POINTS >
  static matrix_t< numFunctions, NUM_POINTS >
    Eval(const matrix_t< 2, NUM_POINTS >& local)
```

It stores the values of all local shape functions at the point $\widehat{x}$ (passed in `local`) in the reference element in a static matrix of type `matrix_t<numFunctions, functionDim*NUM_POINTS>`, which is a double valued, dense Eigen matrix of fixed size `(numFunctions,functionDim*NUM_POINTS)`, where `numFunctions` refers to the number of local shape functions and is a static member of the class. Also `functionDim` is a static method of the class that tells the dimension of the basis functions' return value, that is, 1 for scalar-valued finite elements, d for vector valued.

Running the executable corresponding to the mainfile listed in Code 3.7.14 provides the following output:

```
Number of local shape functions for element type TRIA is 6
Evaluation of the local shape functions in x = [1 1]^T gives
 0  0  1 -0  0 -0
Evaluation of the gradients of local shape functions in x = [1 1]^T gives
 1 -1  0 -0  4 -4
 0  1  3  0 -4 -0
Number of local shape functions for element type QUAD is 9
Evaluation of the local shape functions in x = [1 1]^T gives
 0  0  1  0 -0 -0 -0 -0  0
Evaluation of the gradients of local shape functions in x = [1 1]^T gives
 0  0  3  1 -0 -0 -4 -0  0
 0  1  3  0 -0 -4 -0 -0  0
```

3.7.2 Example: Quadrilaterial Lagrangian finite elements

So far, see Section 3.4.3 and Eq. (3.4.18), we have adopted the perspective

$$\text{global shape functions} \xrightarrow{\text{Restriction to element}} \text{local shape functions}$$

Now we reverse this construction

$$\text{local shape functions} \xrightarrow{\text{“glueing”}} \text{global shape functions} \tag{3.7.15}$$

In fact, when building the global basis functions for quadratic Lagrangian finite elements we already proceeded this way, see Ex. 3.5.3. Fig. 129 lucidly conveys what is meant by “glueing”.

Be aware that the possibility to achieve a continuous global basis function by glueing together local shape function on adjacent cells, entails a judicious choice of the local shape functions.

This section will demonstrate how the policy (3.7.15) together with the formula (3.7.5) will enable us to extend Lagrangian finite element beyond the meshes discussed in Section 3.5.

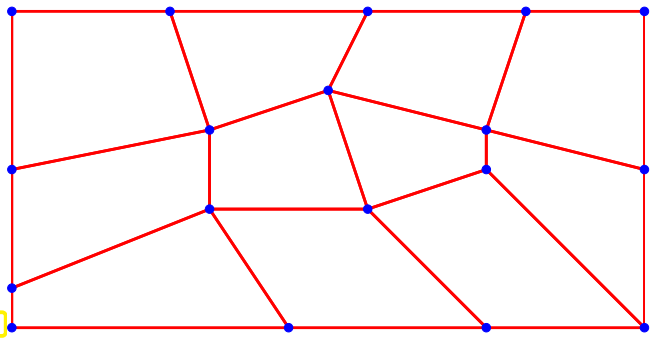


Fig. 171

◁ quadrilateral mesh $\mathcal{M}$ in 2D

What is "$\mathcal{S}_1^0(\mathcal{M})$"?

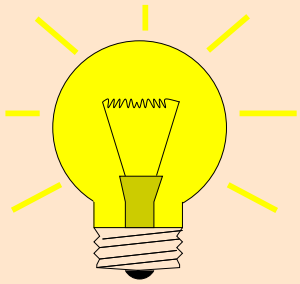
So far we know Lagrangian finite elements only on rectangles, see Section 3.5.2, for which the local spaces are given by $\mathcal{Q}_p(K)$ (→ Def. 3.5.12).

(3.7.16) Bilinear transformations

Clear: If K is a rectangle, $\widehat{K}$ the unit square, then there is a unique affine transformation $\mathbf{\Phi}_K$ (→ Def. 3.6.136) with $K = \mathbf{\Phi}_K(\widehat{K})$.

In this case (3.7.5) holds for the local shape functions of bilinear Lagrangian finite elements from Ex. 3.5.8 (and all tensor product Lagrangian finite elements introduced in Section 3.5.2)

Princple of constructing of parametric finite elements



Idea:

- ✦ local shape functions $\xrightarrow{\text{"glueing"}}$ global shape functions
- ✦ Build local shape functions by "inverse pullback"

$$b_K^i = (\mathbf{\Phi}_K^{-1})^* \widehat{b}^i \,, \tag{3.7.18}$$

where $\left\{\widehat{b}^i\right\}_{i=1}^{Q} \hat{=}$ set of shape functions on reference element $\widehat{K}$.

➢ What is $\mathbf{\Phi}_K$ for a general quadrilateral ?

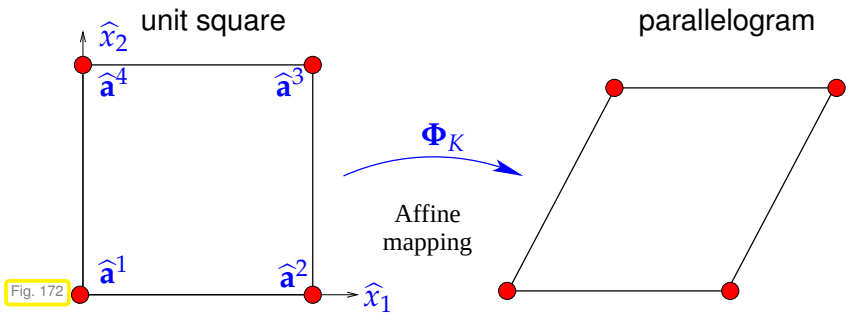


Fig. 172

Affine transformations fail to produce general quadrilaterals from a square. They only give parallelograms.

It takes *bilinear transformations* to obtain a generic quadrilateral from the unit square.

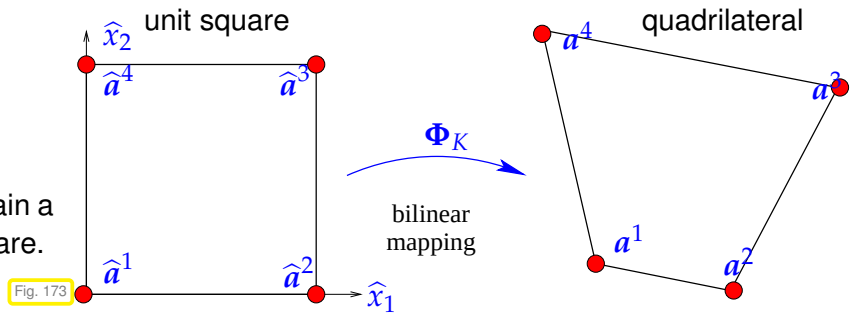


Fig. 173

Bilinear transformation of unit square to quadrilateral with vertices $\boldsymbol{a}^i$, $i = 1,2,3,4$:

$$\mathbf{\Phi}_K(\widehat{\boldsymbol{x}}) = (1-\widehat{x}_1)(1-\widehat{x}_2)\,\boldsymbol{a}^1 + \widehat{x}_1(1-\widehat{x}_2)\,\boldsymbol{a}^2 + \widehat{x}_1\widehat{x}_2\,\boldsymbol{a}^3 + (1-\widehat{x}_1)\widehat{x}_2\,\boldsymbol{a}^4 \,. \tag{3.7.19}$$

⇓

$$\mathbf{\Phi}_K(\widehat{\mathbf{x}}) = \begin{bmatrix} \alpha_1 + \beta_1\widehat{x}_1 + \gamma_1\widehat{x}_2 + \delta_1\widehat{x}_1\widehat{x}_2 \\ \alpha_2 + \beta_2\widehat{x}_1 + \gamma_2\widehat{x}_2 + \delta_2\widehat{x}_1\widehat{x}_2 \end{bmatrix}, \quad \alpha_i, \beta_i, \gamma_i, \delta_i \in \mathbb{R}\,.$$

The mapping property $\mathbf{\Phi}_K(\widehat{\mathbf{a}}^i) = \mathbf{a}^i$ is evident. In order to see $\mathbf{\Phi}_K(\widehat{K}) = K$ ($\widehat{K} \mathrel{\hat{=}}$ unit square) for (3.7.19), verify that $\mathbf{\Phi}_K$ maps all parallels to the coordinate axes to straight lines.

Moreover, a simple computation establishes:

If $\widehat{K}$ is the unit square, $\mathbf{\Phi}_K : \widehat{K} \mapsto K$ a bilinear transformation, and $\widehat{b}^i$ the bilinear local shape functions (3.5.10) on $\widehat{K}$,

then $(\mathbf{\Phi}_K^{-1})^*\widehat{b}^i$ are linear on the edges of K.

(3.7.20) Glueing of local shape functions on quadrilateral meshes

The last observation in § 3.7.16 makes possible the "glueing" of local shape functions obtained by inverse pullback from a nodal basis of $\mathcal{Q}_1(\widehat{K})$ on the unit square $\widehat{K}$.

Explanation:

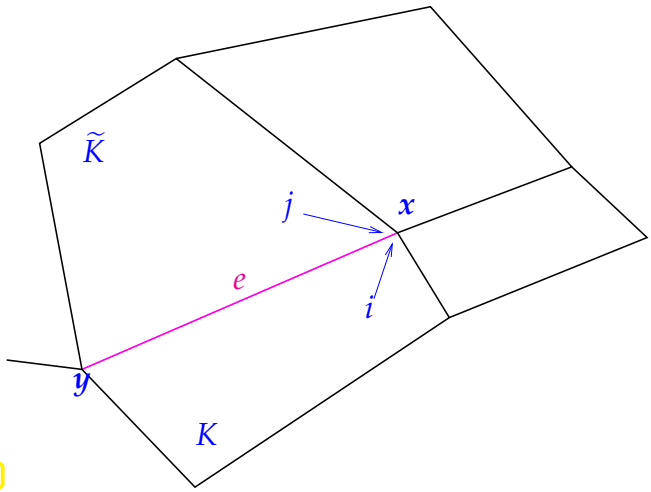


Fig. 174

❶ Pick a vertex $\boldsymbol{x} \in \mathcal{V}(\mathcal{M})$ and consider an adjacent quadrilateral K, on which there is a local shape function b_K^i such that $b_K^i(\boldsymbol{x}) = 1$ and b_K^i vanishes on all other vertices of K. This local shape function is obtained by inverse pullback of the $\widehat{b}^i$ associated with $\mathbf{\Phi}_K^{-1}(\boldsymbol{x})$.

❷ The same construction can be carried out for another quadrilateral $\widetilde{K}$ that shares the vertex $\boldsymbol{x}$ and an edge e with K. On that quadrilateral we find the local shape function $b_{\widetilde{K}}^j$

❸ Both $b_{K\,|e}^i$ and $b_{\widetilde{K}\,|e}^j$ are linear and attain the same values, that is 0 and 1 at the endpoints $\mathbf{x}$ and $\mathbf{y}$ of e, respectively.

▶ $$b_{K\,|e}^i = b_{\widetilde{K}\,|e}^j$$

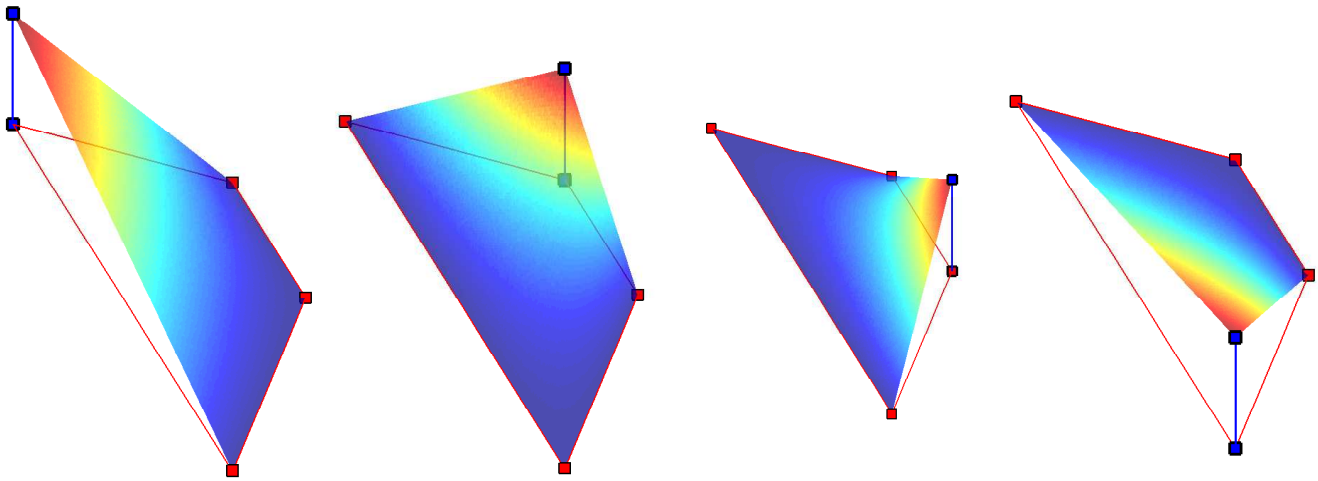
▶ Continuity of global shape function (defined by interpolation conditions at nodes)

Remark 3.7.21 (Non-polynomial "bilinear" local shape functions)

Note that the components of $\mathbf{\Phi}_K^{-1}$ are *not polynomial* even if $\mathbf{\Phi}_K$ is a bilinear transformation (3.7.19).

The local shape functions b_K^i defined by (3.7.18), where $\mathbf{\Phi}_K$ is a bilinear transformation and $\widehat{b}^i$ are the bilinear local shape functions on the unit square, are **not polynomial** in general.

Visualization of local shape functions on trapezoidal cell $K := \operatorname{convex}\left\{ \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 3 \\ 0 \end{bmatrix}, \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix} \right\}$:



3.7.3 Transformation techniques

In the previous section we already generalized the notion of affine equivalent finite element spaces from Section 3.7.1.

"Bilinear" Lagrangian finite elements = a specimen of parametric finite elements

Definition 3.7.22. Parametric finite elements

A finite element space on a mesh $\mathcal{M}$ is called parametric, if there exists a reference element $\widehat{K}$, $Q \in \mathbb{N}$, and functions $\widehat{b}^i \in C^0(\overline{\widehat{K}})$, $i = 1, \dots, Q$, such that

$$\forall K \in \mathcal{M}: \quad \exists \text{ bijection } \mathbf{\Phi}_K : \widehat{K} \mapsto K: \quad \widehat{b}^i = \mathbf{\Phi}_K^* b_K^i, \quad i = 1, \dots, Q \,,$$

where $\{b_K^1, \dots, b_K^Q\}$ = set of local shape functions on K.

This definition takes the possibility of "glueing" for granted: the concept of a local shape function, see (3.4.18), implies the existence of a global shape function with the right continuity properties (C^0-continuity for $H^1(\Omega)$-conforming finite element spaces).

How to implement parametric finite elements ?

We consider a generic elliptic 2nd-order variational Dirichlet problem

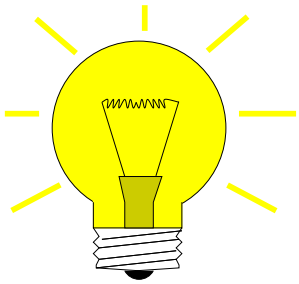
$$\begin{array}{l} u \in H^1(\Omega) \,, \\ u = g \text{ on } \partial\Omega \end{array}: \quad \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x}) \,\mathbf{grad}\, u(\boldsymbol{x})) \cdot \mathbf{grad}\, v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,. \tag{2.4.5}$$

(3.7.23) Local computations for parametric finite elements

We focus on the computation of element (stiffness) matrices and element (load) vectors (→ Def. 3.6.69), a key step in the set-up of the Galerkin matrix and right hand side vector.

Both challenge and opportunities arise from the implicit definition of the local local shape functions via pullback (→ Def. 3.7.2)

$$\boxed{b_K^i = (\mathbf{\Phi}_K^{-1})^* \widehat{b}^i \quad \Leftrightarrow \quad \widehat{b}^i = \mathbf{\Phi}_K^* b_K^i \,, \quad i = 1, \dots, Q} \,.$$



Known: transformation $\mathbf{\Phi}_K : \widehat{K} \mapsto K$ reference element $\widehat{K}$

Idea: use transformation to $\widehat{K}$ to compute element stiffness matrix $\mathbf{A}_K$, and element load vector $\vec{\varphi}_K$:

Detailed formulas for entries of element matrix $\mathbf{A}_K$ and element vector $\vec{\varphi}_K$:

$$\begin{aligned}
(\mathbf{A}_K)_{ij} &= \int_K \boldsymbol{\alpha}(\boldsymbol{x})\, \mathbf{grad}\, b_K^j(\boldsymbol{x}) \cdot \mathbf{grad}\, b_K^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \\
&= \int_{\widehat{K}} (\mathbf{\Phi}_K^* \boldsymbol{\alpha})(\widehat{\boldsymbol{x}}) \underbrace{(\mathbf{\Phi}_K^*(\mathbf{grad}\, b_K^j))}_{=\,?}(\widehat{\boldsymbol{x}}) \cdot \underbrace{(\mathbf{\Phi}_K^*(\mathbf{grad}\, b_K^i))}_{=\,?}(\widehat{\boldsymbol{x}})\, |\det D\mathbf{\Phi}_K(\widehat{\boldsymbol{x}})|\, \mathrm{d}\widehat{\boldsymbol{x}} \,, \\
(\vec{\varphi}_K)_i &= \int_K f(\boldsymbol{x}) b_K^i(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} (\mathbf{\Phi}_K^* f)(\widehat{\boldsymbol{x}})\, \widehat{b}^i(\widehat{\boldsymbol{x}})\, |\det D\mathbf{\Phi}_K(\widehat{\boldsymbol{x}})|\, \mathrm{d}\widehat{\boldsymbol{x}} \,,
\end{aligned}$$

by transformation formula (for multidimensional integrals, see also (3.6.150)):

$$\int_K f(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_{\widehat{K}} f(\widehat{\boldsymbol{x}}) |\det D\mathbf{\Phi}_K(\widehat{\boldsymbol{x}})|\, \mathrm{d}\widehat{\boldsymbol{x}} \quad \text{for } f : K \mapsto \mathbb{R} \,, \quad \boldsymbol{x} = \mathbf{\Phi}_K(\widehat{\boldsymbol{x}}) \,, \tag{3.7.24}$$

All integrals have been transformed to the reference element $\widehat{K}$, where we can now apply a quadrature formula:

$$\int_{\widehat{K}} \widehat{f}(\widehat{\boldsymbol{x}})\, \mathrm{d}\widehat{\boldsymbol{x}} \approx \sum_{l=1}^{P} \widehat{\omega}_l \widehat{f}(\widehat{\boldsymbol{\zeta}}_l) \,, \quad \widehat{\boldsymbol{\zeta}}_l \in \widehat{K},\ \widehat{\omega}_l \in \mathbb{R} \,, \tag{3.6.151}$$

which can be combined with (3.7.24):

$$\int_K f(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \approx \sum_{l=1}^{P} \widehat{\omega}_l f(\mathbf{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l))\, |\det D\mathbf{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l)| \,. \tag{3.7.25}$$

Required information and evaluations:

- values $\widehat{b}^i(\widehat{\boldsymbol{\zeta}}_l)$, $i = 1, \dots, Q$, $l = 1, \dots, P$,
- gradients $\mathbf{\Phi}^*(\mathbf{grad}\, b_K^i)$ at quadrature nodes $\widehat{\boldsymbol{\zeta}}_l \in \widehat{K}$ **!?**
- metric factors at quadrature nodes in $\widehat{K}$: $\det D\mathbf{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l)$
- values $\boldsymbol{\alpha}(\mathbf{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l)) \in \mathbb{R}^{d,d}$ and $f(\mathbf{\Phi}_K(\widehat{\boldsymbol{\zeta}}_l)) \in \mathbb{R}$ from point evaluations of functions $\boldsymbol{\alpha} : \overline{\Omega} \to \mathbb{R}^{d,d}$, $f : \overline{\Omega} \to \mathbb{R}$.

The gradients seem to pose a problem as b_K^i may be elusive, *cf.* Rem. 3.7.21! Fortunately we can compute them from the gradients of the local shape functions $\widehat{b}^j$ on the reference element using the formulas given in the next lemma.

Lemma 3.7.26. Transformation formula for gradients

For differentiable $u : K \mapsto \mathbb{R}$ and any diffeomorphism $\mathbf{\Phi} : \widehat{K} \mapsto K$ we have

$$(\mathbf{grad}_{\widehat{x}}(\mathbf{\Phi}^* u))(\widehat{x}) = (D\mathbf{\Phi}(\widehat{x}))^T \underbrace{(\mathbf{grad}_x u)(\mathbf{\Phi}(\widehat{x}))}_{=\mathbf{\Phi}^*(\mathbf{grad}\, u)(\widehat{x})} \quad \forall \widehat{x} \in \widehat{K} \,. \tag{3.7.27}$$

Proof. Use **chain rule** for components of the gradient

$$\frac{\partial \mathbf{\Phi}^* u}{\partial \widehat{x}_i}(\widehat{x}) = \frac{\partial}{\partial \widehat{x}_i} u(\mathbf{\Phi}(\widehat{x})) = \sum_{j=1}^{d} \frac{\partial u}{\partial x_j}(\mathbf{\Phi}(\widehat{x}))\, \frac{\partial \mathbf{\Phi}_j}{\partial \widehat{x}_i}(\widehat{x}) \,.$$

$$\blacktriangleright \quad \begin{bmatrix} \frac{\partial \mathbf{\Phi}^* u}{\partial \widehat{x}_1}(\widehat{x}) \\ \vdots \\ \frac{\partial \mathbf{\Phi}^* u}{\partial \widehat{x}_d}(\widehat{x}) \end{bmatrix} = (\mathbf{grad}_{\widehat{x}}\, \mathbf{\Phi}^* u)(\widehat{x}) = D\mathbf{\Phi}(\widehat{x})^T \begin{bmatrix} \frac{\partial u}{\partial x_1}(\mathbf{\Phi}(\widehat{x})) \\ \vdots \\ \frac{\partial u}{\partial x_d}(\mathbf{\Phi}(\widehat{x})) \end{bmatrix} = D\mathbf{\Phi}(\widehat{x})^T (\mathbf{grad}_x u)(\mathbf{\Phi}(\widehat{x})) \,.$$

Here, $D\mathbf{\Phi}(\widehat{x}) \in \mathbb{R}^{d,d}$ is the Jacobian of Φ at $\widehat{x} \in \widehat{K}$, see [18, Bem. 7.6.1]. □

Using Lemma 3.7.26 we arrive at a tractable expression for the entries of the element matrix:

$$\begin{aligned} (\mathbf{A}_K)_{ij} &= \int_{\widehat{K}} (\boldsymbol{\alpha}(\mathbf{\Phi}(\widehat{x}))(D\mathbf{\Phi})^{-T}\, \mathbf{grad}\, \widehat{b}^i) \cdot ((D\mathbf{\Phi})^{-T}\, \mathbf{grad}\, \widehat{b}^j) |\det D\mathbf{\Phi}|\, \mathrm{d}\widehat{x} \\ &= \int_{\widehat{K}} ((D\mathbf{\Phi})^{-1} \boldsymbol{\alpha}(\mathbf{\Phi}(\widehat{x}))(D\mathbf{\Phi})^{-T})\, \mathbf{grad}\, \widehat{b}^i \cdot \mathbf{grad}\, \widehat{b}^j |\det D\mathbf{\Phi}|\, \mathrm{d}\widehat{x} \,. \end{aligned} \tag{3.7.28}$$

Note that the argument $\widehat{x}$ is suppressed for some terms in the integrand.

✎ notation: for matrix $\mathbf{S}$ write $\mathbf{S}^{-T} := \left(\mathbf{S}^{-1}\right)^T = \left(\mathbf{S}^T\right)^{-1}$

The next step is the approximation of (3.7.28) by means of quadrature rule (3.6.132) on $\widehat{K}$, see (3.6.151):

$$(\mathbf{A}_K)_{ij} \approx \sum_{l=1}^{P} \widehat{\omega}_l \left(\mathbf{M}_K(\widehat{\zeta}_l)\, \mathbf{grad}\, \widehat{b}^i(\widehat{\zeta}_l)\right) \cdot \mathbf{grad}\, \widehat{b}^j(\widehat{\zeta}_l) |\det D\mathbf{\Phi}(\widehat{\zeta}_l)| \,, \tag{3.7.29}$$

$$\text{with} \quad \mathbf{M}_K(\widehat{x}) := (D\mathbf{\Phi})^{-1}(\widehat{x}) \boldsymbol{\alpha}(\mathbf{\Phi}(\widehat{x}))(D\mathbf{\Phi})^{-T}(\widehat{x}) \,, \quad \widehat{x} \in \widehat{K} \,.$$

The vectors $\mathbf{grad}\, \widehat{b}^i(\widehat{\zeta}_l)$ are easily computed, since the local shape functions $\widehat{b}^i$ will usually be simple polynomials. In addition, they are independent of K, so they can be precomputed and stored in a table. The same holds for the numbers $|\det D\mathbf{\Phi}(\widehat{\zeta}_l)|$.

Remark 3.7.30 (BETL support for transformation of gradients)

From (3.7.28) and (3.7.29) we see that $D\mathbf{\Phi}(\widehat{x})$ for certain points $\widehat{x} \in \widehat{K}$ is required for the computation of entries of element matrices. BETL supplies this matrix through a function of **Geometry**:

```
template< int NUM_POINTS > matrix_t< dimFrom, dimTo*NUM_POINTS >
jacobianInverseTransposed( const matrix_t< dimFrom, NUM_POINTS
   >& local );
```

This function takes a matrix argument whose columns give point coordinates in the reference element. The `dimFrom`/`dimTo` template arguments specify the dimension of the ambient space of the reference element/actual element, 2 throughout in this course. The function returns the `NUM_POINTS` inverses of the transposed Jacobian horizontally concatenated into a big matrix.

Example 3.7.31 (Transformation techniques for bilinear transformations)

In Section 3.7.2 we saw that it takes a general bilinear transformation (3.7.19) to map a square onto a general quadrilateral cell, see Fig. 173 on page 341. It turned out that these bilinear mappings are key to defining *parametric* Lagrangian finite elements on general quadrilaterals.

In order to compute the element (stiffness) matrices according to (3.7.29), we have to evaluate the Jacobians for bilinear transformations and their determinants. This can be done through the following formulas:

$$\begin{aligned} \mathbf{\Phi}(\widehat{\boldsymbol{x}}) &= \begin{bmatrix} \alpha_1 + \beta_1\widehat{x}_1 + \gamma_1\widehat{x}_2 + \delta_1\widehat{x}_1\widehat{x}_2 \\ \alpha_2 + \beta_2\widehat{x}_1 + \gamma_2\widehat{x}_2 + \delta_2\widehat{x}_1\widehat{x}_2 \end{bmatrix}, \quad \alpha_i, \beta_i, \gamma_i, \delta_i \in \mathbb{R}\,, \\ \Rightarrow \qquad D\mathbf{\Phi}(\widehat{\boldsymbol{x}}) &= \begin{bmatrix} \beta_1 + \delta_1\widehat{x}_2 & \gamma_1 + \delta_1\widehat{x}_1 \\ \beta_2 + \delta_2\widehat{x}_2 & \gamma_2 + \delta_2\widehat{x}_1 \end{bmatrix}, \\ \Rightarrow \qquad \det(D\mathbf{\Phi}(\widehat{\boldsymbol{x}})) &= \beta_1\gamma_2 - \beta_2\gamma_1 + (\beta_1\delta_2 - \beta_2\delta_1)\widehat{x}_1 + (\delta_1\gamma_2 - \delta_2\gamma_1)\widehat{x}_2\,. \end{aligned} \tag{3.7.32}$$

Both $D\mathbf{\Phi}(\widehat{\boldsymbol{x}})$ and $\det(D\mathbf{\Phi}(\widehat{\boldsymbol{x}}))$ are (componentwise) linear in $\boldsymbol{x}$.

If $\mathbf{\Phi} = \mathbf{\Phi}_K$ for a generic quadrilateral K as in (3.7.19), then the coefficients $\alpha_i, \beta_i, \gamma_1, \delta_i$ depend on the shape of K in a straightforward fashion:

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \end{bmatrix} = \mathbf{a}^1\,, \quad \begin{bmatrix} \beta_1 \\ \beta_2 \end{bmatrix} = \mathbf{a}^2 - \mathbf{a}^1\,, \quad \begin{bmatrix} \gamma_1 \\ \gamma_2 \end{bmatrix} = \mathbf{a}^4 - \mathbf{a}^1\,, \quad \begin{bmatrix} \delta_1 \\ \delta_2 \end{bmatrix} = \mathbf{a}^3 - \mathbf{a}^2 - \mathbf{a}^4 + \mathbf{a}^1\,.$$

Example 3.7.33 (Local computations in BETL based on transformation techniques)

Focus on linear variational problem:

$$u \in H_0^1(\Omega): \quad \int_\Omega \mathbf{M}(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x}) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,.$$

We assume that the (diffusion) coefficient function $\mathbf{M} : \Omega \mapsto \mathbb{R}^{d,d}$ is matrix-valued, uniformly symmetric positive definite, and given in procedural form as a function that allows point evaluations, *cf.* Rem. 1.5.5. Writing b_K^j, $j = 1, \dots, Q(K)$, for the local shape functions on cell (element) $K \in \mathcal{M}$, see Def. 3.4.19, we obtain for the element matrix

$$(\mathbf{A}_K)_{ij} = \int_K \mathbf{M}(\boldsymbol{x})\,\mathbf{grad}\,b_K^j \cdot \mathbf{grad}\,b_K^i\,\mathrm{d}\boldsymbol{x}\,, \quad i,j \in \{1, \dots, Q\}\,. \tag{3.7.34}$$

Writing $\mathbf{\Phi} : \widehat{K} \to K$ for the transformation from the reference element and applying a P-point quadrature formula on $\widehat{K}$, we arrive at

$$(\mathbf{A})_{ij} = \sum_{l=1}^{P} \widehat{\omega}_l |\det D\mathbf{\Phi}(\widehat{\zeta}_l)| \boldsymbol{\alpha}(\mathbf{\Phi}(\widehat{\zeta}_l)) D\mathbf{\Phi}(\widehat{\zeta}_l)^{-\top} (\mathbf{grad}\, \widehat{b}^j)(\widehat{\zeta}_l) \cdot D\mathbf{\Phi}(\widehat{\zeta}_l)^{-\top} (\mathbf{grad}\, \widehat{b}^j)(\widehat{\zeta}_l) \,. \qquad (??)$$

The element matrix can be computed by the following compact formula

$$\mathbf{A}_K = \sum_{l=1}^{P} \widehat{\omega}_l |\det D\mathbf{\Phi}(\widehat{\zeta}_l)| \begin{bmatrix} \mathbf{g}_{1,l}^\top \\ \vdots \\ \mathbf{g}_{Q,l}^\top \end{bmatrix} \boldsymbol{\alpha}(\mathbf{\Phi}(\widehat{\zeta}_l)) \left[\mathbf{g}_{1,l} \cdots \mathbf{g}_{Q,l}\right] , \tag{3.7.35}$$

$$\mathbf{g}_{j,l} := D\mathbf{\Phi}(\widehat{\zeta}_l)^{-\top} (\mathbf{grad}\, \widehat{b}^j)(\widehat{\zeta}_l) = \mathbf{grad}\, b_K^j(\zeta_l) \in \mathbb{R}^d \,. \tag{3.7.36}$$

In order to implement a local assembler in BETL using local computations based on transformation techniques, we require the following two ingredients:

- a **fe::FEBasis**-compatible object (template parameter `FEBASIS`) that handles the basis functions and was introduced in § 3.6.75 and Ex. 3.7.13. In particular, an `FEBASIS` object has to provide the two types **basisFunction_t** and **diffBasisFunction_t**.
- a **QuadRuleList**-compatible object as explained in § 3.6.164 (template parameter `QUADRULES`).

Moreover, as we learnt in Ex. 3.6.94, a `ELEM_MAT_BUILDER` object (for the local element matrix) in BETL gets an argument of type `BUILDER_DATA_T`, which can be used to pass arbitrary information for local computations. Here this role is played by the object of type `MATERIAL`. Here, we expect this type to provide a functor with an evaluation operator according to

```
inline result_t operator()( localPoint_t x, element_t e) const
```

C++11 code 3.7.37: Computation of general element matrix according to (3.7.29) **in BETL**
→ **GITLAB**

```
template< typename FEBASIS, typename QUADRULES >
struct StiffnessLocalMatrixAssembler{
private:
  static const int dim_ = 2;  // world dimension (2D)
public:
  typedef double numeric_t;
  typedef Eigen::Matrix< numeric_t, Eigen::Dynamic, Eigen::Dynamic >
      result_t;
  static void initialize() {}

  template<  class MATERIAL, class ELEMENT>
  static result_t eval( const MATERIAL& M, const ELEMENT& el){
    result_t result;
    switch( el.refElType() ){
    case eth::base::RefElType::TRIA:  {
     eval_< eth::base::RefElType::TRIA > evaluator;
     result = evaluator.compute_( M, el ); break;  }
    case eth::base::RefElType::QUAD:  {
```

```
      eval_< eth::base::RefElType::QUAD > evaluator;
      result = evaluator.compute_( M, el ); break;  }
    default:
      ETH_ASSERT_MSG( false, "Implemented for TRIA and QUAD only");
    }
    return(result);
  }
```

We observe that the structure is rather similar to the implementation of **MySimpleLocalVectorAssembler** in Code 3.6.168, and that the implementation relies on the private member struct whose `compute_` method carries out the quadrature and the computation of the local element matrix. As a template parameter it is passed the reference element type.

C++11 code 3.7.38: Evaluator struct for StiffnessLocalMatrixAssembler ➙ GITLAB

```
template< eth::base::RefElType RET >
struct eval_ {
  template< class MATERIAL, class ELEMENT>
  static result_t compute_(const MATERIAL& M, const ELEMENT& el){
    typedef typename FEBASIS::template diffBasisFunction_t<RET>
        basisFunctGrads;
    typedef betl2::quad::Quadrature< RET, QUADRULES::template get<
        RET >() > quadrule_t;
    // Number of local shape functions
    static const auto nDofs_  = FEBASIS::template numDofs<RET>();
    // Initialize result matrix to zero
    result_t result( nDofs_, nDofs_ ); result.setZero();
    // Get GEOMETRY object for current element
    const auto& geom = el.geometry();
    // Get local quadrature points and weights
    const auto& xi = quadrule_t::getPoints();
    const auto& wi = quadrule_t::getWeights()*quadrule_t::getScale();
    // Number of quadrature points (known at compile time!)
    const int nQuadP = quadrule_t::getNumPoints( );
    // Compute |det DΦ| in quadrature points § 3.6.164
    const auto detJi = geom.template integrationElement<nQuadP>( xi );
    // Multiply weights with metric factors
    const auto coeff = detJi.cwiseProduct(wi);
    // Get DΦ^-T for every quadrature point Rem. 3.7.30
    const auto invJT = geom.template
        jacobianInverseTransposed<nQuadP>(xi);
    // Gradient of every basis function b̂^i in quadrature points
    const Eigen::Matrix< double, nDofs_, dim_*nQuadP > gradEval =
        basisFunctGrads::Eval(xi);
    // Loop over quadrature points
    for( int l=0; l < xi.cols(); l++ ) {
     // DΦ^-T(x̂) in current quadrature point
     const Eigen::Matrix< double, dim_, dim_ >& invJT_l =
         invJT.template block< dim_,dim_ >( 0, dim_*l );
```

```
30      // Compute gradient of actual local shape function according to
           Lemma 3.7.26
31      const Eigen::Matrix< double, nDofs_, dim_ >
32        grad_b = gradEval.template block<
             nDofs_,dim_>(0,l*dim_)*invJT_l.transpose();
33      // evaluate diffusion coefficient at current quadrature point
34      const auto Meval = M( xi.col(l), el );
35      // Implementation of (3.7.35)
36      result += coeff( l ) * grad_b * Meval * grad_b.transpose();
37       }
38       return(result);
39     }};
```

Line 5: Obtain a type that provides the evaluation of the gradients of local shape functions, see § 3.6.75. We assume `FEBASIS` to be of Lagrangian type (c.f. § 3.6.75).

Line 6: Here we fetch the quadrature rule that must be applied to the current reference element type, see § 3.6.164.

Line 22: Results are `nQuadP` blocks of size 2x2. Each block contains the Jacobian $D\mathbf{\Phi}^{-T}(\widehat{x})$ in a quadrature point $\widehat{x}$.

Line 25: In the variable `gradEval`, which is a matrix of size $\texttt{nDofs\_} \times \texttt{2nQuadP}$, every row contains the *transposed* gradients of a single local shape function, i.e. $(\mathbf{grad}\,\widehat{b}^j)^\top$, $j = 0...\text{nDofs\_} - 1$ concatenated horizontally.

Line 32: The j-th row of `grad_b` represents the *transposed* transformed gradient

$$\mathbf{grad}\, b_K^j(\boldsymbol{\zeta}_l) = (D\mathbf{\Phi})^{-T}(\widehat{\boldsymbol{\zeta}}_l)\,\mathbf{grad}\,\widehat{b}^j(\widehat{\boldsymbol{\zeta}}_l)$$

evaluated in the l-the quadrature node $\boldsymbol{\zeta}_l$ in the actual element K. Notice that the gradient is again handled as a row vector.

Line 36: Notice that the object `M` could return a matrix or a scalar. Both will work thanks to EIGEN's operator overloading.

A working example using this local assembler implementation can be found in ➥ GITLAB.

3.7.4 Boundary approximation

Intuition: Approximating a (smooth) curved boundary $\partial\Omega$ by a polygon/polyhedron will introduce a (sort of) discretization error.

> Parametric finite elment constructions provide a tool going beyond polygonal/polyhedral approximation of boundaries (by simple straight lines or flat faces).

An example of a mesh resolving a curved boundary is given in Fig. 112. Here we discuss this for a very simple case of triangular meshes in 2D (more details → [5, Sect, 10.2]).

Idea: Piecewise polynomial approximation of boundary (boundary fitting)
($\partial\Omega$ locally considered as function over straight edge of an element)

Example: Piecewise quadratic boundary approximation
(Part of $\partial\Omega$ between $\mathbf{a}^1$ and $\mathbf{a}^2$ approximated by parabola)

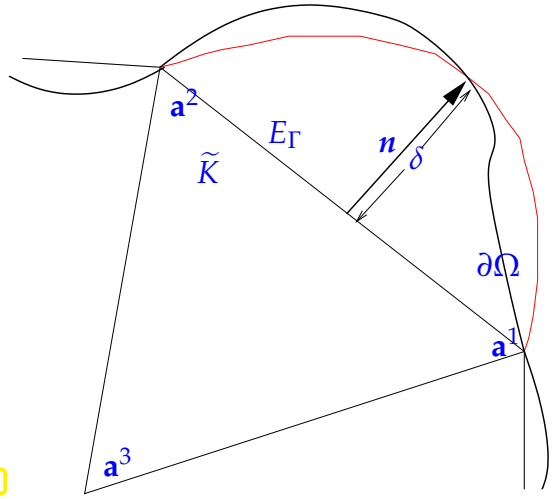


Fig. 175

(3.7.39) Piecewise quadratic polynomial boundary approximation

Mapping $\widetilde{K} \to$ "curved element" K:

$$\widetilde{\boldsymbol{\Phi}}_K(\widetilde{\mathbf{x}}) := \widetilde{\mathbf{x}} + 4\delta\,\lambda_1(\widetilde{\mathbf{x}})\lambda_2(\widetilde{\mathbf{x}})\,\boldsymbol{n}\,. \tag{3.7.40}$$

(λ_i barycentric coordinate functions on $\widetilde{K}$, $\boldsymbol{n}$ normal to E_Γ, see Fig. 175)

Note: Essential: δ sufficiently small $\Longrightarrow$ $\boldsymbol{\Phi}$ bijective

The complete transformation $\boldsymbol{\Phi}_K : \widehat{K} \mapsto K$ is obtained by joining an affine transformation ($\to$ Def. 3.6.136) $\boldsymbol{\Phi}_K^a : \widehat{K} \mapsto \widetilde{K}$, $\boldsymbol{\Phi}_K^a(\widehat{\boldsymbol{x}}) := \mathbf{F}_K\widehat{\boldsymbol{x}} + \boldsymbol{\tau}_K$, and $\widetilde{\boldsymbol{\Phi}}_K$:

$$\boldsymbol{\Phi}_K = \widetilde{\boldsymbol{\Phi}}_K \circ \boldsymbol{\Phi}_K^a\,.$$

For parabolic boundary fitting:

$$D\widetilde{\boldsymbol{\Phi}}_K = \mathbf{I} + 4\delta\,\boldsymbol{n}\cdot\mathbf{grad}(\lambda_1\lambda_2)^\top \in \mathbb{R}^{2,2} \quad , \quad \det(D\widetilde{\boldsymbol{\Phi}}_K) = 1 + 4\delta\,\boldsymbol{n}\cdot\mathbf{grad}(\lambda_1\lambda_2)\,.$$

Example 3.7.41 (Second-order geometry approximation in Gmsh)

The menu item `Mesh->Set Order 2` makes **Gmsh** insert information into the `.msh`-file that is necessary for parabolic boundary approximation.

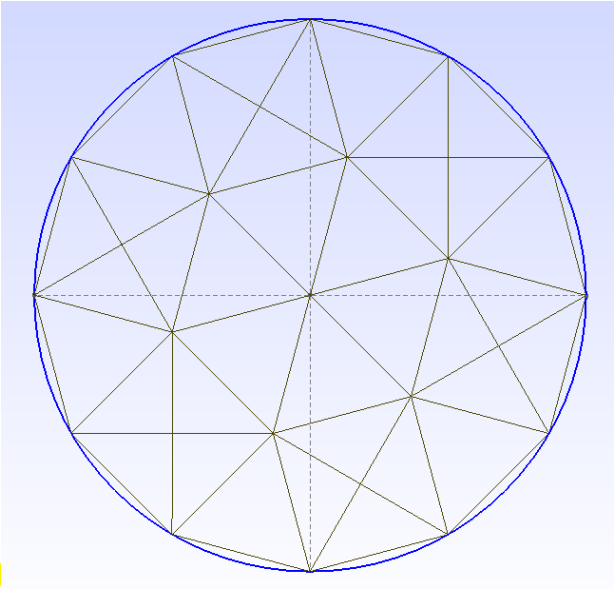


Fig. 176

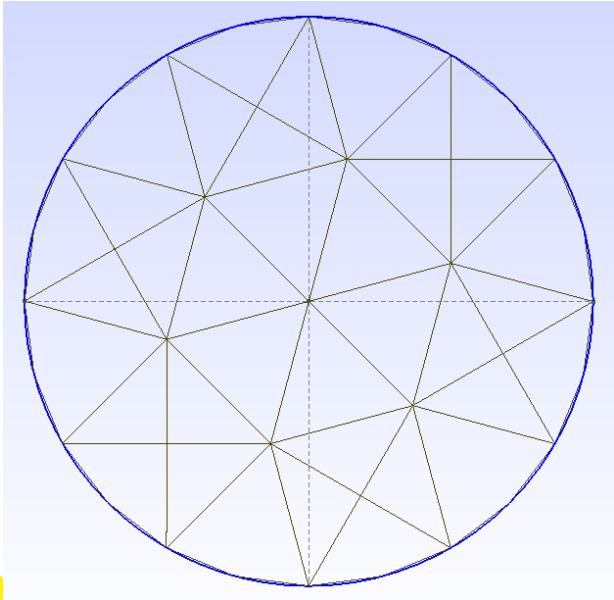


Fig. 177

Polygonal (left) and parabolic (right) approximation of a circular boundary

The `.msh`-file then contains entities of type 8 and 9, which corresponds to 3-node lines and 6-node triangles. The former is described by three locations, the latter by six, where the extra points designate the (shifted) midpoints of edges.

?! *Review question(s) 3.7.42.* (Parametric finite elements)

1. Which data required for the computation of the element matrices for 2nd-order elliptic variational problems discretized by means of Lagrangian finite elements, depend on the current cell, and which do not?
2. Does the numerical quadrature of the pullback of a function to the reference element yield the same value as its local numerical quadrature on a cell based on the same quadrature rule obtained from the reference element by transformation?
3. The mapping $\widehat{\boldsymbol{x}} \mapsto [a\widehat{x}_1 b\widehat{x}_2]^\top$, $a, b > 0$, takes the unit disc $\widehat{D} \subset \mathbb{R}^2$ to a ellipse Ω with axes a and b. Let $u \in H_0^1(\Omega)$ solve $\Delta u = f$, $f \in L^2(\Omega)$. Using Lemma 3.7.26, derive the variational problem solved by the pullback of u to $\widehat{D}$.
4. Give the formula for the bilinear transformation that maps the unit square to the "triangular" quadrilateral with vertices $\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1/2\\1/2\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}$.

3.8 Linearization

3.8.1 Non-linear variational problems

So far we have discussed the finite elements for *linear* second-order variational boundary value problems only.

However, as we have learned in § 1.5.91, in 1D the Galerkin approach based on linear finite elements was perfectly capable of dealing with *non-linear* two-point boundary value problems. Indeed the abstract dis-

cussion of the Galerkin approach in Section 1.5.2 was aimed at general and possibly non-linear variational problems, see (1.5.9), (1.5.23).

It goes without saying that the abstract (and formal) discussion of Section 1.5.2 remains true for *non-linear* second-order boundary value problems in variational form.

Difficult: Characterization of "spaces of functions with finite energy" ($\rightarrow$ Sobolev spaces, Section 2.3) for non-linear variational problems.

(Relief!) In this course we do not worry that much about function spaces.

Recall ($\rightarrow$ Rem. 1.3.31): Non-linear variational problem

$$\boxed{u \in V: \quad \mathsf{a}(u;v) = \ell(v) \quad \forall v \in V_0} \,, \tag{1.3.24}$$

- $V_0 \hat{=}$ test space, (real) vector space (usually a function space, "Sobolev-type" space $\rightarrow$ Section 2.3)
- $V \hat{=}$ trial space, affine space: usually $V = u_0 + V_0$, with offset function $u_0 \in V$,
- $f \hat{=}$ a linear mapping $V_0 \mapsto \mathbb{R}$, a linear form,
- $\mathsf{a} \hat{=}$ a mapping $V \times V_0 \mapsto \mathbb{R}$, *linear in the second argument*, that is

$$\mathsf{a}(u;\alpha v + \beta w) = \alpha \mathsf{a}(u;v) + \beta \mathsf{a}(u;w) \quad \forall u \in V\,,\, v,w \in V_0\,,\, \alpha,\beta \in \mathbb{R}\,. \tag{1.3.25}$$

Remember that linearity in the second argument is a key feature of variational equations (1.3.24) arising in the calculus of variations when seeking extrema of functionals on function spaces, see Section 1.3.1.

Example 3.8.1 (Heat conduction with radiation boundary conditons)

➢ 2nd-order elliptic boundary value problem, *cf.* (2.6.10) & (2.7.4)

$$\begin{aligned} -\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) &= f \quad \text{in } \Omega\,, \\ \kappa(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \boldsymbol{n}(\boldsymbol{x}) + \Psi(u) &= 0 \quad \text{on } \partial\Omega\,. \end{aligned}$$

► Variational formulation from Ex. 2.9.6

$$u \in H^1(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \Psi(u)\,v\,\mathrm{d}S = \int_\Omega f v\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,. \tag{2.9.8}$$

If $\Psi : \mathbb{R} \mapsto \mathbb{R}$ is not an affine linear function, then (2.9.8) represents a non-linear variational problem (1.3.24) with

- trial/test space $V = V_0 = H^1(\Omega)$ ($\rightarrow$ Def. 2.3.25),
- right hand side linear form $\ell(v) := \int_\Omega f v\,\mathrm{d}\boldsymbol{x}$,
- $\mathsf{a}(u;v) := \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \Psi(u)\,v\,\mathrm{d}S$.

Note that the non-linearity enters only through the boundary term.

Example 3.8.2 (Non-linear materials)

In the context of heat conduction (→ Section 2.6) a material is called non-linear, if its heat conductivity may also vary with temperature: Using the notations of § 2.6.4, this means that

$$\boldsymbol{\kappa} : \Omega \times \mathbb{R} \to \mathbb{R}^{3,3} \ , \quad \kappa = \kappa(\boldsymbol{x}, u) \ ,$$

where the uniform positivity (2.6.6) still has to hold, this time uniformly in the temperature, as well.

In the case of Dirichlet boundary conditions this leads to the 2nd-order elliptic boundary value problem (→ § 2.6.7)

$$-\operatorname{div}(\kappa(\boldsymbol{x}, u)\,\mathbf{grad}\, u) = f \quad \text{in } \Omega, \quad u = g \quad \text{on } \partial\Omega \ ,$$

with variational formulation

$$\begin{array}{r} u \in H^1(\Omega) \\ u = g \text{ on } \partial\Omega \end{array} : \quad \int_\Omega \boldsymbol{\kappa}(\boldsymbol{x}, u)\,\mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega) \ ,$$

where a "moderate" increase of $\|\boldsymbol{\kappa}\|$ as a function of u has to be assumed, in order to be able retain $H^1(\Omega)$ as variational space.

3.8.2 Newton in function space

Pursuing the policy of Galerkin discretization (choice of discrete spaces and corresponding bases, → Section 1.5.2) we can convert (1.3.24) into a non-linear system of equations

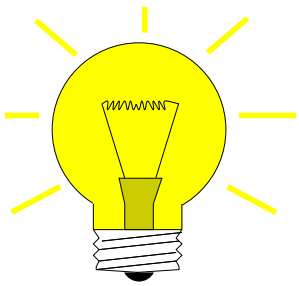
$$\mathsf{a}(u_0 + \sum_{j=1}^{N} \mu_j b_N^j ; b_N^k) = f(b_N^k) \quad \forall k = 1, \dots, N \ . \tag{1.5.23}$$

If the left hand side depends smoothly on the unkowns (the coefficients μ_j of $\vec{\boldsymbol{\mu}}$), then the classical Newton method (→ [14, Section 2.4]) to solve it iteratively.

Here, we focus on a different approach that reverses the order of the steps:

1. Linearization of problem ("Newton in function space"),
2. Galerkin discretization of linearized problems.

Recall idea of Newton's method [14, Section 2.4] for the iterative solution of $F(\mathbf{x}) = 0$, $F : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$ smooth:



Idea: local linearization:

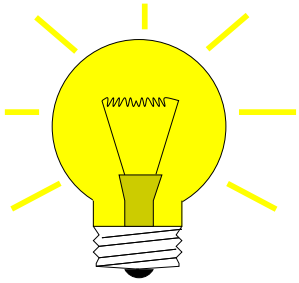
Given $\vec{\xi}^{(k)} \in D$ ➢ $\vec{\xi}^{(k+1)}$ as zero of affine linear model function

$$F(\vec{\xi}) \approx \widetilde{F}(\vec{\xi}) := F(\vec{\xi}^{(k)}) + DF(\vec{\xi}^{(k)})(\vec{\xi} - \vec{\xi}^{(k)}) \ .$$

◤ Newton iteration:

$$\boxed{\vec{\xi}^{(k+1)} := \vec{\xi}^{(k)} - DF(\vec{\xi}^{(k)})^{-1} F(\vec{\xi}^{(k)})} \ , \quad [\text{ if } DF(\vec{\xi}^{(k)}) \text{ regular }] \tag{3.8.3}$$

← apply idea to (1.3.24)



Idea: local linearization:

Given $u^{(k)} \in V$ ➢ $u^{(k+1)}$ from

$$\boxed{\begin{aligned} w \in V_0: \quad & \mathsf{a}(u^{(k)};v) + D_u\mathsf{a}(u^{(k)};v)w = \ell(v) \quad \forall v \in V_0 \,, \\ & u^{(k+1)} := u^{(k)} + w \,. \end{aligned}} \tag{3.8.4}$$

The meaning of $DF(\vec{\xi}^{(k)})$ in (3.8.3) is clear: it stands for the Jacobian of F evaluated at $\vec{\xi}^{(k)}$

But what is the meaning of $D_u\mathsf{a}(u^{(k)};v)w$ in (3.8.4)?

Remember the "definition" of the Jacobian (for sufficiently smooth F)

$$DF(\vec{\xi})\vec{\mu} = \lim_{t\to 0} \frac{F(\vec{\xi}+t\vec{\mu}) - F(\vec{\xi})}{t} \,, \quad \vec{\xi} \in D, \vec{\mu} \in \mathbb{R}^N \,. \tag{3.8.5}$$

➢ try the "definition" in the spirit of directional derivatives as exploited in the calculus of variations, see (1.3.6),

$$\boxed{D_u\mathsf{a}(u^{(k)};v)w = \lim_{t\to 0} \frac{\mathsf{a}(u+tw;v) - \mathsf{a}(u;v)}{t} \,, \quad u^{(k)} \in V \,, \quad v,w \in V_0} \,. \tag{3.8.6}$$

The next statement recalls a fact that we have come across in a similar form when computing directional derivatives of functionals in the calculus of variations approach, see Section 1.3.1: For a sufficiently smooth mapping the directional derivative is linear in the direction (variation), see 45.

In the current context the mapping is $u \mapsto \{v \mapsto \mathsf{a}(u;v)\}$, that is, a mapping from the trial space V into the space of linear forms (→ Def. 1.3.22) on the test space V_0 . Thus, its derivative in some $u \in V$ can be expected to be a linear mapping from $V_0 \to \{\text{linear forms on } V_0\}$.

Directional derivative of a variational form

If $(u,v) \mapsto \mathsf{a}(u;v)$ depends smoothly on u, then

$$(v,w) \mapsto D_u\mathsf{a}(u^{(k)};v)w \quad \text{is a bilinear form } V_0 \times V_0 \mapsto \mathbb{R}.$$

Example 3.8.8 (Derivative of non-linear $u \mapsto \mathsf{a}(u;\cdot)$)

We revisit the non-linear variational problem from Ex. 3.8.1 with

$$\mathsf{a}(u;v) := \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} \Psi(u)\, v \,\mathrm{d}S$$

Evidently, the derivative $D_u\mathsf{a}(u;v)w$ from (3.8.6) is *linear* in the sense that

$$D_u(\mathsf{b}(u;v) + \mathsf{c}(u;v))w = D_u\mathsf{b}(u;v)w + D_u\mathsf{c}(u;v)w \quad \forall v,w \in V_0 \,.$$

Hence we can separately compute the derivative of the two terms contributing to a:

First, we tackle the bilinear term, for which the derivative is straightforward, because for every *bilinear* form (→ Def. 1.3.22) $\mathsf{c} : V_0 \times V_0 \mapsto \mathbb{R}$ holds

$$D_u\mathsf{c}(u,v)w = \lim_{t\to 0} \frac{\mathsf{c}(u+tw,v) - \mathsf{c}(u,v)}{t} = \mathsf{c}(v,w) \,, \tag{3.8.9}$$

analogous the computations on page 148 that yielded the linear variational problem associated with a quadratic minimization problem. (3.8.9) can also be regarded as another incarnation of the fact that the derivative of a linear mapping is constant: $D_u\mathsf{c}(u,v)w$ does not depend on u!

Next, apply formula (3.8.6) to the non-linear boundary term in (2.9.8), that is, here

$$\mathsf{b}(u;v) := \int_{\partial\Omega} \Psi(u)v\,\mathrm{d}S\,, \quad u,v \in H^1(\Omega)\,.$$

$$\blacktriangleright \quad \mathsf{b}(u+tw;v) - \mathsf{b}(u;v) = \int_{\partial\Omega} (\Psi(u+tw) - \Psi(u))v\,\mathrm{d}S\,, \quad u,v \in H^1(\Omega)\,.$$

Assume $\Psi : \mathbb{R} \mapsto \mathbb{R}$ is smooth with derivative Ψ' and employ *Taylor expansion* for fixed $w \in H^1(\Omega)$ and $t \to 0$

$$\mathsf{b}(u+tw;v) - \mathsf{b}(u;v) = \int_{\partial\Omega} t\Psi'(u)wv\,\mathrm{d}S + O(t^2)\,.$$

$$\blacktriangleright \quad D_u\mathsf{b}(u^{(k)};v)w = \lim_{t\to 0} \frac{\mathsf{b}(u+tw;v) - \mathsf{b}(u;v)}{t} = \int_{\partial\Omega} \Psi'(u)w\,v\,\mathrm{d}S\,.$$

= a bilinear form in v, w on $H^1(\Omega) \times H^1(\Omega)$!

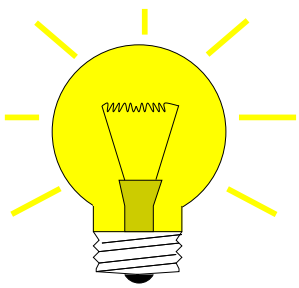
This example also demonstrates how to actually compute $D_u\mathsf{a}(u^{(k)};v)w$ needed in (3.8.4)

The manipulations above rely on techniques already addressed in Ex. 1.3.16 on page 1.3.16.

3.8.3 Galerkin discretization of linearized variational problem

We have found the following variational problem for the computation of the Newton update ("in function space")

$$\begin{aligned} w \in V_0: \quad & \mathsf{a}(u^{(k)};v) + D_u\mathsf{a}(u^{(k)};v)w = \ell(v) \quad \forall v \in V_0\,, \\ & u^{(k+1)} := u^{(k)} + w\,. \end{aligned} \tag{3.8.4}$$



Idea: (3.8.4) is a linear variational problem ($\to$ Def. 1.4.8)!

$$\begin{aligned} & w \in V_0: \quad \mathsf{c}(w,v) = g(v) \quad \forall v \in V_0\,, \\ & \mathsf{c}(w,v) = D_u\mathsf{a}(u^{(k)};v)w\,, \quad g(v) := \ell(v) - \mathsf{a}(u^{(k)};v)\,. \end{aligned}$$

$\blacktriangleright$ Tackle it by means of Galerkin discretization!

$\blacktriangleright$ Newton-Galerkin iteration for (1.3.24)

Given $u_N^{(k)} \in V_N^{(k)}$ $\succ$ $u_N^{(k+1)} \in V_N^{(k+1)}$ from

$$\boxed{\begin{aligned} w_N \in V_{0,N}^{(k+1)}: \quad & D_u\mathsf{a}(u_N^{(k)};v_N)w_N = \ell(v_N) - \mathsf{a}(u_N^{(k)};v_N) \quad \forall v_N \in V_{0,N}^{(k+1)}\,, \\ & u_N^{(k+1)} := \mathsf{P}_N^{(k+1)} u_N^{(k)} + w_N\,. \end{aligned}} \tag{3.8.10}$$

Newton update

Note: different Galerkin trial/test spaces $V_N^{(k)}$, $V_{0,N}^{(k)}$ may be used in different steps of the iteration!

(It may enhance efficiency to use Galerkin trial/test spaces of a rather small dimension in the beginning and switch to larger when the iteration is about to converge.)

Warning! If $V_N^{(k)} \neq V_N^{(k+1)}$ you cannot simply add $u_N^{(k)}$ and w

➢ Linear projection operator $\mathsf{P}_N^{(k+1)} : V_N^{(k)} \mapsto V_N^{(k+1)}$ required in (3.8.10)

Any of the Lagrangian finite element spaces introduced in Section 3.5 will supply valid $V_N/V_{0,N}$. Offset functions can be chosen according to the recipes from Section 3.6.6.

Important aspect: termination of iteration, see [14, Section 2.4.3].

Option: termination based on relative size of Newton update, with w, $u_N^{(k+1)}$ from (3.8.10)

$$\textbf{STOP}, \text{ if } \quad \|w\| \leq \tau \left\| u_N^{(k+1)} \right\| , \tag{3.8.11}$$

where $\|\cdot\|$ is a relevant norm (e.g., energy norm) on $V_N^{(k+1)}$ and $\tau > 0$ a prescribed relative tolerance.

?! *Review question(s) 3.8.12.* (Finite element discretization of non-linear variational problems)

1. Why is numerical quadrature indispensable for a general purpose finite element code?
2. The unit square $[0,1]^2$ can be mapped onto a general non-degenerate quadrilateral K by means of a mapping $\mathbf{\Phi}_K$ composed of (i) an affine mapping $\mathbf{\Phi}_K^{\mathtt{aff}}$ ($\rightarrow$ Def. 3.6.136) and (ii) a mapping $\mathbb{R}^2 \rightarrow \mathbb{R}^2$, $\widehat{\boldsymbol{x}} \mapsto \boldsymbol{d}\widehat{x}_1\widehat{x}_2$. Find formulas for both $\mathbf{\Phi}_K^{\mathtt{aff}}$ and $\boldsymbol{d} \in \mathbb{R}^2$ in terms of the vertex coordinates $\boldsymbol{a}^1, \dots, \boldsymbol{a}^4$ of K.

Learning outcomes

Skills to be acquired in Chapter 3:

- Familiarity with all aspects of abstract Galerkin discretization of a linear variational problem.
- Knowledge of the role of the main ingredients for a finite element Galerkin discretization: variational problem, mesh, global and local shape functions.
- Understanding of properties of finite element Galerkin matrices, in particular, their sparsity patterns.
- Ability to implement the (approximate, by means of quadrature) computation of element matrices and element right hand side vectors for Lagrangian finite elements and rather general 2nd-order elliptic boundary value problems
- Grasp of rationale and realization of *local assembly* of finite element Galerkin matrices and right hand side vectors.
- Use of LehrFEM finite element MATLAB library to implement a finite element simulation code for a given 2nd-order elliptic boundary value problem.
- Ability to deal with non-zero essential boundary conditions in a finite element context.

  vectors, and dealing with more general shapes of cells.

Bibliography

[1] J. Alberty, C.Carstensen, and A. Funken. Remarks around 50 lines of Matlab: short finite element implementation. *Numerical Algorithms*, 20:117–137, 1999.

[2] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. I. Abstract framework. *Computing*, 82(2-3):103–119, 2008.

[3] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. *Computing*, 82(2-3):121–138, 2008.

[4] Daniele Boffi, Franco Brezzi, and Michel Fortin. *Mixed finite element methods and applications*, volume 44 of *Springer Series in Computational Mathematics*. Springer, Heidelberg, 2013.

[5] S. Brenner and R. Scott. *Mathematical theory of finite element methods*. Texts in Applied Mathematics. Springer–Verlag, New York, 2nd edition, 2002.

[6] A. Burtscher, E. Fonn, P. Meury, and C. Wiesmayr. *LehrFEM - A 2D Finite Element Toolbox*. SAM, ETH Zürich, Zürich, Switzerland, 2010. http://www.sam.math.ethz.ch/ hiptmair/tmp/LehrFEMManual.pdf.

[7] Z.-X. Chen. *The Finite Element Method*. World Scientific, Singapore, 2011.

[8] A. Dedner, R. Klöfkorn, and M. Nolte. The dune-alugrid module. Preprint arXiv:1407.6954v2 [cs.MS], arXiv, 2014.

[9] Andreas Dedner, Robert Klöfkorn, Martin Nolte, and Mario Ohlberger. A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module. *Computing*, 90(3-4):165–196, 2010.

[10] D.A. Dunavant. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *Int. J. Numer. Meth. Engr.*, 21:1129–1148, 1985.

[11] Stefan Funken, Dirk Praetorius, and Philipp Wissgott. Efficient implementation of adaptive P1-FEM in Matlab. *Comput. Methods Appl. Math.*, 11(4):460–490, 2011.

[12] J.R. Gilbert, C.Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[13] R. Hiptmair. Finite elements in computational electromagnetism. *Acta Numerica*, 11:237–339, 2002.

[14] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[15] P. Knabner and L. Angermann. *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*, volume 44 of *Texts in Applied Mathematics*. Springer, Heidelberg, 2003.

[16] J. N. Lyness and Ronald Cools. A survey of numerical cubature over triangles. In *Mathematics of Computation 1943–1993: a half-century of computational mathematics (Vancouver, BC, 1993)*, volume 48 of *Proc. Sympos. Appl. Math.*, pages 127–150. Amer. Math. Soc., Providence, RI, 1994.

[17] F. Sayas. A gentle introduction to the finite element method. Online Lecture notes, www.math.umn.edu/~sayas002/anIntro2FEM.pdf, 2008.

[18] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

Chapter 4

Finite Differences (FD) and Finite Volume Methods (FV)

Now we examine two approaches to the discretization of scalar linear 2nd-order elliptic BVPs that offer an alternative to finite element Galerkin methods discussed in Chapter 3.

What these methods have in common with (low degree) Lagrangian finite element methods is

- that they rely on meshes (→ Section 3.4.1) tiling the computational domain Ω,
- they lead to *sparse* linear systems of equations.

Contents

Remark 4.0.1 (Collocation approach on "complicated" domains)

Section 1.5.3.2 taught us spline collocation methods. A crucial insight was that collocation methods (see beginning of Section 1.5.3 for a presentation of the idea), which target the boundary value problem in ODE/PDE form, have to employ discrete trial spaces comprised of *continuously differentiable* functions, see Rem. 1.5.120.

It is very difficult to contruct spaces of piecewise polynomial C^1-functions on non-tensor product domains for $d = 2, 3$ and find suitable collocation nodes, *cf.* (1.5.116).

Therefore we skip the discussion of collocation methods for 2nd-order elliptic BVPs on $\Omega \subset \mathbb{R}^d$, $d = 2, 3$.

4.1 Finite differences

A finite difference scheme for a 2-point boundary value problem was presented in Section 1.5.4, which you are advised to browse again. The gist of this finite difference method was the following:

Construction of finite difference methods

Replace the derivatives in the *differential equation* with *difference quotients* connecting approximate values of the solutions *at the nodes of a grid/mesh*.

Recall: Finite differences target the "ODE/PDE-formulation" of the boundary value problem.

Our current goal: extension to higher dimensions

2D model problem:

Homogeneous Dirichlet BVP for Laplacian:

$$-\Delta u = -\frac{\partial^2 u}{\partial x_1^2} - \frac{\partial^2 u}{\partial x_2^2} = f \quad \text{in } \Omega :=]0,1[^2 \,,$$
$$u = 0 \quad \text{on } \partial\Omega \,.$$

Discretization based on
$\mathcal{M}$ = (triangular) tensor-product grid
(meshwidth $h = (1+N)^{-1}$, $N \in \mathbb{N}$)

lexikographic (line-by-line) ordering/numbering of nodes of $\mathcal{M}$

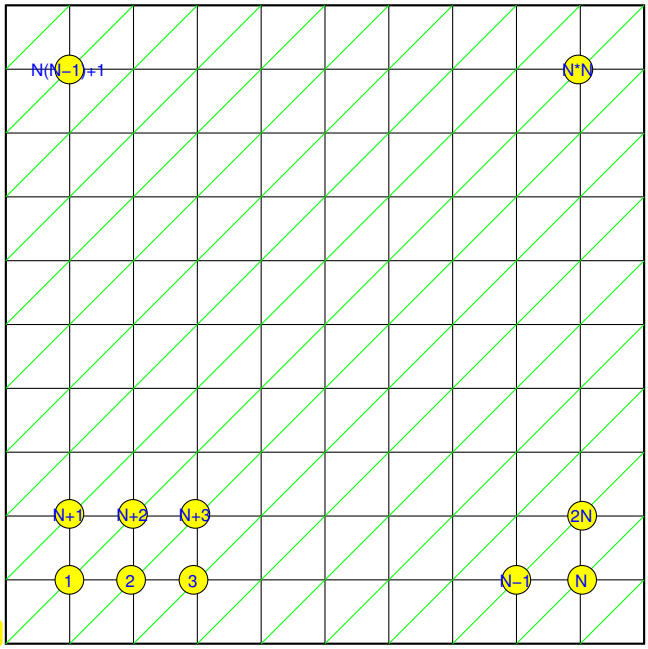


Fig. 178

4.1.1 Grid-based difference quotients

☞ First step of finite difference approach to $-\Delta$: approximation of derivatives by symmetric difference quotients.

This is nothing new: we did this in (1.5.144). The following formulas generalize the symmetric second difference quotient (1.5.138) to partial derivatives.

$$\begin{aligned} \frac{\partial^2}{\partial x_1^2} u_{|\mathbf{x}=(\xi,\eta)} &\approx \frac{u(\xi-h,\eta) - 2u(\xi,\eta) + u(\xi+h,\eta)}{h^2} \,, \\ \frac{\partial^2}{\partial x_2^2} u_{|\mathbf{x}=(\xi,\eta)} &\approx \frac{u(\xi,\eta-h) - 2u(\xi,\eta) + u(\xi,\eta+h)}{h^2} \,. \end{aligned} \tag{4.1.2}$$

$$-\Delta u_{|\mathbf{x}=(\xi,\eta)} \approx \frac{1}{h^2}\big(4u(\xi,\eta) - u(\xi-h,\eta) - u(\xi+h,\eta) - u(\xi,\eta-h) - u(\xi,\eta+h)\big) \,.$$

Second step: use this approximation at grid point $\boldsymbol{p} = (ih, jh)$. This will connect the five point values $u(ih,jh)$, $u((i-1)h,jh)$, $u((i+1)h,jh)$, $u(ih,(j-1)h)$, $u(ih,(j+1)h)$.

Approximations $\mu_{i,j}$ to the *point values* $u(ih, jh)$
will be the unknowns of the finite difference method.

Centering the above difference quotients at grid points yields linear relationships between the unknowns:

$$\frac{1}{h^2}\big(4u(ih,jh) - u(ih-h,jh) - u(ih+h,jh) - u(ih,jh-h) - u(ih,jh+h)\big) = f(ih,jh)\ ,$$

$$\frac{1}{h^2}\big(4\mu_{i,j} - \mu_{i-1,j} - \mu_{i+1,j} - \mu_{i,j-1} - \mu_{i,j+1}\big) = f(ih,jh)\ . \tag{4.1.3}$$

Also this is familiar from the discussion in 1D, see (1.5.143). Yet, in 1D the association of the point values and of components of the vector $\vec{\mu}$ of unknowns was straightforward and suggested by the linear ordering of the nodes of the grid. In 2D we have much more freedom.

One option on tensor-product grids is the line-by-line ordering (lexikographic ordering) depicted in Fig. 178. This allows a simple indexing scheme:

$$u(\mathbf{p}) \quad \leftrightarrow \quad \mu_{i,j} \quad \leftrightarrow \quad \mu_{(j-1)N+i}$$

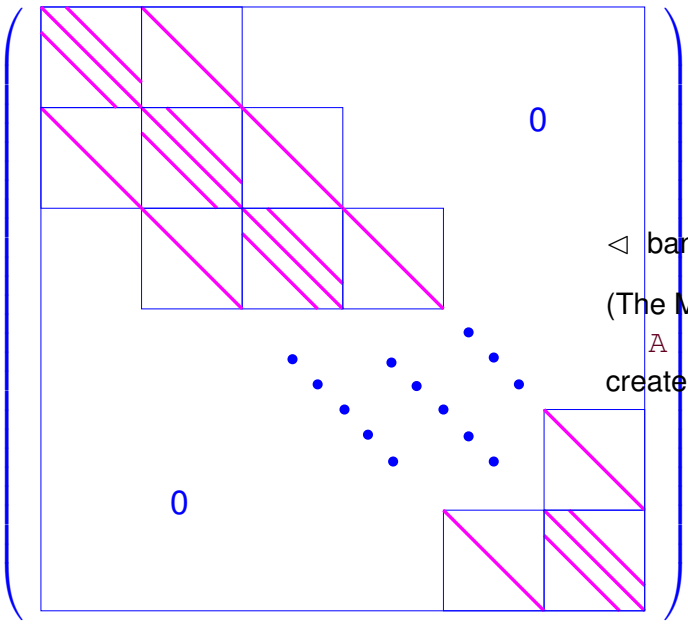
$$\Downarrow$$

$$(4.1.3) \blacktriangleright \quad \frac{-\mu_{(j-2)N+i} - \mu_{(j-1)N+i-1} + 4\mu_{(j-1)N+i} - \mu_{(j-1)N+i+1} - \mu_{jN+i}}{h^2} = \underbrace{f(ih,jh)}_{=\varphi_{(j-1)N+i}}\ . \tag{4.1.4}$$

➤ linear system of N^2 equations $\mathbf{A}\vec{\mu} = \vec{\varphi}$ with $N^2 \times N^2$ block-tridiagonal Poisson matrix

$$\mathbf{A} := \frac{1}{h^2}\begin{pmatrix} \mathbf{T} & -\mathbf{I} & 0 & \cdots & \cdots & 0 \\ -\mathbf{I} & \mathbf{T} & -\mathbf{I} & & & \vdots \\ 0 & -\mathbf{I} & \mathbf{T} & -\mathbf{I} & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & -\mathbf{I} & \mathbf{T} & -\mathbf{I} \\ 0 & \cdots & \cdots & 0 & -\mathbf{I} & \mathbf{T} \end{pmatrix}, \mathbf{T} := \begin{pmatrix} 4 & -1 & 0 & & & 0 \\ -1 & 4 & -1 & & & \vdots \\ 0 & -1 & 4 & -1 & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \\ \vdots & & & -1 & 4 & -1 \\ 0 & \cdots & \cdots & 0 & -1 & 4 \end{pmatrix} \in \mathbb{R}^{N,N} \tag{4.1.5}$$

$\mathbf{A} =$



◁ band structure of Poisson matrix

(The MATLAB command
`A = gallery('poisson',n)`
creates a sparse $n^2 \times n^2$ Poisson matrix.)

Remark 4.1.6 (Extra smoothness of source function in finite difference approach)

Obviously, we have to require $f \in C^0(\overline{\Omega})$ in order to render (4.1.3) meaningful.

► FD approach entails more regular source functions compared to finite element methods, for which $f \in L^2(\Omega)$ is enough.

(However, when numerical quadrature (→ Section 3.6.5) is used for the computation of the right hand side vector $\vec{\boldsymbol{\varphi}}$ as in Section 3.3.6, then point evaluation of f has to be possible and $f \in C^0(\overline{\Omega})$ will also be required.)

Remark 4.1.7 (Stencil notation)

One row of the linear system of equations arising from the finite difference discretization of the 2D model problem on the tensor product mesh depicted in Fig. 178:

$$\frac{1}{h^2}\left(4\mu_{i,j} - \mu_{i-1,j} - \mu_{i+1,j} - \mu_{i,j-1} - \mu_{i,j+1}\right) = f(ih, jh) \ . \qquad ((4.1.3))$$

Note: unknowns $\mu_{i,j}$ indexed by position of the grid node they are associated with.

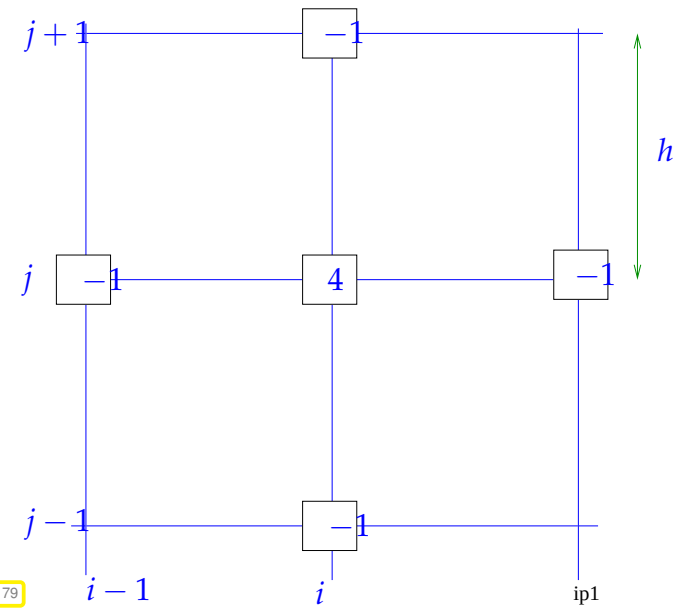


Fig. 179

← stencil anchored at $(ih, jh) \in \Omega$.

stencil notation:

$$\begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix}_h$$

A stencil as given above is a way to describe a row of a linear system of equations operating on unknowns associated with the nodes of a grid *without imposing a global numbering*.

The stencil description is particularly convenient in the case of *translation invariance*, where the stencils are the same for (almost) all nodes of the grid. Then, instead of specifying the matrix of the linear system of equations, it suffices to describe the stencil.

Stencils are not confined to tensor product grids:

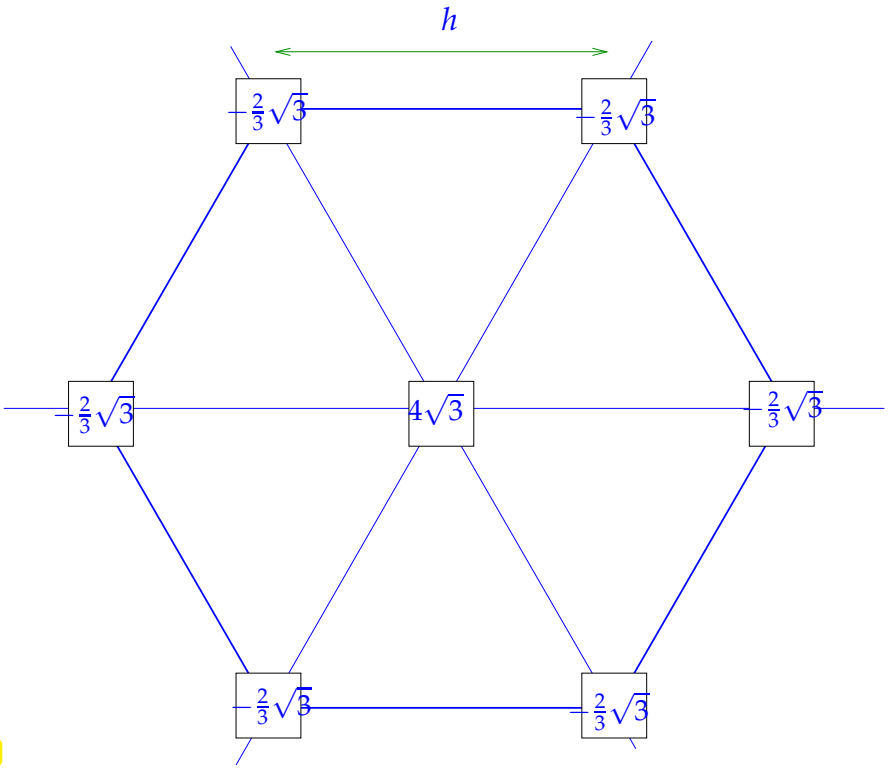


Fig. 180

← Stencil for discretization of $-\Delta$ by means of linear Lagrangian finite elements on a grid consisting of equilateral triangles.

$$\frac{2}{3}\sqrt{3} \cdot \begin{bmatrix} & -1 & & -1 & \\ -1 & & 6 & & -1 \\ & -1 & & -1 & \end{bmatrix}_h$$

4.1.2 Finite differences and finite elements

Already in Section 1.5.4 we saw that the linear system of equations popping out of the finite difference discretization of the linear two-point BVP (1.5.119) was the same as that obtained via the linear finite Galerkin approach on the same mesh.

In two dimensions we will also come to this conclusion! So, let us derive the Galerkin matrix and right hand side vect

for the 2D model problem on the tensor product mesh depicted in Fig. 178. To begin with we convert it into a triangular mesh $\mathcal{M}$ by splitting each square into two equal triangles by inserting a diagonal (green lines in Fig. 178). On this mesh we use linear Lagrangian finite elements as in Section 3.3.

Then we repeat the considerations of Section 3.3.

Linear Lagrangian finite element Galerkin discretization of 2D model problem $\rightarrow$ Section 3.3: $V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M})$ (global shape functions $\hat{=}$ "tent functions", $\rightarrow$ Fig. 92)

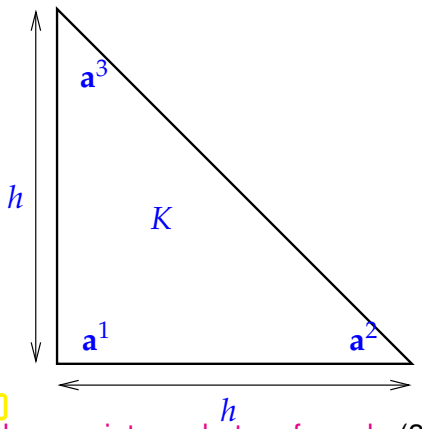


Fig. 181

Element stiffness matrix from (3.3.23):

$$\mathbf{A}_K = \frac{1}{2}\begin{pmatrix} 2 & -1 & -1 \\ -1 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} .$$

($\leftarrow$ numbering of local shape functions)

Elem

use three-point quadrature formula (3.6.159)

$$\blacktriangleright \quad \vec{\varphi}_K = \frac{1}{6}h^2 \begin{pmatrix} f(\mathbf{a}^1) \\ f(\mathbf{a}^2) \\ f(\mathbf{a}^3) \end{pmatrix} .$$

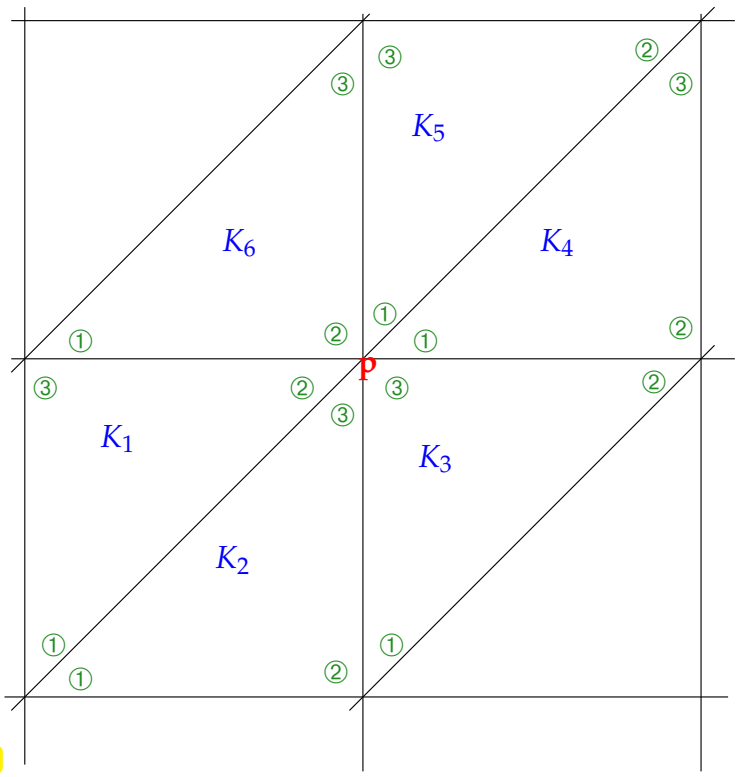


Fig. 182

Local assembly:

$\leftarrow$ green: local vertex numbers

Contributions to load vector component associated with node $\mathbf{p}$:

From K_1 : $(\vec{\varphi}_{K_1})_2$
From K_2 : $(\vec{\varphi}_{K_2})_3$
From K_3 : $(\vec{\varphi}_{K_3})_3$
From K_4 : $(\vec{\varphi}_{K_4})_1$
From K_5 : $(\vec{\varphi}_{K_5})_1$
From K_6 : $(\vec{\varphi}_{K_6})_2$

$$\blacktriangledown$$

$$\vec{\varphi}_{\mathbf{p}} = h^2 f(\mathbf{p}) .$$

Assembly of finite element Galerkin matrix from element (stiffness) matrices ($\rightarrow$ Section 3.6.4):

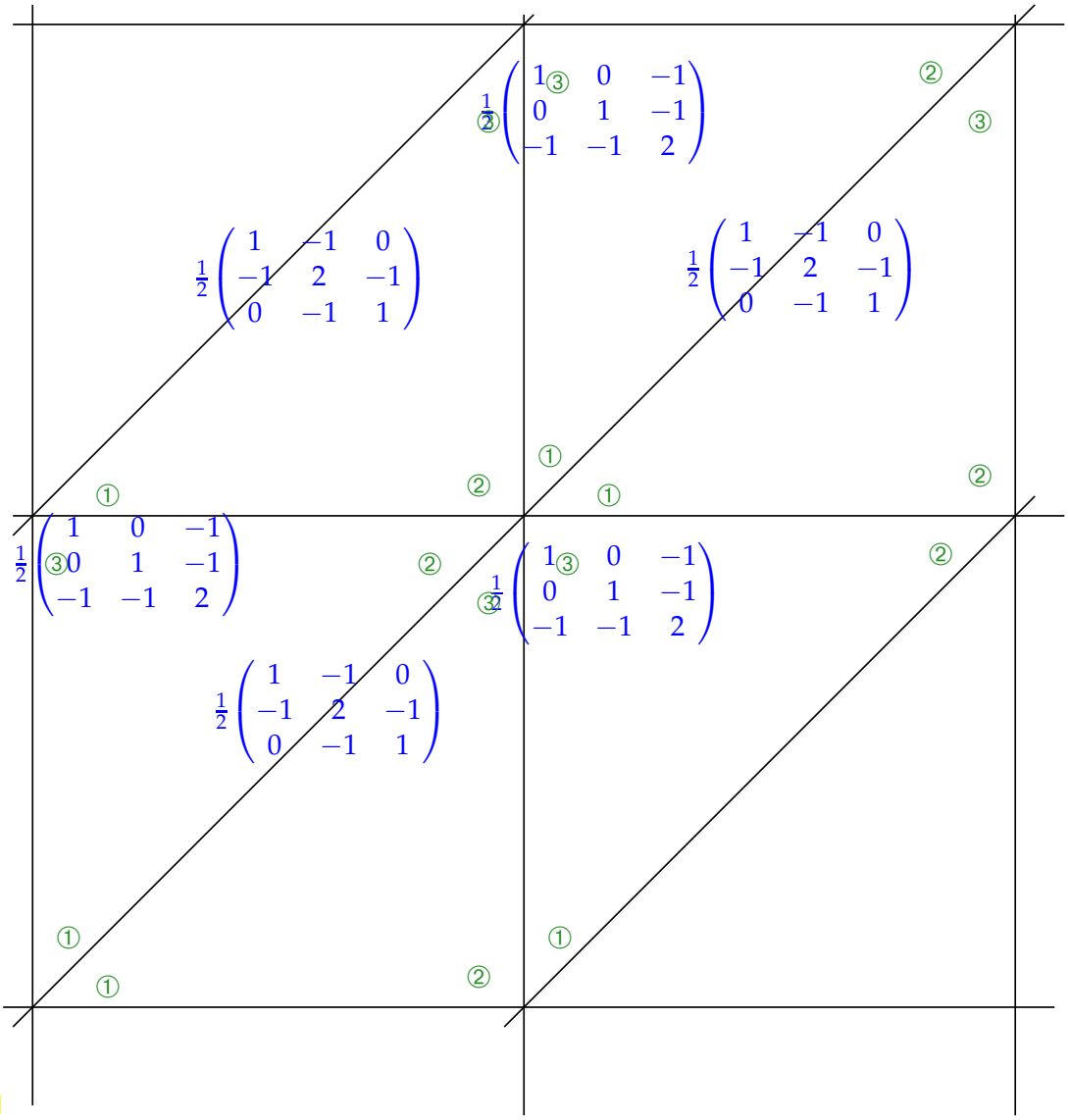

$\frac{1}{2}\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ -1 & -1 & 2 \end{pmatrix}$
$\frac{1}{2}\begin{pmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{pmatrix}$

Fig. 183

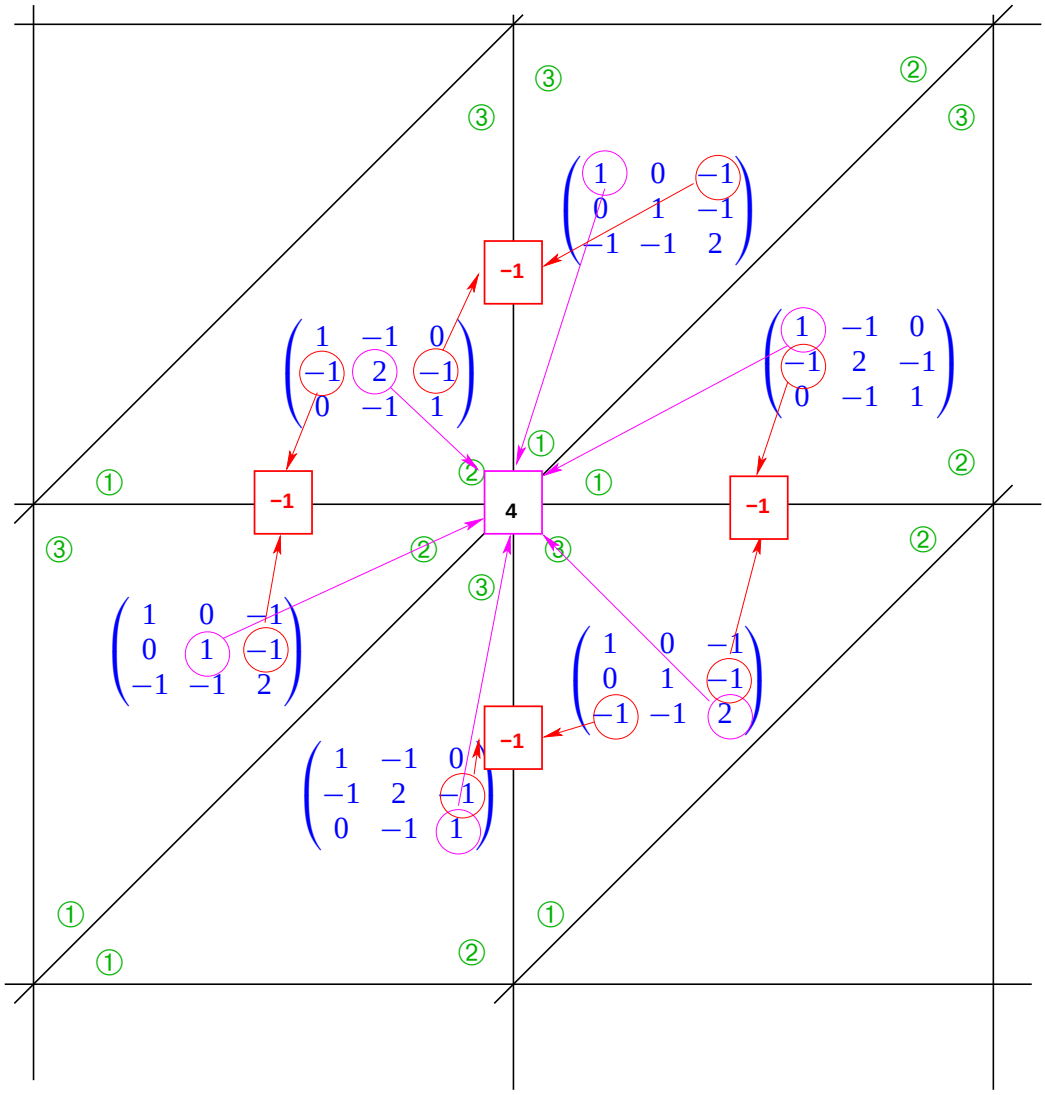


Fig. 184

➤ $N^2 \times N^2$ linear system of equations $h^2\mathbf{A}\vec{\mu} = h^2\vec{\varphi}$, $\mathbf{A} \hat{=}$ Poisson matrix (4.1.5)

(Most) finite difference schemes	$\leftrightarrow$	finite element Galerkin schemes with numerical quadrature on structured meshes

(4.1.8) Pros and cons of "finite difference approach"

Discussion: finite differences vs. finite element Galerkin methods (here focused on 2nd-order linear scalar problems)

- Finite element methods can be used on general triangulations and structured (tensor-product) meshes alike, which delivers superior flexibility in terms of geometry resolution (advantage FEM).
- The correct treatment of all kinds of boundary conditions ($\rightarrow$ Section 2.7). naturally emerges from the variational formulations in the finite element method (advantage FEM).
- Finite difference approach cannot deal with second-order elliptic boundary value problems with *discontinuous* diffusion coefficient (α in (2.4.5), (2.9.16)), which does not cause difficulties for finite element methods.
- Finite element methods have built-in "safety rails" because there are clear criteria for choosing viable

finite element spaces and once this is done, there is no freedom left to go astray (advantage FEM).

✦ Finite element methods are harder to understand (advantage FD, but only with students who have not attended this course!)

Then, why are "finite difference methods" ubiquitous in scientific and engineering simulations **?**

When people talk "finite differences" they have in mind structured meshes (translation invariant, tensor product structure) and use the term as synonym for "discretization on structured meshes". The popularity of structured meshes is justified:

- structured meshes allow regular data layout and vectorization, which boost the performance of algorithms on high performance computing hardware.

  $\rightarrow$ course "High Performance Computing"

- translation invariant PDE operators give rise to simple Galerkin matrices that need not be assembled and stored (recall the 5-point-stencil for $-\Delta$) and support very efficient matrix$\times$vector operations.

Use structured meshes whenever possible!

4.2 Finite volume methods (FVM)

4.2.1 Discrete balance laws

Focus: linear scalar 2nd-order elliptic boundary value problem in 2D ($\rightarrow$ Section 2.6), homogeneous Dirichlet boundary conditions ($\rightarrow$ Section 2.7), uniformly positive scalar heat conductivity $\kappa = \kappa(\boldsymbol{x})$

$$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad u = 0 \quad \text{on } \partial\Omega \,.$$

Finite volume methods for 2nd-order elliptic BVP are inspired by the *conservation principle* (2.6.3).

$$\int_{\partial V} \mathbf{j}\cdot\boldsymbol{n}\,\mathrm{d}S = \int_V f\,\mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes" } V \,. \tag{2.6.3}$$

Physics requires that this holds for all (infinitely many) "control volumes" $V \subset \Omega$.

Since discretization has to lead to a finite number of equations, the idea is to demand that (2.6.3) holds for only a *finite number of special control volumes.*

► First ingredient of FVM: (finitely many) control volumes

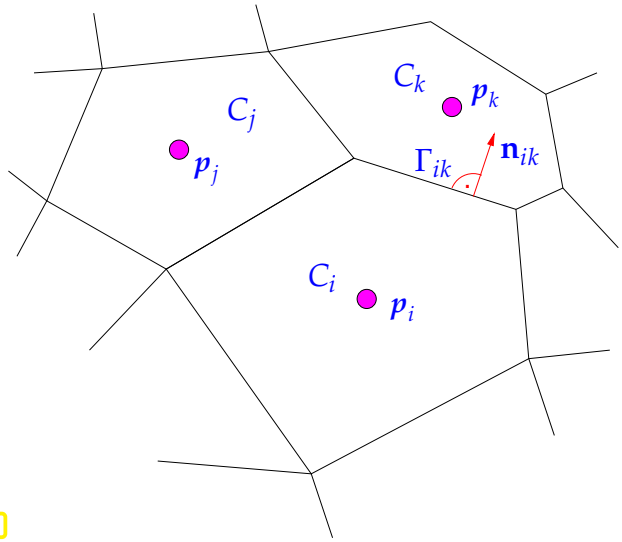


Fig. 185

Concrete choice:

Control volumes =

(polygonal) cells of a mesh $\widetilde{\mathcal{M}} = \{C_i\}_i$ covering computational domain Ω.

Associate cell C_i $\leftrightarrow$ nodal value μ_i

Meaning: $\mu_i \approx u(\boldsymbol{p}_i)$, $\boldsymbol{p}_i$ = "center" of C_i

The conservation law (2.6.3) had to be linked to the flux law (2.6.5) in order to give rise to a 2nd-order scalar PDE see (2.6.8)–(2.6.10).

Correspondingly, "heat conservation in control volumes" has to be supplemented by a rule that furnishes the heat flux between two adjacent control volumes.

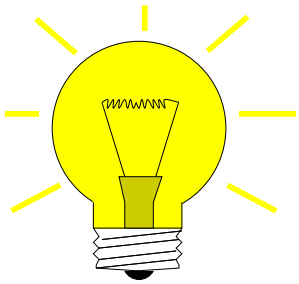
Second ingredient of FVM: local numerical fluxes

For two adjacent cells C_k, C_i with common edge $\Gamma_{ik} := \overline{C}_i \cap \overline{C}_k$.

$$\boxed{\text{Numerical flux} \quad J_{ik} = \Psi(\mu_i, \mu_k) \approx \int_{\Gamma_{ik}} \mathbf{j} \cdot \boldsymbol{n}_{ik} \, \mathrm{d}S}$$

(Ψ = numerical flux function, $\mathbf{j}$ = (heat) flux, see (2.6.2), $\boldsymbol{n}_{ik} \hat{=}$ edge normal.)

How to obtain a system of equations from combining (2.6.3) with a numerical flux?



Idea: consider balance law on (finitely many **!**) control volumes C_i

$$\int_{\partial C_i} \mathbf{j} \cdot \mathbf{n}_i \, \mathrm{d}S = \int_{C_i} f \, \mathrm{d}\boldsymbol{x} \quad \Rightarrow \quad \sum_{k \in \mathcal{U}_i} J_{ik} = \int_{C_i} f \, \mathrm{d}\boldsymbol{x} \, .$$

✎ notation: $\mathcal{U}_i := \{j : C_i \text{ and } C_j \text{ share edge}, C_j \in \widetilde{\mathcal{M}}\}$,
$\boldsymbol{p}_i$ = node associated with control volume C_i.

System of equations ($\widetilde{M} := \sharp\widetilde{\mathcal{M}}$ equations, unknowns μ_i):

$$\sum_{k \in \mathcal{U}_i} \Psi(\mu_i, \mu_k) = \int_{C_i} f \, \mathrm{d}\boldsymbol{x} \quad \forall i = 1, \dots, \widetilde{M} \, . \tag{4.2.1}$$

Further approximation: 1-point quadrature for approximate evaluation of integral over C_i,

$$\sum_{k \in \mathcal{U}_i} \Psi(\mu_i, \mu_k) = |C_i| \, f(\boldsymbol{p}_i) \quad \forall i = 1, \dots, \widetilde{M} \, . \tag{4.2.2}$$

Note: homogeneous Dirichlet problem ➢ only "interior" control volumes in (4.2.1)

4.2.2 Dual meshes

Dual meshes are a commonly used technique for the construction of control volumes for FVM, based on conventional FE triangulation $\mathcal{M}$ of Ω ($\rightarrow$ Section 3.4.1).

Focus: dual mesh for triangular mesh $\mathcal{M}$ in 2D, Ω polygon (this triangular mesh is often called the primal mesh)

(4.2.3) Voronoi dual meshes

Popular choice: Voronoi dual mesh

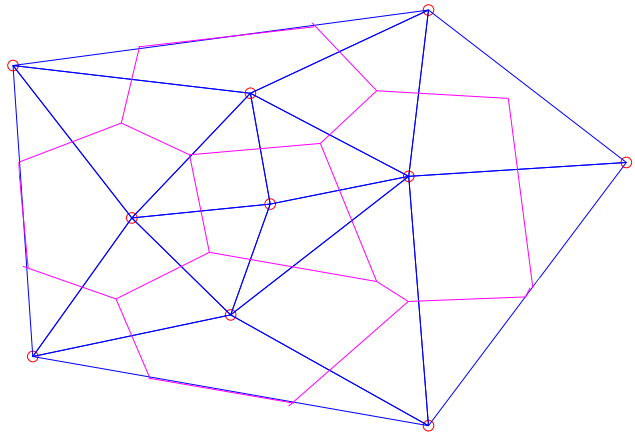


Fig. 186

Write $\mathcal{V}(\mathcal{M}) = \{\boldsymbol{p}_1, \dots, \boldsymbol{p}_M\}$ = nodes of $\mathcal{M}$.

Define Voronoi cells

$$C_i := \{\mathbf{x} \in \Omega \colon |\mathbf{x} - \boldsymbol{p}_i| < |\mathbf{x} - \boldsymbol{p}_j| \ \forall j \neq i\} \ . \tag{4.2.4}$$

▶ Voronoi dual mesh $\widetilde{\mathcal{M}} := \{C_i\}_{i=1}^{M}$

Construction of Voronoi dual cells:

edges $\rightarrow$ perpendicular bisectors
nodes $\rightarrow$ circumcenters of triangles

▶ straightforward generalization to 3D

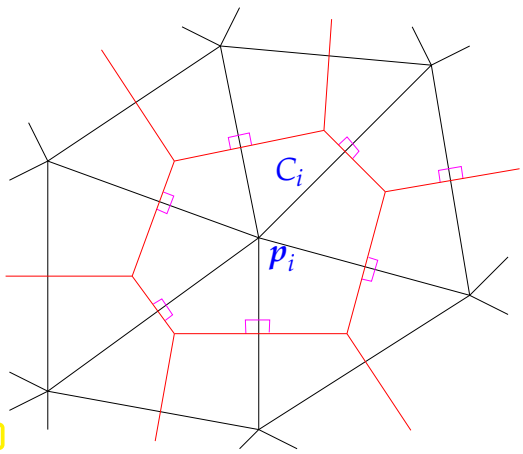


Fig. 187

Remark 4.2.5 (Geometric obstruction to Voronoi dual meshes)

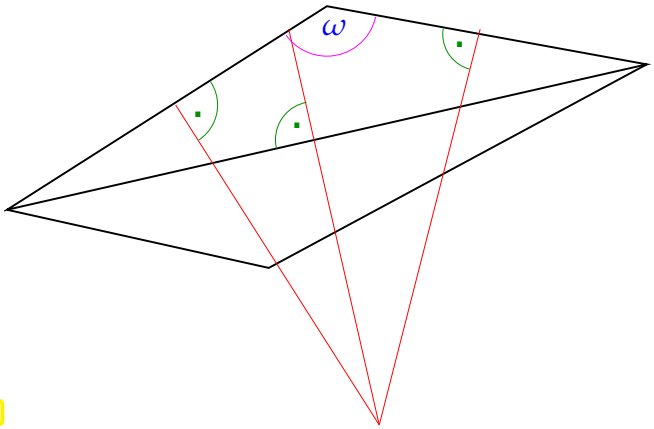


Fig. 188

$\Leftarrow$ Obtuse angle ω:

- circumcenter $\notin$ triangle
- $\overline{C}_i \cap \overline{C}_j \neq \emptyset \quad \not\Rightarrow$ nodes i, j connected by edge
- geometric construction breaks down
- connectivity of unknowns hard to determine

Theorem 4.2.6. Angle condition for Voronoi dual meshes

The following Angle condition ensures that the Voronoi cells belonging to adjacent nodes of a triangular mesh have a common edge ($\overline{C}_i \cap \overline{C}_j \neq \emptyset \quad \Leftrightarrow \quad$ nodes i, j connected by edge of $\mathcal{M}$):

(i) sum of angles facing interior edge $\leq \pi$,

(ii) angles facing boundary edges $\leq \pi/2$.

Note: Condition (ii) important only for FV methods with unknowns attached to boundary vertices, that is, in the case of non-Dirichlet boundary conditions, *cf.* Rem. 4.2.11.

Definition 4.2.7. Delaunay triangulation

Triangular meshes satisfying the angle conditions (i) and (ii) from Thm. 4.2.6 are called Delaunay triangulations.

(4.2.8) Barycentric dual meshes

Another popular choice: Barycentric dual mesh

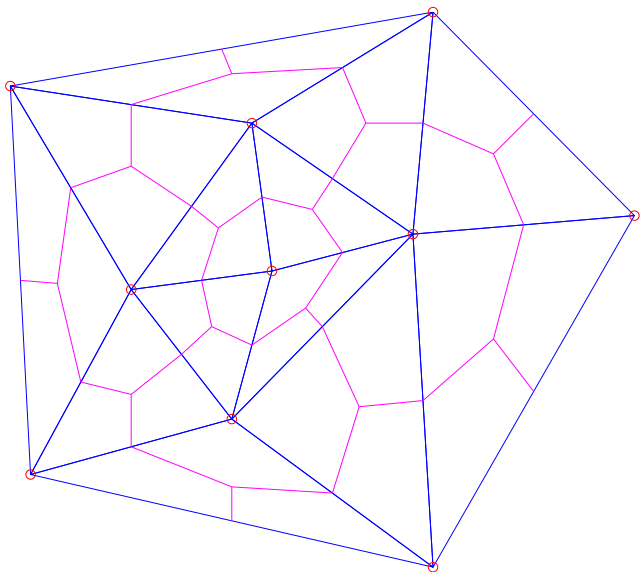


Fig. 189

Dual cells:

edges $\rightarrow$ union of lines connecting barycenters and midpoints of edges of $\mathcal{M}$

nodes $\rightarrow$ barycenters of triangles

$\blacktriangleright$ No geometric obstructions

Definition 4.2.9. Barycenter of a triangle

The barycenter of a triangle with vertices $\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3 \in \mathbb{R}^d$ is the point $\boldsymbol{p} := \frac{1}{3}(\boldsymbol{a}^1 + \boldsymbol{a}^2 + \boldsymbol{a}^3)$.

(4.2.10) "Centers" of dual control volumes

For both the Voronoi construction and barycentric dual meshes: natural choice for "centers" $\boldsymbol{p}_i$ of control volumes = vertices of triangular (primal) mesh.

Remark 4.2.11 (FV: Incorporation of homogeneous Dirichlet boundary conditions)

Assume choice of "centers" of control volumes according to § 4.2.10

Homogeneous Dirichlet boundary conditions ($\rightarrow$ Section 2.7) $u = 0$ on $\partial\Omega$ are taken into account by

- considering only control volumes (dual cells) located in the interior of Ω in (4.2.1),
- setting $\mu_k = 0$ for neighboring control volumes (dual cells) that abut $\partial\Omega$. This makes sense, because the centers of those control volumes will be located on $\partial\Omega$, where u is supposed to vanish.

Remark 4.2.12 (Treatment of Neumann boundary conditions in finite volume schemes)

Consider finite volume method based on dual meshes for 2nd-order elliptic Neumann problem:

$$-\operatorname{div}(\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\boldsymbol{n} = h \quad \text{on } \partial\Omega\,. \tag{2.10.2}$$

The value of u on $\partial\Omega$ is unknown.
➢ keep balance equations for control volumes associated with (primal) vertices on the boundary $\partial\Omega$.

Remember (2.7.3): $h = -\mathbf{j}\cdot\boldsymbol{n}$, that is, the Neumann data $h \in L^2(\partial\Omega)$ already provide the flux!

Taking for granted a numerical flux function Ψ, we find the following modified instance of (4.2.1) for a control volume C_i adjacent to $\partial\Omega$:

$$\sum_{k\in\mathcal{U}_i} \Psi(\mu_i,\mu_k) - \int_{\partial C_i\cap\partial\Omega} h\,\mathrm{d}S = |C_i| f(\boldsymbol{p}_i)\,.$$

4.2.3 Relationship of finite element and finite volume methods

Hardly surprising, finite volume methods and finite element Galerkin discretizations are closely related. This will be explored in this section for a model problem.

Setting:

- We consider the homogeneous Dirichlet problem for the Laplacian Δ

$$-\Delta u = f \quad \text{in } \Omega \quad , \quad u = 0 \quad \text{on } \partial\Omega\,. \tag{4.2.13}$$

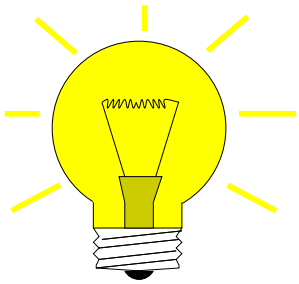
- Discretization by finite volume method based on a triangular mesh $\mathcal{M}$ and on Voronoi dual cells $\rightarrow$ Fig. 186:

  Assumption: $\mathcal{M}$ = Delaunay triangulation of Ω $\Leftrightarrow$ angle condition

  Number of control volumes = number of interior nodes of $\mathcal{M}$

(4.2.14) Numerical flux by interpolation

Still missing: specification of numerical flux function $\Psi : \mathbb{R}^2 \mapsto \mathbb{R}$ for each dual edge



Idea: obtain numerical flux from
Fourier's law (2.6.5) applied to a (sufficiently smooth) $u_N : \Omega \mapsto \mathbb{R}$
reconstructed from dual cell values μ_i.

Natural approach, since μ_i is read as approximation of $u(\boldsymbol{p}_i)$, where the "center" $\boldsymbol{p}^i$ of the dual cell C_i coincides with an *interior node* $\boldsymbol{x}^i \in \mathcal{V}(\mathcal{M})$ of the triangular mesh $\mathcal{M}$:

$$u_N = \mathsf{I}_1 \vec{\mu} := \sum_{i=1}^{N} \mu_i b_N^i \,, \tag{4.2.15}$$

where $N = \sharp\mathcal{V}(\mathcal{M})$ = number of dual cells, size of vector $\vec{\mu}$,
$b_N^i \hat{=}$ nodal basis function ("tent function") of $\mathcal{S}_{1,0}^0(\mathcal{M})$ belonging to the node inside C_i.

$u_N \hat{=}$ piecewise linear interpolant of vertex values μ_i

Note that u_N is not smooth across inner edges of $\mathcal{M}$. However, we do not care when computing $\mathbf{j} := \kappa(\boldsymbol{x})\,\mathbf{grad}\, u_N$, because this flux is *only needed at edges of the dual mesh*, which lie inside triangles of $\mathcal{M}$ (with the exception of single points that are irrelevant for the flux integrals).

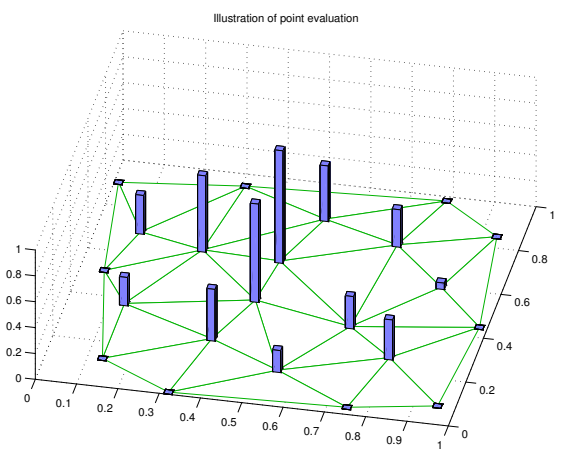


Fig. 190

vertex values μ_i on $\mathcal{V}(\mathcal{M})$

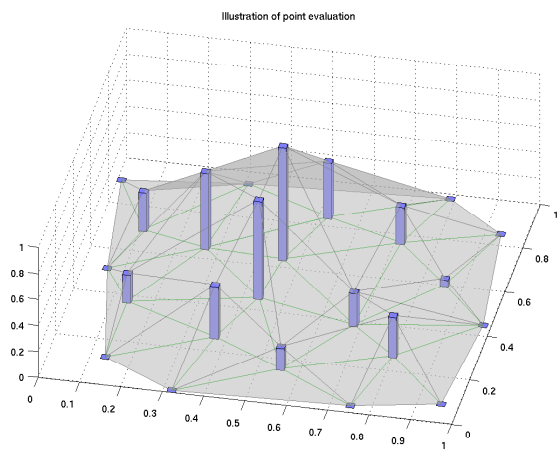


Fig. 191

p.w. linear interpolant $u_N := \mathsf{I}_1 \vec{\mu} \in \mathcal{S}_{1,0}^0(\mathcal{M})$

Choice of numerical flux: $$J_{ik} := -\int_{\Gamma_{ik}} \mathbf{grad}\, \mathsf{I}_1 \vec{\mu} \cdot \mathbf{n}_{ik} \,\mathrm{d}S \tag{4.2.16}$$

Using the numerical flux (4.2.16) in (4.2.1) ➢ one row of finite volume discretization matrix from the equations

$$\sum_{k\in\mathcal{U}_i}\int_{\Gamma_{ik}} \mathbf{grad}\, \mathsf{l}_1\vec{\mu}\cdot\mathbf{n}_{ik}\,\mathrm{d}S = \mu_i \underbrace{\int_{\partial C_i}\mathbf{grad}\, b_N^i\,\mathrm{d}S}_{=\text{ matrix entry } -(\mathbf{A})_{ii}} + \sum_{j\in\mathcal{U}_i}\mu_j \underbrace{\left(\sum_{k\in\mathcal{U}_i}\int_{\Gamma_{ik}}\mathbf{grad}\, b_N^j\cdot\mathbf{n}_{ik}\,\mathrm{d}S\right)}_{=\text{ matrix entry } -(\mathbf{A})_{ij}}$$

$$= -\int_{\mathcal{C}_i} f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

$$\Rightarrow \qquad (\mathbf{A})_{ij} = -\int_{\partial\mathcal{C}_i}\mathbf{grad}\, b_N^j\cdot\mathbf{n}_i\,\mathrm{d}S\,, \quad i,j\in\{1,\dots,N\}\,. \tag{4.2.17}$$

where $\boldsymbol{n}_i \mathrel{\hat{=}}$ exterior unit normal vector to ∂C_i.

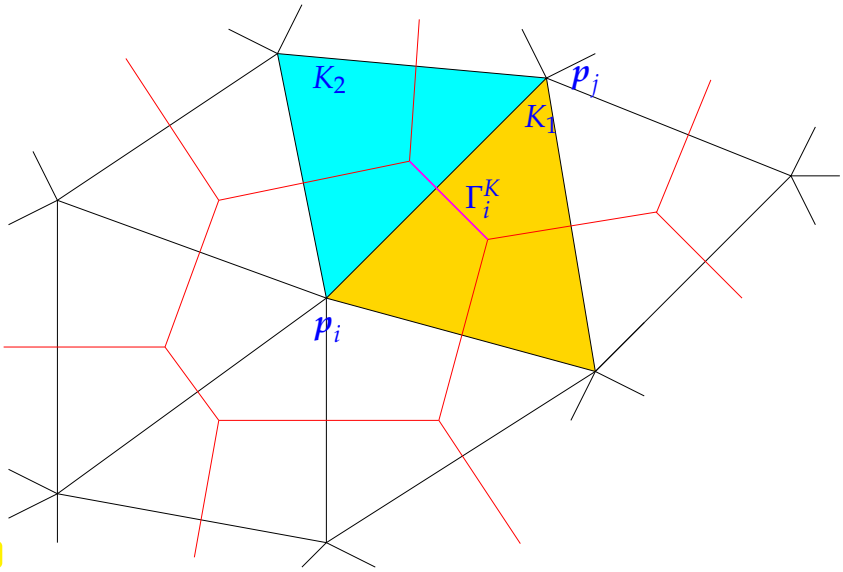


Fig. 192

Notations used in the formulas below:

$\boldsymbol{p}_i$, $\boldsymbol{p}_j \mathrel{\hat{=}}$ vertices of primal mesh ("location of unknowns μ_i, μ_j)

$K_1, K_2 \mathrel{\hat{=}}$ triangles adjacent to edge connecting $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$

Part of the boundary of the control volume C_i:

$$\Gamma_i^K := \partial\mathcal{C}_i \cap K\,.$$

Now, consider $i \neq j \leftrightarrow$ off-diagonal entries of $\mathbf{A}$:

First, we recall that the intersection of the support of the "tent function" b_N^j with ∂C_i is located inside $K_1 \cup K_2$, see Fig. 192.

$$\blacktriangleright \quad (\mathbf{A})_{ij} = -\int_{\Gamma_i^{K_1}}\mathbf{grad}\, b_N^j\cdot\mathbf{n}_i\,\mathrm{d}S - \int_{\Gamma_i^{K_2}}\mathbf{grad}\, b_N^j\cdot\mathbf{n}_i\,\mathrm{d}S\,.$$

$\leftrightarrow$ assembly of $(\mathbf{A})_{ij}$ from contributions of the two cells K_1 and K_2, *cf.* Section 3.3.5, page 3.3.5.

Next observe that $\mathbf{grad}\, b_N^j$ is piecewise constant, which implies

$$\operatorname{div}\mathbf{grad}\, b_N^j = 0 \quad \text{in } K_1 \quad , \quad \operatorname{div}\mathbf{grad}\, b_N^j = 0 \quad \text{in } K_2\,. \tag{4.2.18}$$

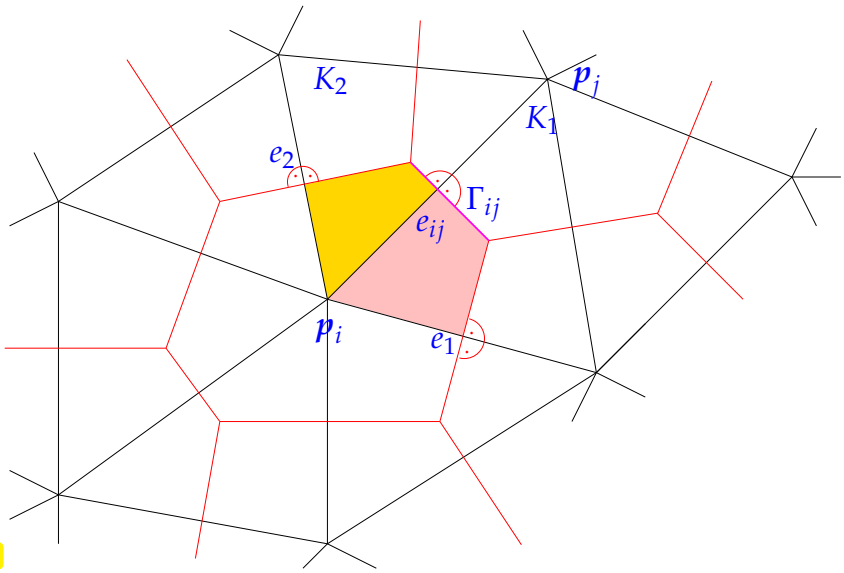


Fig. 193

Now apply Gauss' theorem Thm. 2.5.7 to the domains $C_i \cap K_1$ and $C_i \cap K_2$ (shaded in figure).

Also use again that $\mathbf{grad}\, b_N^j \equiv \text{const}$ on K_1 and K_2.
Another important observation; conclusion from $\mathbf{grad}\,\lambda_i$-formula from Section 3.3.5:

$$\mathbf{grad}\, b_N^j \perp e_1 \quad \text{in } K_1 \,,$$
$$\mathbf{grad}\, b_N^j \perp e_2 \quad \text{in } K_2 \,.$$

$$\text{(4.2.18)} \quad \Rightarrow \quad \int_{\partial(K_i \cap C_i)} \mathbf{grad}\, b_N^j \cdot \boldsymbol{n}\, \mathrm{d}S = 0 \,,$$

$$(\mathbf{A})_{ij} = \frac{1}{2}\int_{e_1} \mathbf{grad}\, b^j_{N\,|K_1} \cdot \boldsymbol{n}_{e_1}\, \mathrm{d}S + \frac{1}{2}\int_{e_{ij}} \mathbf{grad}\, b^j_{N\,|K_1} \cdot \boldsymbol{n}^1_{e_{ij}}\, \mathrm{d}S$$
$$+ \frac{1}{2}\int_{e_{ij}} \mathbf{grad}\, b^j_{N\,|K_2} \cdot \boldsymbol{n}^2_{e_{ij}}\, \mathrm{d}S + \frac{1}{2}\int_{e_2} \mathbf{grad}\, b^j_{N\,|K_1} \cdot \boldsymbol{n}_{e_2}\, \mathrm{d}S \,. \quad \text{(4.2.19)}$$

On the other hand, an entry of finite element Galerkin matrix $\widetilde{\mathbf{A}}$ based on linear Lagrangian finite element space $\mathcal{S}_1^0(\mathcal{M})$ can be computed as, see Section 3.3.5:

$$(\widetilde{\mathbf{A}})_{ij} = \int_{K_1} \mathbf{grad}\, b_N^j \cdot \mathbf{grad}\, b_N^i\, \mathrm{d}\boldsymbol{x} + \int_{K_2} \mathbf{grad}\, b_N^j \cdot \mathbf{grad}\, b_N^i\, \mathrm{d}\boldsymbol{x} \,.$$

Conduct local integration by parts using Green's first formula from Thm. 2.5.9 and taking into account (4.2.18) and the linearity of the local shape functions

$$(\widetilde{\mathbf{A}})_{ij} = \int_{\partial K_1} (\mathbf{grad}\, b^j_{N\,|K_1} \cdot \boldsymbol{n}_1) b^i_N\, \mathrm{d}S + \int_{\partial K_2} (\mathbf{grad}\, b^j_{N\,|K_2} \cdot \boldsymbol{n}_2) b^i_N\, \mathrm{d}S$$
$$= \tfrac{1}{2}|e_1|\, \mathbf{grad}\, b^j_{N\,|K_1} \cdot \boldsymbol{n}_{e_1} + \tfrac{1}{2}|e_{ij}|\, \mathbf{grad}\, b^j_{N\,|K_1} \cdot \boldsymbol{n}^1_{e_{ij}} +$$
$$\tfrac{1}{2}|e_2|\, \mathbf{grad}\, b^j_{N\,|K_2} \cdot \boldsymbol{n}_{e_2} + \tfrac{1}{2}|e_{ij}|\, \mathbf{grad}\, b^j_{N\,|K_2} \cdot \boldsymbol{n}^2_{e_{ij}} \,.$$

This is the same value as for $(\mathbf{A})_{ij}$ from (4.2.19)! Similar considerations apply to the diagonal entries $(\mathbf{A})_{ii}$ and $(\widetilde{\mathbf{A}})_{ii}$.

The finite volume discretization and the finite element Galerkin discretization spawn the same system matrix for the model problem (4.2.13).

Learning outcomes

The chapter aims to impart

- the gist of the "finite difference approach": starting from strong form of a partial differential equation replace derivatives by difference quotients anchored on a regular grid (finite lattice).
- awareness that finite difference schemes can usually be recovered as finite element discretization (plus quadrature) on special (regular) meshes.
- the principles of the finite volume discretization of 2nd-order elliptic boundary value problems.
- the idea of using dual meshes as a tool to construct control volumes for a finite volume discretization.

Bibliography

[1] Patrick Mullen, Pooran Memari, Fernando de Goes, and Mathieu Desbrun. Hot: Hodge-optimized triangulations. *ACM Trans. Graph.*, 30(4):103:1–103:12, July 2011.

Chapter 5

Convergence and Accuracy

In this chapter we resume the discussion of Section 1.6 of accuracy of a Galerkin solution u_N of a variational boundary value problem. More precisely, we are going to study the *asymptotic convergence* of relevant norms $\|u - u_N\|$ of the discretization error $u - u_N$ ($u \mathrel{\hat{=}}$ exact solution) as we let the dimension of the discrete trial space tend to ∞, see Rem. 1.6.2.

Focus: (as in all previous chapters) finite element Galerkin discretization of *linear* scalar 2nd-order elliptic boundary value problems in 2D, 3D

Contents

(5.0.1) Prerequisite knowledge

Familiarity with the following concepts is essential for understanding the material in this chapter:

- Boundary value problems (from equilibrium models, diffusion models): Section 2.5, Section 2.7,
- Variational formulation: Section 2.9, see also (2.4.5), (2.9.16), (3.1.4),
- Some Sobolev spaces and their norms: Section 2.3
- Abstract Galerkin discretization: Section 3.2,
- Lagrangian finite elements: Section 3.5, Section 3.3.

5.1 Galerkin Error Estimates

(5.1.1) Linear variational problems revisited → **§ 1.4.7, Section 2.4.1**

Abstract setting for this section: linear variational problem (1.4.9) in the form

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{3.2.3}$$

- $V_0 \hat{=}$ (real) vector space, a space of functions $\Omega \mapsto \mathbb{R}$ for scalar 2nd-order elliptic variational problems,
- $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R} \hat{=}$ a bilinear form, see Def. 1.3.22,
- $\ell : V_0 \mapsto \mathbb{R} \hat{=}$ a linear form, see Def. 1.3.22,

☞ We want (3.2.3) to be related to a quadratic minimization problem (→ Def. 2.2.32):

Assumption 5.1.2.

The bilinear form $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R}$ in (3.2.3) is symmetric and positive definite (→ Def. 2.2.40).

▶ a supplies an inner product on V_0

▶ a induces energy norm $\|\cdot\|_{\mathsf{a}}$ on V_0 (→ Def. 2.2.43)

☞ We want (3.2.3) to be well posed, remember Def. 2.4.13 and the discussion in Section 2.4.2. Thm. 2.4.21 motivates the next assumption.

Assumption 5.1.3.

The right hand side functional $\ell : V_0 \mapsto \mathbb{R}$ from (3.2.3) is continuous w.r.t. to the energy norm (→ Def. 2.2.43) induced by a:

$$\exists C > 0: \quad |\ell(u)| \leq C\|u\|_{\mathsf{a}} \quad \forall u \in V_0 \,. \tag{2.2.55}$$

☞ An assumption to appease fastidious mathematicians, see § 2.4.18 for further discussion:

Assumption 5.1.4.

V_0 equipped with the energy norm $\|\cdot\|_{\mathsf{a}}$ is a Hilbert space, that is, complete (→ Def. 2.3.10).

Theorem 5.1.5. Existence and uniqueness of solution of linear variational problem

Under Ass. 5.1.2–Ass. 5.1.4 the linear variational problem has a unique solution $u \in V_0$.

This repeats Cor. 2.4.20 from § 2.4.18 and is also known as Riesz representation theorem for continuous linear functionals.

Remark 5.1.6 (Well-posed 2nd-order linear elliptic variational problems)

For instance, thanks to the Poincaré-Friedrichs inequality from Thm. 2.3.31, Ass. 5.1.2 is satisfied for the bilinear form of a second-order linear elliptic (pure) Dirichlet problem, see (2.5.16), (2.4.5), with

$$\mathsf{a}(u,v) := \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u)\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}\,, \quad u,v\in H_0^1(\Omega)\,, \tag{5.1.7}$$

and uniformly positive definite ($\rightarrow$ Def. 2.2.18) coefficient tensor $\boldsymbol{\alpha}:\Omega\mapsto\mathbb{R}^{d,d}$, see Section 2.2.3.

A second-order linear elliptic Neumann problem involves the sam bilinear form (5.1.7), but Ass. 5.1.2 holds only on the smaller space

$$H_*^1(\Omega) := \{v\in H^1(\Omega)\colon \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0\}\,, \tag{2.9.15}$$

thanks to second Poincaré-Friedrichs inequality from Thm. 2.9.20.

For the right hand side functional of a 2nd-order Neumann problem, see (2.10.2), (2.9.16),

$$\ell(v) := \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega} h(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}S\,, \quad v\in H^1(\Omega)\,,$$

we found in Section 2.3, see (2.3.30), and § 2.10.7, that $f\in L^2(\Omega)$ and $h\in L^2(\partial\Omega)$ ensures Ass. 5.1.3.

Ass. 5.1.4 for a from (5.1.7) is a deep result in the theory of Sobolev spaces [3, Sect. 5.2.3, Thm. 2]. It has been stated earlier as Thm. 2.3.27.

(5.1.8) Galerkin discretization error

Now consider Galerkin discretization of (3.2.3) ($\rightarrow$ Section 3.2) based on Galerkin trial/test space

$$V_{0,N}\subset V_0 \quad , \quad N := \dim V_{0,N} < \infty\,,$$

which leads to the discrete variational problem

$$\boxed{u_N\in V_{0,N}\colon \quad \mathsf{a}(u_N,v_N) = \ell(v_N) \quad \forall v_N\in V_{0,N}}\,. \tag{3.2.8}$$

Thm. 3.2.9 guarantees existence and uniqueness of the Galerkin solution $u_N\in V_{0,N}$

Goal:	Bound *relevant norm* of discretization error $u-u_N$
Here:	Relevant norm = energy norm $\|\cdot\|_{\mathsf{a}}$

Remember why the energy norm is a "relevant norm":

➢ Bounds of $\|u - u_N\|_{\mathsf{a}}$ provide bounds for the *error in energy*, see § 1.6.8, (1.6.11)

$$|J(u) - J(u_N)| = \tfrac{1}{2}|\mathsf{a}(u,u) - \mathsf{a}(u_N,u_N)| = |\tfrac{1}{2}\mathsf{a}(u+u_N, u-u_N)| \overset{(2.2.44)}{\leq} \tfrac{1}{2}\|u-u_N\|_{\mathsf{a}} \cdot \|u+u_N\|_{\mathsf{a}} \,.$$

(No doubt, energy is a key quantity for the solution of an equilibrium problem, which is defined as the minimizer of a potential energy functional.)

Other "relevant norms" were discussed in Section 1.6.1,Section 2.3:

- the mean square norm or $L^2(\Omega)$-norm, see Def. 2.3.4,
- the supremum norm or $L^\infty(\Omega)$-norm, see Def. 1.6.5.

(5.1.9) Bounds for error of Galerkin solutions in energy norm

The Galerkin approach allows a remarkably simple bound of the energy norm of the discretization error $u - u_N$:

$$\begin{array}{rcl} \mathsf{a}(u,v) & = & \ell(v) \quad \forall v \in V_0 \,, \\ \mathsf{a}(u_N,v_N) & = & \ell(v) \quad \forall v_N \in V_{0,N} \end{array} \quad \overset{V_{0,N}\subset V_0}{\Longrightarrow} \quad \mathsf{a}(u-u_N, v_N) = 0 \quad \forall v_N \in V_{0,N} \,.$$

Galerkin orthogonality

$$\mathsf{a}(u-u_N, v_N) = 0 \quad \forall v_N \in V_{0,N} \,. \qquad (5.1.10)$$

[Geometric meaning for inner product $a(\cdot,\cdot)$ →]

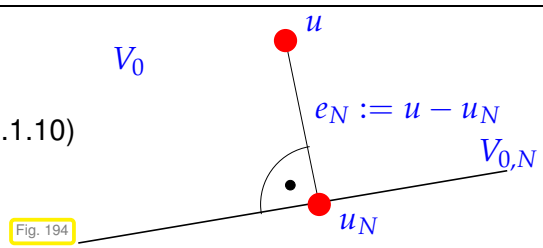


Fig. 194

Supplement 5.1.11.

In linear algebra we learned that an symmetric positive definite bilinear form (→ Def. 2.2.40) on a (finite-dimensional) vector spaces induces a Euclidean geometry with meaningful notions of length (→ Def. 2.2.43) and angle. This carries over to Hilbert spaces and makes it possible for us to draw "geometric" pictures like Fig. 194.

△

Parlance: Discretization error $e_N := u - u_N$ "$a(\cdot,\cdot)$-orthogonal" to discrete trial/test space V_N

Remark 5.1.12 (Pythagoras' theorem)

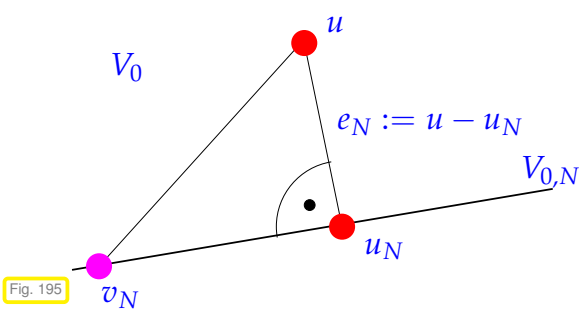


Fig. 195

If $a(\cdot,\cdot)$ is inner product on V "Pythagoras' theorem" tells us, see Fig. 195:

$$\|u - v_N\|_{\mathsf{a}}^2 = \|u - u_N\|_{\mathsf{a}}^2 + \|u_N - v_N\|_{\mathsf{a}}^2 \,. \qquad (5.1.13)$$

This is immediate from (5.1.10) and the bilinearity of $\mathsf{a}(\cdot,\cdot)$.

In Euclidean geometry: the point in a hyperplane nearest to a given point is its orthogonal projection onto the hyperplane. The next theorem states this for the inner product $\mathsf{a}(\cdot,\cdot)$ and $V_{0,N}$ instead of a hyperplane, see Fig. 195 for an illustration.

Notice that (5.1.13) with $v_N = 0$ gives a simple formula for computation of energy norm of Galerkin discretization error in numerical experiments with known solution u of (3.2.3) and u_N of (3.2.8):

$$\|u - u_N\|_{\mathsf{a}}^2 = \|u\|_{\mathsf{a}}^2 - \|u_N\|_{\mathsf{a}}^2 \,. \tag{5.1.14}$$

Theorem 5.1.15. Cea's lemma

Under Ass. 5.1.2–Ass. 5.1.4 the energy norm of the Galerkin discretization error for (3.2.3) *satisfies*

$$\|u - u_N\|_{\mathsf{a}} = \inf_{v_N \in V_{0,N}} \|u - v_N\|_{\mathsf{a}} \,.$$

Proof. Use bilinearity of a and Galerkin orthogonality (5.1.10): for *any* $v_N \in V_{0,N}$

$$\|u - u_N\|_{\mathsf{a}}^2 = \mathsf{a}(u - u_N, u - u_N) = \mathsf{a}(u - v_N, u - u_N) + \underbrace{\mathsf{a}(v_N - u_N, u - u_N)}_{=0} \,.$$

Next, use the Cauchy-Schwartz inequality for the inner product a:

$$\mathsf{a}(u,v) \leq \|u\|_{\mathsf{a}} \|v\|_{\mathsf{a}} \quad \forall u,v \in V_0 \,.$$

$$\blacktriangleright \qquad \|u - u_N\|_{\mathsf{a}}^2 \leq \|u - v_N\|_{\mathsf{a}} \cdot \|u - u_N\|_{\mathsf{a}} \,,$$

and cancel one factor $\|u - u_N\|_{\mathsf{a}}$. □

An alternative proof can invoke Pythagoras' theorem (5.1.13):

$$(5.1.13) \quad \Rightarrow \quad \|u - u_N\|_{\mathsf{a}}^2 = \|u - v_N\|_{\mathsf{a}}^2 - \|u_N - v_N\|_{\mathsf{a}}^2 \leq \|u - v_N\|_{\mathsf{a}}^2 \quad \forall v_N \in V_{0,N} \,.$$

We highlight an obvious, but fundamental consequence of Thm. 5.1.15:

Optimality of Galerkin solutions:

$$\underbrace{\|u - u_N\|_{\mathsf{a}}}_{\uparrow} = \underbrace{\inf_{v_N \in V_{0,N}} \|u - v_N\|_{\mathsf{a}}}_{\uparrow} \,, \tag{5.1.17}$$

((energy) norm of) discretization error best approximation error

☞ As regards the energy norm, the Galerkin solution is the best possible solution we can obtain in a given trial space.

Thus, Cea's lemma Thm. 5.1.15 permits us to assess accuracy of Galerkin solution w.r.t. the energy norm $\|\cdot\|_{\mathsf{a}}$ by just studying the capability of functions in $V_{0,N}$ to approximate u!

(5.1.18) More accurate solutions by refinement

Thm. 5.1.15 ▶ "Monotonicity" of best approximation: consider different trial/test spaces

$$\begin{array}{c} V_{0,N}, V'_{0,N} \subset V_0 \,, \\ V_{0,N} \subset V'_{0,N} \end{array} \quad \Rightarrow \quad \inf_{v_N \in V'_{0,N}} \|u - v_N\|_{\mathsf{a}} \leq \inf_{v_N \in V_{0,N}} \|u - v_N\|_{\mathsf{a}} \,.$$

Thus, when we measure the discretization error in the energy norm we can

enhance accuracy by simply enlarging ("refining") the trial space.

(5.1.19) Refinement of finite element spaces

Now return to Lagrangian finite element (→ Section 3.5) Galerkin discretization of linear 2nd-order elliptic variational problems.

How to achieve refinement of a (Lagrangian) FE space **?**

- h-refinement: replace $\mathcal{M}$ (underlying $V_{0,N}$) → $\mathcal{M}'$ (underlying larger discrete trial space $V'_{0,N'}$)

Example 5.1.20 (Regular/uniform refinement of triangular mesh in 2D)

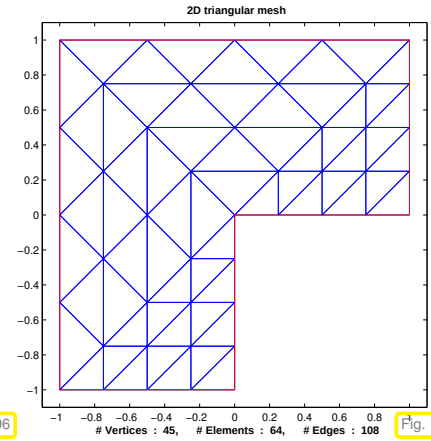


Fig. 196

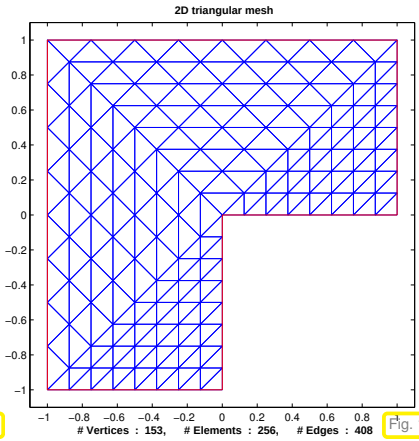


Fig. 197

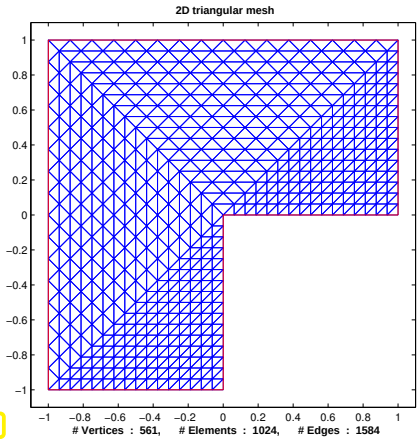


Fig. 198

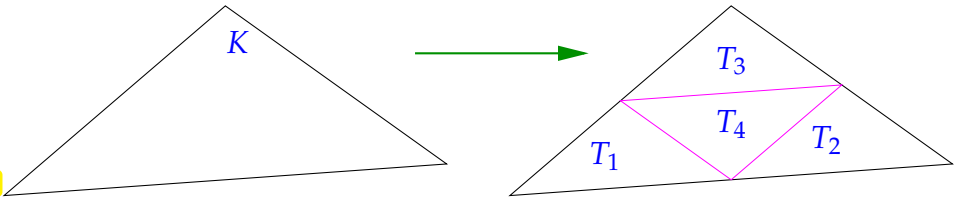


Fig. 199

Regular refinement of triangle K into four congruent triangles T_1, T_2, T_3, T_4

➢ Creates so-called nested meshes
(Each cell of the coarse mesh $\mathcal{M}'$ is a union of cells of the fine mesh $\mathcal{M}$)

$$\blacktriangleright \quad \mathcal{S}_p^0(\mathcal{M}) \subset \mathcal{S}_p^0(\mathcal{M}') \,,$$

that is, h-refinement through global regular refinement is a true refinement in the sense that its creates a larger finite element space, which contains the original finite element space.

- p-refinement: replace $V_{0,N} := \mathcal{S}_p^0(\mathcal{M})$, $p \in \mathbb{N}$ with $V'_{0,N} := \mathcal{S}_{p+1}^0(\mathcal{M}) \quad \Rightarrow \quad V_{0,N} \subset V'_{0,N}$

The extreme case of p-refinement amounts to the use of *global* polynomials on Ω as trial and test functions
➢ (polynomial) spectral Galerkin method, see Section 1.5.2.1.

Combination of h-refinement and p-refinement **?** OF COURSE (**hp-refinement**, [7])

Example 5.1.21 (Global regular refinement in BETL)

The implementation of the class **betl2::volume2dGrid::hybrid::Grid** in BETL provides the member function

```
void globalRefine(int numRefines);
```

that carries out `numRefines` global regular refinement steps of the initial mesh. So far, it is only implemented for triangular meshes.

To perform refinements, additional information about the geometry of the object, that is approximated by the initial mesh, is needed. In BETL, this is provided via the so-called **GeometryMapper**. The only additional geometry specification that is implemented so far is the specification for a disc (see **SphereMapper** in `Library/grid/geometry_sphere_mapper.hpp`), projecting the new vertices onto the sphere. If no specific geometry information is known, the **IdentityMapper** should be used for a specification (see `Library/grid/geometry_identity_mapper.hpp`). For this choice, the new vertices will not be transformed. See Code 5.1.22, Line 11-Line 21, for the usage of the **GeometryMapper**s.

To access the different meshes created by refinement, one has to use different specializations of **GridViewFactory**:

```
typedef eth::grids::utils::GridViewFactory<grid_t,view> gridview_t;
```

- In order to access the bottom level, i.e. the finest grid, the so-called *leaf view* we define

```
typedef eth::grid::GridViewTypes::LeafView view;
```

- If we want to access higher level, i.e. coarser meshes, we have to rely on

```
typedef eth::grid::GridViewTypes::LevelView view;
```

  In the latter case, in addition to the `grid_ptr` object, the instantiation also needs an integer `i`, specifying the index of the level, see Code 5.1.22, Line 5. BETL adopts the convention that level `i` represents the mesh after `numRefines - i` refinements.

C++11 code 5.1.22: Reading and refining a mesh with BETL ➙ GITLAB

```
1 // read mesh from file, see also Code 3.6.20
2 big::Input input( basename );
3 // wrap input interface around the given input
4 typedef betl2::input::InputInterface< big::Input > inpInterface_t;
5 inpInterface_t inpInterface( input );
```

```
// instantiate the grid implementation as a dynamic object
typedef betl2::volume2dGrid::hybrid::Grid grid_t;
typedef std::shared_ptr< eth::grid::Grid<grid_t::gridTraits_t> >
    grid_ptr_t;
grid_ptr_t grid_ptr( new grid_t( inpInterface ));
// Types for selecting details of refinement.
typedef betl2::GeometryMapper<grid_t::gridTraits_t> geomBaseMapper_t;

typedef betl2::IdentityMapper<grid_t::gridTraits_t> geomMapper_t;
typedef std::shared_ptr<geomBaseMapper_t> baseMapper_ptr_t;
// get a reference to the member data of the grid managing the
    refinement
auto& refhandler = grid_ptr -> refinement( );
// create an instance ot the geomMapper_t, which specifies
// the geometry used for the refinement.
baseMapper_ptr_t geomMapper( new geomMapper_t );
// select refinement procedure (only regular refinement available)
const auto elementCollection = grid_ptr -> leafEntities<0>();
refhandler.registerMapping( geomMapper, elementCollection );
// refine numRefines times
grid_ptr -> globalRefine( numRefines );
// create a grid view factory of the finest level (leaf view)
typedef eth::grids::utils::GridViewFactory<grid_t,
    eth::grid::GridViewTypes::LeafView > grid_factory_t;
grid_factory_t grid_factory( grid_ptr );
// get number of elements, edges and vertices from the leaf view
const int numElements = grid_factory.getView().size(0);
const int numEdges    = grid_factory.getView().size(1);
const int numNodes    = grid_factory.getView().size(2);
```

The next code demonstrates the use of `LevelView`s in order to access intermediate refinement level of a sequence of meshes created by refinement.

C++11 code 5.1.23: Accessing refined meshes in BETL, Code 5.1.22 cnt'd ➙ GITLAB

```
1 // fetch numbers of geometric entities for all levels starting with the
    second finest, i.e. level 1
2 for( int lev = 1; lev <= numRefines; ++lev ) {
3   typedef eth::grids::utils::GridViewFactory< grid_t,
4       eth::grid::GridViewTypes::LevelView > level_grid_factory_t;
5    level_grid_factory_t level_grid_factory(grid_ptr, lev);
6    const int levelNumElements = level_grid_factory.getView().size(0);
7    const int levelNumEdges    = level_grid_factory.getView().size(1);
8    const int levelNumNodes    = level_grid_factory.getView().size(2);
9 }
```

Note that the loop of Line 2 traverses the meshes from fine to coarse.

?! *Review question(s) 5.1.24.* (Estimates for Galerkin discretization error)

1. Explain the notion of "Galerkin orthogonality" and prove it for the Galerkin discretization of a linear variational problem with symmetric positive definite bilinear form.
2. Let the bilinear form $\mathsf{a} : V \times V \to \mathbb{R}$ on a normed real vector space satisfy

$$|\mathsf{a}(u,v)| \leq C\|u\|\|v\| \quad \forall u,v \in V \quad , \quad |\mathsf{a}(v,v)| \geq \gamma\|v\|^2 \quad \forall v \in V \ ,$$

   for some constants $C > 0$, $\gamma > 0$. Derive a bound for the norm $\|u - u_N\|$ of the Galerkin discretization error for a linear variational problem with bilinear form a in terms of the best approximation error for its exact solution $u \in V$.
3. Give an example for a 2nd-order linear elliptic boundary value problem for $-\Delta$ for which finite element discretization by means of $\mathcal{S}_1^0(\mathcal{M})$, $\mathcal{M}$ a triangular mesh, will always produce the exact solution.
4. Start from a hybrid mesh with n_c cells, n_e edges, and n_p vertices. Develop a formula that gives the numbers of cells, edges, and vertices of the mesh created by k steps of regular refinement.

5.2 Empirical (Asymptotic) Convergence of FEM

(5.2.1) Introduction and fundamental notions

Already in Section 1.6.2 we examined with the "convergence" of approximate solutions obtained through discretization. Recall that studying convergence boils down to determining dependence of some norm of the discretization error on a discretization parameter (→ § 1.6.22).

In this section we study the convergence of Galerkin solutions obtained from Lagrangian finite element discretization of linear scalar 2nd-order elliptic variational problems (→ Section 2.9) *empirically*. This means that we conduct numerical experiments, in which we measure norms of the discretization errors. Of course, this can be done only for finite sequences of discrete models. However, if these cover a sufficiently wide range of discretization parameters, they will provide evidence of general laws governing convergence. We have already seen this in Exp. 1.6.23 and Exp. 1.6.35. Yet, Exp. 1.6.34 warns that one must not jump to premature conclusions from poorly chosen sample sets of discretization parameters.

Our model problem: Dirichlet problem for Poisson equation on a polygonal domain $\Omega \subset \mathbb{R}^2$:

$$-\Delta u = f \in L^2(\Omega) \quad \text{in } \Omega \quad , \quad u = g \in C^0(\partial\Omega) \quad \text{on } \partial\Omega \ , \tag{5.2.2}$$

+ Lagrangian finite element discretization on triangular meshes (→ Section 3.5.1).

From Section 1.6.2 recall

that convergence is an *asymptotic* notion with focus on infinite sequences of discrete models associated with a sequence of discretization parameters that tends to a limit value.

Focus on asymptotics entails studying a

norm of the discretization error as function of a (real, cardinal) discretization parameter.

The discretization parameter must be linked to the resolution ("capability to approximate generic solution") of the Galerkin trial/test space $V_{0,N}$. Possible choices are

- the number of unknowns $N := \dim V_{0,N}$ as a measure of the "cost" of a discretization, see Section 1.6.2. The implied limit is $N \to \infty$.
- the maximum "size" of mesh cells, expressed by the mesh width $h_{\mathcal{M}}$ ($\to$ Def. 5.2.3), see below. The natural limit is $h_{\mathcal{M}} \to 0$.
- the local polynomial degree p in the limit $p \to \infty$, in the case of p-refinement.

Definition 5.2.3. Mesh width

Given a mesh $\mathcal{M} = \{K\}$, its mesh width $h_{\mathcal{M}}$ is defined as

$$h_{\mathcal{M}} := \max\{\operatorname{diam} K \colon K \in \mathcal{M}\} \quad , \quad \operatorname{diam} K := \max\{|\mathbf{p} - \mathbf{q}| \colon \mathbf{p}, \mathbf{q} \in K\} \ .$$

This generalizes the concept of "mesh width" introduced in Section 1.5.2.2.

Also remember

- the two main types of asymptotic converge ($\to$ Def. 1.6.24, [5, Def. 4.1.31]):

  algebraic convergence and exponential convergence.

- how to gather qualitatively and quantitative information about convergence from error norms measured in a numerical experiment, see § 1.6.27, [5, Rem. 4.1.33].

Remark 5.2.4 (Approximate computation of norms (II), see also Rem. 1.6.20)

Even if the exact solution u of a boundary value problem is known and a finite element solution u_N, it will usually be all but impossible to determine the exact value of $\|u - u_N\|$ for all relevant norms like $\|\cdot\|_{L^2(\Omega)}$, $|\cdot|_{H^1(\Omega)}$, $\|\cdot\|_{L^\infty(\Omega)}$, etc.

In the case of norms involving an integral $\|u - u_N\|$ has to be computed by means of numerical quadrature, which will boil down to the cell-wise application of a local quadrature rule (3.6.132), as discussed in Section 3.6.5.

Danger: The inevitable quadrature error may dominate the discretization error!

Safeguard: (Bolstered by theory) Chose local quadrature of "sufficiently high order"!

Guideline: Chose order $2p + 1$, when using finite element methods based on local polynomial degree p. In this case and for h-refinement the quadrature error will shrink faster than the finite element discretization error and it will not "pollute" the observed asymptotic behavior of $\|u - u_N\|_{L^2(\Omega)}$, $|u - u_N|_{H^1(\Omega)}$.

All examples in this section rely on "overkill quadrature": the order of the local quadrature rule is much higher than even demanded by the above guideline. Hence, the impact of quadrature errors can be ignored.

Example 5.2.5 (Approximate computation of norms of the discretization errors in BETL)

Code 5.2.6 listed below demonstrates how to compute $L^2(\Omega)$-norm and $H^1(\Omega)$-seminorm (*energy norm*) of the discretization error for Lagrangian finite elements.

It handles the following three steps:

- *Reconstruction* of the finite element solution (for the $L^2(\Omega)$-norm) and its gradient (for the $H^1(\Omega)$-seminorm) from a vector of basis expansion coefficients, see Code 5.2.6, Line 2-Line 10. The interpretation of the entries of the coefficient vector is supplied by an associated **FESpace** object.
- Definition of the *reference solution*. In this example we will use an *analytic solution* as a reference solution (exact solution), see Code 5.2.6, Line 12-Line 23.
- Actual *computation of the error norm*, see Code 5.2.6, Line 25-Line 26, based on the data prepared in the previous steps.

For all steps, so-called *grid functions* are used in BETL:

- In the *reconstruction*-step, objects of type **InterpolationGridFunction** are used, see `Library/functional/interpolation_grid_function.hpp`,
- while for the *reference solution* we use the class type **fem::AnalyticalGridFunction**, see `Library/functional/analytical_grid_function.hpp`.

The instantiation of an object of class type **InterpolationGridFunction** needs a **eth::grids::utils::GridViewFactory** object, a **fe::FESpace** object and a vector of type **Eigen::Matrix< numeric_t, Eigen::Dynamic, 1 >** whose length corresponds to the number of d.o.f. stored in the **fe::FESpace** object. It reconstructs the finite element solution from the supplied vector.

The class type **fem::AnalyticalGridFunction** was already used in Code 3.6.184, Line 18, to create a grid function describingthe Dirichlet data. Remember that in addition to a **eth::grids::utils::GridViewFactory** object, the class **fem::AnalyticalGridFunction** requires an functor as input argument that provides an evaluation operator

```
result_t operator() (globalCoord_t x) const;
```

The method **fem::makeAnalyticalGridFunction** from ➔ GITLAB acts as a wrapper so that analytical grid functions can also be built from `lambda` functions.

Grid functions provide the following member functions:

- The bracket operator `operator()(q,e)` does an *evaluation on the grid* of the underlying function `f`, i.e. `f(q,e)`, where `q` represents a local quadrature point and `e` corresponds to a cell.
- `inner(g)` takes another grid function `g` and calculates the inner product $\langle f, g\rangle_{L^2(\Omega)}$ based on a quadrature rule defined by the following predefined **QuadRuleList**, see § 3.6.164 for a definition:

```
using DefaultQuadratureList = QuadRuleList<
                QuadRule< eth::base::RefElType::SEGMENT, 6 >,
                QuadRule< eth::base::RefElType::TRIA   ,25 >,
                QuadRule< eth::base::RefElType::QUAD   ,25 > >
```

- `norm()` calculates the $L^2(\Omega)$-norm of the underlying function f, i.e. $\|f\|_{L^2(\Omega)}$, using the above mentioned `DefaultQuadratureList`.

If the user wants to use other quadrature rules to evaluate an inner product of two grid functions, the class **InnerProduct**, that is called by the member functions above, can be used directly, see `Library/functional/g`

For the error computations, we will additionally need operations that work for all grid functions (see Code 5.2.6, Line 25). The operations are defined in `Library/functional/grid_function_operations` Let f and g be two grid functions and λ be a scalar of a certain type. Then the following operations (and combinations of them) can be performed:

- $f+g$: returns the grid function representing the sum of the two grid functions.
- $f-g$: returns the grid function representing the difference of the two grid functions.
- $\lambda * f$: returns the grid function representing the grid function f scaled by a scalar λ.

The partial listing of the mainfile ➙ GITLAB shown in Code 5.2.6 computes the discretisation error of the Galerkin solution of a Dirichlet boundary value problem by means of linear Lagrangian finite elements on a sequence of meshes that is obtained via the global refinement routine from Code 5.1.22. Listed are only the error computations on the leaf view, i.e. the finest level 0. For the other levels, the computations are analogous and can be found in the mainfile itself.

C++11 code 5.2.6: Computation of $L^2(\Omega)$- and $H^1(\Omega)$-norm of the finite element discretization error ➙ GITLAB

```
// types for handling the finite element solution and its gradient
typedef InterpolationGridFunction< grid_factory_t ,typename
          DH_t::fespace_t,numeric_t> func_t;
typedef InterpolationGridFunction< grid_factory_t ,typename
          DH_t::fespace_t,numeric_t,fe::FEDiff::Grad>  grad_func_t;
// wrap the finite element solution into a grid function
// (sol represents the coefficient vector of the solution)
func_t uh (grid_factory,dh.fespace(),sol);
// get the gradient of the finite element solution
grad_func_t grad_uh (grid_factory,dh.fespace(),sol);
// Define the analytic solution from a lambda function
const auto sol_exact_double = [](const coord_t& x){
  Eigen::Matrix<numeric_t, 1, 1> res; res << 0.5*cos(x(0))*cos(x(1));
     return res; };
// create an AnalyticalGridFunction object
const auto exactFunc =
 fem::makeAnalyticalGridFunction(grid_factory,sol_exact_double );
// define the gradient of the analytic solution
const auto grad_sol_exact_double = [](const coord_t& x){
  Eigen::Matrix<numeric_t, 2, 1> res;
  res << -0.5*sin(x(0))*cos(x(1)),-0.5*cos(x(0))*sin(x(1)); return
     res; };
// create another AnalyticalGridFunction object
const auto grad_exactFunc =
  fem::makeAnalyticalGridFunction(grid_factory,grad_sol_exact_double);

  // Compute the L^2(Ω)/H^1(Ω)-norm of the error
L2_error_vector(numRefines) = (uh - exactFunc).norm();
H1_error_vector(numRefines) = (grad_uh - grad_exactFunc).norm();
```

Experiment 5.2.7 (Convergence for linear and quadratic Lagrangian finite elements in energy norm)

Setting: $\Omega =]0,1[^2$, $f(x_1,x_2) = 2\pi^2 \sin(\pi x_1)\sin(\pi x_2)$, $\boldsymbol{x} \in \Omega$, $g = 0$

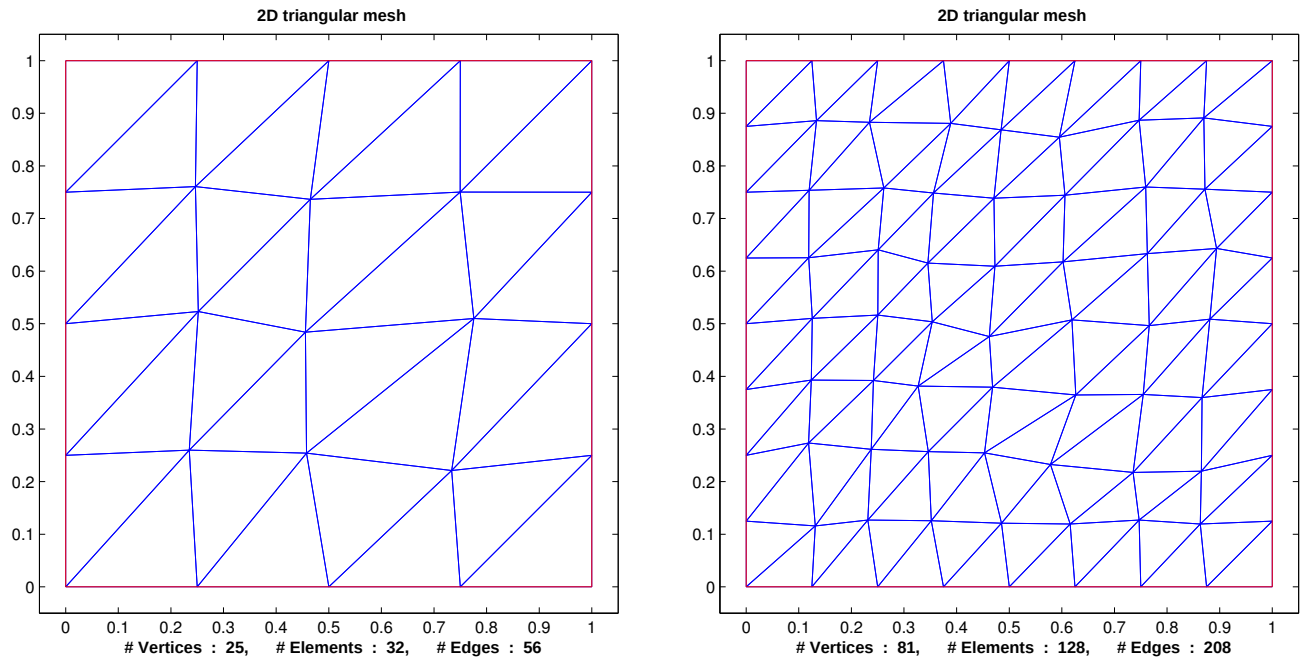
➢ Smooth solution $u(x,y) = \sin(\pi x)\sin(\pi y)$.

- Galerkin finite element discretization based on triangular meshes and
  - linear Lagrangian finite elements, $V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M}) \subset H^1_0(\Omega)$ (→ Section 3.3),
  - quadratic Lagrangian finite elements, $V_{0,N} = \mathcal{S}^0_{2,0}(\mathcal{M}) \subset H^1_0(\Omega)$ (→ Ex. 3.5.3),
- quadrature rule (3.6.162) for assembly of local load vectors (→ Section 3.6.5),

Monitored: $H^1(\Omega)$-semi-norm (→ Def. 2.3.23) of the Galerkin discretization error $u - u_N$

Approximate $(*)$ computation of $|u - u_N|_{H^1(\Omega)}$ on a sequence of meshes (created by successive regular refinement (→ Ex. 5.1.20) of coarse initial mesh)

$(*)$: use of local quadrature rule (3.6.162) (on current FE mesh)



Unstructured triangular meshes of $\Omega =]0,1[^2$ (two coarsest specimens)

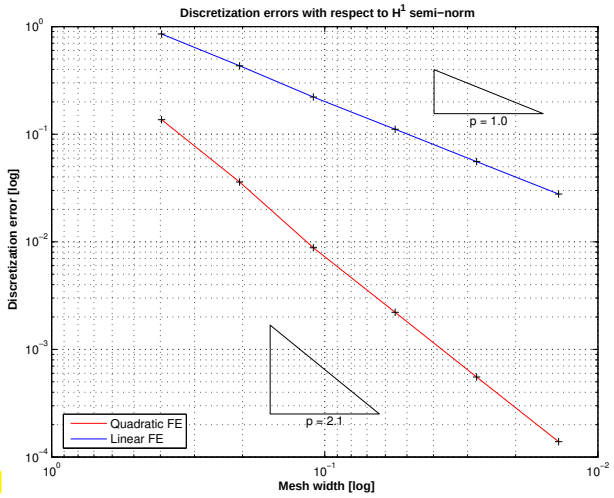


Fig. 200

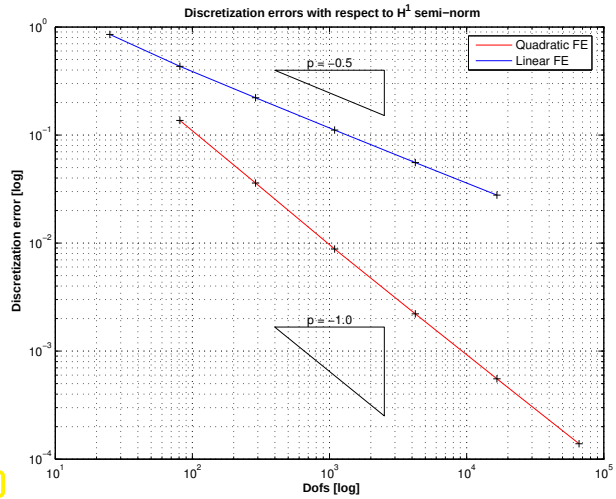


Fig. 201

$H^1(\Omega)$-semi-norm of discretization error on unit square ($- \leftrightarrow p = 1$, $- \leftrightarrow p = 2$)

Again, recall the two types of convergence (algebraic convergence vs. exponential convergence) from Def. 1.6.24 and how to detect them in a numerical experiment by inspecting appropriate graphs, see § 1.6.27.

Observations:
- Algebraic rates of convergence in terms of N and h
- Quadratic Lagrangian FE converge with double the rate of linear Lagrangian FE

Recall: Rates of algebraic convergence can be estimated by linear least squares fitting $\rightarrow$ Code 1.6.30. In Fig. 200 and Fig. 201 these estimated rates are indicated by the slopes of hypothenuses of triangles (ger. "Steigungsdreieck"). We find

linear Lagrangian finite elements:	$\lvert u - u_N \rvert_{H^1(\Omega)} = O(h_{\mathcal{M}}) = O(N^{-\frac{1}{2}})$
quadratic Lagrangian finite elements:	$\lvert u - u_N \rvert_{H^1(\Omega)} = O(h_{\mathcal{M}}^2) = O(N^{-1})$

Experiment 5.2.8 (Convergence of linear and quadratic Lagrangian finite elements in L^2-norm)

Setting as above in Exp. 5.2.7, $\Omega =]0,1[^2$.

Monitored: asymptotics of the $L^2(\Omega)$-semi-norm of the Galerkin discretization error (approximate computation of $\|u - u_N\|_{L^2(\Omega)}$ by means of local quadrature rule (3.6.162) on a sequence of meshes created by successive regular refinement ($\rightarrow$ Ex. 5.1.20) of coarse initial mesh).

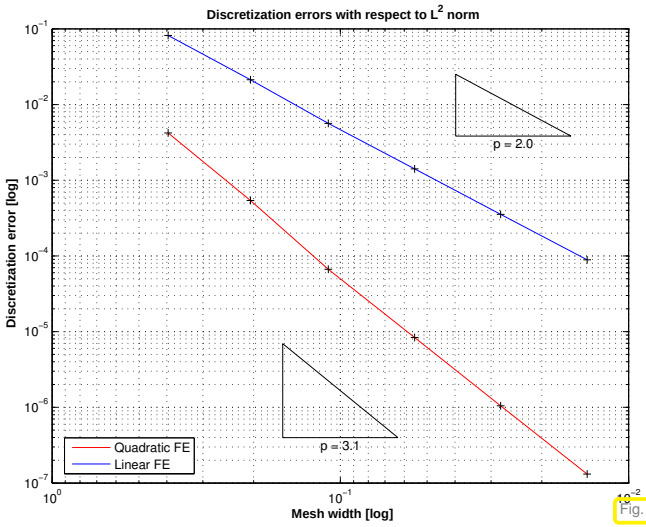


Fig. 202

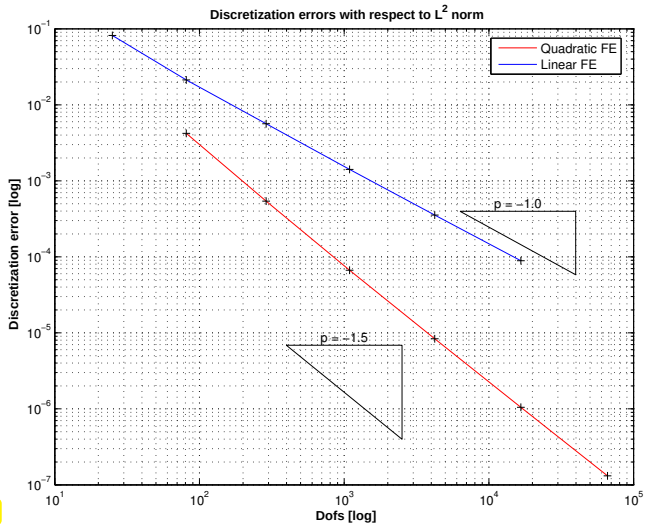


Fig. 203

$L^2(\Omega)$-norm of discretization error on unit square ($- \leftrightarrow p = 1$, $- \leftrightarrow p = 2$)

Observations:
- Linear Lagrangian FE ($p = 1$) ➽ $\|u - u_N\|_0 = O(h_{\mathcal{M}}^2) = O(N^{-1})$
- Quadratic Lagrangian FE ($p = 2$) ➽ $\|u - u_N\|_0 = O(h_{\mathcal{M}}^3) = O(N^{-1.5})$

For the "conversion" of convergence rates with respect to the mesh width $h_{\mathcal{M}}$ and $N := \dim \mathcal{S}_p^0(\mathcal{M})$, note that in 2D for Lagrangian finite element spaces with fixed polynomial degree ($\rightarrow$ Section 3.5) and meshes created by global (that is, carried out everywhere) regular refinement

$$N = O(h_{\mathcal{M}}^{-2}) \; . \tag{5.2.9}$$

See Section 5.3.5, page 414 for further discussion, (5.3.66) for a more general relationship.

Experiment 5.2.10 (h-convergence of Lagrangian FEM on L-shaped domain)

Setting: model problem (5.2.2) on $\Omega =]-1,1[^2 \setminus (]0,1[\times]-1,0[)$, exact solution (in polar coordinates, see § 2.4.24)

$$u(r,\varphi) = r^{2/3}\sin(2/3\varphi) \qquad \succ \quad f = 0, g = u_{|\partial\Omega}.$$

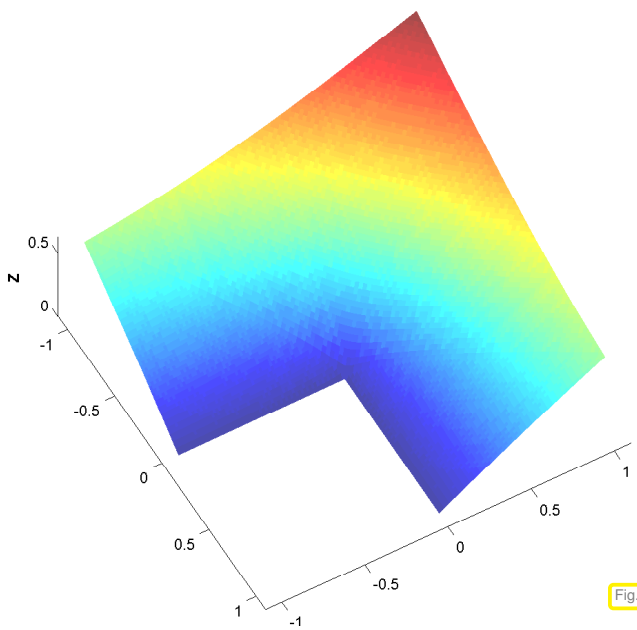


Fig. 204

Exact solution u

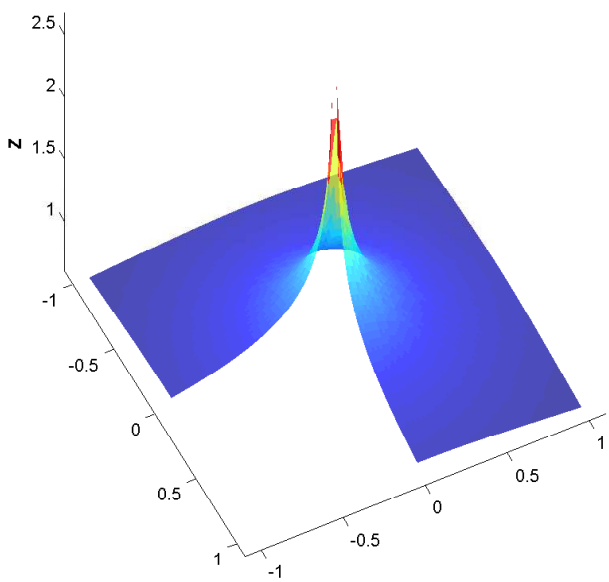


Fig. 205

Norm of gradient: $\|\mathbf{grad}\, u\|$

Note: $\mathbf{grad}\, u$ has a so-called singularity at 0, that is, "$\|\mathbf{grad}\, u(0)\| = \infty$".

- Galerkin finite element discretization based on triangular meshes and
  - linear Lagrangian finite elements, $V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M}) \subset H^1_0(\Omega)$ ($\rightarrow$ Section 3.3),
  - quadratic Lagrangian finite elements, $V_{0,N} = \mathcal{S}^0_{2,0}(\mathcal{M}) \subset H^1_0(\Omega)$ ($\rightarrow$ Ex. 3.5.3),
- linear/quadratic interpolation of Dirichlet data to obtain offset function $u_0 \in \mathcal{S}^0_{p,0}(\mathcal{M})$, $p = 1, 2$, see Section 3.6.6, Ex. 3.6.174.

Sequence of meshes created by successive regular refinement ($\rightarrow$ Ex. 5.1.20) of coarse initial mesh, see Fig. 206 and Fig. 207.

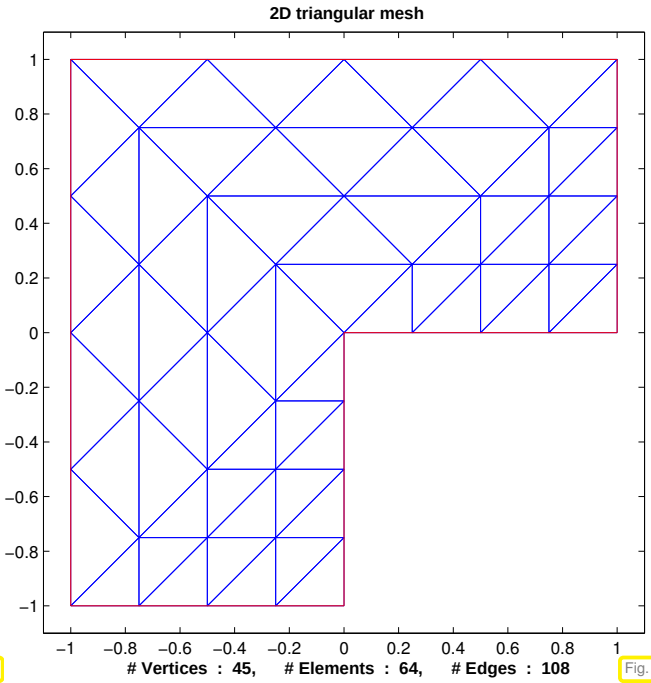


Fig. 206

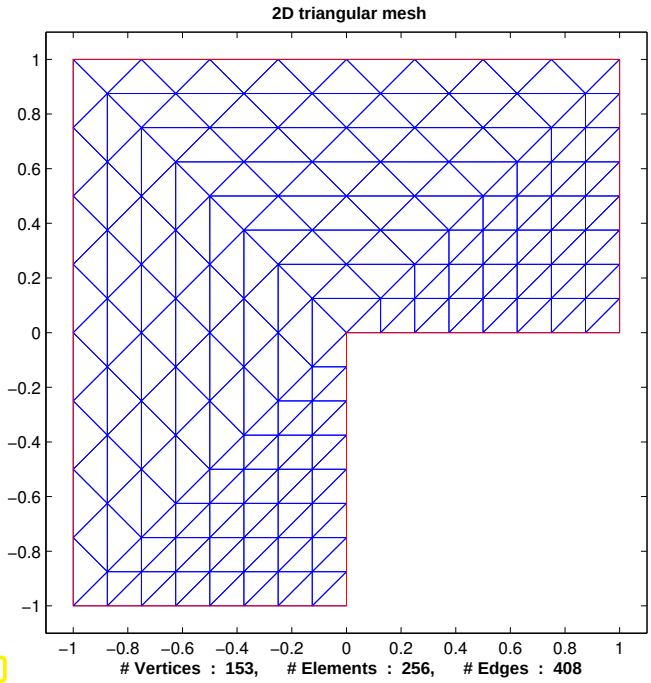


Fig. 207

Unstructured triangular meshes of $\Omega =]-1,1[^2 \setminus (]0,1[\times]-1,0[)$ (two coarsest specimens)

Approximate computation of $|u - u_N|_{H^1(\Omega)}$ by using local quadrature formula (3.6.162) on FE meshes.

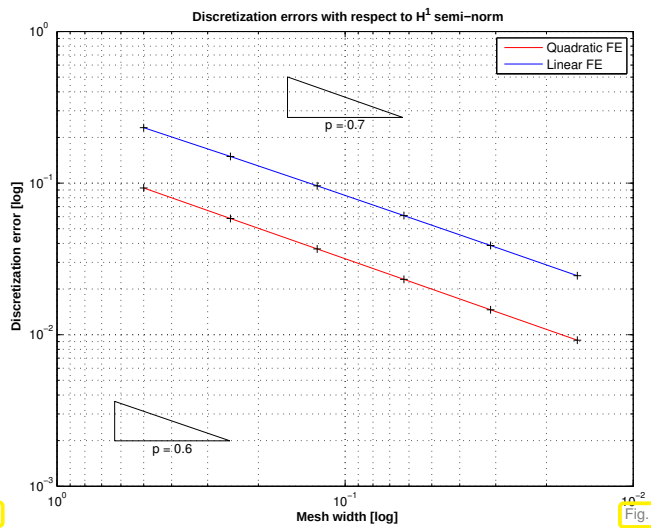


Fig. 208

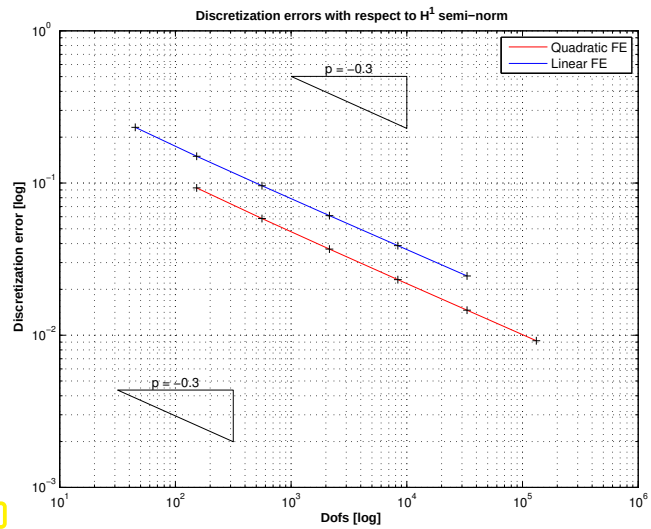


Fig. 209

$H^1(\Omega)$-semi-norm of discretization error on "L-shaped" domain ($- \leftrightarrow p = 1$, $- \leftrightarrow p = 2$)

Observations:
- For **both** $p = 1, 2$: $\|u - u_N\|_1 = O(N^{-1/3})$
- No gain from higher polynomial degree

Conjecture: singularity of $\mathbf{grad}\, u$ at $x = 0$ seems to foil faster algebraic convergence of quadratic Lagrangian finite element solutions!

Experiment 5.2.11 (Convergence of Lagrangian FEM for p-refinement)

- Model BVP as in Exp. 5.2.7 ➢ unit square domain $\Omega =]0,1[^2$,

  Exp. 5.2.10 ➢ L-shaped domain $\Omega =]-1,1[^2 \setminus (]0,1[\times]-1,0[)$.
- Galerkin finite element discretization based on $\mathcal{S}_p^0(\mathcal{M})$, $p = 1,2,3,5,6,7,8,9,10$, built on a *fixed* coarse triangular mesh of Ω.

  ➢ p-refinement

Monitored: $H^1(\Omega)$-semi-norm (energy norm) and $L^2(\Omega)$-norm of discretization error as functions of polynomial degree p and $N := \dim \mathcal{S}_p^0(\mathcal{M})$.

(Computation of norms by means of local quadrature rule of order 19!. This renders the error in norm computations introduced by numerical quadrature negligible.)

Meaningful discretization parameters for asymptotic study of error norms:

- polynomial degree p for Lagrangian finite element space,
- $N := \dim V_{0,N}$ as a measure of the "cost" of a discretization, see Section 1.6.2.

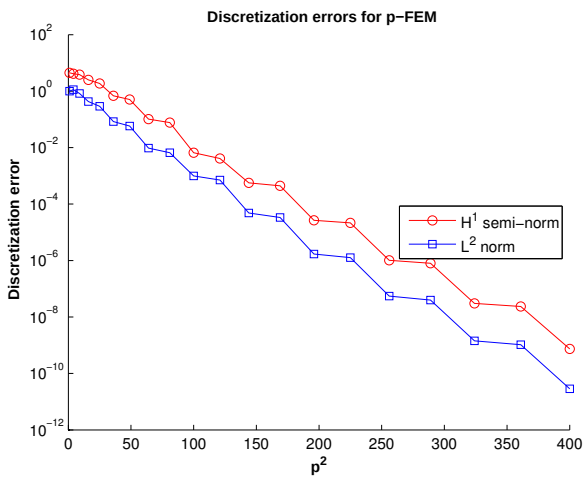


Fig. 210

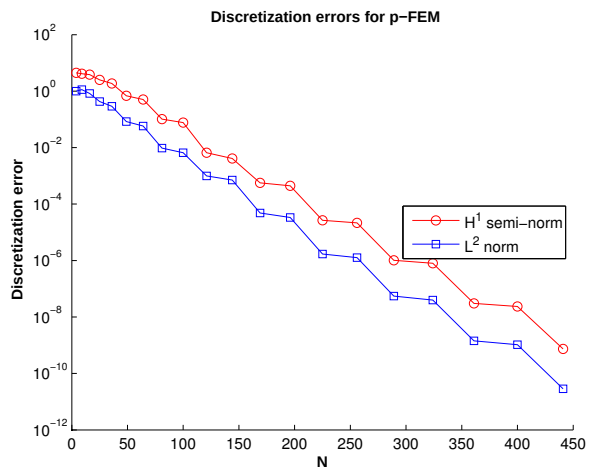


Fig. 211

$\Omega =]0,1[^2$: behavior of $|u - u_N|_{H^1(\Omega)}$ for different polynomial degrees.
Lagrangian FEM: p-convergence for smooth (analytic) solution

Observation: exponential convergence of FE discretization error, *cf.* the behavior of the discretization error of spectral collocation and polynomial spectral Galerkin methods in 1D, Exp. 1.6.23.

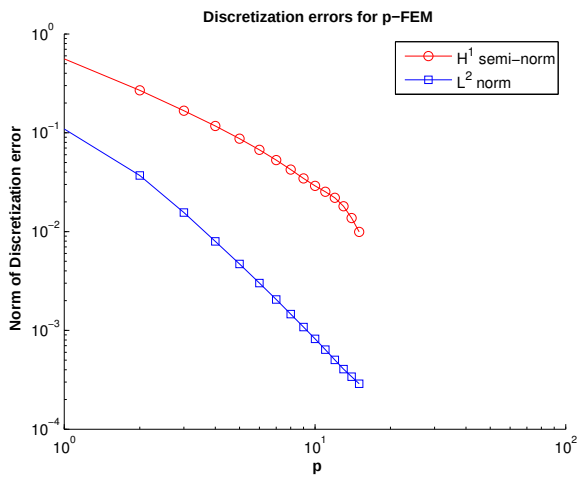


Fig. 212

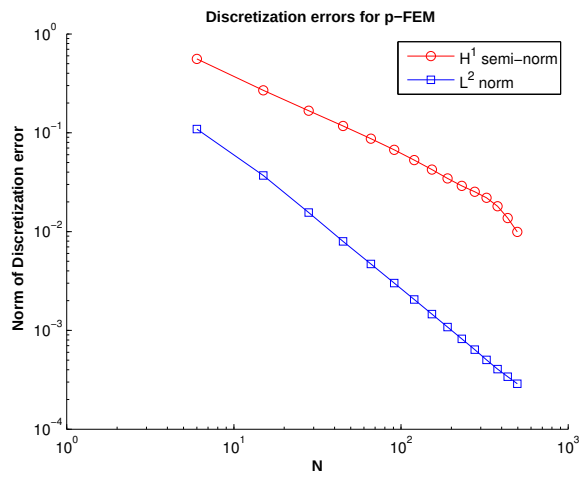


Fig. 213

Lagrangian FEM: p-convergence for solution with singular gradient (L-shaped domain)

Observation: Only algebraic convergence of FE discretization error!

The suspect: "singular behavior" of $\mathbf{grad}\, u$ at $\boldsymbol{x} = 0$.

(5.2.12) Summary of observations

Observations on convergence of Galerkin finite element solutions of 2nd-order elliptic BVPs obtained by means of Lagrangian finite elements:

- For h-refinement we generally observe algebraic convergence of $H^1(\Omega)$-/$L^2(\Omega)$-norms of the discretization errors in meshwidth/problem size.
- The rate of convergence seems to depend on
  - the kind of error norm considered,
  - properties of the exact solution u of the boundary value problem,
  - the (uniform) polynomial degree of the Lagrangian finite element space.
- In general $\|u - u_N\|_{L^2(\Omega)}$ seems to converge faster than $|u - u_N|_{H^1(\Omega)}$.

The following sections will be devoted to providing some mathematical underpinning for these observation, which will yield deeper insights into the asymptotic behavior of finite element discretization errors.

5.3 A Priori Finite Element Error Estimates

(5.3.1) A priori versus a posteriori error estimates

We are interested in a priori estimates of norms of the discretization error $u - u_N$, where u is the exact solution of a linear 2nd-order elliptic boundary value problem and u_N its finite element Galerkin approximation.

A priori estimate: bounds for error norms available **before** computing approximate solutions.

$$\Updownarrow$$

A posteriori estimate: bounds for error norms based on an approximate solution **already computed**.

We repeat our assumptions: The variational formulation of the elliptic boundary value problem leads to a linear variational problem (3.2.3) with symmetric and positive definite bilinear form a ($\rightarrow$ Ass. 5.1.2) and a-continuous right hand side functional ($\rightarrow$ Ass. 5.1.3).

(5.3.2) General policy for obtaining a priori error estimates in energy norm

➢ Results of Section 5.1 provide handle on a priori estimate for Galerkin discretization error:

Optimality (5.1.17) of Galerkin solution ► a priori error estimates

Thm. 5.1.15 ➤ Estimate energy norm of Galerkin discretization error $u - u_N$ by bounding the best approximation error for exact solution u in finite element space:

$$\underbrace{\|u - u_N\|_{\mathsf{a}}}_{\uparrow} \leq \underbrace{\inf_{v_N \in V_{0,N}} \|u - v_N\|_{\mathsf{a}}}_{\uparrow} , \tag{5.1.17}$$

(norm of) discretization error best approximation error

How to estimate best approximation error $\inf_{v_N \in V_{0,N}} \|u - v_N\|_V$?

➢ Well, given solution u seek candidate function $w_N \in V_{0,N}$ with

$$\|u - w_N\|_V \approx \inf_{v_N \in V_N} \|u - v_N\|_V .$$

Natural choice: w_N by interpolation/averaging of (*unknown, but existing*) u

Thus, the task of bounding the Galerkin discretization error is reduced to interpolation error estimates.

5.3.1 Estimates for linear interpolation in 1D

In this section we first study interpolation error estimates in one spatial dimension, in order to elucidate the general approach and the structure of the estimates.

Computational domain ($\rightarrow$ Section 1.4): interval $\Omega = [a, b]$

Given: mesh of Ω ($\rightarrow$ Section 1.5.2.2): $\mathcal{M} := \{]x_{j-1}, x_j[: j = 1, \dots, M\}$, $M \in \mathbb{N}$

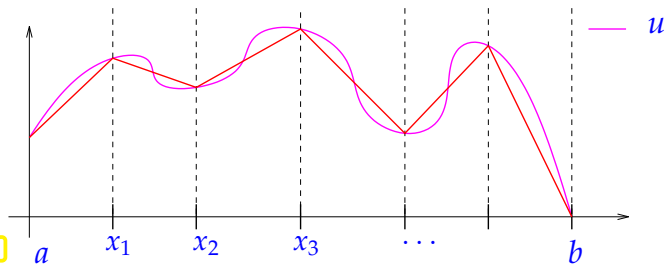


Fig. 214

Piecewise linear interpolant of $u \in C^0([a,b])$

$$\mathsf{I}_1 u \in \mathcal{S}_1^0(\mathcal{M}) \,, \tag{5.3.3}$$

$$(\mathsf{I}_1 u)(x_j) = u(x_j) \,, \quad j = 0, \dots, M \,. \tag{5.3.4}$$

➢ [5, Section 3.3.2]

Goal: Bound suitable norm (→ Section 1.6.1) of interpolation error $u - \mathsf{I}_1 u$ in terms of geometric quantities $(*)$ characterizing $\mathcal{M}$.

$(*)$: A typical such quantity is the mesh width $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|$, *cf.* Def. 5.2.3.

Now we investigate *different norms* of the interpolation error. Beforehand, recall the various norms on spaces of (bounded/integrable) functions, the supremum norm $\|\cdot\|_{L^\infty(]a,b[)}$ (→ Def. 1.6.5), the L^2-norm $\|\cdot\|_{L^2(]a,b[)}$ (→ Def. 1.6.7), and the H^1-(semi)norm $|\cdot|_{H^1(]a,b[)}$ (→ Def. 1.6.14, see also Def. 2.3.23).

✦ $\|u - \mathsf{I}_1 u\|_{L^\infty([a,b])}$, see [5, Section 4.1.2] and [5, Section 4.5.1]: from [5, Thm. 4.1.37] for $n = 1$: for $u \in C^2([a,b])$

$$\max_{x_{j-1} \le x \le x_j} u(x) - (\mathsf{I}_1 u)(x) = \tfrac{1}{4} u''(\xi_t)(x_j - x_{j-1})^2 \,, \quad \text{for some } \xi_t \in]x_{j-1}, x_j[\,, \tag{5.3.5}$$

with local linear interpolant $$(\mathsf{I}_1 u)(x) = \frac{x - x_{j-1}}{x_j - x_{j-1}} u(x_j) - \frac{x_j - x}{x_j - x_{j-1}} u(x_{j-1}) \,. \tag{5.3.6}$$

(5.3.5) ▶ interpolation error estimate in $L^\infty([a,b])$

$$\boxed{\|u - \mathsf{I}_1 u\|_{L^\infty([a,b])} \le \tfrac{1}{4} h_{\mathcal{M}}^2 \|u''\|_{L^\infty([a,b])}} \,. \tag{5.3.7}$$

This is obtained by simply taking the maximum over all *local* norms of the interpolation error.

However, we should actually target the energy norm. Hence, we also have to study other norms of the interpolation error:

✦ norm $\|u - \mathsf{I}_1 u\|_{L^2([a,b])}$: All mesh cells contribute to this norm: with $\mathsf{I}_1 u$ from (5.3.6)

$$\|u - \mathsf{I}_1 u\|^2_{L^2([a,b])} = \sum_{j=1}^M \|u - \mathsf{I}_1 u\|^2_{L^2(]x_{j-1},x_j[)} = \sum_{j=1}^M \int_{x_{j-1}}^{x_j} |(u - \mathsf{I}_1 u)(x)|^2 \, \mathrm{d}x \,. \tag{5.3.8}$$

➢ Idea: *localization*
(Estimate error on individual mesh cells and sum local bounds)

This idea is very natural for piecewise linear interpolation, because it is local in the sense that $\mathsf{I}_1 u$ on a cell K depends only on the values of u on $\overline{K}$!

Recall integrating by parts

$$\int_0^1 u(\xi) v'(\xi) \, \mathrm{d}\xi = -\int_0^1 u'(\xi) v(\xi) + \underbrace{(u(1)v(1) - u(0)v(0))}_{\text{boundary terms}} \quad \forall u, v \in C^1_{\mathrm{pw}}([0,1]) \,. \tag{1.3.40}$$

Apply this formula *twice*, for $u \in C^2([x_{j-1}, x_j])$, $x \in [x_{j-1}, x_j]$, thus removing derivatives from u:

$$\begin{aligned}
&\int_{x_{j-1}}^{x} \frac{(x_j - x)(\xi - x_{j-1})}{x_j - x_{j-1}} u''(\xi)\,\mathrm{d}\xi + \int_{x}^{x_j} \frac{(x - x_{j-1})(x_j - \xi)}{x_j - x_{j-1}} u''(\xi)\,\mathrm{d}\xi \\
&\qquad = -\int_{x_{j-1}}^{x} \frac{x_j - x}{x_j - x_{j-1}} u'(\xi)\,\mathrm{d}\xi + \frac{(x_j - x)(x - x_{j-1})}{x_j - x_{j-1}} u'(x) \\
&\qquad\quad + \int_{x}^{x_j} \frac{x - x_{j-1}}{x_j - x_{j-1}} u'(\xi)\,\mathrm{d}\xi - \frac{(x_j - x)(x - x_{j-1})}{x_j - x_{j-1}} u'(x) \\
&\qquad = \frac{x_j - x}{x_j - x_j}\big(u(x_{j-1}) - u(x)\big) + \frac{x - x_{j-1}}{x_j - x_{j-1}}\big(u(x_j) - u(x)\big) \\
&\qquad\qquad = \underbrace{\frac{x_j - x}{x_j - x_{j-1}} u(x_{j-1}) + \frac{x - x_{j-1}}{x_j - x_{j-1}} u(x_j)}_{=\mathsf{I}_1 u(x)} - u(x)\,. \qquad (5.3.9)
\end{aligned}$$

We also appealed to the fundamental theorem of calculus, which is (1.3.40) for $v \equiv 1$. What we have obtained is a (kernel) representation formula for the local interpolation error $\mathsf{I}_1 u - u$ of the form

$$(\mathsf{I}_1 u - u)(x) = \int_{x_{j-1}}^{x_j} G(x,\xi)\, u''(\xi)\,\mathrm{d}\xi\,. \qquad (5.3.10)$$

with $$G(x,\xi) = \begin{cases} \dfrac{(x_j - x)(\xi - x_{j-1})}{x_j - x_{j-1}} & \text{for } x_{j-1} \le \xi < x\,, \\[2ex] \dfrac{(x - x_{j-1})(x_j - \xi)}{x_j - x_{j-1}} & \text{for } x \le \xi \le x_j\,. \end{cases}$$, which satisfies

$$|G(x,\xi)| \le |x_j - x_{j-1}| \quad \Rightarrow \quad \int_{x_{j-1}}^{x_j} G(x,\xi)^2\,\mathrm{d}\xi \le |x_j - x_{j-1}|^3\,. \qquad (5.3.11)$$

The following figures display the kernel function G for 1D linear interpolation and for $x_{j-1} = 0$, $x_j = 1$.

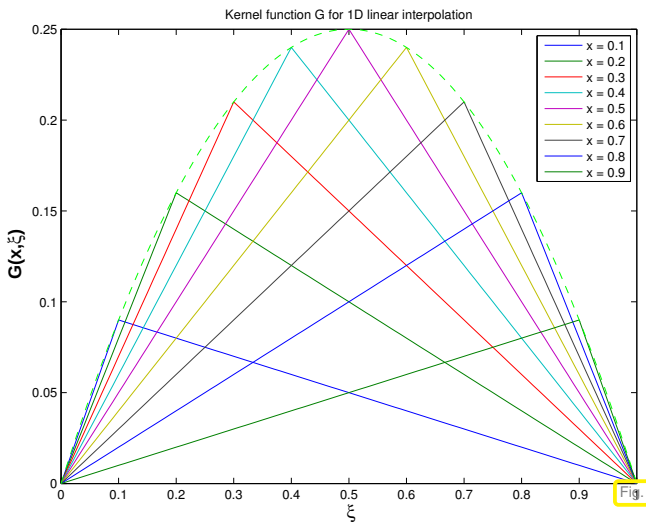


Fig. 215

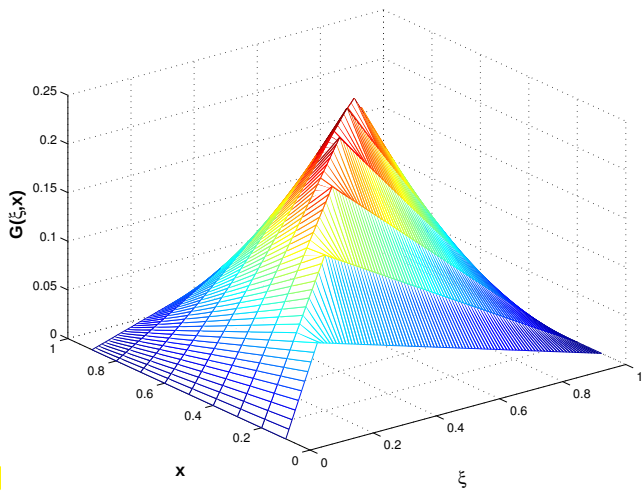


Fig. 216

The next step relies on the Cauchy-Schwarz inequality (2.2.44) in the form

$$\int_\Omega u(\boldsymbol{x}) v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \le \|u\|_{L^2(\Omega)} \|v\|_{L^2(\Omega)} \quad \forall u, v \in L^2(\Omega)\,, \qquad (1.6.13)$$

which is applied to the representation (5.3.10):

$$\blacktriangleright \quad \int_{x_{j-1}}^{x_j} |u(x) - \mathsf{I}_1 u(x)|^2 \,\mathrm{d}x = \int_{x_{j-1}}^{x_j} \left| \int_{x_{j-1}}^{x_j} G(x,\xi)\, u''(\xi)\,\mathrm{d}\xi \right|^2 \mathrm{d}x \tag{5.3.12}$$

$$\overset{(2.3.30)}{\leq} \int_{x_{j-1}}^{x_j} \left\{ \int_{x_{j-1}}^{x_j} G(x,\xi)^2\,\mathrm{d}\xi \cdot \int_{x_{j-1}}^{x_j} |u''(\xi)|^2\,\mathrm{d}\xi \right\} \mathrm{d}x\,.$$

As a consequence of (5.3.11) we can drop the kernel function G from right hand side of (5.3.12)

$$\overset{(5.3.11)}{\Rightarrow} \quad \boxed{\|u - \mathsf{I}_1 u\|^2_{L^2(]x_{j-1},x_j[)} = \int_{x_{j-1}}^{x_j} |u(x) - \mathsf{I}_1 u(x)|^2\,\mathrm{d}x \leq |x_j - x_{j-1}|^4 \int_{x_{j-1}}^{x_j} |u''(\xi)|^2\,\mathrm{d}\xi}\,. \tag{5.3.13}$$

Apply this estimate on $[x_{j-1}, x_j]$, note that $|x_j - x_{j-1}| \leq h_{\mathcal{M}}$ for any j, sum over all cells of the mesh $\mathcal{M}$ and take square root.

$$(5.3.13) \quad \Rightarrow \quad \boxed{\|u - \mathsf{I}_1 u\|_{L^2([a,b])} \leq h^2_{\mathcal{M}} \|u''\|_{L^2([a,b])}}\,. \tag{5.3.14}$$

✦ norm $|u - \mathsf{I}_1 u|_{H^1([a,b])}$: In light of Def. 2.3.23 of the $H^1([a,b])$-seminorm, we first differentiate representation formula (5.3.10): for $x_{j-1} < x < x_j$, using the explicit piecewise linear representation of G,

$$\begin{aligned} \frac{d}{dx}(\mathsf{I}_1 u - u)(x) &= \int_{x_{j-1}}^{x_j} \frac{\partial G}{\partial x}(x,\xi) u''(\xi)\,\mathrm{d}\xi \\ &= \int_{x_{j-1}}^{x_j} -\frac{\xi - x_{j-1}}{x_j - x_{j-1}} u''(\xi)\,\mathrm{d}\xi + \int_{x_{j-1}}^{x_j} \frac{x_j - \xi}{x_j - x_{j-1}} u''(\xi)\,\mathrm{d}\xi\,. \end{aligned}$$

Again, the Cauchy-Schwarz inequality (1.6.13) is useful and yields

$$\begin{aligned} \int_{x_{j-1}}^{x_j} \left| \frac{d}{dx}(\mathsf{I}_1 u - u)(x) \right|^2 \mathrm{d}x &= \int_{x_{j-1}}^{x_j} \left| \int_{x_{j-1}}^{x_j} \frac{\partial G}{\partial x}(x,\xi) u''(\xi)\,\mathrm{d}\xi \right|^2 \mathrm{d}x \\ &\leq \int_{x_{j-1}}^{x_j} \left\{ \int_{x_{j-1}}^{x_j} \underbrace{\left| \frac{\partial G}{\partial x}(x,\xi) \right|^2}_{\leq 1} \mathrm{d}\xi \cdot \int_{x_{j-1}}^{x_j} |u''(\xi)|^2\,\mathrm{d}\xi \right\} \mathrm{d}x\,. \end{aligned} \tag{5.3.15}$$

$$\blacktriangleright \quad \boxed{|u - \mathsf{I}_1 u|^2_{H^1(]x_{j-1},x_j[)} \leq (x_j - x_{j-1})^2 \int_{x_{j-1}}^{x_j} |u''(\xi)|^2\,\mathrm{d}\xi}\,. \tag{5.3.16}$$

As above, apply this estimate on $[x_{j-1}, x_j]$, use $|x_j - x_{j-1}| \leq h_{\mathcal{M}}$, sum over all cells of the mesh $\mathcal{M}$ and take square root.

$$(5.3.16) \quad \Rightarrow \quad \boxed{|u - \mathsf{I}_1 u|_{H^1([a,b])} \leq h_{\mathcal{M}} \|u''\|_{L^2([a,b])}} \; . \tag{5.3.17}$$

What we learn from this example:

1. We have to rely on smoothness of the interpolant u to obtain bounds for norms of the interpolation error. In the above estimates, we have to take for granted boundedness/square integrability of the second derivative.
2. The bounds for the norms of the interpolation error involve norms of derivatives of the interpolant.
3. For smooth u we find algebraic convergence ($\rightarrow$ Def. 1.6.24) of norms of the interpolation error *in terms of mesh width* $h_{\mathcal{M}} \rightarrow 0$.

5.3.2 Error estimates for linear interpolation in 2D

Given:

- ✦ polygonal domain $\Omega \subset \mathbb{R}^2$
- ✦ triangular mesh $\mathcal{M}$ of Ω

($\rightarrow$ Def. 3.4.2)

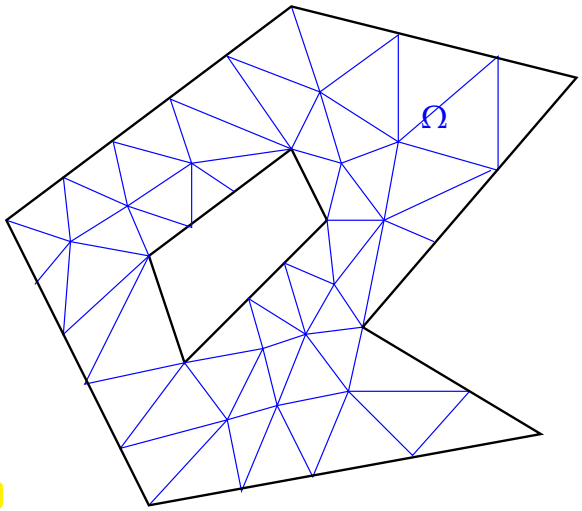


Fig. 217

Section 5.3.1 introduced piecewise linear interpolation on a mesh/grid in 1D. The next definition gives the natural 2D counterpart on a triangular mesh, which is closely related to the piecewise linear reconstruction (interpolation) operator from (4.2.15), see Fig. 190,Fig. 191.

Definition 5.3.18. Linear interpolation in 2D

The linear interpolation operator $\mathsf{I}_1 : C^0(\bar{\Omega}) \mapsto \mathcal{S}_1^0(\mathcal{M})$ is defined by

$$\mathsf{I}_1 u \in \mathcal{S}_1^0(\mathcal{M}) \quad , \quad \mathsf{I}_1 u(\boldsymbol{p}) = u(\boldsymbol{p}) \quad \forall \boldsymbol{p} \in \mathcal{V}(\mathcal{M}) \; .$$

! This is a valid definition only because a function $v_N \in \mathcal{S}_1^0(\mathcal{M})$ is *uniquely determined* by its values in the vertices of the mesh ($\rightarrow$ 3.3.10), which are the interpolation nodes for the linear Lagrangian finite element space.

Recalling the definition of the nodal basis $\mathfrak{B} = \{b_N^{\boldsymbol{p}} : \boldsymbol{p} \in \mathcal{V}(\mathcal{M})\}$ of $\mathcal{S}_1^0(\mathcal{M})$ from (3.3.13), where $b_N^{\boldsymbol{p}}$ is the "tent function" associated with node $\boldsymbol{p}$, an equivalent definition is, *cf.* (3.6.175),

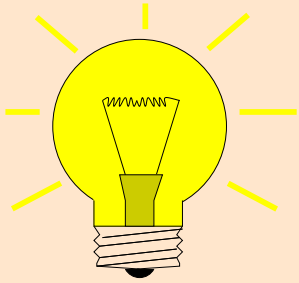
$$\mathsf{I}_1 u = \sum_{\boldsymbol{p} \in \mathcal{V}(\mathcal{M})} u(\boldsymbol{p})\, b_N^{\boldsymbol{p}} \; , \quad u \in C^0(\overline{\Omega}) \; . \tag{5.3.19}$$

Task: ☞ For "sufficiently smooth" $u : \Omega \mapsto \mathbb{R}$ ($\leftrightarrow u \in C^\infty(\bar{\Omega})$ to begin with) *estimate*

interpolation error norm $\|u - \mathsf{I}_1 u\|_{H^1(\Omega)}$.

Interpolation error estimation: localization trick

Again, linear interpolation in 2D according to Def. 5.3.18 is *local* in the sense that $\mathsf{I}_1 u$ inside a triangle K depends only on u on $\overline{K}$.



Idea: Localization

I_1 local ➢ first, estimate $\|u - \mathsf{I}_1 u\|^2_{H^1(K)}$, $K \in \mathcal{M}$,
then, global estimate via summation as in Section 5.3.1.

➢ Focus on single triangle $K \in \mathcal{M}$

Crucial for localization to work: linear interpolation operator $\mathsf{I}_1 : C^0(\bar{\Omega}) \mapsto \mathcal{S}_1^0(\mathcal{M})$ can be defined purely locally by

$$\mathsf{I}_1 u_{|K} = u(\boldsymbol{a}^1)\lambda_1 + u(\boldsymbol{a}^2)\lambda_2 + u(\boldsymbol{a}^3)\lambda_3 \,, \tag{5.3.21}$$

for each triangle $K \in \mathcal{M}$ with vertices $\boldsymbol{a}^1$, $\boldsymbol{a}^2$, $\boldsymbol{a}^3$ ($\lambda_k \hat{=}$ barycentric coordinate functions = local shape functions for $\mathcal{S}_1^0(\mathcal{M})$, see Fig. 99).

(5.3.22) Representation formula for interpolation error

The main steps parallel those for the 1D case in Section 5.3.1 though the technicalities are much more intricate. We start with a representation formula for local interpolation errors, *cf.* (5.3.9). Its derivation solely relies on elementary formulas from calculus.

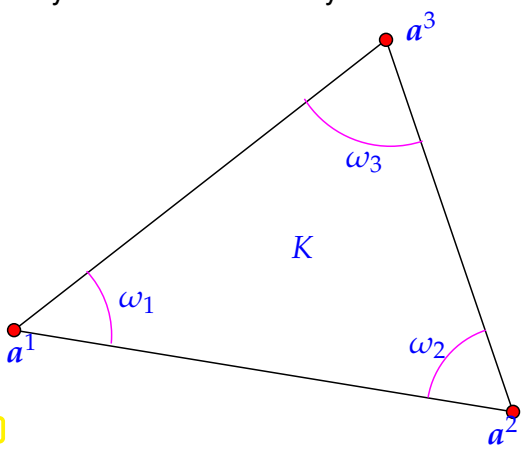


Fig. 218

$u \in C^2(\bar{K})$: by mean value formula $\forall \boldsymbol{x} \in K$,

$$u(\boldsymbol{a}^j) = u(\boldsymbol{x}) + \mathbf{grad}\, u(\boldsymbol{x}) \cdot (\boldsymbol{a}^j - \boldsymbol{x}) + \int_0^1 (\boldsymbol{a}^j - \boldsymbol{x})^\top D^2 u(\boldsymbol{x} + \xi(\boldsymbol{a}^j - \boldsymbol{x}))(\boldsymbol{a}^j - \boldsymbol{x})(1-\xi)\,\mathrm{d}\xi \,, \tag{5.3.23}$$

$$D^2 u(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 u}{\partial x_1{}^2}(\boldsymbol{x}) & \frac{\partial^2 u}{\partial x_1 \partial x_2}(\boldsymbol{x}) \\ \frac{\partial^2 u}{\partial x_1 \partial x_2}(\boldsymbol{x}) & \frac{\partial^2 u}{\partial x_2{}^2}(\boldsymbol{x}) \end{pmatrix} \hat{=} \text{Hessian.}$$

The formula (5.3.23) is easily verified by applying integration by parts (1.3.40) in the form

$$f(b) - f(a) = \left[\xi f'(\xi)\right]_a^b - \int_a^b \xi f''(\xi)\,\mathrm{d}\xi = f'(a)(b-a) + \int_a^b (b-\xi) f''(\xi)\,\mathrm{d}\xi \,.$$

to the function $f(t) = u(t\boldsymbol{a}^j + (1-t)\boldsymbol{x})$ with $a = 0$, $b = 1$. Use the multi-dimensional chain rule to express the derivatives of f through derivatives of u:

$$f'(t) = \mathbf{grad}\, u(t\boldsymbol{a}^j + (1-t)\boldsymbol{x}) \cdot (\boldsymbol{a}^j - \boldsymbol{x}) \,,$$
$$f''(t) = (\boldsymbol{a}^j - \boldsymbol{x})^\top D^2 u(t\boldsymbol{a}^j + (1-t)\boldsymbol{x})(\boldsymbol{a}^j - \boldsymbol{x}) \,.$$

Next, use (5.3.23) to replace $u(\boldsymbol{a}^j)$ in the formula (5.3.21) for local linear interpolation. Also use the identities for the barycentric coordinate functions

$$\sum_{j=1}^{3} \lambda_j(\boldsymbol{x}) = 1 \quad , \quad \boldsymbol{x} = \sum_{j=1}^{3} \boldsymbol{a}^j \lambda_j(\boldsymbol{x}) \; . \tag{5.3.24}$$

$$\mathsf{I}_1 u(\boldsymbol{x}) = \sum_{j=1}^{3} u(\boldsymbol{a}^j)\lambda_j(\boldsymbol{x}) = u(\boldsymbol{x}) \cdot \underbrace{\sum_{j=1}^{3} \lambda_j(\boldsymbol{x})}_{=1} + \mathbf{grad}\, u(\boldsymbol{x}) \cdot \underbrace{\sum_{j=1}^{3} (\boldsymbol{a}^j - \boldsymbol{x})\lambda_j(\boldsymbol{x})}_{=0} + R(\boldsymbol{x}) \; ,$$

with $$R(\boldsymbol{x}) := \sum_{j=1}^{3} \left(\int_0^1 (\boldsymbol{a}^j - \boldsymbol{x})^\top D^2 u(\boldsymbol{x} + \xi(\boldsymbol{a}^j - \boldsymbol{x}))(\boldsymbol{a}^j - \boldsymbol{x})(1-\xi)\,\mathrm{d}\xi \right) \lambda_j(\boldsymbol{x}) \; . \tag{5.3.25}$$

Again, as in the case of (5.3.9) for 1D linear interpolation we have arrived at an integral representation formula for the local interpolation error:

$$\boxed{(u - \mathsf{I}_1 u)(\boldsymbol{x}) = \sum_{j=1}^{3} \left(\int_0^1 (\boldsymbol{a}^j - \boldsymbol{x})^T D^2 u(\boldsymbol{x} + \xi(\boldsymbol{a}^j - \boldsymbol{x}))(\boldsymbol{a}^j - \boldsymbol{x})(1-\xi)\,\mathrm{d}\xi \right) \lambda_j(\boldsymbol{x})} \; . \tag{5.3.26}$$

(5.3.27) Estimate for L^2-norm of interpolation error

Together with the triangle inequality, the trivial bound $|\lambda_j| \leq 1$ yields

$$\|u - \mathsf{I}_1 u\|_{L^2(K)} \leq \sum_{j=1}^{3} \left(\int_K \left(\int_0^1 (\boldsymbol{a}^j - \boldsymbol{x})^T D^2 u(\boldsymbol{x} + \xi(\boldsymbol{a}^j - \boldsymbol{x}))(\boldsymbol{a}^j - \boldsymbol{x})(1-\xi)\,\mathrm{d}\xi \right)^2 \mathrm{d}\boldsymbol{x} \right)^{\frac{1}{2}} \; .$$

To estimate an expression of the form

$$\int_K \left(\int_0^1 (\boldsymbol{a}^j - \boldsymbol{x})^T D^2 u(\boldsymbol{x} + \xi(\boldsymbol{a}^j - \boldsymbol{x}))(\boldsymbol{a}^j - \boldsymbol{x})(1-\xi)\,\mathrm{d}\xi \right)^2 \mathrm{d}\boldsymbol{x} \; , \tag{5.3.28}$$

we may assume, without loss of generality, that $\boldsymbol{a}^j = 0$.

➢ Task: estimate terms (where 0 is a vertex of K!)

$$\int_K \left(\int_0^1 \boldsymbol{x}^\top D^2 u((1-\xi)\boldsymbol{x})\boldsymbol{x}(1-\xi)\,\mathrm{d}\xi \right)^2 \mathrm{d}\boldsymbol{x} = \int_K \left(\int_0^1 \boldsymbol{x}^\top D^2 u(\xi\boldsymbol{x})\boldsymbol{x}\,\xi\,\mathrm{d}\xi \right)^2 \mathrm{d}\boldsymbol{x} \; .$$

Denote $\gamma \;\hat{=}\;$ angle of K at vertex 0,
$h \;\hat{=}\;$ length of longest edge of K.

K is contained in the sector
$S := \{\boldsymbol{x} = \binom{r\cos\varphi}{r\sin\varphi} : 0 \le r < h, 0 \le \varphi \le \gamma\}$

Lemma 5.3.29. Auxiliary estimate on sector

For any $\psi \in L^2(S)$ holds

$$\int_S \left(\int_0^1 |\boldsymbol{y}|^2 \psi(\tau \boldsymbol{y}) \tau \,\mathrm{d}\tau \right)^2 \mathrm{d}\boldsymbol{y} \le \frac{h^4}{8} \|\psi\|^2_{L^2(S)} \,.$$

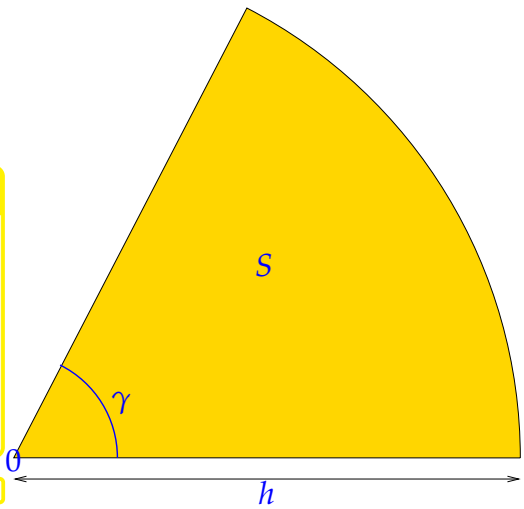


Fig. 219

Using polar coordinates (r, φ), $\widehat{\boldsymbol{s}}_\varphi = \binom{\cos\varphi}{\sin\varphi}$, see [8, Bsp. 8.5.3], and Cauchy-Schwarz inequality (2.3.30):

$$\begin{aligned}
\int_S \left(\int_0^1 |\boldsymbol{y}|^2] \psi(\tau \boldsymbol{y}) \tau \,\mathrm{d}\tau \right)^2 \mathrm{d}\boldsymbol{y} &= \int_0^\gamma \int_0^h \left(\int_0^1 r^2 \psi(\tau r \widehat{\boldsymbol{s}}_\varphi) \tau \,\mathrm{d}\tau \right)^2 r \,\mathrm{d}r \mathrm{d}\varphi \\
&= \int_0^\gamma \int_0^h \left(\int_0^r \psi(\sigma \widehat{\boldsymbol{s}}_\varphi) \sigma \,\mathrm{d}\sigma \right)^2 r \,\mathrm{d}r \mathrm{d}\varphi \le \int_0^\gamma \int_0^h \int_0^r \psi^2(\sigma \widehat{\boldsymbol{s}}_\varphi) \sigma \,\mathrm{d}\sigma \cdot \int_0^r \sigma \,\mathrm{d}\sigma \, r \,\mathrm{d}r \mathrm{d}\varphi \\
&\le \tfrac{1}{2} \int_0^\gamma \int_0^h \psi^2(\sigma \widehat{\boldsymbol{s}}_\varphi) \sigma \,\mathrm{d}\sigma \mathrm{d}\varphi \cdot \int_0^h r^3 \,\mathrm{d}r \,.
\end{aligned}$$

Use $|\boldsymbol{z}^\top \mathbf{A} \boldsymbol{y}| \le \|\mathbf{A}\|_F |\boldsymbol{z}| |\boldsymbol{y}|$, $\mathbf{A} \in \mathbb{R}^{n,n}$, $\boldsymbol{z}, \boldsymbol{y} \in \mathbb{R}^n$, and then apply § 5.3.27 with $\boldsymbol{y} := \boldsymbol{x} - \boldsymbol{a}^j$, $\tau = 1 - \xi$

$$\|u - \mathsf{I}_1 u\|^2_{L^2(K)} \le \tfrac{3}{8} h_K^4 \left\| \left\| D^2 u \right\|_F \right\|^2_{L^2(K)} \,, \tag{5.3.30}$$

with Frobenius matrix norm $\left\| D^2 u(\boldsymbol{x}) \right\|_F^2 := \sum_{i,j=1}^{2} \left| \frac{\partial^2 u}{\partial x_i \partial x_j}(\boldsymbol{x}) \right|^2$ (→ [5, Def. 7.5.37]),

size of triangle $h_K := \operatorname{diam} K := \max\{|\boldsymbol{p} - \boldsymbol{q}| : \boldsymbol{p}, \boldsymbol{q} \in K\}$

(5.3.31) Estimate of local H^1-seminorm of the interpolation error

Estimate for gradient: from (5.3.23) we infer the local integral representation formula, which can also be obtained by taking the gradient of (5.3.26).

$$\begin{aligned}
\mathbf{grad}\, \mathsf{I}_1 u(\boldsymbol{x}) &= \sum_{j=1}^{3} u(\boldsymbol{a}^j) \,\mathbf{grad}\, \lambda_j(\boldsymbol{x}) \\
&= \sum_{j=1}^{3} \left(u(\boldsymbol{x}) + \mathbf{grad}\, u(\boldsymbol{x}) \cdot (\boldsymbol{a}^j - \boldsymbol{x}) + \int_0^1 \ldots \,\mathrm{d}\xi \right) \mathbf{grad}\, \lambda_j(\boldsymbol{x}) \\
&= u(\boldsymbol{x}) \underbrace{\sum_{j=1}^{3} \mathbf{grad}\, \lambda_j(\boldsymbol{x})}_{=0} + \underbrace{\sum_{j=1}^{3} (\boldsymbol{a}^j - \boldsymbol{x})^\top \mathbf{grad}\, \lambda_j(\boldsymbol{x})}_{=\mathbf{I}} \cdot \mathbf{grad}\, u(\boldsymbol{x}) + G(\boldsymbol{x}) \,,
\end{aligned}$$

$$\text{with} \quad G(\boldsymbol{x}) := \sum_{j=1}^{3} \underbrace{\left(\int_0^1 (\boldsymbol{a}^j - \boldsymbol{x})^\top D^2 u(\boldsymbol{x} + \xi(\boldsymbol{a}^j - \boldsymbol{x}))(\boldsymbol{a}^j - \boldsymbol{x})(1-\xi)\,\mathrm{d}\xi \right)}_{\textit{cf.}\ (5.3.28)} \mathbf{grad}\,\lambda_j(\boldsymbol{x})\,.$$

Note that $\mathbf{grad} \sum_{j=1}^{3} \lambda_j(\boldsymbol{x}) = \mathbf{grad}\,1 = 0$ and

$$\sum_{j=1}^{3} \mathbf{grad}\,\lambda_j(\boldsymbol{x})(\boldsymbol{a}^j - \boldsymbol{x})^\top = \sum_{j=1}^{3} \mathbf{grad}\,\lambda_j(\boldsymbol{x})(\boldsymbol{a}^j)^\top = \mathbf{grad}\big(\sum_{j=1}^{3} \lambda_j(\boldsymbol{x})\boldsymbol{a}^j\big) = \mathbf{grad}\,\boldsymbol{x} = \mathbf{I}\,. \tag{5.3.32}$$

As an immediate consequence of the formulas from Section 3.3.5

$$\begin{aligned}
\mathbf{grad}\,\lambda_1 &= -\frac{|e_1|}{2|K|}\boldsymbol{n}^1 = \frac{1}{2|K|}(\boldsymbol{a}^2 - \boldsymbol{a}^3)^\perp = \frac{1}{2|K|}\begin{pmatrix} a_2^2 - a_2^3 \\ a_1^3 - a_1^2 \end{pmatrix}, \\
\mathbf{grad}\,\lambda_2 &= -\frac{|e_2|}{2|K|}\boldsymbol{n}^2 = \frac{1}{2|K|}(\boldsymbol{a}^3 - \boldsymbol{a}^1)^\perp = \frac{1}{2|K|}\begin{pmatrix} a_2^3 - a_2^1 \\ a_1^1 - a_1^3 \end{pmatrix}, \\
\mathbf{grad}\,\lambda_3 &= -\frac{|e_3|}{2|K|}\boldsymbol{n}^3 = \frac{1}{2|K|}(\boldsymbol{a}^1 - \boldsymbol{a}^2)^\perp = \frac{1}{2|K|}\begin{pmatrix} a_2^1 - a_2^2 \\ a_1^2 - a_1^1 \end{pmatrix},
\end{aligned}$$

we conclude

$$(3.6.119) \quad \blacktriangleright \quad \boxed{|\,\mathbf{grad}\,\lambda_j(\boldsymbol{x})| \le \frac{h_K}{2|K|}\,, \quad \boldsymbol{x} \in K}\,. \tag{5.3.33}$$

$$\blacktriangleright \quad \|\mathbf{grad}(u - \mathsf{I}_1 u)\|^2_{L^2(K)} \le \frac{h_K^2}{4|K|^2}\|R\|^2_{L^2(K)} \overset{(5.3.30)}{\le} \tfrac{3}{8}\frac{h_K^6}{4|K|^2}\Big\|\big\|D^2 u\big\|_F\Big\|^2_{L^2(K)}\,. \tag{5.3.34}$$

Summary of *local* interpolation error estimates for linear interpolation according to Def. 5.3.18:

Lemma 5.3.35. Local interpolation error estimates for 2D linear interpolation

For any triangle K and $u \in C^2(\overline{K})$ the following holds

$$\|u - \mathsf{I}_1 u\|^2_{L^2(K)} \le \tfrac{3}{8}h_K^4\Big\|\big\|D^2 u\big\|_F\Big\|^2_{L^2(K)}\,, \tag{5.3.30}$$

$$\|\mathbf{grad}(u - \mathsf{I}_1 u)\|^2_{L^2(K)} \le \tfrac{3}{24}\frac{h_K^6}{|K|^2}\Big\|\big\|D^2 u\big\|_F\Big\|^2_{L^2(K)}\,. \tag{5.3.34}$$

(5.3.36) Shape regularity

Note: the estimates of Lemma 5.3.35 are structurally similar to the 1D estimates (5.3.13) and (5.3.16) in the sense that the bounds involve L^2-norms of second derivatives and factors depending on the cell size.

New aspect compared to Section 5.3.1: *shape* of K enters error bounds of Lemma 5.3.35.

This dependence on shape can be reduced to a single number:

Definition 5.3.37. Shape regularity measure

For a simplex $K \in \mathbb{R}^d$ we define its shape regularity measure as the ratio

$$\rho_K := h_K^d : |K| \ ,$$

and the shape regularity measure of a simplicial mesh $\mathcal{M} = \{K\}$

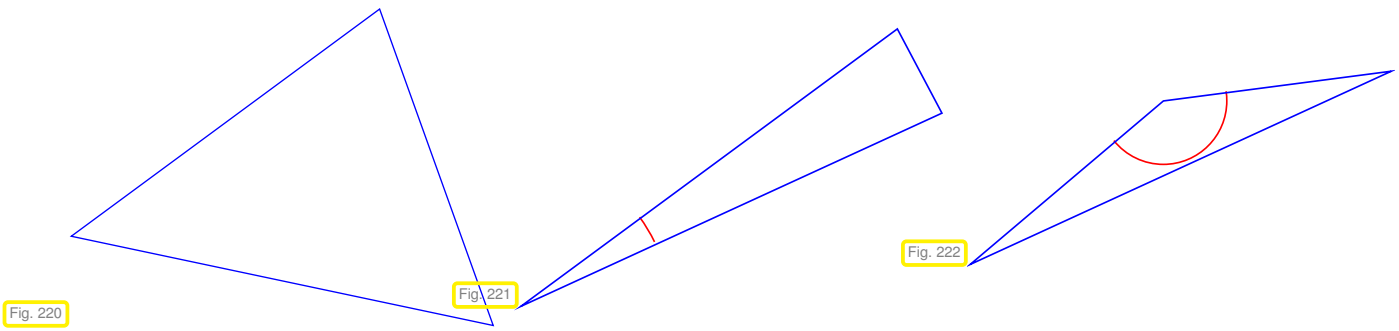
$$\rho_{\mathcal{M}} := \max_{K \in \mathcal{M}} \rho_K \ .$$

Important: shape regularity measure ρ_K is an invariant of a similarity class of triangles.

(= if a triangle is transformed by scaling, rotation, and translation, the shape regularity measure does not change)

➢ Sloppily speaking, ρ_K depends only on the shape, not the size of K

For triangle K: ρ_K large $\Leftrightarrow$ K "distorted" $\Leftrightarrow$ K has small angles



ρ_k small ρ_k large ρ_k large

The shape regularity measure $\rho_{\mathcal{M}}$ is often used to gauge the *quality* of meshes produced by mesh generators.

Now we return to estimates for norms of the interpolation error for piecewise linear interpolation I_1. The final step is to add up the local estimates from Lemma 5.3.35 over all triangles of the mesh and take the square root.

Theorem 5.3.38. 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 ($\to$ Def. 5.2.3) and $\rho_{\mathcal{M}}$ the shape regularity measure ($\to$ Def. 5.3.37) of $\mathcal{M}$.

Remark 5.3.39 (Energy norm and $H^1(\Omega)$-norm)

Objection! Well, Cea's lemma Thm. 5.1.15 refers to the energy norm, but Thm. 5.3.38 provides estimates in $H^1(\Omega)$-norm only!

☞ For uniformly positive definite ($\rightarrow$ Def. 2.2.18) and bounded coefficient tensor $\boldsymbol{\alpha} : \Omega \mapsto \mathbb{R}^{d,d}$, *cf.* (2.2.17),

$$\exists 0 < \alpha^- < \alpha^+ : \quad \alpha^- \|\boldsymbol{z}\|^2 \leq \boldsymbol{z}^T \boldsymbol{\alpha}(\boldsymbol{x}) \boldsymbol{z} \leq \alpha^+ \|\boldsymbol{z}\|^2 \quad \forall \boldsymbol{z} \in \mathbb{R}^d, \boldsymbol{x} \in \Omega ,$$

and the energy norm ($\rightarrow$ Def. 2.2.43) induced by

$$\mathsf{a}(u,v) := \int_\Omega (\boldsymbol{\alpha}(\boldsymbol{x}) \,\mathbf{grad}\, u) \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} , \quad u,v \in H_0^1(\Omega) , \tag{5.1.7}$$

we immediately find the equivalence (= two-sided uniform estimate)

$$\sqrt{\alpha^-} |v|_{H^1(\Omega)} \leq \|v\|_a \leq \sqrt{\alpha^+} |v|_{H^1(\Omega)} \quad \forall v \in H^1(\Omega) . \tag{5.3.40}$$

Thus, interpolation error estimates in $|\cdot|_{H^1(\Omega)}$ immediately translate into estimates in terms of the energy norm (with bounds for the coefficient entering the constants).

5.3.3 The Sobolev Scale of Function Spaces

Interpolation error estimates like in Thm. 5.3.38 hinge on smoothness of the interpoland u: the bounds in Thm. 5.3.38 the term $\left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)}$. This norm conveys two messages:

- $\left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)} < \infty$ is a smoothness requirement.
- The size of $\left\| \left\| D^2 u \right\|_F \right\|_{L^2(\Omega)}$ is a measure for the smoothness of u.

In this section we take a closer look at norms involving derivatives and their capability to indicate the smoothness of a function. In fact, in the guise of the $H^1(\Omega)$-seminorm from Def. 2.3.23 we have already come across an example for such a norm. Thus, what we pursue in this section can also be regarded as a generalization of $H^1(\Omega)$.

Definition 5.3.41. Higher order Sobolev spaces/norms

The m-th order Sobolev norm, $m \in \mathbb{N}_0$, for $u : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ (sufficiently smooth) is defined by

$$\|u\|_{H^m(\Omega)}^2 := \sum_{k=0}^{m} \sum_{\boldsymbol{\alpha} \in \mathbb{N}^d, |\boldsymbol{\alpha}|=k} \int_\Omega |D^{\boldsymbol{\alpha}} u|^2 \,\mathrm{d}\boldsymbol{x} , \quad \text{where} \quad D^{\boldsymbol{\alpha}} u := \frac{\partial^{|\boldsymbol{\alpha}|} u}{\partial x_1^{\alpha_1} \cdots \partial x_d^{\alpha_d}} .$$

Sobolev space $\quad H^m(\Omega) := \{ v : \Omega \mapsto \mathbb{R} : \|v\|_{H^m(\Omega)} < \infty \}$.

To understand this definition recall the multi-index notation (3.4.9), (3.4.10)

(5.3.42) Purposes of Sobolev spaces

Gripe (as in Section 2.3): Don't bother me with these Sobolev spaces !

Response: Well, these concepts are pervasive in the numerical analysis literature and you have to be familiar with them, in particular, with the notations.

Reassuring: Again, it is only the norms $\|u\|_{H^m(\Omega)}$ that matter for us !

Now, we have come across an additional purpose of Sobolev spaces and their norms:

provide framework for variational formulation of elliptic BVP ($\rightarrow$ Section 2.3)	◀	Sobolev spaces	▶	provide norms $\|\cdot\|_{H^m(\Omega)}$ that measure strength of singularities of functions

---

Sobolev scale: $\ldots \subset H^3(\Omega) \subset H^2(\Omega) \subset H^1(\Omega) \subset L^2(\Omega)$

Observation: bounds in Thm. 5.3.38 = "principal parts" of Sobolev norms, that is, the parts containing the highest partial derivatives.

Definition 5.3.43. Higher order Sobolev semi-norms

The m-th order Sobolev semi-norm, $m \in \mathbb{N}$, for sufficiently smooth $u : \Omega \mapsto \mathbb{R}$ is defined by

$$|u|^2_{H^m(\Omega)} := \sum_{\alpha \in \mathbb{N}^d, |\alpha|=m} \int_\Omega |D^\alpha u|^2 \, \mathrm{d}x \, .$$

Elementary observation: $|p|_{H^m(\Omega)} = 0 \quad \Leftrightarrow \quad p \in \mathcal{P}_{m-1}(\mathbb{R}^d)$

▶ By density arguments we can rewrite the interpolation error estimates of Thm. 5.3.38 in terms of Sobolev semi-norms:

Corollary 5.3.44. Error estimate for piecewise linear interpolation in 2D

Under the assumptions/with notations of Thm. 5.3.38

$$\left.\begin{aligned} \|u - \mathsf{I}_1 u\|_{L^2(\Omega)} &\leq \sqrt{\tfrac{3}{8}}\, h^2_{\mathcal{M}} |u|_{H^2(\Omega)} \, , \\ |u - \mathsf{I}_1 u|_{H^1(\Omega)} &\leq \sqrt{\tfrac{3}{24}}\, \rho_{\mathcal{M}}\, h_{\mathcal{M}} |u|_{H^2(\Omega)} \, , \end{aligned}\right. \quad \forall u \in H^2(\Omega) \, .$$

Remark 5.3.45 (Continuity of interpolation operators)

An interpolation operator like I_1 maps functions to functions; it represents a *linear* mapping (= operator) between two function spaces. If we specify norms on these spaces, we may ask whether this mapping is continuous in the sense of the following definition, which generalizes Def. 2.2.56.

Definition 5.3.46. Continuous linear operator

A linear mapping $\mathsf{T} : X \to Y$ between two normed vector spaces X and Y is continuous or bounded, if and only if

$$\exists C > 0 : : \quad \|\mathsf{T}v\|_Y \leq C\|v\|_X \quad \forall v \in X \,.$$

In order to investigate the continuity of I_1 apply the $\triangle$-inequality to the estimates of Cor. 5.3.44:

$$\|\mathsf{I}_1 u\|_{L^2(\Omega)} \leq \|u\|_{L^2(\Omega)} + \sqrt{\tfrac{3}{8}}\, h_{\mathcal{M}}^2 |u|_{H^2(\Omega)} \leq 2\|u\|_{H^2(\Omega)} \,, \tag{5.3.47}$$

if lengths are scaled such that $h_{\mathcal{M}} \leq 1$. In light of Def. 5.3.46 estimate (5.3.47) means that $\mathsf{I}_1 : H^2(\Omega) \mapsto L^2(\Omega)$ is a continuous linear mapping.

The same conclusion could have been drawn from the following fundamental result:

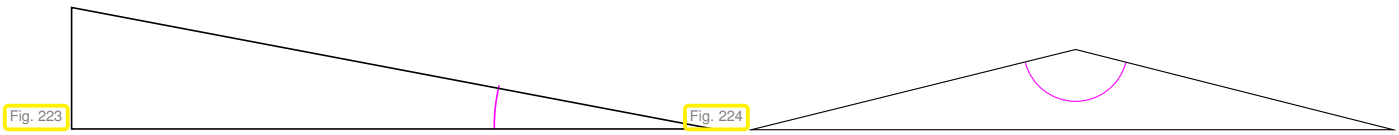
Theorem 5.3.48. Sobolev embedding theorem

$$m > \frac{d}{2} \quad \Rightarrow \quad H^m(\Omega) \subset C^0(\overline{\Omega}) \quad \wedge \quad \exists C = C(\Omega) > 0 : \quad \|u\|_\infty \leq C\|u\|_{H^m(\Omega)} \quad \forall u \in H^m(\Omega) \,.$$

Yet, for $d > 1$ the nodal interpolation operator $\mathsf{I}_1 : H^1(\Omega) \mapsto L^2(\Omega)$ is **not** continuous, as we learn from Ex. 2.4.22.

5.3.4 Anisotropic interpolation error estimates

Look at the following triangular cells with "bad shape regularity" (ρ_K "large"): very small/large angles:



The estimates of Lemma 5.3.35 might suggest that we face huge local interpolation errors, once ρ_K becomes large.

Issue: are the estimates of Lemma 5.3.35 *sharp* **?**

We will try to find this out experimentally by computing the best possible constants in the estimates

$$\|u - \mathsf{I}_1 u\|_{L^2(K)} \leq C_{K,2} h_k^2 \|u\|_{H^2(K)} \,, \quad \|u - \mathsf{I}_1 u\|_{H^1(K)} \leq C_K h_K \|u\|_{H^2(K)} \,.$$

Note: Merely translating, rotating, or scaling K does not affect the constants $C_{K,2}$ and C_K. Therefore, we can restrict ourselves to "canonical triangles". Every general triangle can be mapped to one of these by translating, rotating, and scaling.

$$C_{K,2} := \sup_{u \in H^2(K)\setminus\{0\}} \frac{\|u - \mathsf{I}_1 u\|_{L^2(K)}}{\|u\|_{H^2(K)}} \,, \quad C_K := \sup_{u \in H^2(k)\setminus\{0\}} \frac{\|u - \mathsf{I}_1 u\|_{H^1(K)}}{\|u\|_{H^2(K)}} \,,$$

on triangle $K := \operatorname{convex}\left\{ \binom{0}{0}, \binom{1}{0}, \binom{p_x}{p_y} \right\}$.

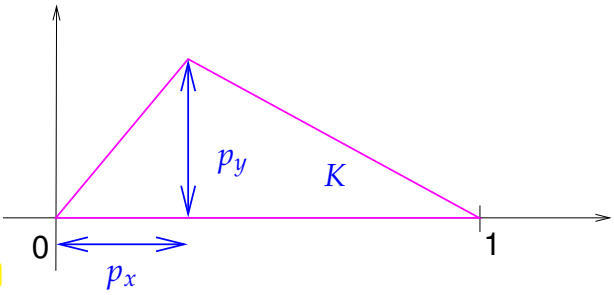


Fig. 225

Sampling the space of "canonical" triangles (modulo similarity)

$$0 \le p_x, p_y \le 1 \,.$$

+ Numerical computation of C_K, $C_{K,2}$ implementation by A. Inci (spectral polynomial Galerkin method)

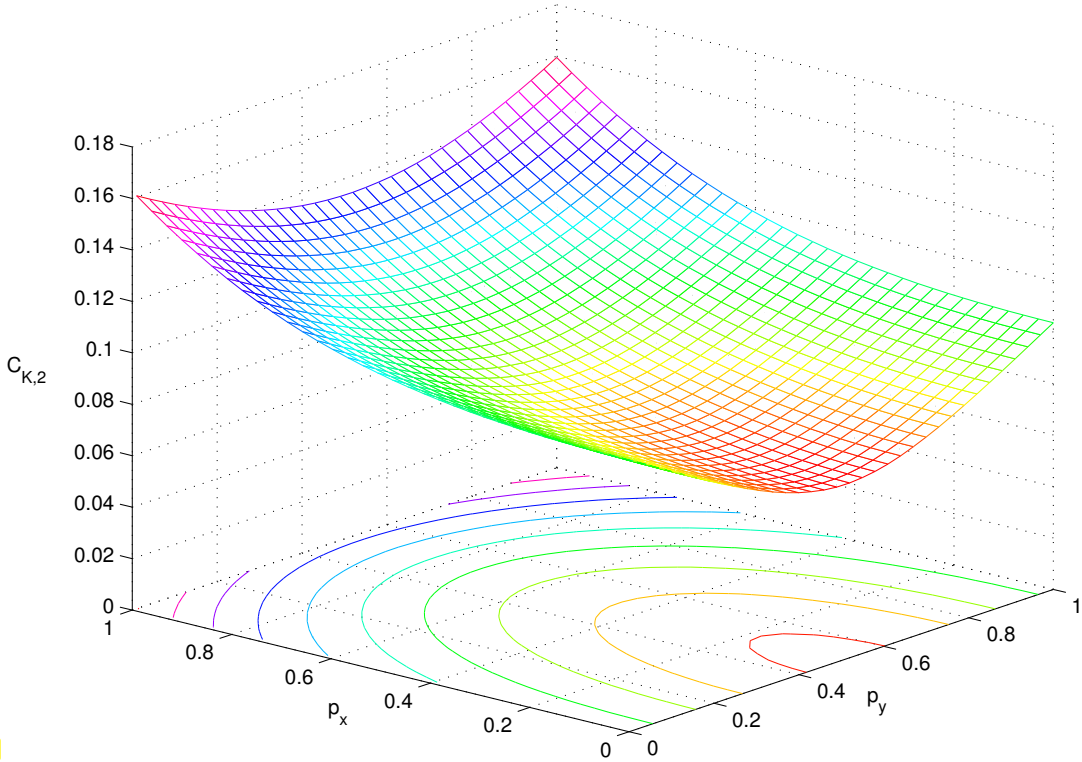


Fig. 226

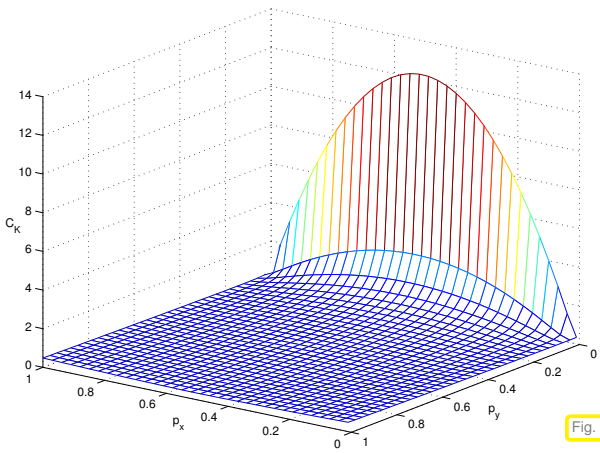


Fig. 227

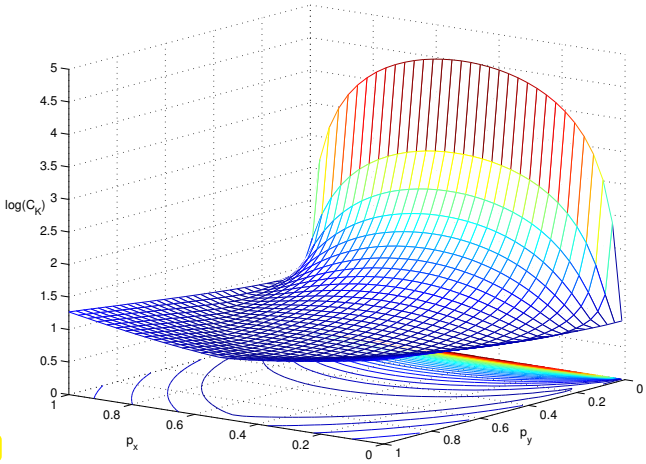


Fig. 228

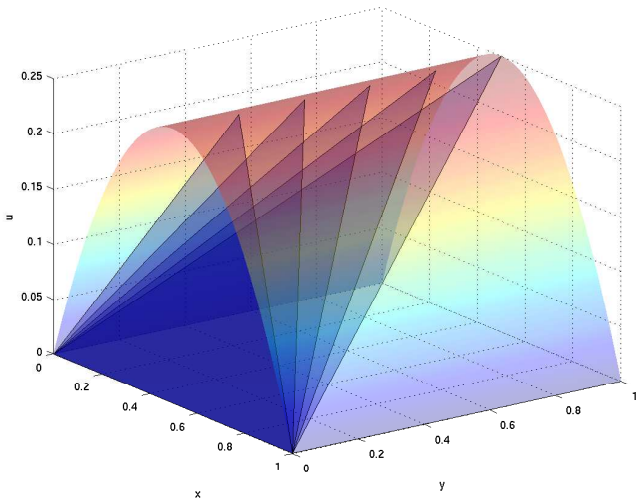


Fig. 229

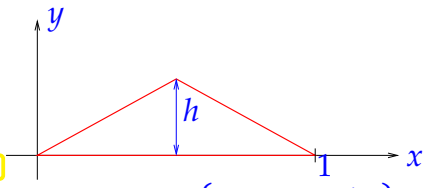


Fig. 230

triangle $K := \operatorname{convex}\left\{\binom{0}{0}, \binom{1}{0}, \binom{1/2}{h}\right\}$, $h > 0$, $u(x,y) = x(1-x)$, $0 < x < 1$.

◁ linear interpolant of u on K as $h \to 0$

The interpolant becomes steeper and steeper as $h \to 0$:

$$\|u\|^2_{H^2(K)} = \frac{3031}{1440}h\,, \quad \|u - \mathsf{I}_1 u\|^2_{H^1(K)} = \frac{29}{2880}h + \frac{1}{12}h + \frac{1}{32}h^{-1}\,, \quad \|u - \mathsf{I}_1 u\|^2_{L^2(K)} = \frac{29}{2889}h$$

$$\frac{\|u - \mathsf{I}_1 u\|^2_{H^1(K)}}{\|u\|^2_{H^2(K)}} \geq \frac{269}{6062} + \frac{45}{3031}h^{-2} \quad , \quad \frac{\|u - \mathsf{I}_1 u\|^2_{L^2(K)}}{\|u\|^2_{H^2(K)}} = \frac{29}{6062}\,.$$

Experiment 5.3.49 (Good accuracy on "bad" meshes)

$\Omega =]0,1[^2$, $u(x_1,x_2) = \sin(\pi x_1)\sin(\pi x_2)$, BVP $-\Delta u = f$, $u_{|\partial\Omega} = 0$, finite element Galerkin discretization on triangular meshes, $V_N = \mathcal{S}^0_{1,0}(\mathcal{M})$.

☞ meshes created by random distortion of tensor product grids

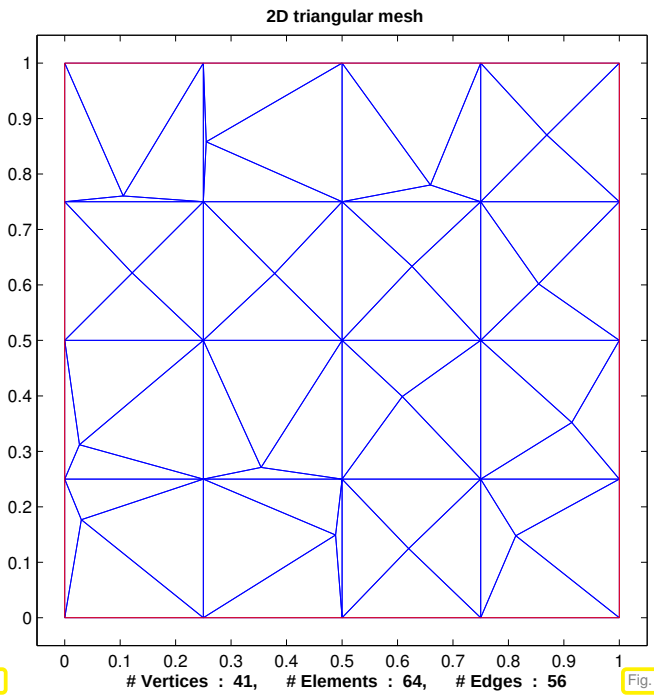


Fig. 231

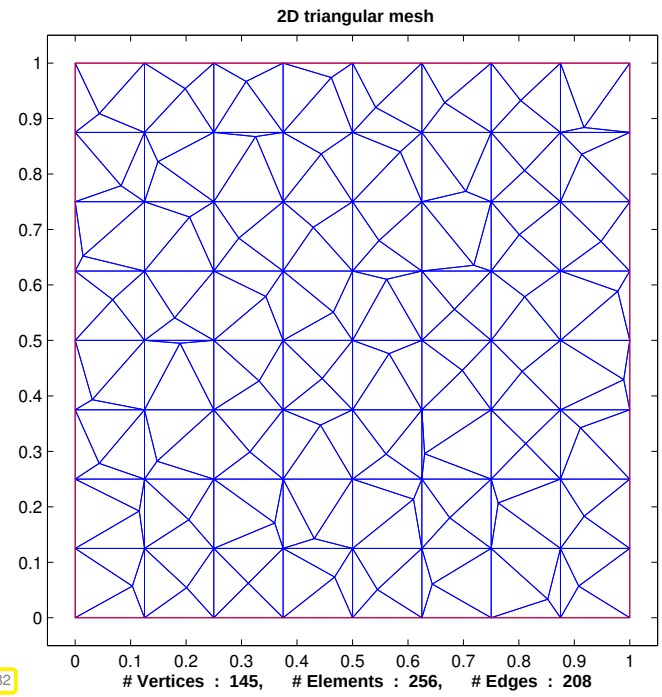


Fig. 232

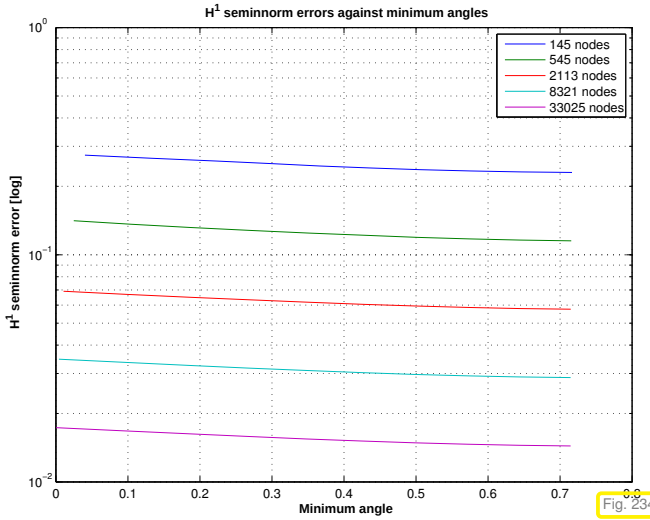


Fig. 233

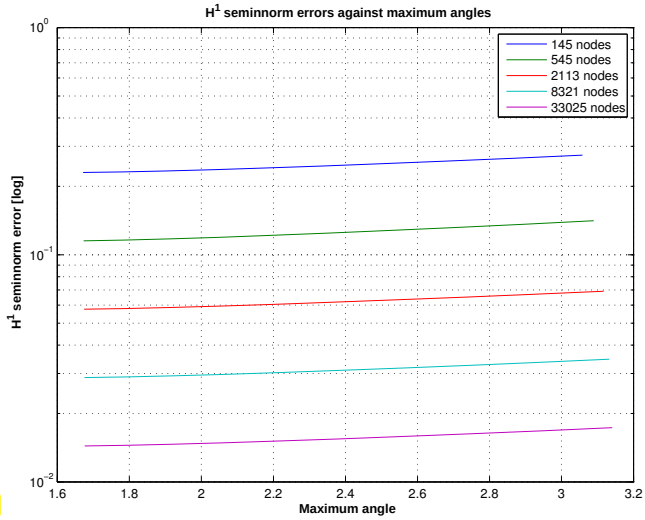


Fig. 234

Monitored: for different mesh resolutions, $H^1(\Omega)$-seminorm of discretization error as function of smallest/largest angle in the mesh.

Observation: Accuracy does *not* suffer much from distorted elements !

Remark 5.3.50 (Gap between interpolation error and best approximation error)

Exp. 5.3.49 raises doubts whether the interpolation error can be trusted to provide good, that is, reasonably sharp bounds for the best approximation error.

In this example we will see that

$$\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_N\|_1 \ll \|u - \mathsf{I}_p u\|_{H^1(\Omega)} \quad \text{is possible !}$$

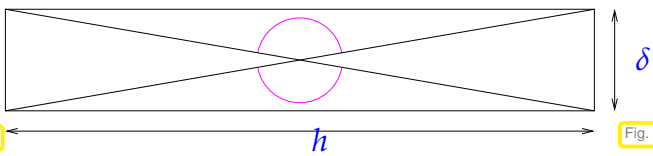


Fig. 235

Elementary cell of "bad mesh" $\mathcal{M}_{\text{bad}}$

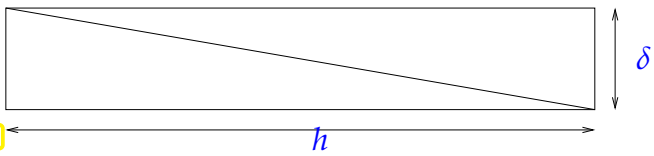


Fig. 236

Elementary cell of "good mesh" $\mathcal{M}_{\text{good}}$

$$\text{On "bad" mesh} \quad : \quad \sup_{u \in H^2(\Omega)} \frac{\|u - \mathsf{I}_1 u\|_{H^1(\Omega)}}{\|u\|_{H^2(\Omega)}} \to \infty \text{ as } h/\delta \to \infty,$$

$$\text{On "good" mesh} : \quad \sup_{u \in H^2(\Omega)} \frac{\|u - \mathsf{I}_1 u\|_{H^1(\Omega)}}{\|u\|_{H^2(\Omega)}} \quad \text{uniformly bounded in } h/\delta.$$

$$\text{Yet,} \quad \inf_{v_N \in \mathcal{S}_1^0(\mathcal{M}_{\text{bad}})} \|u - v_N\|_{H^1(\Omega)} \leq \inf_{v_N \in \mathcal{S}_1^0(\mathcal{M}_{\text{good}})} \|u - v_N\|_{H^1(\Omega)} \quad \forall u \in H^2(\Omega) \ .$$

5.3.5 General approximation error estimates

In Section 5.3.2 we only examined the behavior of norms of the interpolation error for piecewise linear interpolation into $\mathcal{S}_1^0(\mathcal{M})$, that is, the case of Lagrangian finite elements of degree $p = 1$.

However, Exp. 5.2.7 sent the clear message that quadratic Lagrangian finite elements achieve faster convergence of the energy norm of the Galerkin discretization error, see Fig. 200,Fig. 201.

$$\Updownarrow$$

On the other hand quadratic finite elements could not deliver faster convergence in Exp. 5.2.10.

In this section we learn about theoretical results that shed light on these observations and extend the results of Section 5.3.2.

(5.3.51) L^∞ interpolation error estimate in 1D

The faster convergence of quadratic Lagrangian FE in Exp. 5.2.7 does not come as a surprise: recall the estimate from [5, Eq. (4.5.12)]:

$$\left\|u - \mathsf{I}_p u\right\|_{L^\infty([a,b])} \leq \frac{h_{\mathcal{M}}^{p+1}}{(p+1)!}\left\|u^{(p+1)}\right\|_{L^\infty([a,b])} \quad \forall u \in C^{p+1}([a,b]) \ ,$$

where $\mathsf{I}_p u$ is the $\mathcal{M}$-piecewise polynomial interpolant of u of local degree p. It generalizes (5.3.7), where this result was stated for $p = 1$.

➢ $$\left\|u - \mathsf{I}_p u\right\|_{L^\infty([a,b])} = O(h_{\mathcal{M}}^{p+1}) \ !$$

(5.3.52) Local interpolation onto higher degree Lagrangian finite element spaces

$\mathcal{M}$: triangular/tetrahedral/quadrilateral/hybrid mesh of domain Ω ($\rightarrow$ Section 3.4.1)

Recall ($\rightarrow$ Section 3.5): nodal basis functions of p-th degree Lagrangian finite element space $\mathcal{S}_p^0(\mathcal{M})$ defined via interpolation nodes, *cf.* (3.5.4).

Set of interpolation nodes: $\mathcal{N} = \{\boldsymbol{p}_1, \dots, \boldsymbol{p}_N\} \subset \overline{\Omega}$, $N = \dim \mathcal{S}_p^0(\mathcal{M})$.

➢ General nodal Lagrangian interpolation operator (agrees with I_1 from Def. 5.3.18 for $p = 1$)

$$\mathsf{I}_p : \begin{cases} C^0(\overline{\Omega}) & \mapsto \quad \mathcal{S}_p^0(\mathcal{M}) \\ u & \mapsto \quad \mathsf{I}_p(u) := \sum_{l=1}^{N} u(\boldsymbol{p}_l) b_N^l \end{cases} , \tag{5.3.53}$$

where b_N^l are the nodal basis functions.

(3.5.4) $\Rightarrow$ $\mathsf{I}_p(u)(\boldsymbol{p}_l) = u(\boldsymbol{p}_l)$, $l = 1, \dots, N$ (Interpolation property!) .

By virtue of the location of the interpolation nodes, see Ex. 3.5.3, Ex. 3.5.7, and Fig. 138, the nodal interpolation operators are purely local:

$$\forall K \in \mathcal{M}: \quad \mathsf{I}_p u_{|K} = \sum_{i=1}^{Q} u(\mathbf{q}_i^K)\, b_K^i \ , \tag{5.3.54}$$

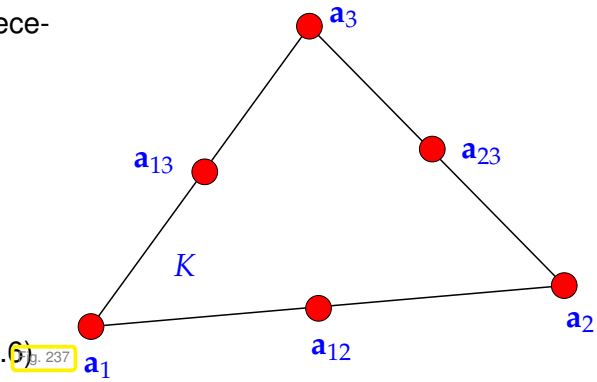
$\boldsymbol{q}_i^K$, $i = 1,\dots,Q$ = local interpolation nodes in cell $K \in \mathcal{M}$, see Ex. 3.5.3, Ex. 3.5.7, and Fig. 138,
b_i^K, $i = 1,\dots,Q$ = local shape functions: $b_i^K(\boldsymbol{q}_j^K) = \delta_{ij}$.

Example 5.3.55 (Piecewise quadratic interpolation $\rightarrow$ Ex. 3.5.3)

For a triangle $K = \operatorname{convex}\{\boldsymbol{a}^1, \boldsymbol{a}^2, \boldsymbol{a}^3\}$ and $p = 2$ the piecewise quadratic interpolation operator on K is given by

$$\begin{aligned} \mathsf{I}_2 u_{|K} = & \quad -\sum_{i=1}^{3} \lambda_i(1-2\lambda_i)\, u(\mathbf{a}^i) \\ & + \sum_{1\le i<j\le 3} 4\lambda_i\lambda_j\, u(\tfrac{1}{2}(\mathbf{a}^i + \mathbf{a}^j)) \ . \end{aligned}$$

local shape functions, see (3.5.6)



The following theorem summarizes best approximation results for affine equivalent Lagrangian FE spaces $\mathcal{S}_p^0(\mathcal{M})$ ($\rightarrow$ Section 3.5) on mesh $\mathcal{M}$ of a bounded polygonal/polyhedral domain $\Omega \subset \mathbb{R}^d$. It is the result of many years of research in approximation theory, see [7, Sect. 3.3], [1].

Theorem 5.3.56. 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

$$\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_N\|_{H^1(\Omega)} \le C \left(\frac{h_{\mathcal{M}}}{p}\right)^{\min\{p+1,k\}-1} \|u\|_{H^k(\Omega)} \quad \forall u \in H^k(\Omega) \ . \tag{5.3.57}$$

This theorem is a typical example of finite element analysis results that you can find in the literature. It is important to know what kind of information can be gleaned from statements like that of Thm. 5.3.56.

Remark 5.3.58 ("Generic constants")

A statement like (5.3.57) is typical of a priori error estimates in the numerical analysis literature, which often come in the form

$$\|u - u_N\|_X \le C \cdot \text{"discretization parameter"} \cdot \|u\|_Y \ ,$$

where

- $C > 0$ is not specified precisely or only claimed to exist ("there is", though, in principle, they could be computed),
- C must neither depend on the exact solution u nor the discrete solution u_N,
- the possible dependence of C on problem parameters or discretization parameters has to stated unequivocally.

Such constants $C > 0$ are known as generic constants. Customarily, different generic constants are even denoted by the same symbol ("C" is most common). The use of generic constants is an alternative to the Landau-O notation (1.6.25).

(5.3.59) Nature of a priori estimates $\rightarrow$ **Section 1.6.2**

In combination with Cea's lemma (Thm. 5.1.15) Thm. 5.3.56 implies a priori estimates of the energy norm of the finite element Galerkin discretization error (see also Rem. 5.3.39) of the form

$$\|u - u_N\|_{\mathsf{a}} \leq C \left(\frac{h_{\mathcal{M}}}{p}\right)^{\min\{p+1,k\}-1} \|u\|_{H^k(\Omega)} \,, \tag{5.3.60}$$

where u is the exact solution of the discretized 2nd-order elliptic boundary value problem.

(5.3.60) does not give concrete information about $\|u - u_N\|_{\mathsf{a}}$, because

- ✦ we do not know the value of the "generic constant" $C > 0$, see Rem. 5.3.58,
- ✦ as u is unknown, a bound for $\|u\|_{H^k(\Omega)}$ may not be available.

A priori error estimates like (5.3.60) exhibit only the *trend* of the (norm of) the discretization error as discretization parameters $h_{\mathcal{M}}$ (mesh width), p (polynomial degree) are varied.

Supplement 5.3.61.

The estimate of Thm. 5.3.56 is *sharp*: the powers of $h_{\mathcal{M}}$ and p cannot be increased. △

(5.3.62) The message of asymptotic a priori convergence estimates

What questions can Thm. 5.3.56 and (5.3.60) answer? What do they tell us about the accuracy and *efficiency* of a Lagrangian finite element Galerkin discretization of a 2nd-order elliptic BVP**?** Closely related discussions have been have developed for numerical quadrature, see [5, Section 5.4], and higher order single step methods for initial value problems from ODEs, see [5, Rem. 11.4.1]. You are advised to review these passages in order to understand the parallels.

Question 5.3.63. *What computational effort buys us what error (measured in energy norm)?*

Bad luck ($\rightarrow$ § 5.3.59): actual error norm remains elusive! Therefore, rephrase the question so that it fits the available information about the effect of changing discretization parameters on the error:

Question 5.3.64. *What* increase *in computational effort buys us a prescribed* decrease *of the (energy norm of the) error?*

The answer to this question offers an a priori gauge of the *asymptotic efficiency* of a discretization method.

Convention:	computational effort $\approx$ number of unknowns $N = \dim \mathcal{S}_p^0(\mathcal{M})$ (problem size)

(5.3.65) The price of a finite element space

Framework: family $\mathbb{M}$ of simplicial meshes of domain $\Omega \subset \mathbb{R}^d$, $d = 1,2,3$, created by *global* regular refinement of a single initial mesh: h-refinement

Important specimens of such families of meshes are provided by sequences of simplicial meshes created by global regular refinement ($\rightarrow$ Ex. 5.1.20). This refinement rule has distinct benefits:

- it avoids greater distortion of "child cells" w.r.t. their parents,
- it spawns meshes with fairly uniform size h_K of cells.

A mathematical way to express these insights:

$$\begin{array}{lll} \text{uniform shape-regularity}: & \exists C > 0: & \rho_{\mathcal{M}} \leq C \,, \\ \text{local quasi-uniformity} & \exists C > 0: & \max\{h_K/h_{K'},\ K,K' \in \mathcal{M}\} \leq C \,, \end{array} \qquad \forall \mathcal{M} \in \mathbb{M} \,.$$

Now, for meshes $\in \mathbb{M}$, we investigate "N-dependence", $N = \dim \mathcal{S}_p^0(\mathcal{M})$, of energy norm of finite element discretization error:

Counting argument $\quad N = \dim \mathcal{S}_p^0(\mathcal{M}) \approx p^d h_{\mathcal{M}}^{-d} \quad \Rightarrow \quad \boxed{\dfrac{h_{\mathcal{M}}}{p} \approx N^{-1/d}}$. (5.3.66)

dimensions of local spaces, Lemma 3.4.11 $\qquad \sim \sharp\mathcal{M} \sim \sharp\mathcal{V}(\mathcal{M}), \mathcal{E}(\mathcal{M})$ etc.

Notation: $\approx \;\hat{=}\;$ uniform equivalence on the set $\mathbb{M}$, that is, each side can be bounded by a constant times the other, and the constants can be chosen independently of the mesh $\mathcal{M} \in \mathbb{M}$

(5.3.67) Dimensions of Lagrangian finite element spaces on triangular meshes

$d = 2$: for triangular meshes $\mathcal{M}$, by Lemma 3.4.11

$$\dim \mathcal{S}_p^0(\mathcal{M}) = \sharp\{\text{nodes}(\mathcal{M})\} + \sharp\{\text{edges}(\mathcal{M})\}\,(p-1) + \sharp\mathcal{M}\,\tfrac{1}{2}(p-1)(p-2) \,.$$

1 basis function per vertex

$p-1$ basis functions per edge

$\frac{1}{2}(p-1)(p-2)$ "interior" basis functions

Geometric considerations: the number of triangles sharing a vertex can be bounded in terms of $\rho_{\mathcal{M}}$, because $\rho_{\mathcal{M}}$ implies a lower bound for the smallest angles of the triangular cells.

$$\exists C = C(\rho_{\mathcal{M}}): \quad \sharp\{K_j \in \mathcal{M}: \overline{K}_i \cap \overline{K}_j \neq \emptyset\} \leq C \quad (i = 1,2,\ldots,\#\mathcal{M}) \,.$$

If every vertex belongs only to a small number of triangles, the number $\sharp\{\text{nodes}(\mathcal{M})\}$ can be bounded by $C \cdot \sharp\mathcal{M}$, where $C > 0$ will depend on $\rho_{\mathcal{M}}$ only. The same applies to the edges.

$$\Rightarrow \quad \sharp\{\text{nodes}(\mathcal{M})\}, \sharp\{\text{edges}(\mathcal{M})\} \;\approx\; \sharp\mathcal{M} \,.$$

$$\Downarrow$$

$$\dim \mathcal{S}_p^0(\mathcal{M}) \approx (\sharp\mathcal{M})p^2 \,, \tag{5.3.68}$$

with constants hidden in $\approx$ depending on $\rho_{\mathcal{M}}$ only.

Now, we merge (5.3.60) and (5.3.66):

$$\boxed{u \in H^k(\Omega)} \overset{\text{Thm. 5.3.56}}{\Rightarrow} \boxed{\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_N\|_{H^1(\Omega)} \leq CN^{-\frac{\min\{p,k-1\}}{d}} \|u\|_{H^k(\Omega)}} \,, \tag{5.3.69}$$

with $C > 0$ depending *only* on d, k, and $\rho_{\mathcal{M}}$.

Convergence of best approximation error for Lagrangian finite elements

(5.3.69) ➢ Energy norm of the discretization error features algebraic convergence ($\rightarrow$ Def. 1.6.24) in the problem size (= number of unknowns) with a

$$\text{rate} \quad = \frac{\min\{p, k-1\}}{d}.$$

We observe that

- ✦ the rate of convergence is limited by the polynomial degree p of the Lagrangian FEM,
- ✦ the rate of convergence is limited by the smoothness of the exact solution u, measured by means of the Sobolev index k, see Section 5.3.3,
- ✦ the rate of convergence will be worse for $d = 3$ than for $d = 2$, the effect being more pronounced for small k or p.

(5.3.71) Asymptotic efficiency of Lagrangian finite elements

Now we answer Question 5.3.64 ("What increase in computational effort buys us a prescribed decrease of the (energy norm of the) error?"):

Assumption: a priori error estimate (5.3.69) is *sharp*

$$\exists C = C(u, \ldots) > 0: \quad \text{error norm}(N) \approx CN^{-\frac{\min\{p,k-1\}}{d}} \quad \forall \mathcal{M} \in \mathbb{M} \,.$$

$$\blacktriangleright \quad \frac{\text{error norm}(N_1)}{\text{error norm}(N_2)} \approx \left(\frac{N_1}{N_2}\right)^{-\frac{\min\{p,k-1\}}{d}} .$$

$\blacktriangleright$ reduction of (the energy norm of) the error by a factor $\rho > 1$ requires increase of the problem size by factor $\rho^{\frac{d}{\min\{p,k-1\}}}$

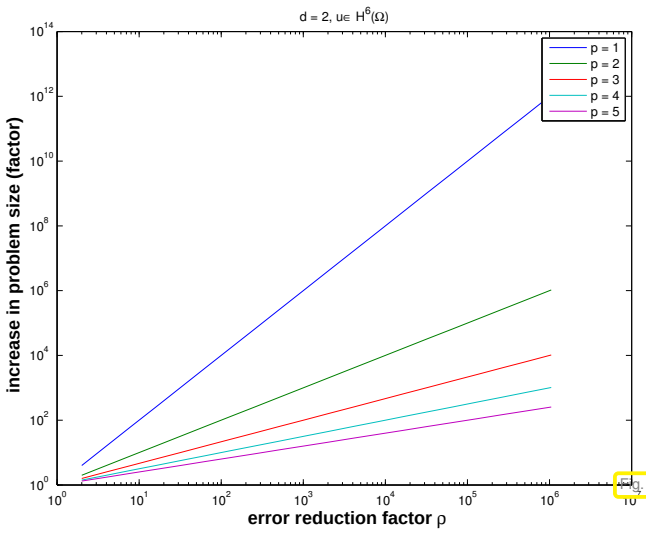


Fig. 238

exact solution $u \in H^6(\Omega)$

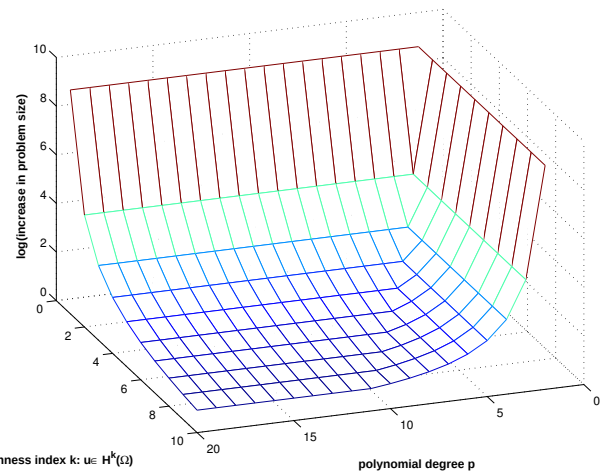


Fig. 239

error reduction by factor $\rho = 100$

Discussion: Solution $u \in H^k(\Omega)$ ➢ optimal asymptotic efficiency for $p = k - 1$

(Here, $u \in H^k(\Omega)$ is supposed to be sharp in the sense that we can**not** take for granted $u \in H^{k+1}(\Omega)$.)

Remark 5.3.72 (General asymptotic estimates)

Recall (→ Section 1.6.2): convergence is an asymptotic notion

Now we deduce asymptotic estimates for the best approximation errors from Thm. 5.3.56, and (5.3.69), in particular, for the case $N \to \infty$, where N is the dimension of the finite element space:

For the exact solution u we assume: $u \in H^k(\Omega)$, $k \in \mathbb{N}$.

- h-refinement: p fixed, $h_{\mathcal{M}} \to 0$ for $\mathcal{M} \in \mathbb{M}$:

  (5.3.69) ⇒ algebraic convergence w.r.t. N

  ☞ $p \leq k - 1$ ▶

$$\boxed{\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_N\|_1 = O(N^{-p/d})} \tag{5.3.73}$$

  Here the polynomial degree of the Lagrangian finite elements limits the rate of algebraic convergence.

  ☞ $k \leq p + 1$ ▶

$$\boxed{\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_N\|_1 = O(N^{-(k-1)/d})} \tag{5.3.74}$$

  Here, the smoothness (measured in the Sobolev scale) is the limiting factor for the rate of algebraic convergence.

  Note: for very smooth solution u, i.e. $k \gg 1$, polynomial degree p limits speed of convergence

- p-refinement: $\mathcal{M} \in \mathbb{M}$ fixed, $p \to \infty$:

☞ p large ►

$$\boxed{\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} \|u - v_N\|_1 = O(N^{-(k-1)/d})} \tag{5.3.75}$$

Note: arbitrarily fast (super-)algebraic convergence for very smooth solutions $u \in C^\infty(\overline{\Omega})$. (However, the exponential convergence observed in Exp. 5.2.11 is not captured by the approximation error estimates of Thm. 5.3.56.)

?! *Review question(s) 5.3.76.* (Convergence of finite element solutions)

1. What does the statement "Exponentially convergent Galerkin schemes are better than algebraically convergent methods" allude to?
2. It is known that the solution u of a scalar 2nd-order elliptic boundary value problem belongs to $H^2(\Omega)$, but fails to be contained in $H^3(\Omega)$.
   (a) Describe the convergence of the $H^1(\Omega)$-norm of the discretization error one can expect from a finite element Galerkin discretization by means of degree p, $p \in \mathbb{N}$, Lagrangian finite elements on a sequence of meshes obtained by regular refinement.
   (b) Which convergence of $\|u - u_N\|_{H^1(\Omega)}$ in terms of polynomial degree p will probably be observed for finite element solutions $u_N \in \mathcal{S}_p^0(\mathcal{M})$, $\mathcal{M}$ fixed, and increasing p?
3. Appealing to Thm. 8.4.42 explain why for a bounded polygonal domain $\Omega \subset \mathbb{R}^2$ and $\mathsf{I}_1 : C^0(\overline{\Omega}) \to \mathcal{S}_1^0(\mathcal{M})$ denoting the nodal interpolation operator according to Def. 5.3.18 the *interpolation* error estimate
   $$\|u - \mathsf{I}_1 u\|_{L^2(\Omega)} \leq C h_{\mathcal{M}} \|u\|_{H^1(\Omega)} \quad \forall u \in H^1(\Omega) \,,$$
   with $C > 0$ depending only on the shape regularity measure $\rho_{\mathcal{M}}$ of a triangular mesh $\mathcal{M}$ can**not** be true.
4. If $\mathcal{M}'$ has been created by regular refinement of a triangular mesh $\mathcal{M}$, how are $\rho_{\mathcal{M}}$ and $\rho_{mesh'}$ related?
5. For which exponents $a > 0$ does the function $x \mapsto x^\alpha$ belong to the Sobolev space $H^m(]0,1[)$, $m \in \mathbb{N}$?

5.4 Elliptic Regularity Theory

Crudely speaking, in Section 5.3.5 we saw that the asymptotic behavior of the Lagrangian finite element Galerkin discretization error (for 2nd-order elliptic BVPs) can be predicted provided that

- we use families of meshes, whose cells have rather uniform size and whose shape regularity measure is uniformly bounded,

- we have an *idea about the smoothness of the exact solution* u, that is, we know $u \in H^k(\Omega)$ for a (maximal) k, see Thm. 5.3.56.

Knowledge about the mesh can be taken for granted, but

how can we guess the smoothness of the (unknown !) exact solution u **?**

A (partial) answer is given in this section.

Focus: Scalar 2nd-order elliptic BVP with homogeneous Dirichlet boundary conditions

$$-\operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega \,.$$

To begin with, we summarize the available information:

➢ Known: $\underbrace{\boxed{u \text{ solves BVP}} \;+\; \boxed{\text{Information about coefficient } \sigma\text{, domain } \Omega\text{, source function } f\text{, boundary data } g}}$

u will belong to a certain class of functions (e.g. subspace $S \subset H^1(\Omega)$)

Example 5.4.1 (Elliptic lifting result in 1D)

$d = 1$, $\Omega =]0,1[$, coefficient $\sigma \equiv 1$, homogeneous Dirichlet boundary conditions:

$$u'' = f \quad , \quad u(0) = u(1) = 0 \,.$$

Obvious from Def. 5.3.41: $\quad f \in H^k(\Omega) \quad \Rightarrow \quad u \in H^{k+2}(\Omega)$ (a lifting theorem)

Can this be generalized to higher dimensions $d > 1$**?**

Partly so:

Theorem 5.4.2. 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 $k \in \mathbb{N}$,

$$\left.\begin{array}{l} u \in H^1_0(\Omega) \quad \text{and} \quad -\operatorname{div}(\sigma\,\mathbf{grad}\,u) \in H^k(\Omega) \\ u \in H^1(\Omega)\,,\; -\operatorname{div}(\sigma\,\mathbf{grad}\,u) \in H^k(\Omega) \text{ and } \quad \mathbf{grad}\,u \cdot \mathbf{n} = 0 \text{ on } \partial\Omega \end{array}\right. \Rightarrow \quad u \in H^{k+2}(\Omega)\,.$$

In addition, for such u there is $C = C(k,\Omega,\sigma)$ such that

$$\|u\|_{H^{k+2}(\Omega)} \le C\|\operatorname{div}(\sigma\,\mathbf{grad}\,u)\|_{H^k(\Omega)}\,.$$

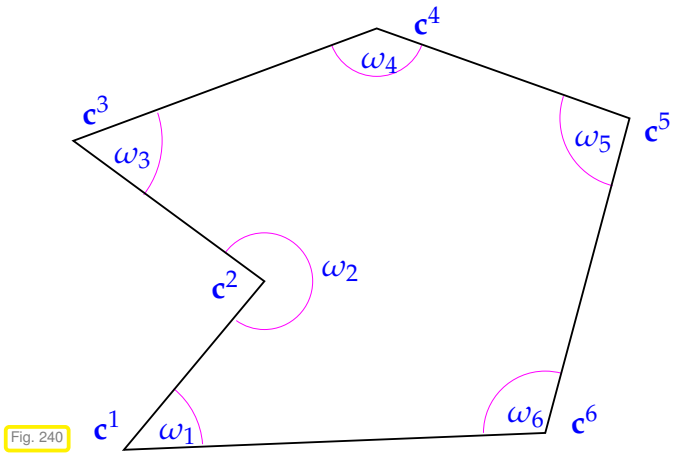


Fig. 240

What about non-smooth $\partial\Omega$ **?**
These are very common in engineering applications ("CAD-geometries").

◁ polygonal domain with corners $\boldsymbol{c}^i$

How will the corners affect the smoothness of solutions of

$$u \in H_0^1(\Omega): \quad \Delta u = f \in C^\infty(\overline{\Omega})?$$

Example 5.4.3 (Corner singular functions)

This example answers some of the questions asked above by exhibiting locally harmonic functions satisfying homogeneous Dirichlet boundary conditions that, nevertheless, feature a singularity at a corner of the domain:

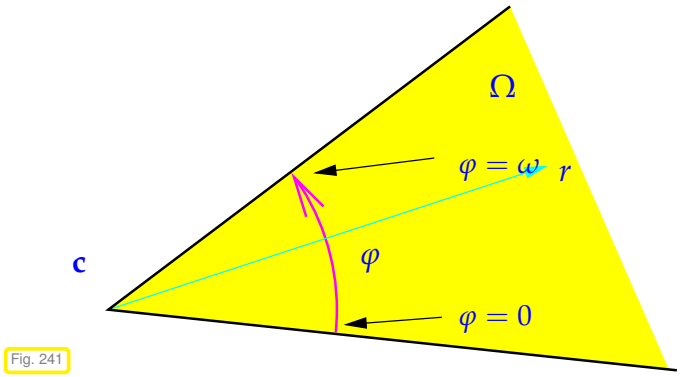


Fig. 241

corner singular function

$$u_s(r,\varphi) = r^{\frac{\pi}{\omega}} \sin(\frac{\pi}{\omega}\varphi)\,, \qquad r \geq 0\,, \quad 0 \leq \varphi \leq \omega\,. \tag{5.4.4}$$

(in local polar coordinates)

▶ $u_s = 0$ on $\partial\Omega$ locally at $\boldsymbol{c}$!

Straightforward computation (in polar coordinates): $\Delta u_s = 0$ in Ω **!**

To see this recall: Δ in polar coordinates:

$$\Delta u = \frac{1}{r}\frac{\partial}{\partial r}(r\frac{\partial u}{\partial r}) + \frac{1}{r^2}\frac{\partial^2 u}{\partial \varphi^2}\,. \tag{5.4.5}$$

$$\overset{(5.4.4)}{\Longrightarrow} \quad \Delta u_s(r,\varphi) = \frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\pi}{\omega}r^{\frac{\pi}{\omega}-1}\sin(\frac{\pi}{\omega}\varphi)\right) + \frac{1}{r^2}r^{\frac{\pi}{\omega}}\frac{\partial}{\partial\varphi}\cos(\frac{\pi}{\omega}\varphi)\frac{\pi}{\omega}$$
$$= \left(\frac{\pi}{\omega}\right)^2 r^{\frac{\pi}{\omega}-2}\sin(\frac{\pi}{\omega}\varphi) - \left(\frac{\pi}{\omega}\right)^2 r^{\frac{\pi}{\omega}-2}\sin(\frac{\pi}{\omega}\varphi) = 0\,.$$

What is "singular" about these functions? Plot them for $\omega = \frac{3\pi}{2}$, *cf.* Exp. 5.2.10

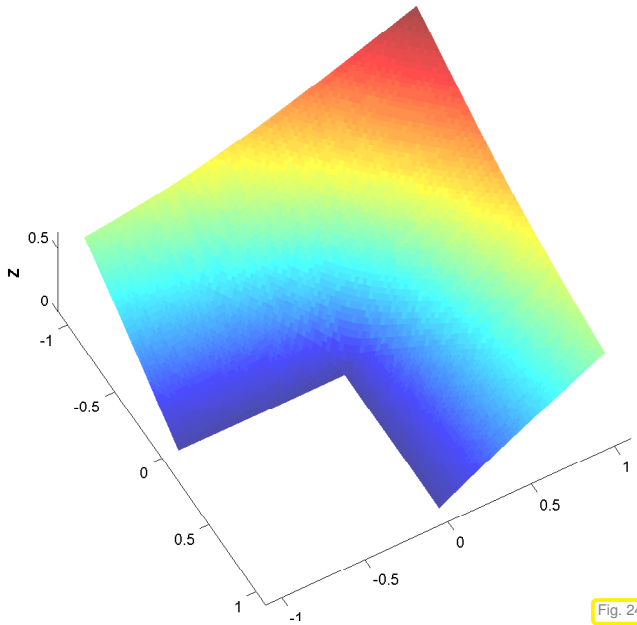


Fig. 242

u_s for $\omega = \frac{3\pi}{2}$

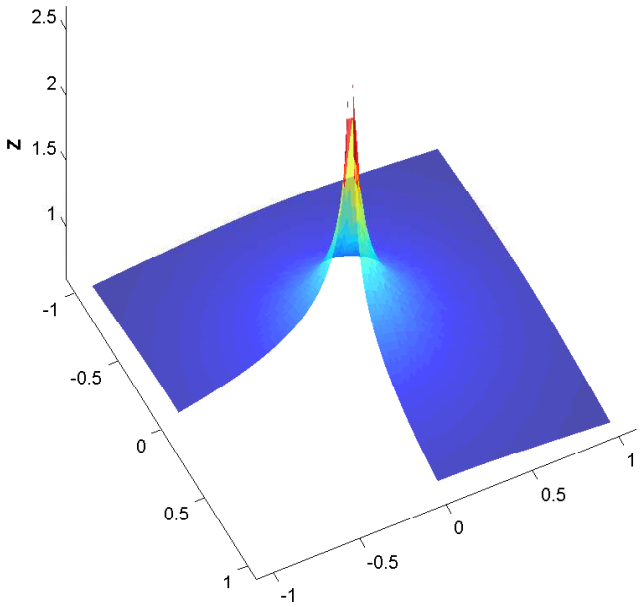


Fig. 243

$\|\mathbf{grad}\, u_s\|$ for $\omega = \frac{3\pi}{2}$

Recall gradient (2.4.26) in polar coordinates

$$\mathbf{grad}\, u = \frac{\partial u}{\partial r}\mathbf{e}_r + \frac{1}{r}\frac{\partial u}{\partial \varphi}\mathbf{e}_\varphi \,. \tag{2.4.26}$$

$$\overset{(5.4.4)}{\Longrightarrow} \quad \mathbf{grad}\, u_s(r,\varphi) = \frac{\pi}{\omega} r^{\frac{\pi}{\omega}-1}\left(\sin(\tfrac{\pi}{\omega}\varphi)\mathbf{e}_r + \cos(\tfrac{\pi}{\omega}\varphi)\right)\mathbf{e}_\varphi \,.$$

$\omega > \pi$ ("re-entrant corner") $\Longrightarrow$ "$\mathbf{grad}\, u_s(0) = \infty$"

How does this "blow-up" of the gradient affect the Sobolev regularity (that is, the smoothness as expressed through "$u_s \in H^k(\Omega)$") of the corner singular function u_s**?**

We try to compute $|u|_{H^2(D)}$, with (in polar coordinates, see Fig. 241)

$$D := \{(r,\varphi) : 0 < r < 1,\, 0 < \varphi < \omega\} \,.$$

By tedious computations we find

$$\omega > \pi \quad \Rightarrow \quad \int_D \left\| D^2 u_s(r,\varphi) \right\|_F^2 r \mathrm{d}(r,\varphi) = \infty \,.$$

$$\overset{\text{Def. 5.3.41}}{\Longrightarrow} \quad \left\{ \quad \omega > \pi \quad \Rightarrow \quad u_s \notin H^2(D) \quad \right\} .$$

Bad news: With the exception of "concocted/manufactured" examples,

> corner singular functions like (5.4.4) will be present in the solution of linear scalar 2nd-order elliptic BVP on polygonal domains!

The meaning of "being present" is elucidated in the following theorem:

Theorem 5.4.6. Corner singular function decomposition

Let $\Omega \subset \mathbb{R}^2$ be a polygon with J corners $\mathbf{c}^i$. Denote the polar coordinates in the corner $\mathbf{c}^i$ by (r_i, φ_i) and the inner angle at the corner $\mathbf{c}^i$ by ω_i. Additionally, let $f \in H^l(\Omega)$ with $l \in \mathbb{N}_0$ and $l \neq \lambda_{ik} - 1$, where the λ_{ik} are given by the singular exponents

$$\lambda_{ik} = \frac{k\pi}{\omega_i} \quad \text{for } k \in \mathbb{N} \,. \tag{5.4.7}$$

Then $u \in H_0^1(\Omega)$ with $-\Delta u = f$ in Ω can be decomposed

$$u = u^0 + \sum_{i=1}^{J} \psi(r_i) \sum_{\lambda_{ik} < l+1} \kappa_{ik}\, s_{ik}(r_i, \varphi_i) \,, \quad \kappa_{ik} \in \mathbb{R} \quad , \tag{5.4.8}$$

with regular part $u^0 \in H^{l+2}(\Omega)$, cut-off functions $\psi \in C^\infty(\mathbb{R}^+)$ ($\psi \equiv 1$ in a neighborhood of 0), and corner singular functions

$$\begin{aligned} \lambda_{ik} \notin \mathbb{N}: &\quad s_{ik}(r,\varphi) = r^{\lambda_{ik}} \sin(\lambda_{ik}\varphi) \,, \\ \lambda_{ik} \in \mathbb{N}: &\quad s_{ik}(r,\varphi) = r^{\lambda_{ik}} (\ln r) \sin(\lambda_{ik}\varphi). \end{aligned} \tag{5.4.9}$$

$\Omega \subset \mathbb{R}^2$ has re-entrant corners	$\Rightarrow$	if u solves $\Delta u = f$ in Ω, $u = 0$ on $\partial\Omega$, then $u \notin H^2(\Omega)$ in general.

Theorem 5.4.10. Elliptic lifting theorem on convex domains [?, Thm. 3.2.1.2]

[Elliptic lifting theorem on convex domains] If $\Omega \subset \mathbb{R}^d$ convex, $u \in H_0^1(\Omega)$, $\Delta u \in L^2(\Omega)$ $\Rightarrow$ $u \in H^2(\Omega)$.

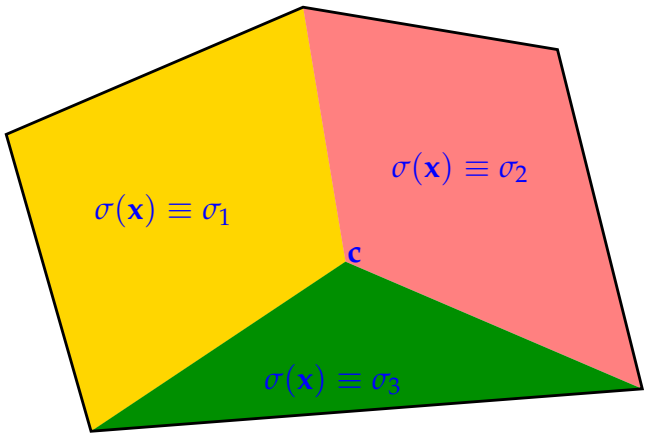
Terminology: if conclusion of Thm. 5.4.10 true $\rightarrow$ Dirichlet problem 2-regular.

Similar lifting theorems also hold for Neumann BVPs, BVPs with *smooth* coefficients.

(5.4.11) Causes for non-smoothness of solutions of elliptic BVPs

Causes for poor Sobolev regularity of solution u of BVPs for $-\operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\, u) = f$:

- Corners of $\partial\Omega$, see above
- Discontinuities of σ
  $\rightarrow$ singular functions at "material corners"
- Mixed boundary conditions
- Non-smooth source function f



"material corner" at $\mathbf{c}$

?! *Review question(s) 5.4.12.* (Elliptic regularity)

1. Compute the gradient of the corner singular function from (5.4.4) in Cartesian coordinates.
2. Exhibit a corner singular function u_n that satisfies, on the wedge shaped domain Ω displayed in Fig. 241, $\Delta u_n = 0$ and $\mathbf{grad}\, u_n \cdot \boldsymbol{n} = 0$ on $\partial\Omega$ (close to the corner $\boldsymbol{c}$). Use polar coordinates.
3. Consider the one-dimensional boundary value problem $u'' = f$ in $]0,1[$, $u(0) = u(1) = 0$. Which Sobolev regularity will the weak solution $u \in H_0^1(]0,1[)$ posses, if f is piecewise smooth with a single discontinuity?

5.5 Variational Crimes

Variational crime = instead of solving (exact) discrete (linear) variational problem

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = f(v_N) \quad \forall v_N \in V_{0,N}\,, \tag{3.2.8}$$

we solve the perturbed variational problem

$$\widetilde{u}_N \in V_{0,N}: \quad \mathsf{a}_N(\widetilde{u}_N, v_N) = f_N(v_N) \quad \forall v_N \in V_{0,N}\,. \tag{5.5.1}$$

this causes a perturbation of Galerkin solution u_N and we end up with a perturbed solution $\widetilde{u}_N \in V_{0,N}$.

Approximations $\mathsf{a}_N(\cdot,\cdot) \approx \mathsf{a}(\cdot,\cdot)$, $f_N(\cdot) \approx f(\cdot)$ are usually due to

- the use of numerical quadrature $\rightarrow$ Section 3.6.5,
- an approximation of the oundary $\partial\Omega$ $\rightarrow$ Section 3.7.4.

We are all sinners! Variational crimes are *inevitable* in practical FEM, recall Rem. 1.5.5!

Which "variational petty crimes" can be tolerated**?**

Guideline for acceptable variational crimes

Variational crimes must not affect (type and rate) of asymptotic convergence!

This requirement must be met for *all* boundary value problems the finite element methods has been designed to solve, in particular, for problems with smooth solutions, for which maximal rates of algebraic convergence can be achieved (→ Rem. 5.3.72).

Hence, when probing the impact of variational crimes in a numerical experiment, always choose test problems with smooth solutions.

5.5.1 Impact of numerical quadrature

Model problem: on polygonal/polyhedral $\Omega \subset \mathbb{R}^d$:

$$u \in H_0^1(\Omega): \quad \mathsf{a}(u,v) := \int_\Omega \sigma(\mathbf{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = f(v) := \int_\Omega f v \,\mathrm{d}\boldsymbol{x}\,. \tag{5.5.3}$$

Assumptions: σ satisfies (2.6.6), $\sigma \in C^0(\overline{\Omega})$, $f \in C^0(\overline{\Omega})$

- Galerkin finite element discretization, $V_N := \mathcal{S}_p^0(\mathcal{M})$ on simplicial mesh $\mathcal{M}$
- Approximate evaluation of $\mathsf{a}(u_N, v_N)$, $f(v_N)$ by a fixed stable local numerical quadrature rule (→ Section 3.6.5)

  ➤ perturbed bilinear form a_N, right hand side f_N (see (5.5.1))

Focus: h-refinement (key discretization parameter is the mesh width $h_\mathcal{M}$)

Experiment 5.5.4 (Impact of numerical quadrature on finite element discretization error)

$\Omega =]0,1[^2$, $\sigma \equiv 1$, $f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$, $(x,y)^T \in \Omega$

➤ solution $u(x,y) = \sin(\pi x)\sin(\pi y)$, $g = 0$.

Details of numerical experiment:

- *Quadratic* Lagrangian FE ($V_N = \mathcal{S}_2^0(\mathcal{M})$) on triangular meshes $\mathcal{M}$, obtained by regular refinement
- "Exact" evaluation of bilinear form by very high order quadrature
- f_N from one point quadrature rule (3.6.161) *of order 2*

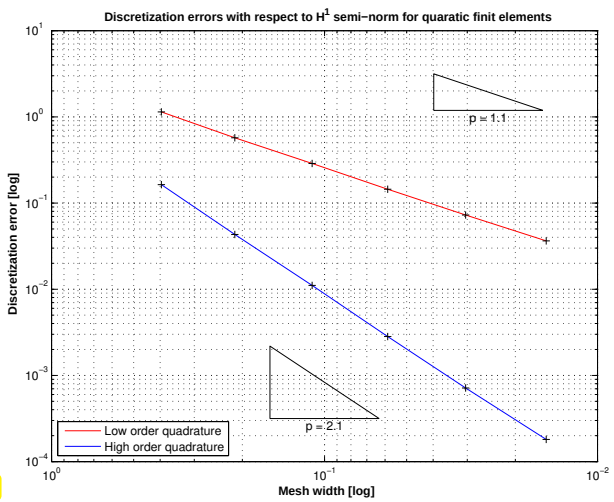


Fig. 244

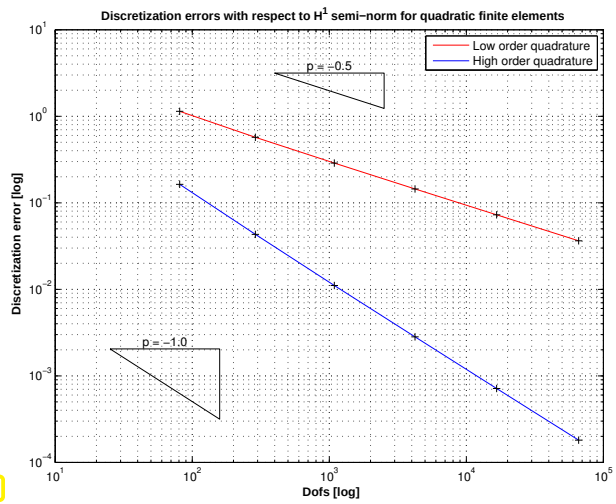


Fig. 245

$H^1(\Omega)$-norm of discretization error on unit square ($-$ $\leftrightarrow$ rule (3.6.161), $-$ $\leftrightarrow$ rule (3.6.162))

Observation: Use of quadrature rule of order 2 $\Rightarrow$ Algebraic rate of convergence (w.r.t. N) drops from $\alpha = 1$ to $\alpha = 1/2$!

Finite element theory [2, Ch. 4,§4.1] tells us that the above guideline can be met, if the local numerical quadrature rule has sufficiently high order. The quantitative results can be condensed into the following rules of thumb:

$\|u - u_N\|_1 = O(h_{\mathcal{M}}^p)$ at best	▶	Quadrature rule of order $2p-1$ sufficient for right hand side functional f_N.
$\|u - u_N\|_1 = O(h_{\mathcal{M}}^p)$ at best	▶	Quadrature rule of order $2p-1$ sufficient for bilinear form a_N.

5.5.2 Approximation of boundary

We focus on 2nd-order scalar linear variational problems as in the previous section.

Experiment 5.5.5 (Impact of linear boundary approximation on FE convergence)

Setting: $\Omega := B_1(0) := \{\mathbf{x} \in \mathbb{R}^2 : |\mathbf{x}| < 1\}$, $u(r,\varphi) = \cos(r\pi/2)$ (polar coordinates)
➢ $f = \frac{\pi}{2r}\sin(r\pi/2) + \frac{\pi}{2}\cos(r\pi/2)$

- Sequences of unstructured triangular meshes $\mathcal{M}$ obtained by regular refinement (of coarse mesh with 4 triangles) + linear boundary fitting.
- Galerkin FE discretization based on $V_N := \mathcal{S}_{1,0}^0(\mathcal{M})$ or $V_N := \mathcal{S}_{2,0}^0(\mathcal{M})$.
- Recorded: approximate norm $|u - u_N|_{1,\Omega_h}$, evaluated using numerical quadrature rule (3.6.162).

  (FE solution extended beyond the domain covered by $\mathcal{M}$ ("mesh interior") to Ω ("full domain") by means of polynomial extrapolation.)

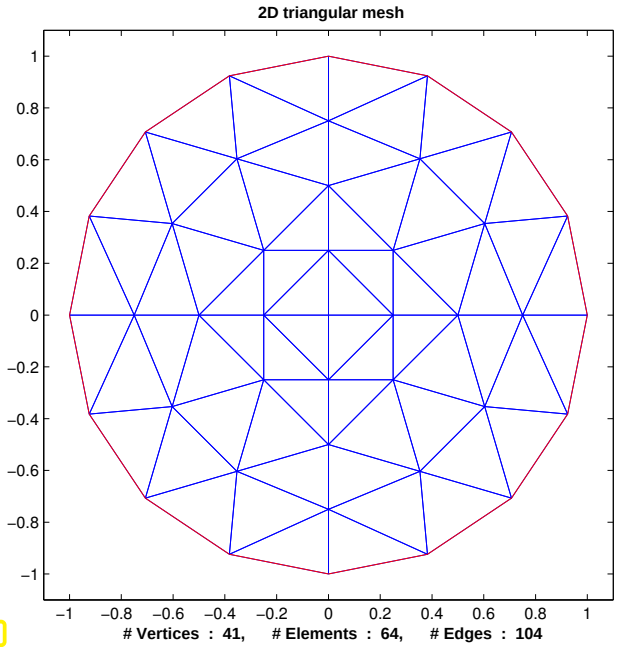


Fig. 246

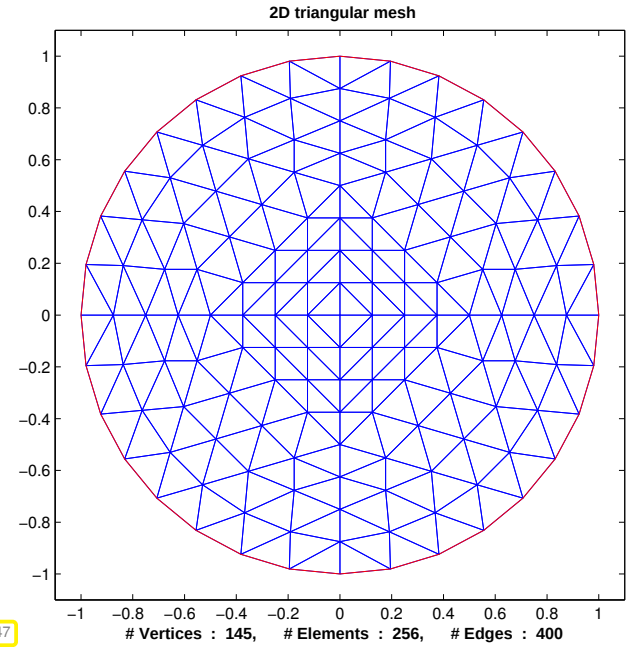


Fig. 247

Linearly boundary fitted unstructured triangular meshes of $\Omega = B_1(0)$.

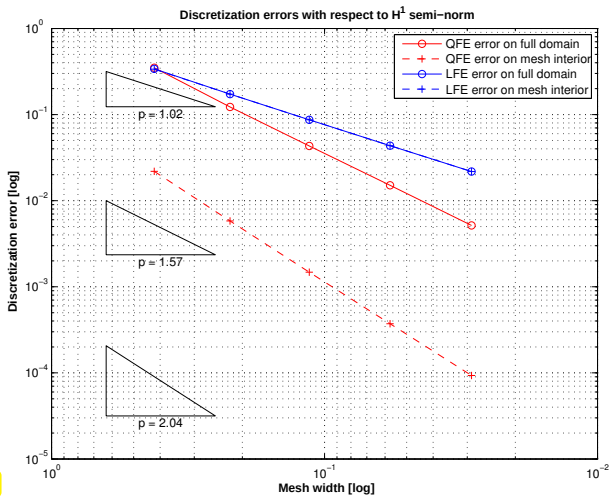


Fig. 248

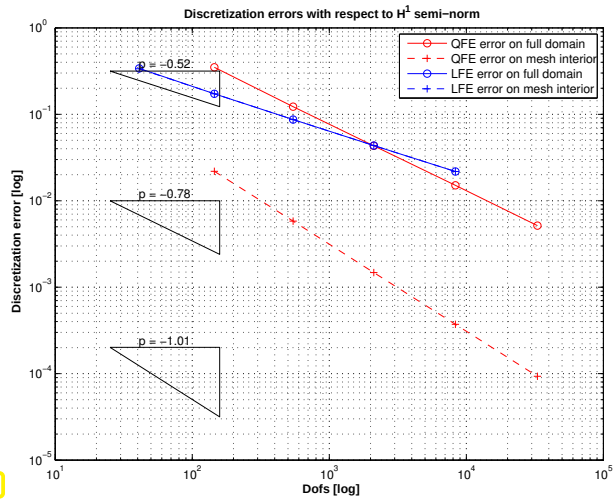


Fig. 249

$H^1(\Omega)$-norm of discretization error on unit ball ($- \leftrightarrow p = 1$, $- \leftrightarrow p = 2$)

Dashed lines in Fig. 248, Fig. 249: error norms computed on polygonal domain covered by the mesh $\neq \Omega$; this spurious "error norm" suggests no deterioration of the convergence!

Rule of thumb deduced from sophisticated finite element theory:

If $V_{0,N} = \mathcal{S}_p^0(\mathcal{M})$, use boundary fitting with polynomials of degree p.

A technique for higher order boundary approximation by means of parametric finite elements on curved cells was presented in Section 3.7.4.

5.6 Duality Techniques

5.6.1 Linear output functionals

(5.6.1) Setting

Adopt abstract setting of Section 5.1:

linear variational problem (1.4.9) in the form

$$u \in V_0: \quad \mathsf{a}(u,v) = \ell(v) \quad \forall v \in V_0 \,, \tag{3.2.3}$$

- $V_0 \hat{=}$ (real) vector space, a space of functions $\Omega \mapsto \mathbb{R}$ for scalar 2nd-order elliptic variational problems, usually "energy space" $H^1(\Omega)/H^1_0(\Omega)$, see Section 2.3
- $\mathsf{a} : V_0 \times V_0 \mapsto \mathbb{R} \hat{=}$ a bilinear form, see Def. 1.3.22,
- $\ell : V_0 \mapsto \mathbb{R} \hat{=}$ a linear form, see Def. 1.3.22,
- Ass. 5.1.2,Ass. 5.1.3,Ass. 5.1.4 are supposed to hold ➢ existence, uniqueness, and stability of solution u by Thm. 5.1.5.

(Examples of 2nd-order linear BVPs discussed in Rem. 5.1.6, Section 2.9)

Galerkin discretization using $V_{0,N} \subset V_0$ ➢ discrete variational problem

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = f(v_N) \quad \forall v_N \in V_{0,N} \,. \tag{3.2.8}$$

New twist: we are interested mainly/only in the *number* $F(u)$, where

$$\boxed{F : V_0 \mapsto \mathbb{R}} \quad \text{is an output functional.}$$

Mathematical terminology: functional $\hat{=}$ mapping from a function space into $\mathbb{R}$

Example 5.6.2 (Output functionals)

Some output functionals for solutions of PDEs commonly encountered in applications:

- mean values, see Exp. 5.6.6 below
- total heat flux through a surface (for heat conduction model $\to$ Section 2.6), see Exp. 5.6.12 below
- total surface charge of a conducting body (for electrostatics $\to$ Section 2.2.2)
- total heat production (Ohmic losses) by stationary currents
- total force on a charged conductor (for electrostatics $\to$ Section 2.2.2)
- lift and drag in computational fluid dynamics (aircraft simulation)
- and many more ...
- monostatic radar cross section for wave scattering problems in frequency domain

---

We consider output functionals with special properties, which are rather common in practice:

Assumption 5.6.3. Linearity of output functional

The output functional F is a linear form ($\rightarrow$ Def. 1.3.22) on V_0

To put the next assumption into context, please recall Ass. 5.1.3 and § 2.4.31.

Assumption 5.6.4. Continuity of output functional $\rightarrow$ Def. 2.2.56

The output functional is continuous w.r.t. the energy norm in the sense that

$$\exists C_f > 0: \quad |F(v)| \leq C_f \|v\|_{\mathsf{a}} \quad \forall v \in V_0 \,.$$

Now consider Galerkin discretization of (3.2.3) based on Galerkin trial/test space $V_{0,N} \subset V_0$, $N := \dim V_{0,N} < \infty$ ➢ discrete variational problem

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in V_{0,N} \,. \tag{3.2.8}$$

What would you dare to sell as an approximation of $F(u)$? Of course, ...

Galerkin solution $u_N \in V_{0,N}$ ➥ approximate output value $F(u_N)$

(5.6.5) A simple estimate

How accurate is $F(u_N)$, that is, how big is the output error $|F(u) - F(u_N)|$?

Linearity ($\rightarrow$ Ass. 5.6.3) and continuity Ass. 5.6.4 conspire to furnish a very simple estimate

$$|F(u) - F(u_N)| \leq C_f \, \|u - u_N\|_{\mathsf{a}} \,.$$

A priori estimates for $\|u - u_N\|_{\mathsf{a}}$ ➽ estimates for $|F(u) - F(u_N)|$

Hence, Thm. 5.3.56 immediately tells us the asymptotic convergence of linear and continuous output functionals defined for solutions of 2nd-order scalar elliptic BVPs and Lagrangian finite element discretization.

Experiment 5.6.6 (Approximation of mean temperature)

Heat conduction model ($\rightarrow$ Section 2.6), scaled heat conductivity $\kappa \equiv 1$, on domain $\Omega =]0,1[^2$, fixed temperature $u = 0$ on $\partial\Omega$:

$$-\Delta u = f \quad \text{in } \Omega \quad , \quad u = 0 \quad \text{on } \partial\Omega \,.$$

Heat source function $f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$, $(x,y)^T \in \Omega$
➢ solution $u(x,y) = \sin(\pi x)\sin(\pi y)$.

$$\text{mean temperature} \qquad F(u) = \frac{1}{|\Omega|} \int_\Omega u \, \mathrm{d}\boldsymbol{x} \,.$$

Details of finite element Galerkin discretization:

- Sequence of triangular meshes $\mathcal{M}$ created by regular refinement.

- Galerkin discretization: $V_{0,N} := \mathcal{S}^0_{1,0}(\mathcal{M})$ (linear Lagrangian finite elements $\rightarrow$ Section 3.3).
- Quadrature rule (3.6.162) of order 6 for assembly of right hand side vector (more than sufficiently accurate $\rightarrow$ guidelines from Section 5.5.1)

Expected: algebraic convergence in $h_{\mathcal{M}}$ with rate 1 of approximate mean temperature

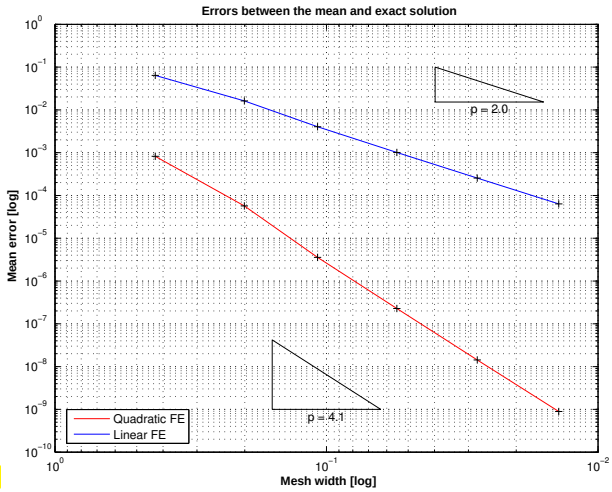


Fig. 250

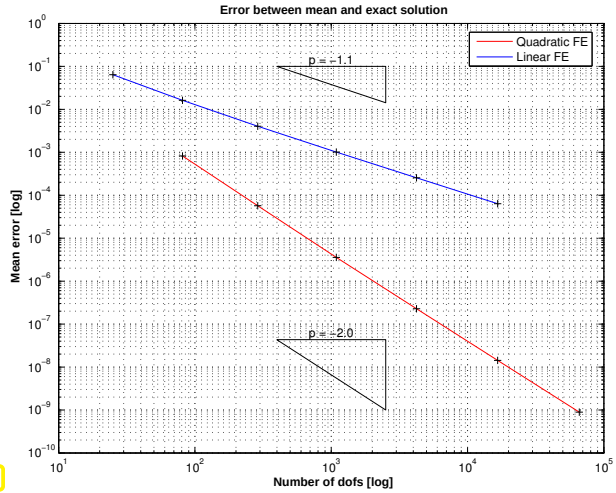


Fig. 251

Error in mean value on unit square ($- \leftrightarrow p = 1$, $- \leftrightarrow p = 2$)

Observation: Mean value converges twice as fast as expected: algebraic convergence $O(h^2_{\mathcal{M}})$!

Theorem 5.6.7. 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

$$|F(u) - F(u_N)| \leq \|u - u_N\|_{\mathsf{a}} \inf_{v_N \in V_{0,N}} \|g_F - v_N\|_{\mathsf{a}} \,. \tag{5.6.8}$$

Proof. For *any* $v_N \in V_{0,N}$:

$$F(u) - F(u_N) = \mathsf{a}(u - u_N, g_F) \overset{(*)}{=} \mathsf{a}(u - u_N, g_F - v_N) \leq \|u - u_N\|_{\mathsf{a}} \|g_F - v_N\|_{\mathsf{a}} \,.$$

$(*) \leftarrow$ by Galerkin orthogonality (5.1.10). □

If g_F can be approximated well in $V_{0,N}$, then the output error can converge $\rightarrow 0$ (much) faster than $\|u - u_N\|_{\mathsf{a}}$.

Example 5.6.9 (Approximation of mean temperature cnt'd $\rightarrow$ Exp. 5.6.6)

- The mean temperature functional (5.6.8) is obviously linear $\rightarrow$ Ass. 5.6.3
- By the Cauchy-Schwarz inequality (2.3.30) it clearly satisfies Ass. 5.6.4 even with $\|\cdot\|_{\mathsf{a}} = \|\cdot\|_{L^2(\Omega)}$, let alone for $\|\cdot\|_{\mathsf{a}} = |\cdot|_{H^1(\Omega)}$ on $H^1_0(\Omega)$.

What is $g_F \in H_0^1(\Omega)$ in this case? By Thm. 5.6.7 it is the solution of the variational problem

$$\int_\Omega \mathbf{grad}\, g_F \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = F(v) = \frac{1}{|\Omega|}\int_\Omega v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\ .$$

The associated 2nd-order BVP reads

$$-\Delta g_F = \frac{1}{|\Omega|} \quad \text{in } \Omega, \quad g_F = 0 \quad \text{on } \partial\Omega\ .$$

Now recall the elliptic lifting theory Thm. 5.4.10 for convex domains: since $\Omega =]0,1[^2$ is convex, we conclude $g_F \in H^2(\Omega)$.

By interpolation estimate of Thm. 5.3.38 ($\mathsf{I}_1 \hat{=}$ linear interpolation onto $\mathcal{S}_1^0(\mathcal{M})$)

$$\inf_{v_N \in \mathcal{S}_1^0(\mathcal{M})} |g_F - v_N|_{H^1(\Omega)} \leq |g_F - \mathsf{I}_1 g_F|_{H^1(\Omega)} \leq C h_{\mathcal{M}} |g_F|_{H^2(\Omega)}\ ,$$

where $C > 0$ may depend on Ω and the shape regularity measure ($\rightarrow$ Def. 5.3.37) of $\mathcal{M}$.

Plug this into the duality estimate (5.6.8) of Thm. 5.6.7 and note that $u \in H^2(\Omega)$ by virtue of Thm. 5.4.10 and $f \in L^2(\Omega)$:

$$|F(u) - F(u_N)| \leq C h_{\mathcal{M}} \cdot \underbrace{|u - u_N|_{H^1(\Omega)}}_{\leq C h_{\mathcal{M}} \text{ if } u \in H^2(\Omega)} \leq C h_{\mathcal{M}}^2\ ,$$

where the "generic constant" $C > 0$ depends only on Ω, u, $\rho_{\mathcal{M}}$.

Again, by the elliptic lifting theory Thm. 5.4.10 we infer that $u \in H^2(\Omega)$ holds for this example since $f \in L^2(\Omega)$.

5.6.2 Case study: Boundary flux computation

Model problem (process engineering):

Long pipe carrying turbulent flow of coolant (water)

$\Omega \subset \mathbb{R}^2$: cross-section of pipe
κ : (scaled) heat conductivity of pipe material (assumed homogeneous, $\kappa = \mathrm{const}$)

Assumption: Constant temperatures u_o, , u_i at outer/inner wall Γ_o, Γ_i of pipe

Task: Compute heat flow pipe $\rightarrow$ water

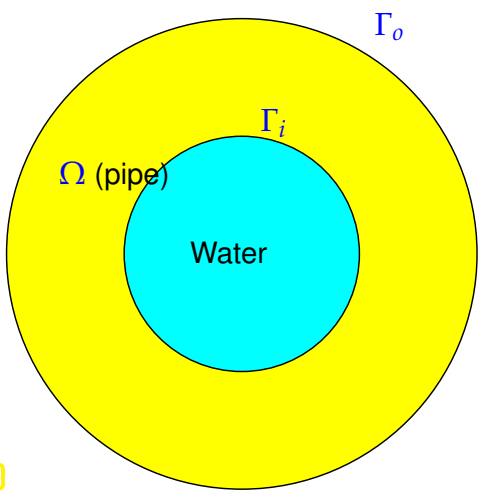


Fig. 252

Mathematical model: elliptic boundary value for stationary heat conduction ($\rightarrow$ Section 2.6)

$$-\operatorname{div}(\kappa\, \mathbf{grad}\, u) = 0 \quad \text{in } \Omega \quad , \quad u = u_x \quad \text{on } \Gamma_x,\ x \in \{i, o\}\ . \tag{5.6.10}$$

Heat flux through Γ_i: $$J(u) := \int_{\Gamma_i} \kappa\, \mathbf{grad}\, u \cdot \boldsymbol{n} \,\mathrm{d}S\ . \tag{5.6.11}$$

Relate to abstract framework: (5.6.10) $\cong$ (3.2.3), $V_0 \cong H_0^1(\Omega)$ ($\rightarrow$ Section 2.9)

(Actually, $u \in H^1(\Omega)$, but by means of offset functions we can switch to the variational space $H_0^1(\Omega)$, see Section 2.2.3,Section 3.6.6.)

Numerical method: finite element computation of heat conduction in pipe
(e.g. linear Lagrangian finite element Galerkin discretization, Section 3.3)

Expectation: Algebraic convergence $|J(u) - J(u_N)| = O(h_{\mathcal{M}}^2)$ for regular h-refinement

This expectation is based on the analogy to Exp. 5.6.6 (Approximation of mean temperature), where duality estimates yielded $O(h_{\mathcal{M}}^2)$ convergence of the mean temperature error in the case of Galerkin discretization by means of linear Lagrangian finite elements on a sequence of meshes obtained by regular refinement. Now, it seems, we can follow the same reasoning.

Experiment 5.6.12 (Computation of heat flux)

✦ Setting: model problem "heat flux pipe *to* water", see (5.6.10) and Fig. 252.

✦ Linear output functional from (5.6.11)

✦ Domain $\Omega = B_{R_o}(0) \setminus B_{R_i}(0) := \{\mathbf{x} \in \mathbb{R}^2 \colon R_i < |\mathbf{x}| < R_o\}$ with $R_o = 1$ and $R_i = 1/2$

✦ Dirichlet boundary data $u_i = 60°\mathrm{C}$ on Γ_i, $u_o = 10°\mathrm{C}$ on Γ_o, heat source $f \equiv 0$, heat conductivity $\kappa \equiv 1$.

➢ Exact solution: $u(r,\varphi) = C_1 \ln(r) + C_2$, with $C_1 := (u_o - u_i)/(\ln R_i - \ln R_o)$,
➢ Exact heat flux: $J = 2\pi\kappa C_1$, $C_2 := (\ln R_o u_i - \ln R_i u_o)/(\ln R_i - \ln R_o)$.

Details of linear Lagrangian finite element Galerkin discretization:

- Sequences of unstructured triangular meshes $\mathcal{M}$ obtained by regular refinement of coarse mesh (from grid generator).
- Galerkin FE discretization based on $V_{0,N} := \mathcal{S}_{1,0}^0(\mathcal{M})$.
- Approximate evaluation of $\mathsf{a}(u_N, v_N)$, $f(v_N)$ by six point quadrature rule (3.6.162) ("overkill quadrature", see Section 5.5.1)
- Approximate evaluation of $J(u_N)$ by 4 point Gauss-Legendre quadrature rule on boundary edges of $\mathcal{M}$.
- Linear boundary approximation (circle replaced by polygon).
- Recorded: errors $|J - J(u_N)|$ on sequence of meshes.

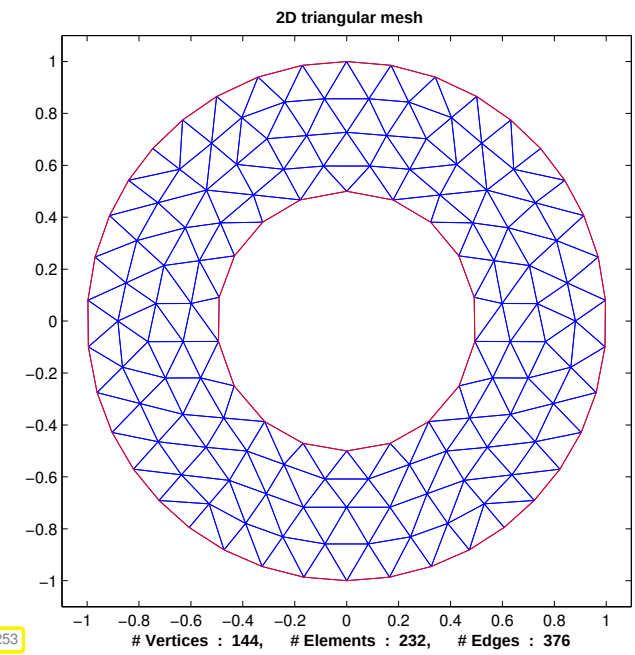


Fig. 253

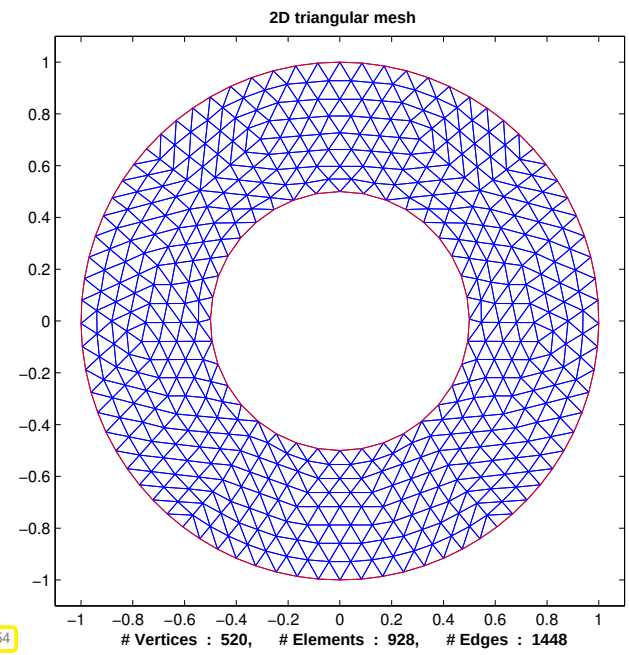


Fig. 254

Unstructured triangular meshes for $\Omega = B_1(0) \setminus B_{1/2}(0)$ (two coarsest specimens).

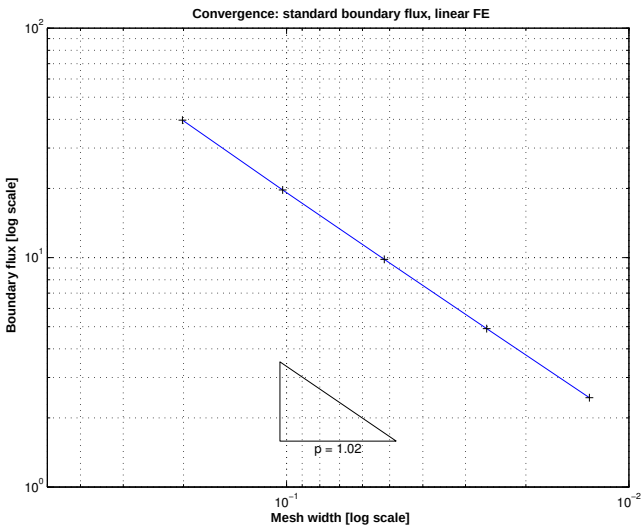


Fig. 255

Observation:

Algebraic convergence of output error for J from (5.6.11) *only with rate 1* (in mesh width $h_{\mathcal{M}}$)!
(This is not the fault of the piecewise linear boundary approximation, which is sufficient when using piecewise linear Lagrangian finite elements, see Section 5.5.2.)

Why was our expectation mistaken **?**

Suspicion: the output functional J fails to meet requirements of duality estimates of Thm. 5.6.7:

boundary flux functional J from (5.6.11) is **not** continuous on $H^1(\Omega)$!

(5.6.13) Non-continuity of boundary flux functional

How can we corroborate our suspicion that J from (5.6.11) fails to be continuous? First remember Def. 2.2.56.

Idea: find $u \in H^1(\Omega)$, for which "$J(u) = \infty$",
cf. investigation of non-continuity of point evaluation functional on $H^1(\Omega) \to$ Ex. 2.4.22.

On $\Omega = \{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\| < 1\}$ (unit disk) consider

$$u(\boldsymbol{x}) = (1 - \|\boldsymbol{x}\|)^\alpha =: g(\|\boldsymbol{x}\|) \,, \quad \frac{1}{2} < \alpha < 1 \,,$$

and the boundary flux functional (5.6.11) on $\partial\Omega$.

☞ On the one hand, using the expression (2.4.26) for the gradient in polar coordinates,

$$J_0(v) = \int_{\partial\Omega} \frac{\partial u}{\partial r}(\boldsymbol{x}) \, \mathrm{d}S(\boldsymbol{x}) = 2\pi\alpha(1-r)^{\alpha-1}{}_{|r=1}{}'' = \infty'' \,.$$

☞ On the other hand, straightforward computation of improper integral using (2.4.27):

$$|u|^2_{H^1(\Omega)} = \int_\Omega \|\mathbf{grad}\, u(\boldsymbol{x})\|^2 \, \mathrm{d}\boldsymbol{x} = 2\pi \int_0^1 |g'(r)|^2 r \, \mathrm{d}r = 2\pi\alpha^2 \int_0^1 (1-r)^{2\alpha-2} r \, \mathrm{d}r$$

$$= 2\pi\alpha^2 \int_0^1 s^{2\alpha-2}(1-s) \, \mathrm{d}s = 2\pi\alpha \left[\frac{s^{2\alpha-1}}{2\alpha-1} - \frac{s^{2\alpha}}{2\alpha} \right]_{s=0}^{s=1} = 2\pi \frac{1}{2\alpha-1} < \infty \,.$$

$$\overset{\text{Def. 2.3.25}}{\Longrightarrow} \quad u \in H^1(\Omega) \quad (u \in C^0(\overline{\Omega}) \text{ and } u \in C^\infty(\Omega \setminus \{0\}) \text{ !}) \,.$$

§ 5.6.13 ➢ Thm. 5.6.7 cannot be applied

(Potentially) poor convergence of flux obtained from straightforward evaluation of $J(u_N)$ for FE solution $u_N \in \mathcal{S}^0_{1,0}(\mathcal{M})$!

Apparently there is no remedy, because the boundary flux functional (5.6.11) seems to be enforced on us by the problem: we are not allowed to tinker with it, are we**?**

"Recovering" continuity of boundary flux functional

Trick:

use fixed cut-off function $\psi \in C^0(\overline{\Omega}) \cap H^1(\Omega)$, $\psi \equiv 1$ on Γ_i, $\psi_{|\Gamma_o} = 0$

$$\int_{\Gamma_i} \kappa \, \mathbf{grad}\, u \cdot \boldsymbol{n} \, \mathrm{d}S = \int_{\Gamma_i} (\kappa \, \mathbf{grad}\, u \cdot \boldsymbol{n}) \, \psi \, \mathrm{d}S = \int_\Omega \underbrace{\mathrm{div}(\kappa \, \mathbf{grad}\, u)}_{=0} \psi + \kappa \, \mathbf{grad}\, u \cdot \mathbf{grad}\, \psi \, \mathrm{d}\boldsymbol{x}$$

$$\blacktriangleright \quad \text{use} \quad J^*(u) := \int_\Omega \kappa \, \mathbf{grad}\, u \cdot \mathbf{grad}\, \psi \, \mathrm{d}\boldsymbol{x} \,. \tag{5.6.15}$$

Obviously $(*)$: $\quad J^* : H^1(\Omega) \mapsto \mathbb{R}$ continuous & $J^*(u) = J(u)$ for solution of (5.6.10)

$(*)$: By the Cauchy-Schwarz inequality (2.3.30), since $\kappa = \mathrm{const}$,

$$|J^*(u)| \leq \kappa \|\mathbf{grad}\, u\|_{L^2(\Omega)} \|\mathbf{grad}\, \psi\|_{L^2(\Omega)} \leq C |u|_{H^1(\Omega)} \,,$$

with $C := \kappa \|\mathbf{grad}\, \psi\|_{L^2(\Omega)}$, which is a constant independent of u, as ψ is a fixed function.

Objection: You cannot just tamper with the output functional of a problem just because you do not like it!

Rebuttal: Of course, one can replace the output function J with another one J^* as long as

$$\boxed{J(u) = J^*(u) \quad \text{for the exact solution } u \text{ of the BVP,}}$$

because the objective is not to "evaluate J", but to obtain an approximation for $J(u)$!

Experiment 5.6.16 (Computation of heat flux cnt'd $\rightarrow$ Exp. 5.6.12)

Further details on flux evaluation:

- Galerkin FE discretization based on $V_{0,N} := \mathcal{S}^0_{1,0}(\mathcal{M})$ or $V_{0,N} := \mathcal{S}^0_{2,0}(\mathcal{M})$.
- Approximate evaluation of $J^*(u_N)$ by six point quadrature rule (3.6.162) ("overkill quadrature", see Section 5.5.1)
- Cut-off function with linear decay in radial direction
- Recorded: errors $|J - J(u_N)|$ and $|J - J^*(u_N)|$.

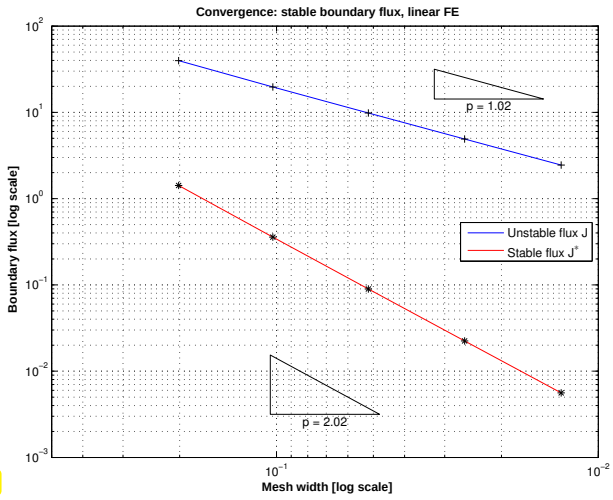


Fig. 256

◁ Convergence of $|J(u) - J(u_N)|$ and $|J(u) - J^*(u_N)|$ for linear Lagrangian finite element discretization.

Additional observations:

- Algebraic convergence $|J(u) - J^*(u_N)| = O(h^2_{\mathcal{M}})$ (rate 2 !) for alternative output functional J^* from (5.6.15).
- Dramatically reduced output error!

Remark 5.6.17 (Finding continuous replacement functionals)

Now you will ask: How can we find good (continuous) replacement functionals, if we are confronted with an unbounded output functional on the energy space**?**

Unfortunately, there is *no recipe*, and sometimes it does not seem to be possible to find a suitable J^* at all, for instance in the case of point evaluation, *cf.* Ex. 2.4.22.

Good news: another opportunity to show off how smart you are!

5.6.3 L^2-estimates

So far we have only studied the energy norm ($\leftrightarrow$ $H^1(\Omega)$-norm, see Rem. 5.3.39) of the finite element discretization error for 2nd-order elliptic BVP.

The reason was the handy tool of Cea's lemma Thm. 5.1.15.

What about error estimates in other "relevant norms", e.g„

- in the mean square norm or $L^2(\Omega)$-norm, see Def. 2.3.4,
- in the supremum norm or $L^\infty(\Omega)$-norm, see Def. 1.6.5?

In this section we tackle $\|u - u_N\|_{L^2(\Omega)}$. We largely reuse the abstract framework of Section 5.6.1: linear variational problem (3.2.3) with exact solution $u \in V_0$, Galerkin finite element solution $u_N \in V_{0,N}$, see p. 425, and the special framework of linear 2nd-order elliptic BVPs, see Rem. 5.1.6: concretely,

$$\mathsf{a}(u,v) := \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H_0^1(\Omega)\,.$$

Experiment 5.6.18 (L^2-convergence of FE solutions $\rightarrow$ Exp. 5.2.8)

Setting: $\Omega =]0,1[^2$, $D \equiv 1$, $f(x,y) = 2\pi^2 \sin(\pi x)\sin(\pi y)$, $(x,y)^\top \in \Omega$
$\rightarrow$ $u(x,y) = \sin(\pi x)\sin(\pi y)$.

- Sequence of triangular meshes $\mathcal{M}$, created by regular refinement.
- FE Galerkin discretization based on $\mathcal{S}^0_{1,0}(\mathcal{M})$ or $\mathcal{S}^0_2(\mathcal{M})$.
- Quadrature rule (3.6.162) for assembly of local load vectors ($\rightarrow$ Section 3.6.5).
- Approximate $L^2(\Omega)$-norm by means of quadrature rule (3.6.162).

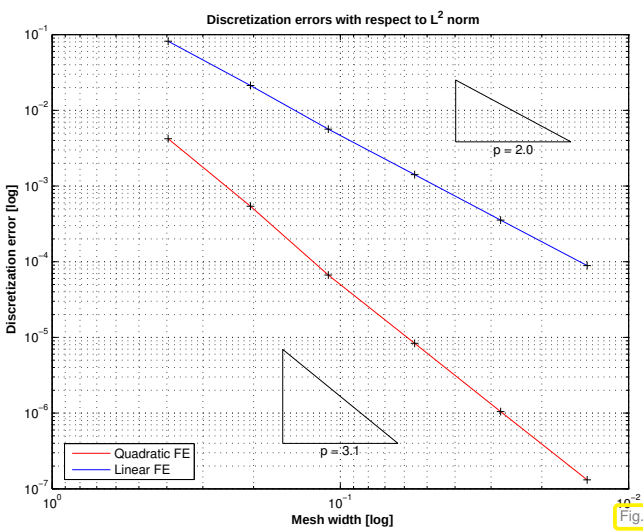


Fig. 257

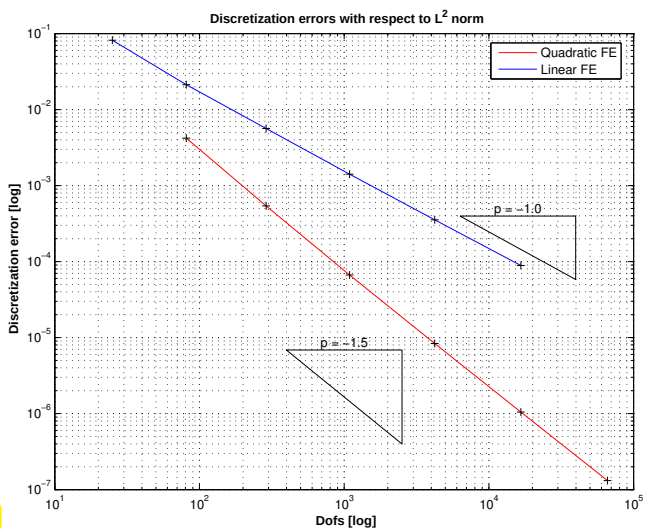


Fig. 258

$L^2(\Omega)$-norm of discretization error on unit square ($-\leftrightarrow p = 1$, $-\leftrightarrow p = 2$)

Observations: • Linear Lagrangian FE ($p = 1$) ➽ $\|u - u_N\|_0 = O(N^{-1})$
• Quadratic Lagrangian FE ($p = 2$) ➽ $\|u - u_N\|_0 = O(N^{-1.5})$

(5.6.19) L^2 interpolation error

Recall the interpolation error estimate of Thm. 5.3.38

$$\|u - \mathsf{I}_1 u\|_{L^2(\Omega)} = O(h_{\mathcal{M}}^2) \quad \text{vs.} \quad |u - \mathsf{I}_1 u|_{H^1(\Omega)} = O(h_{\mathcal{M}}) \,,$$

on a family of meshes with uniformly bounded shape regularity measure.

☞ Higher rate of algebraic convergence of the interpolation error when measured in the weaker $L^2(\Omega)$-norm compared to the stronger $H^1(\Omega)$-norm.

Therefore a similar observation in the case of the finite element approximation error is not so surprising.

(5.6.20) Duality techniques for L^2-estimates

Now we supply a rigorous underpinning and explanation of the behavior of $\|u - u_N\|_{L^2(\Omega)}$ that we have observed and expect.

Idea: Consider special continuous linear "output functional"

$$F(v) := \int_\Omega v \cdot (u - u_N)\,\mathrm{d}\boldsymbol{x} \quad !$$

This is not a practical output functional, because its evaluation will not be possible even if the finite element solution u_N is available. Nevertheless, this F is well defined, because existence and uniqueness of both u and u_N are guaranteed.

This functional is highly relevant for L^2-estimates, because

$$F(u) - F(u_N) = \|u - u_N\|_{L^2(\Omega)}^2 \quad !$$

➢ estimates for the output error will provide bounds for $\|u - u_N\|_{L^2(\Omega)}$!

Note: Both u and u_N are *fixed* functions $\in H^1(\Omega)$!

➢ Linearity of F ($\rightarrow$ Ass. 5.6.3) is obvious.

➢ Continuity $F : H_0^1(\Omega) \mapsto \mathbb{R}$ ($\rightarrow$ Ass. 5.6.4) is clear, use Cauchy-Schwarz inequality (2.3.30).

Duality estimate of Thm. 5.6.7 can be applied:

Thm. 5.6.7 ▶

$$F(u) - F(u_N) = \|u - u_N\|_{L^2(\Omega)}^2 \leq C|u - u_N|_{H^1(\Omega)} \inf_{v_N \in V_{0,N}} |g_F - v_N|_{H^1(\Omega)} \,, \tag{5.6.21}$$

where $C > 0$ may depend only on κ, and the dual solution $g_F \in H_0^1(\Omega)$ satisfies

$$\mathsf{a}(g_F, v) = F(v) \quad \forall v \in V_0 \quad \Leftrightarrow \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\, g_F \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega v(u - u_N)\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)$$

$$\Downarrow$$

$$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, g_F) = u - u_N \quad \text{in } \Omega \quad , \quad g_F = 0 \quad \text{on } \partial\Omega \,. \tag{5.6.22}$$

Assumption 5.6.23. 2-regularity of homogeneous Dirichlet problem

We assume that the homogeneous Dirichlet problem with coefficient κ is 2-regular on Ω: There is $C > 0$, which depends on Ω only such that

$$\begin{array}{c} u \in H_0^1(\Omega) \\ \operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, u) \in L^2(\Omega) \end{array} \quad \Rightarrow \quad u \in H^2(\Omega) \quad \text{and} \quad |u|_{H^2(\Omega)} \leq C \|\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, u)\|_{L^2(\Omega)} \,.$$

By the elliptic lifting theorem for convex domains Thm. 5.4.10 we know

$$\kappa\ C^1\text{-smooth} \quad \& \quad \Omega \text{ convex} \quad \Longrightarrow \quad \text{Ass. 5.6.23 is satisfied.}$$

(5.6.24) Estimates under assumption of 2-regularity

Ass. 5.6.23 in conjunction with (5.6.22) yields

$$|g_F|_{H^2(\Omega)} \leq C \|u - u_N\|_{L^2(\Omega)} \,, \tag{5.6.25}$$

where $C > 0$ depends only on Ω.

Now we can appeal to the general best approximation theorem for Lagrangian finite element spaces Thm. 5.3.56:

$$\inf_{v_N \in \mathcal{S}_p^0(\mathcal{M})} |g_F - v_N|_{H^1(\Omega)} \leq C \frac{h_\mathcal{M}}{p} |g_F|_{H^2(\Omega)} \overset{(5.6.25)}{\leq} C \frac{h_\mathcal{M}}{p} \|u - u_N\|_{L^2(\Omega)} \,, \tag{5.6.26}$$

where the "generic constants" $C > 0$ depend only on Ω and the shape regularity measure $\rho_\mathcal{M}$ ($\rightarrow$ Def. 5.3.37).

Combine (5.6.21) and (5.6.26) and cancel one power of $\|u - u_N\|_{L^2(\Omega)}$: With $C > 0$ depending only on Ω, κ, and the shape regularity measure $\rho_\mathcal{M}$ we conclude

$$\boxed{\text{Ass. 5.6.23} \quad \Rightarrow \quad \|u - u_N\|_{L^2(\Omega)} \leq C \frac{h_\mathcal{M}}{p} \|u - u_N\|_{H^1(\Omega)}} \,.$$

► for h-refinement: gain of one factor $O(h_\mathcal{M})$ (vs. $H^1(\Omega)$-estimates)

Is it important to assume 2-regularity, Ass. 5.6.23 or merely a technical requirement of the theoretical approach?

Experiment 5.6.27 (L^2-estimates on non-convex domain *cf.* Exp. 5.2.10)

Setting: $\Omega =]-1,1[^2 \setminus (]0,1[\times]-1,0[)$, $D \equiv 1$, $u(r,\varphi) = r^{2/3}\sin(2/3\varphi)$ (polar coordinates)
➢ $f = 0$, Dirichlet data $g = u_{|\partial\Omega}$.

Finite element Galerkin discretization and evaluations as in Exp. 5.6.18.

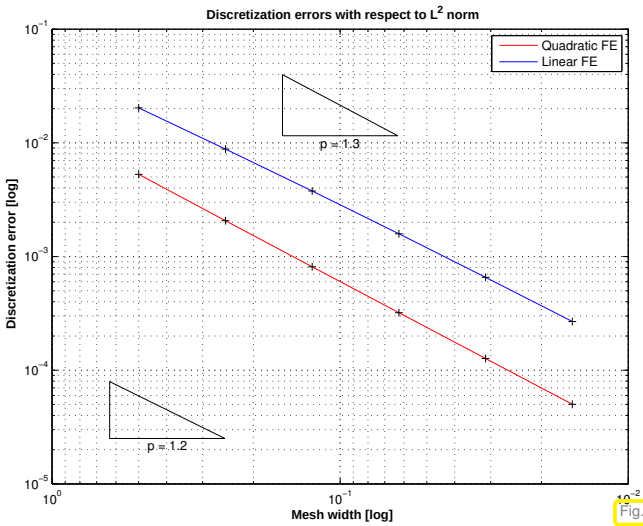


Fig. 259

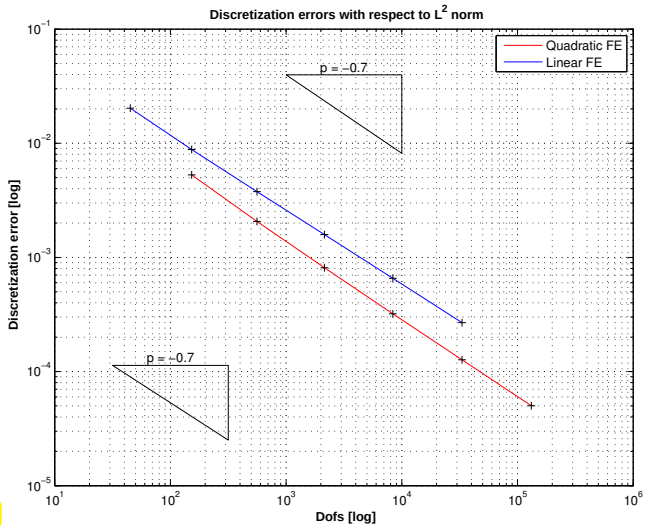


Fig. 260

$L^2(\Omega)$-norm of discretization error on "L-shaped" domain ($- \leftrightarrow p = 1$, $- \leftrightarrow p = 2$)

Observation: For both ($p = 1, 2$) ➽ algebraic convergence $\|u - u_N\|_0 = O(N^{-2/3})$

Comparison with Exp. 5.2.10: for both linear and quadratic Lagrangian FEM

$$\|u - u_N\|_{L^2(\Omega)} = O(N^{-2/3}) \quad \longleftrightarrow \quad \|u - u_N\|_{H^1(\Omega)} = O(N^{-1/3}) \,,$$

that is, we again observe a doubling of the rate of convergence for the weaker norm.

No gain through the use of quadratic FEM, because of limited smoothness of both u and dual solution g_F. For both the solution and the dual solution the gradient will have a singularity at 0.

Remark 5.6.28 (Usefulness of L^2-estimates)

To begin with, the L^2-estimates derived in this section are mainly motivated by curiosity: can we expect the higher rates of convergence that we are accustomed to for weaker norms of interpolation errors also from weaker norm of Galerkin discretization errors.

However, comparing observed convergence in L^2-norm with what is predicted by theory, should be used for testing the correctness of finite element codes, following the procedure of § 5.8.8.

5.7 Discrete Maximum Principle

So far we have investigated the accuracy of finite element Galerkin solutions: we studied relevant norms $\|u - u_N\|$ of the discretization error.

Now new perspective: structure preservation by FEM

To what extent does the finite element solution u_N inherit key structural properties of the solution u of a 2nd-order scalar elliptic BVP**?**

This issue will be discussed for a special structural property of the solution of the linear 2nd-order elliptic BVP (inhomogeneous Dirichlet problem) in variational form ($\rightarrow$ Section 2.9)

$$u \in \widetilde{g} + H_0^1(\Omega): \quad \mathsf{a}(u,v) := \int_\Omega \kappa \,\mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,. \tag{5.7.1}$$

where $\widetilde{g} \mathrel{\hat{=}}$ offset function, extension of Dirichlet data $g \in C^0(\partial\Omega)$, see Section 2.4.1, (2.4.7),
$\kappa \mathrel{\hat{=}}$ bounded and uniformly positive definite diffusion coefficient, see (2.6.6).

(5.7.1) $\longleftrightarrow$ BVP (PDE-form)

$$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, u) = f \quad \text{in } \Omega \quad , \quad u = g \quad \text{on } \partial\Omega \,.$$

Recall ($\rightarrow$ Section 2.6): (5.7.1) models *stationary* temperature distribution in body, when temperature on its surface is prescribed by g.

Intuition:

- ✦ In the absence of heat sources maximal and minimal temperature attained on surface.
- ✦ In the presence of a heat source ($f \geq 0$) the temperature minimum will be attained on surface $\partial\Omega$.
- ✦ If $f \leq 0$ (heat sink), then the maximal temperature will be attained on the surface.

In fact this is a theorem, *cf.* Section 2.8.

Theorem 5.7.2. Maximum principle for 2nd-order elliptic BVP

For $u \in C^0(\overline{\Omega}) \cap H^1(\Omega)$ holds the maximum principle

$$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, u) \geq 0 \quad \Longrightarrow \quad \min_{\boldsymbol{x}\in\partial\Omega} u(\boldsymbol{x}) = \min_{\boldsymbol{x}\in\Omega} u(\boldsymbol{x}) \,,$$
$$-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\, u) \leq 0 \quad \Longrightarrow \quad \max_{\boldsymbol{x}\in\partial\Omega} u(\boldsymbol{x}) = \max_{\boldsymbol{x}\in\Omega} u(\boldsymbol{x}) \,.$$

$\Delta u = 0$

Maximum/minimum on $\partial\Omega$

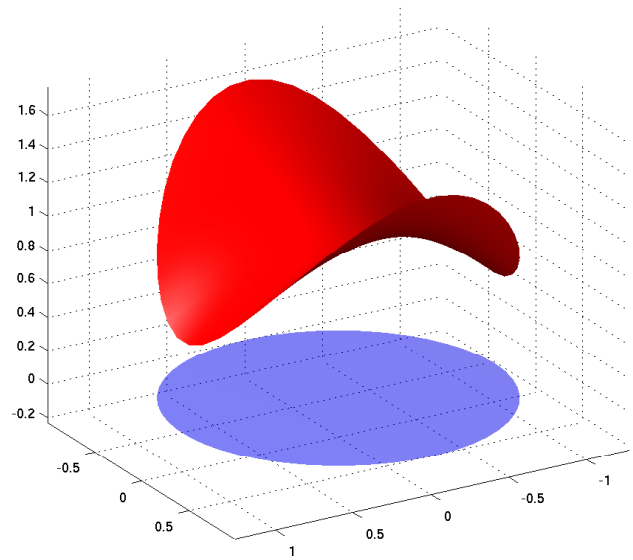


Fig. 261

Proof. ❶: case $-\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = 0$

Section 2.2.3➢ u solves quadratic minimization problem

$$u = \operatorname*{argmin}_{\substack{v\in H^1(\Omega)\\ v=g \text{ on } \partial\Omega}} \int_\Omega \kappa(\boldsymbol{x})\|\mathbf{grad}\,v(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x}\,.$$

If u had a global maximum at $\boldsymbol{x}^*$ in the interior of Ω, that is

$$\exists\delta > 0:\quad u(\boldsymbol{x}^*) \geq \max_{\boldsymbol{x}\in\partial\Omega} u(\boldsymbol{x}) + \delta\,.$$

Now "chop off" the maximum and define

$$w(\boldsymbol{x}) := \min\{u(\boldsymbol{x}), u(\boldsymbol{x}^*) - \delta\}\,,\quad \boldsymbol{x}\in\Omega\,. \tag{5.7.3}$$

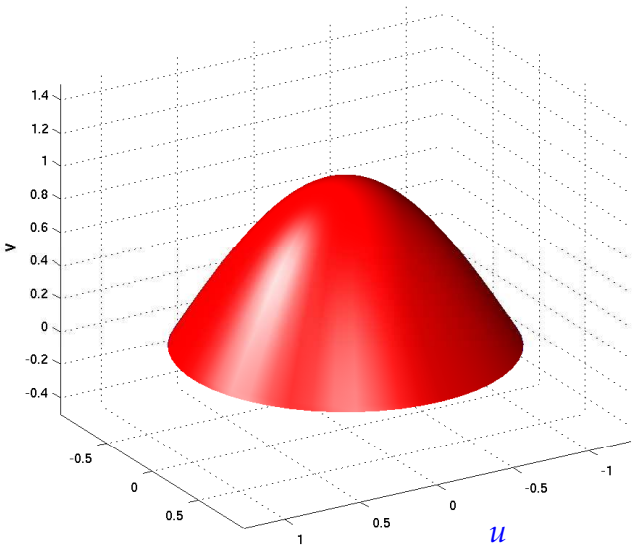


Fig. 262

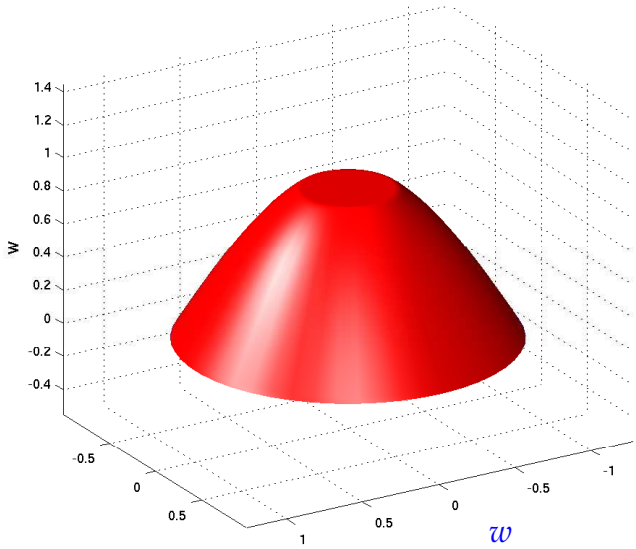


Fig. 263

$$\int_\Omega \kappa(\boldsymbol{x})\|\mathbf{grad}\,u(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x} \geq \int_\Omega \kappa(\boldsymbol{x})\|\mathbf{grad}\,w(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x}\,.$$

Obviously, $w \in C^0(\overline{\Omega})$, and as a continuous function which is piecewise in H^1 the function w will also belong to $H^1(\Omega)$ ($\rightarrow$ Thm. 2.3.35). However

$$\int_\Omega \kappa(\boldsymbol{x})\|\mathbf{grad}\,w(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x} < \int_\Omega \kappa(\boldsymbol{x})\|\mathbf{grad}\,u(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x}\,,$$

which contradicts the definition of u as the global minimizer of the quadratic energy functional.

❷: case $f := -\operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) < 0$

Section 2.2.3➢ u solves quadratic minimization problem

$$u = \operatorname*{argmin}_{\substack{v\in H^1(\Omega)\\ v=g \text{ on } \partial\Omega}} \int_\Omega \tfrac{1}{2}\kappa(\boldsymbol{x})\|\mathbf{grad}\,v(\boldsymbol{x})\|^2 - f(\boldsymbol{x})u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

The function w from (5.7.3) satisfies $w \leq u$. Thus

$$\int_\Omega \underbrace{-f(\boldsymbol{x})}_{\geq 0} u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \geq \int_\Omega \underbrace{-f(\boldsymbol{x})}_{\geq 0} w(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,.$$

Hence, again w realizes a smaller value of the energy functional than u. □

Now we consider a finite element Galerkin discretization of (5.7.1) by means of linear Lagrangian finite elements (→ Section 3.5), using offset functions supported near $\partial\Omega$ as explained in Section 3.6.6.

➢ finite element Galerkin solution $u_N \in \mathcal{S}_1^0(\mathcal{M}) \subset C^0(\overline{\Omega})$

Issue: does u_N satisfy a maximum principle, that is, can we conclude

$$\begin{aligned} f \geq 0 \quad &\Longrightarrow \quad \min_{\boldsymbol{x}\in\partial\Omega} u_N(\boldsymbol{x}) = \min_{\boldsymbol{x}\in\Omega} u_N(\boldsymbol{x})\,, \\ f \leq 0 \quad &\Longrightarrow \quad \max_{\boldsymbol{x}\in\partial\Omega} u_N(\boldsymbol{x}) = \max_{\boldsymbol{x}\in\Omega} u_N(\boldsymbol{x})\ ? \end{aligned} \tag{5.7.4}$$

(5.7.5) Maximum principle for finite difference discretization

Recall from Section 4.1: finite difference discretization of

$$-\Delta u = 0 \quad \text{in } \Omega :=]0,1[^2 \quad , \quad u = g \quad \text{on } \partial\Omega,$$

on an $M \times M$ tensor product mesh

$$\mathcal{M} := \{[(i-1)h, ih] \times [(j-1)h, jh], 1 \leq i,j \leq M\}\,, \quad M \in \mathbb{N}\,.$$

Unknowns in the finite difference method: $\mu_{ij} \approx u((ih, jh)^T), 1 \leq i,j \leq M-1$.

Unknowns are solutions of a linear system of equations, see (4.1.4)

$$\frac{1}{h^2}(4\mu_{i,j} - \mu_{i-1,j} - \mu_{i+1,j} - \mu_{i,j-1} - \mu_{i,j+1}) = 0\,, \quad 1 \leq i,j \leq M-1\,, \tag{5.7.6}$$

where values corresponding to points on the boundary are gleaned from g:

$$\mu_{0,j} := g(0,hj)\,, \quad \mu_{M,j} := g(1,hj)\,, \quad \mu_{i,0} := g(hi,0)\,, \quad \mu_{i,M} := g(hi,1)\,, \quad 1 \leq i,j < M\,.$$

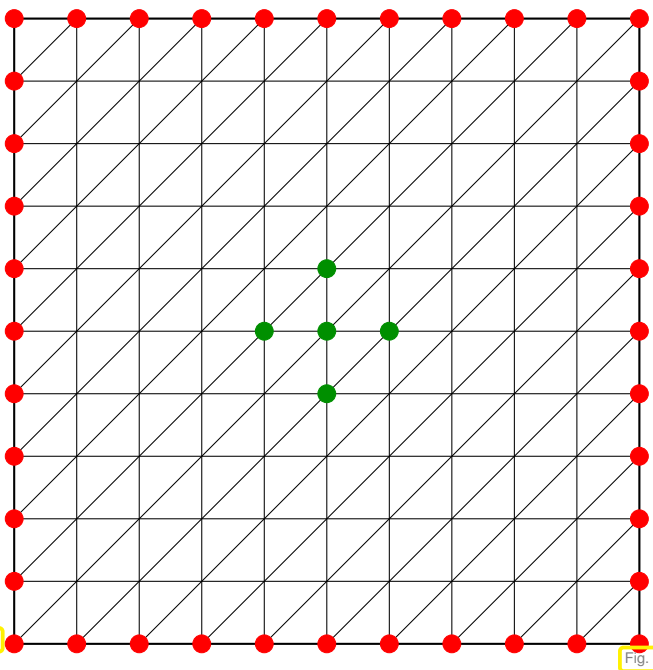


Fig. 264

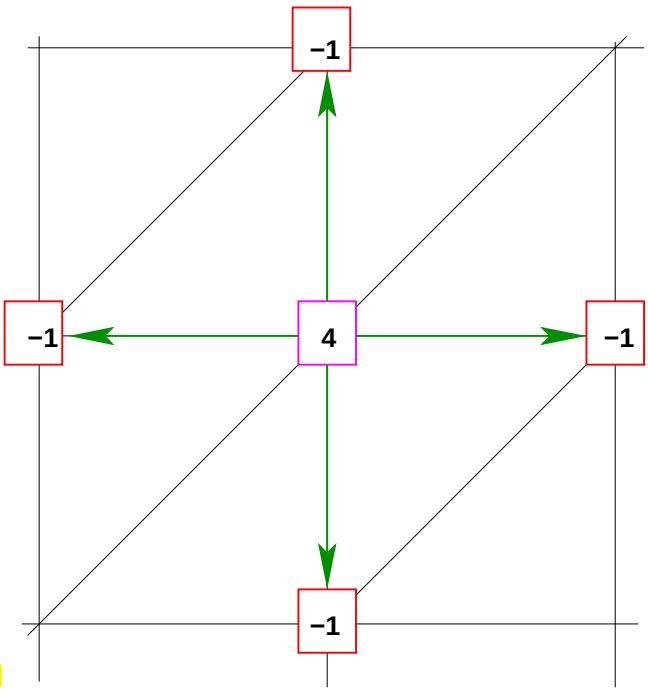


Fig. 265

The finite difference solution $(\mu_{i,j})_{1\le i,j<M}$ will attain its maximal value somewhere:

$$\exists n,m \in \{1,\dots,M-1\}: \quad \mu_{n,m} = \mu_{\max} := \max_{0\le i,j\le M} \mu_{i,j} \,.$$

Assume: $(nh, mh)^T$ in the interior $\Leftrightarrow$ $1 \le n,m < M$

Be aware of the following two facts:

$$\begin{aligned} &\mu_{n-1,m}, \mu_{n+1,m}, \mu_{n,m-1}, \mu_{n,m+1} \le \mu_{n,m} \,, \\ &\mu_{n,m} = \tfrac{1}{4}\big(\mu_{n-1,m} + \mu_{n+1,m} + \mu_{n,m-1} + \mu_{n,m+1}\big) \quad \text{(average!)} \,. \\ &\qquad\qquad \Downarrow\leftarrow \text{“averaging argument”} \\ &\mu_{n-1,m} = \mu_{n+1,m} = \mu_{n,m-1} = \mu_{n,m+1} = \mu_{n,m} \,! \end{aligned} \tag{5.7.7}$$

The same argument can now target the neighboring grid points $((n-1)h, mh)^T$, $((n+1)h, mh)^T$, $(nh, (m-1)h)^T$, $(nh, (m+1)h)^T$. By induction we find:

$$\blacktriangleright \quad \mu_{i,j} = \mu_{\max} \quad \forall 0 \le i,j \le M \,,$$

that is, the finite difference solution has to be *constant*!

$\blacktriangleright$ The finite difference solution can attain its maximum in the interior only in the case of constant boundary data g!

$\blacktriangledown$

Maximum principle satisfied for $f = 0$!

Remark 5.7.8 (Importance of discrete maximum principle)

Discretizations that satisfy the maximum principles will be *positivity preserving*: they yield non-negative solutions for non-negative sources and boundary values (Why?). This can be essential, when we want to compute a quantity that must never drop below zero, like a density or absolute temperatures.

(5.7.9) Maximum principle for linear finite element Galerkin discretization

Now we try to generalize the considerations of the previous paragraph to the discretization by means of *linear Lagrangian finite elements* (space $\mathcal{S}_1^0(\mathcal{M}) \subset H^1(\Omega)$) on a triangular mesh (of a polygonal domain $\Omega \subset \mathbb{R}^2$) see Section 3.3.

$\widetilde{\mathbf{A}} \in \mathbb{R}^{M,M} \mathrel{\hat{=}} \mathcal{S}_1^0(\mathcal{M})$-Galerkin matrix for a from (5.7.1) $\quad (M := \sharp\mathcal{V}(\mathcal{M}))$

Row of this matrix connects all nodal values $\mu_j = u_N(\mathbf{x}^j)$ of the finite element Galerkin solution $u_N \in \mathcal{S}_1^0(\mathcal{M})$ according to

$$\boxed{(\widetilde{\mathbf{A}})_{ii}\mu_i + \sum_{j\ne i} (\widetilde{\mathbf{A}})_{ij}\mu_j = (\vec{\boldsymbol{\varphi}})_i \,, \quad \mathbf{x}^i \text{ interior node}} \,,$$

where $\mu_j := g(\mathbf{x}^j)$ for $\mathbf{x}^j \in \partial\Omega$.

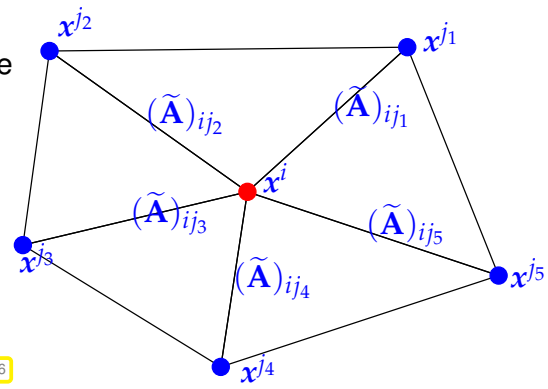


Fig. 266

This formula holds even in the case of Dirichlet boundary conditions, as can be seen from the first row of (3.6.189).

Next we note that the components of the load vector $\vec{\boldsymbol{\varphi}}$ inherit the sign of f, because the nodal basis functions for $\mathcal{S}_1^0(\mathcal{M})$ ($\rightarrow$ Section 3.3.3) are non-negative.

$$(\vec{\boldsymbol{\varphi}})_i = \int_\Omega f(\boldsymbol{x}) b_N^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \Rightarrow \quad \begin{cases} f \geq 0 & \Rightarrow \quad (\vec{\boldsymbol{\varphi}})_i \geq 0 \;\forall i\,, \\ f = 0 & \Rightarrow \quad (\vec{\boldsymbol{\varphi}})_i = 0 \;\forall i\,, \\ f \leq 0 & \Rightarrow \quad (\vec{\boldsymbol{\varphi}})_i \leq 0 \;\forall i\,. \end{cases}$$

The above averaging argument from § 5.7.5 carries over, if the entries of $\widetilde{\mathbf{A}}$ satisfy the following conditions:

- $(\widetilde{\mathbf{A}})_{ii} > 0$ (positive diagonal) , (5.7.10)
- $(\widetilde{\mathbf{A}})_{ij} \leq 0$ for $j \neq i$ (non-positive off-diagonal entries) , (5.7.11)
- $\sum_j (\widetilde{\mathbf{A}})_{ij} = 0$, if $\boldsymbol{x}^i$ is interior node . (5.7.12)

(Recall [5, Def. 1.8.8]: matrix $\widetilde{\mathbf{A}}$ satisfying (5.7.10)–(5.7.12) is diagonally dominant.)

Averaging arguent: For an interior vertex $\boldsymbol{x}^i$ is μ_i a convex combination of the nodal values in adjacent vertices

$$\mu_i = \sum_{j\neq i} \omega_j \mu_j\,, \quad \omega_j > 0\,, \quad \sum_{j\neq i} \omega_j = 1\,, \text{ since } \quad \omega_j := -\frac{(\widetilde{\mathbf{A}}_{ij})}{(\widetilde{\mathbf{A}})_{ii}}\,.$$

$$\blacktriangleright \qquad \min_{j\neq i} \mu_j \leq \mu_i \leq \max_{j\neq i} \mu_j\,,$$

where the index j always runs through all the vertices for which $(\widetilde{\mathbf{A}})_{ij} \neq 0$.

averaging argument $\blacktriangleright$ $u_N(\boldsymbol{x}^i) = \max_{\boldsymbol{y}\in\mathcal{V}(\mathcal{M})} u_N(\boldsymbol{y})$ can only hold for an interior node $\boldsymbol{x}^i$, if $\mu_N = \mathrm{const}$.

$\blacktriangleright$ Since $u_N \in \mathcal{S}_1^0(\mathcal{M})$ attains its extremal values at nodes of the mesh, the maximum principles holds for it in the case $f = 0$ provided that (5.7.10)–(5.7.12) are satisfied.

More general case $f \leq 0 \quad \Rightarrow \quad (\vec{\boldsymbol{\varphi}})_i \leq 0$:

Then the averaging argument again rules out the existence of an interior maximum for an non-constant solution. The case $f \geq 0$ follows similarly.

When will (5.7.10)–(5.7.12) hold for $\mathcal{S}_1^0(\mathcal{M})$-Galerkin matrix**?**

First consider $\kappa \equiv 1, \quad \leftrightarrow \quad -\Delta u = f$
(The linear finite element discretization of this BVP was scrutinized in Section 3.3)

From formula (3.3.23) for element matrix & assembly, see Fig. 101:

$$(\widetilde{\mathbf{A}})_{ij} = -\cot\alpha - \cot\beta = -\frac{\sin(\alpha+\beta)}{\sin\alpha\,\sin\beta}\,.$$

$$\Downarrow$$

$$(\widetilde{\mathbf{A}})_{ij} \leq 0 \quad \Leftrightarrow \quad \alpha+\beta < \pi\,.$$

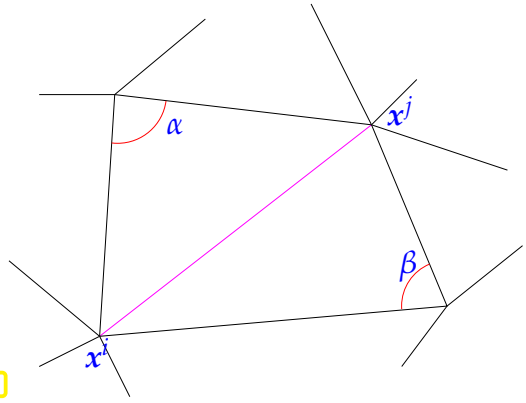


Fig. 267

Moreover

$$\sum_{x\in\mathcal{V}(\mathcal{M})} b_N^x \equiv 1 \quad \Rightarrow \quad \sum_j (\widetilde{\mathbf{A}})_{ij} = 0 \quad (\leftrightarrow (5.7.12))\,.$$

The condition (5.7.10) $\leftrightarrow$ $(\widetilde{\mathbf{A}})_{ii} > 0$ is straightforward.

Theorem 5.7.13. Maximum principle for linear FE solution of Poisson equation

The linear finite element solution of

$$-\Delta u = 0 \quad \text{in } \Omega \subset \mathbb{R}^2 \quad , \quad u = g \quad \text{on } \partial\Omega,$$

on a triangular mesh $\mathcal{M}$ satisfies the maximum principle (5.7.4)*, if $\mathcal{M}$ is a Delaunay triangulation.*

Remark 5.7.14 (Maximum principle for linear FE for 2nd-order elliptic BVPs)

For $\mathcal{S}_1^0(\mathcal{M})$-Galerkin discretization of (5.7.1) on triangular mesh, the conditions (5.7.10)–(5.7.12) are fulfilled,

$$\text{if all angles of triangles of } \mathcal{M} \leq \frac{\pi}{2}.$$

Remark 5.7.15 (Maximum principle for higher order Lagrangian FEM)

Even when using p-degree Lagrangian finite elements with nodal basis functions associated with interpolation nodes, see Section 3.5.1, the discrete maximum principle will fail to hold on *any mesh* for $p > 1$.

5.8 Validation and Debugging of Finite Element Codes

In this section you will learn about an important application of a priori finite element convergence results which you will never find mentioned in any textbook: the detection of programming errors ("debugging") in finite element codes. On one hand, whenever, for a well-defined numerical experiment the observed convergence rates are worse than those predicted by theory, the code must be faulty. One the other hand,

convergence matching theory is *circumstantial evidence* (no proof, however) for the correctness of the implementation.

Also applies to debugging a code ▷

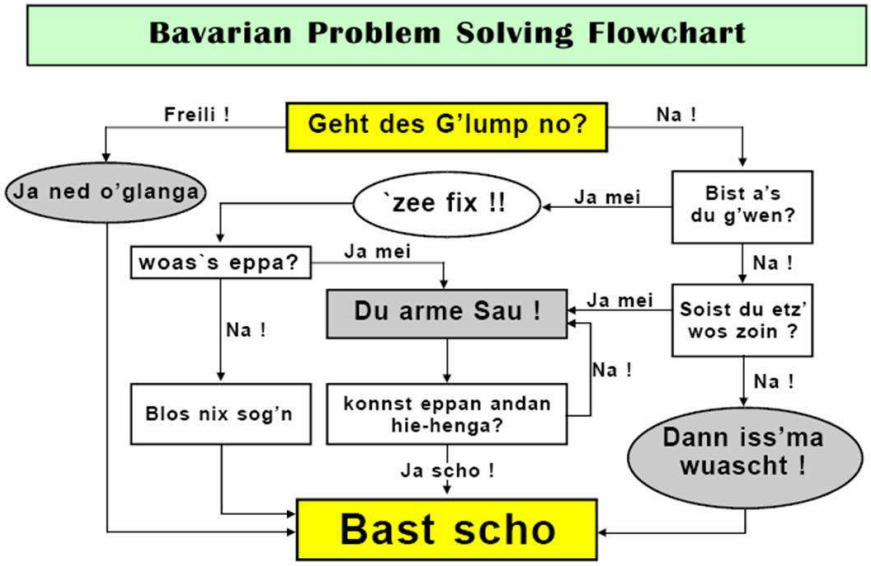


Fig. 268 Bavarian Problem Solving Flowchart

(5.8.1) The code under scrutiny ("model problem")

At our disposal is a code that implements a Lagrangian finite element discretization (→ Section 3.5) of general scalar linear second-order elliptic variational problems (→ Section 2.9, Section 2.10) on domains $\Omega \subset \mathbb{R}^d$, $d = 2,3$:

$$\begin{array}{r} u \in H^1(\Omega) \\ u = g \text{ on } \Gamma_D \end{array}: \quad \int_\Omega \boldsymbol{\alpha}(\boldsymbol{x})\,\mathbf{grad}\,u \cdot \mathbf{grad}\,v + \gamma(\boldsymbol{x}) u\,v\,\mathrm{d}\boldsymbol{x} + \int_{\Gamma_N} \lambda(\boldsymbol{x}) u\,v\,\mathrm{d}S(\boldsymbol{x})$$
$$= \int_\Omega f\,v\,\mathrm{d}\boldsymbol{x} + \int_{\Gamma_N} h\,v\,\mathrm{d}S(\boldsymbol{x}) \quad \forall v \in H^1_{\Gamma_D}(\Omega)\,, \tag{5.8.2}$$

where, based on a partition $\partial\Omega = \overline{\Gamma}_D \cup \overline{\Gamma}_N$, $\Gamma_D \cap \Gamma_N = \emptyset$, the trial and test space is the Sobolev space (→ Def. 2.3.23)

$$H^1_{\Gamma_D}(\Omega) := \left\{ v \in H^1(\Omega) : v = 0 \text{ on } \Gamma_D \right\}\,. \tag{5.8.3}$$

☞ (5.8.2) is the variational formulation for a boundary value problem with mixed Dirichlet, Neumann, and Robin boundary conditions as in Ex. 2.7.8.

- The source function $f \in L^2(\Omega)$, Dirichlet data $g \in C^0(\Gamma_D)$, Neumann data $h \in L^2(\Gamma_N)$, coefficient functions $\boldsymbol{\alpha} : \Omega \to \mathbb{R}^{d,d}$ (uniformly positive definite → Def. 2.2.18), $\gamma : \Omega \to \mathbb{R}_0^+$, $\lambda : \partial\Omega \to \mathbb{R}_0^+$ can be set within the code by defining suitable function classes.
- The code can handle general simplicial meshes (which may be read from file, see Section 3.6.1). The mesh implicitly defines the domain Ω.
- The code can compute the Galerkin finite element solution of (5.8.2) based on the Lagrangian finite element trial and test space $V_{0,N} := \mathcal{S}_p^0(\mathcal{M}) \cap H^1_{\Gamma_D}(\Omega)$ (→ Def. 3.5.2) for *fixed uniform* local polynomial degree $p \in \mathbb{N}$.

Note that the techniques presented in this section are applicable to finite element discretization of variational problems way beyond this model setting.

Task:
- ✦ Code validation: gather evidence for the correctness of the code.
- ✦ Code debugging: detect and located errors in the code.

It will turn out that *asymptotic estimates* for error norms as provided by (5.3.60), Thm. 5.3.56 and in Section 5.6.3 are *key tools* for tackling this task. (This is another reason why finite element convergence theory is relevant for anyone programming finite element methods.)

For testing we will take for granted the availability of sequences of meshes $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \dots$, which satisfy (see § 5.3.65 for related requirements)

1) that the meshwidth decreases geometrically: $h_k = qh_{k-1}$ for some $0 < q < 1$, where h_k is the meshwidth of $\mathcal{M}_k$.
2) that all cells of $\mathcal{M}_k$ have about the same size h_k. This feature is called quasi-uniformity .
3) that the shape regularity measure ($\rightarrow$ Def. 5.3.37) all meshes stays below a common bound, a property called uniform shape regularity.

Note that Item 1 & Item 3 imply that the number of cells increases in geometric progression: $\sharp\mathcal{M}_k = \sigma\,\sharp\mathcal{M}_{k-1}$ for some $\sigma > 1$ (usually $\sigma = 4$ in 2D),

Sequences of meshes complying with the above requirements can, for instance, be generated by successive (global) regular refinement of a coarse initial mesh, see Ex. 5.1.20. Refer to **??** for how to conduct regular refinement with a DUNE-type interface. Also **Gmsh** provides a menu item which triggers global regular refinement of the current mesh.

Simple global regular refinement may sometimes create meshes endowed with "too much structure" to observe "generic convergence behavior".
In this case small random perturbations of vertex positions (mesh jiggling) can restore "truly unstructured meshes".

Sequences of meshes with the above properties were used in the numerical experiments of Section 5.2 and in Exp. 5.5.4, Exp. 5.5.5, Exp. 5.6.6, Exp. 5.6.12, Exp. 5.6.18.

(5.8.4) Observing asymptotic convergence

As explained in Rem. 5.3.72 we expect algebraic convergence of the energy norm (and of the $L^2(\Omega)$-norm as well) of the discretization error in terms of the dimension of the finite element space.

We **assume**: asymptotic convergence estimates are *sharp*, *cf.* § 5.3.71: with a possibly unknown convergence rate $\alpha > 0$ we have for a targeted norm $\|\cdot\|$

$$\exists C = C(u, \dots) > 0: \quad \|u - u_N\| \approx CN^{-\alpha} \quad \forall \mathcal{M}_k \,. \tag{5.8.5}$$

($u \mathrel{\hat{=}}$ exact solution, $u_N \mathrel{\hat{=}}$ finite element Galerkin solution, $\approx \mathrel{\hat{=}}$ "approximate equality"; lower and upper bound with two (slightly) different constants ≈ 1)

According to our assumptions on the sequence of meshes, by § 5.3.67, the dimensions $N_k := \dim \mathcal{S}_p^0(\mathcal{M}_k)$ will also grow in geometric progression ($\kappa = 4$ for 2D triangular mesh)

$$N_k \approx \kappa N_{k-1} \quad \text{for some} \quad \kappa > 1 \quad \Rightarrow \quad N_k \approx \kappa^k N_0 \,. \tag{5.8.6}$$

Write u_k for the finite element Galerkin solution on $\mathcal{M}_k$, combine (5.8.5) and (5.8.6) and use the $\triangle$-inequality

$$\|u_k - u_{k-1}\| \leq \|u_k - u\| + \|u - u_{k-1}\| \approx CN_0\left(\kappa^{-k\alpha} + \kappa^{-(k-1)\alpha}\right) \approx C' N_k^{-\alpha} \,, \tag{5.8.7}$$

with a constant $C' > 0$ independent of N_k.

Measured norms of differences of Galerkin solutions of consecutive meshes in the sequence should display algebraic convergence for $N_k \to \infty$.

Consult § 1.6.27 for instructions on how to tell algebraic convergence from empiric error norms.

Caveat: Computing $\|u_k - u_{k-1}\|$ entails forming the difference of finite element functions on different meshes.

(5.8.8) Method of manufactured solutions $\to$ **[6]**

This technique has widely been used in numerical experiments exploring the asymptotic behavior of norms of discretization errors, as in Exp. 1.6.23, the experiments of Section 5.2, Exp. 5.5.4, and many more.

❶ Pick a simple domain Ω (polygon in 2D) that allows exact triangulation with straight edges.

❷ Choose *smooth* exact solution $u \in C^\infty(\overline{\Omega})$ with a simple analytic expression ($*$) and compute corresponding source function f, boundary data g, and coefficient λ (analytically from the strong form of the BVP). Symbolic computation (Mathematica, MAPLE) should be used.

❸ Choose coefficient functions $\boldsymbol{\alpha}$, γ, and λ given by simple analytic expressions; start with constants.

❹ Solve the resulting "manufactured BVP" on a sequence $(\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2, \dots, \mathcal{M}_L)$ of meshes as introduced above.

❺ Compute the finite element Galerkin solutions $u_k \in \mathcal{S}_p^0(\mathcal{M}_k)$ on mesh $\mathcal{M}_k$, $k = 0, \dots, m$, and the norms $\|u - u_k\|$ of the discretization errors. Use "overkill quadrature" for computation of local error norms, see Rem. 5.2.4.

❻ Estimate the rate of algebraic convergence ($\to$ Def. 1.6.24) following the recipe in § 1.6.27, Code 1.6.30, and plot the errors versus meshwidths in doubly logarithmic scale. Ignore coarse meshes if they give rise to "outliers" due to pre-asymptotic effects as in Exp. 1.6.34.

❼ If the measured rate well matches the predicted rate from (5.3.73)

➢ code has passed test 

Beware of polynomial exact solutions $u \in \mathcal{P}_p$! (Why?) On the other hand, if the above test fails for non-polynomial u, the next step should be to probe $u \in \mathcal{P}_p$ (Why?).

(5.8.9) Direct testing of (bi-)linear forms

This approach can be used to examine specific parts of the variational formulation. We abbreviate with $\mathsf{b}(\cdot,\cdot)$ a continuous and symmetric bilinear form on $H^1(\Omega)$, with ℓ a continuous ($\to$ Def. 2.2.56) linear form on $H^1(\Omega)$. They satisfy

$$\exists C_r > 0: \qquad \ell(v) \leq C_r \|v\|_{H^1(\Omega)} \quad \forall v \in H^1(\Omega) \,, \tag{5.8.10}$$

$$\exists C_c > 0: \qquad \mathsf{b}(v,w) \leq C_c \|v\|_{H^1(\Omega)} \|w\|_{H^1(\Omega)} \quad \forall v,w \in H^1(\Omega) \; . \tag{5.8.11}$$

The Galerkin matrix for b and the right hand side vector associated with ℓ can be tested through the following steps:

❶ Pick a simple domain Ω (polygon in 2D) that allows exact triangulation with straight edges.

❷ Choose a smooth function $w \in C^\infty(\overline{\Omega})$ that is **not** a polynomial.

❸ Compute $\mathsf{b}(w,w)$ and $\ell(w)$ exactly, that is analytically, which is often feasible, if Ω is a square or a circle. Symbolic computation (Mathematica, MAPLE) is advisable.

❹ With the finite element code evaluate $\mathsf{I}_k w$, where $\mathsf{I}_k : C^0(\overline{\Omega}) \to \mathcal{S}_p^0(\mathcal{M}_k)$ is the local nodal interpolation operator as introduced in § 5.3.52, (5.3.53). Write $\vec{v}_k \in \mathbb{R}^{N_k}$ for the vector of basis expansion coefficients of $\mathsf{I}_k w$.

❺ Use the code to compute the $\mathcal{S}_p^0(\mathcal{M}_k)$- Galerkin matrix $\mathbf{B}_k \in \mathbb{R}^{N_k,N_k}$ for b. Also compute the vector $\vec{\rho}_k \in \mathbb{R}^{N_k}$ arising from the $\mathcal{S}_p^0(\mathcal{M}_k)$-Galerkin discretization of ℓ.

❻ Using the asymptotic interpolation error estimates of Thm. 5.3.56 and the continuity of b:

$$\begin{aligned}
\mathsf{b}(w,w) - \vec{v}_k^\top \mathbf{B}_k \vec{v}_k &= \mathsf{b}(w,w) - \mathsf{b}(\mathsf{I}_k w, \mathsf{I}_k w) = \mathsf{b}(w + \mathsf{I}_k w, w - \mathsf{I}_k w) \\
&\overset{(5.8.11)}{\leq} C_c \|w + \mathsf{I}_k w\|_{H^1(\Omega)} \|w - \mathsf{I}_k w\|_{H^1(\Omega)} \\
&\leq C_c \|w\|_{H^1(\Omega)} \left(1 + C h_k^p \|w\|_{H^{p+1}(\Omega)}\right) \left(C h_k^p \|w\|_{H^{p+1}(\Omega)}\right) = O(h_k^p) \; .
\end{aligned}$$

Again, we invoke Thm. 5.3.56 and continuity:

$$\begin{aligned}
\ell(w) - \vec{v}_k^\top \vec{\rho}_k = \ell(w) - \ell(\mathsf{I}_k w) = \ell(w - \mathsf{I}_k w) &\overset{(5.8.10)}{\leq} C_r \|w - \mathsf{I}_k w\|_{H^1(\Omega)} \\
&\leq C_r C h_k^p \|w\|_{H^{p+1}(\Omega)} = O(h_k^p) \; .
\end{aligned}$$

❼ In both estimates the values on left hand side are readily available ($\mathsf{b}(w,w)$ and $\ell(w)$ are supposed to be known!) and theory predicts a rather precise rate p of algebraic convergence for them. If this rate materializes in empiric data

➢ code has passed test 

If code fails test, repeat with "simpler" w, for instance with $w \in \mathcal{P}_p(\mathbb{R}^d)$, which implies $\mathsf{b}(w,w) - \vec{v}_k^\top \mathbf{B}_k \vec{v}_k = 0$. because in this case $w \in \mathcal{S}_p^0(\mathcal{M}_k)$ for all ℓ.

Learning Outcomes

Essential knowledge and skills acquired in this chapter:

- State, prove and understand Cea's Lemma and its relevance for the finite element Galerkin discretization of elliptic BVP.
- Known the meaning of h-refinement and p-refinement.

- Ability to determine empirical (algebraic) convergence rates of various norms of a finite element discretization error.
- Ability to predict the asymptotic algebraic convergence of the energy norm and L^2-norm the finite element discretization error for scalar 2nd-order elliptic BVP.
- Familiarity with features of an elliptic BVP (corners, discontinuous coefficients) that can thwart the fastest possible convergence of a Lagrangian finite element discretization for h-refinement.
- Knowledge of how to choose the appropriate order of quadrature and boundary approximation so as to preserve the optimal rate of convergence (for h-refinement).
- Use duality techniques to obtain improved error estimates for the evaluation of linear and continuous output functionals. Understanding of the importance of continuity of output functionals.
- Knowledge of the (discrete) maximum principle for scalar 2nd-order elliptic boundary value problems.
- An idea of common strategies for the debugging and validation of a general finite element code.

Bibliography

[1] I. Babuška and M. Suri. The optimal convergence rate of the p-version of the finite element method. *SIAM J. Numer. Anal.*, 24(4):750–769, 1987.

[2] P.G. Ciarlet. *The Finite Element Method for Elliptic Problems*, volume 4 of *Studies in Mathematics and its Applications*. North-Holland, Amsterdam, 1978.

[3] L.C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1998.

[4] Helmut Harbrecht. On output functionals of boundary value problems on stochastic domains. *Math. Methods Appl. Sci.*, 33(1):91–102, 2010.

[5] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[6] K. Salari and P. Knupp. Code verification by the method of manufactured solutions. Report SAND2000-1444, Sandia National Lab, Albuquerque, NM, 2000.

[7] C. Schwab. *p- and hp-Finite Element Methods. Theory and Applications in Solid and Fluid Mechanics*. Numerical Mathematics and Scientific Computation. Clarendon Press, Oxford, 1998.

[8] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

Chapter 6

2nd-Order Linear Evolution Problems

(6.0.1) Introduction

This chapter is devoted to time-dependent problems = evolution problems.

Prerequisite knowledge is

- the theory and variational formulation of 2nd-order elliptic BVP Chapter 2,
- basic concepts and algorithms for finite elements, Section 3.2, Section 3.4, Section 3.5,
- knowledge about single step methods for ODEs, [4, Chapter 11].

In particular, we study scalar linear partial differential equations for which *one* coordinate direction is special and identified with time and denoted by the independent variable t. The other coordinates are regarded as spatial coordinates and designated by $\boldsymbol{x} = (x_1, \dots, x_d)^T$.

Why is time special? It seems to be just another dimension.

! In contrast to space, time has a *direction* from past to future and this makes the temporal direction special.

(6.0.2) Outline

Contents

This chapter exclusively deals with *linear* evolution problems. We can distinguish two fundamental classes, dissipative and conservative evolutions. This is reflected by he structure of the chapter, which comprises two sections, Section 6.1 devoted to dissipative (parabolic) evolutions, Section 6.2 addressing conservative (hyperbolic) evolutions. Each section first develops variational formulations, then discretization in space, and, finally, discretization in time. Results on convergence are reviewed and discussed.

(6.0.3) Space-time domains

For time-dependent PDEs ($\boldsymbol{x} \leftrightarrow$ spatial variable, $t \leftrightarrow$ time variable)

➢ solution will be a "function of time and space": $u = u(\boldsymbol{x}, t)$

The domain for such PDEs will have *tensor product structure* (tensor product of spatial domain and a bounded time interval):

Computational domain:

$$\widetilde{\Omega} := \Omega \times]0, T[\subset \mathbb{R}^{d+1} .$$

▶ space-time cylinder

$\Omega \subset \mathbb{R}^d \mathrel{\hat{=}}$ spatial domain (satisfying assumptions of Section 2.2.1)

$T > 0 \mathrel{\hat{=}}$ final time

Data prescribed for u on $\partial\widetilde{\Omega}$ have different names:

On $\Omega \times \{0\}$ $\rightarrow$ initial conditions,
on $\partial\Omega \times]0, T[$ $\rightarrow$ (spatial) boundary conditions.

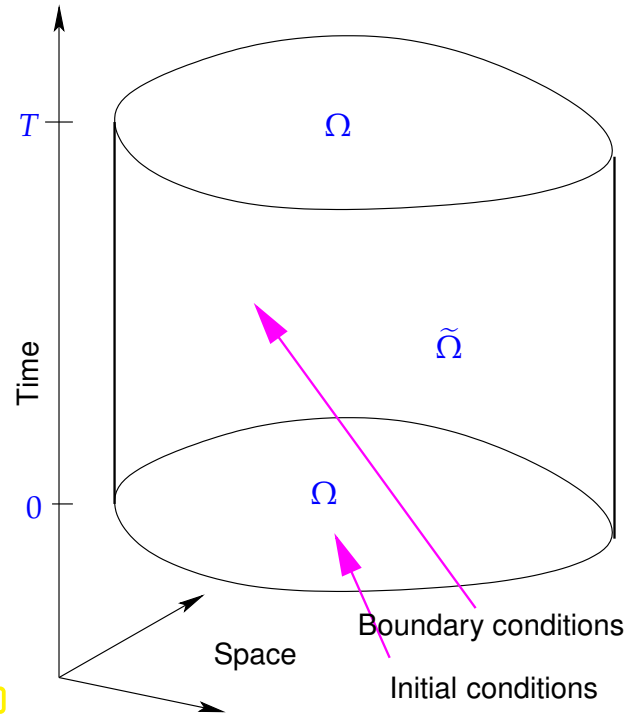


Fig. 269

Remark 6.0.4 (Temporally varying spatial domains)

The spatial sections of a space-time domain $\widetilde{\Omega}$ need not be constant, that is, $\widetilde{\Omega}$ can also be of arbitrary shape. However, the mathematical and numerical treatment of this situation is a challenge; even the distinction between initial conditions and boundary conditions becomes blurred. Thus we confine ourselves to space-time cylinders.

(6.0.5) Terminology for time-dependent problems

Extending the notion of a "boundary value problem":

$$\underbrace{\text{PDE for } u(\boldsymbol{x},t) \quad + \quad \text{initial conditions} \quad + \quad \text{boundary conditions}}_{= \text{ evolution problem}}$$

Note: No boundary conditions are prescribed on $\Omega \times \{T\}$ ("final conditions"): time is supposed to have a "direction" that governs the flow of information in the evolution problem, *cf.* § 6.0.1.

evolution problems (on bounded spatial domains) are also known as

initial-boundary value problems (IBVP).

Remark 6.0.6 (Initial time)

Why do we always pick initial time $t = 0$ in this chapter?

The modelled physical systems will usually be time-invariant, so that we are free to shift time. Remember the analoguous situation with autonomous ODE, see [4, Section 11.1].

6.1 Parabolic initial-boundary value problems

6.1.1 Heat equation

Section 2.6 treated *stationary* heat conduction: no change of temperature with time (temporal equilibrium). For this situation we derived a mathematical model that boils down to a second order scalar linear elliptic boundary value problem for the temperature $u = u(\boldsymbol{x})$ as a function of the spatial variable $\boldsymbol{x} \in \Omega$, see § 2.6.7 and Section 2.7 for a discussion of boundary conditions.

Now we consider the evolution (change in time) of a temperature distribution $u = u(\boldsymbol{x}, t)$ in a solid body occupying a bounded region of space $\Omega \subset \mathbb{R}^d$ over a finite time period $[0, T]$.

(6.1.1) Notations for heat conduction modelling

We use the following symbols in connection with mathematical modelling of transient heat conduction:

$\Omega \subset \mathbb{R}^d$	: space occupied by solid body (bounded spatial computational domain),
$\boldsymbol{x} \in \Omega$	: spatial independent variable
	: (differential operators acting in space are sometimes tagged with subscript $_{\boldsymbol{x}}$)
t	: time variable, $\frac{\partial}{\partial t}/\frac{d}{dt} \hat{=}$ partial/total derivative w.r.t. time,
$\kappa = \kappa(\boldsymbol{x})$	: (spatially varying) heat conductivity ($[\kappa] = \frac{\mathrm{W}}{\mathrm{Km}}$),
$T > 0$	: final time for "observation period" $[0, T]$,
$u_0 : \Omega \mapsto \mathbb{R}$	: initial temperature distribution in Ω,
$g : \partial\Omega \times [0, T] \mapsto \mathbb{R}$	: surface temperature, varying in space and time: $g = g(\boldsymbol{x}, t)$,
$f : \Omega \times [0, T] \mapsto \mathbb{R}$	: time-dependent heat source/sink ($[f] = \frac{\mathrm{W}}{\mathrm{m}^3}$): $f = f(\boldsymbol{x}, t)$.

Goal: derive PDE governing *transient* heat conduction.

The tools and concepts from Section 2.6 will be used again: Heat flux (→ § 2.6.1), energy conservation, and a flux law (→ § 2.6.4).

(6.1.2) Derivation of heat equation

For transient heat conduction the energy balance law (2.6.3) has to be supplemented by a *storage term* reflecting the fact that heat can accumulate:

Conservation of energy:

$$\frac{d}{dt}\int_V \rho u \,\mathrm{d}\boldsymbol{x} + \int_{\partial V} \mathbf{j}\cdot\boldsymbol{n}\,\mathrm{d}S = \int_V f\,\mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes" } V \tag{6.1.3}$$

energy stored in V power flux through ∂V heat generation in V

$\rho = \rho(\mathbf{x})$: (spatially varying) heat capacity ($[\rho] = \mathrm{JK}^{-1}$), uniformly positive, *cf.* (2.6.6).

As in § 2.6.7, now apply Gauss' Theorem Thm. 2.5.7

$$\int_V \operatorname{div}\mathbf{j}(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_{\partial V} \mathbf{j}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})\,\mathrm{dS}(\boldsymbol{x})\,, \quad \mathbf{j}:\Omega\to\mathbb{R}^d\,,$$

to the power flux integral in (6.1.3). This converts the surface integral to a volume integral over $\operatorname{div}\mathbf{j}$ and we get

$$\frac{d}{dt}\int_V \rho u\,\mathrm{d}\boldsymbol{x} + \int_V \operatorname{div}\mathbf{j}\,\mathrm{d}\boldsymbol{x} = \int_V f\,\mathrm{d}\boldsymbol{x} \qquad \text{for all "control volumes" } V$$

Now appeal to another version of the fundamental lemma of the calculus of variations, see Lemma 2.5.12, this time involving piecewise constant test functions.

Local form of energy balance law (Heat equation)

$$\frac{\partial}{\partial t}(\rho u)(\boldsymbol{x},t) + (\operatorname{div}_{\boldsymbol{x}}\mathbf{j})(\boldsymbol{x},t) = f(\boldsymbol{x},t) \quad \text{in} \quad \widetilde{\Omega}\,. \tag{6.1.4}$$

For standard materials the heat flux is linked to temperature variations by Fourier's law (→ § 2.6.4):

$$\mathbf{j}(\boldsymbol{x}) = -\kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\,, \quad \mathbf{x}\in\Omega\,. \tag{2.6.5}$$

From here we let all differential operators like $\mathbf{grad}$ and div act on the spatial independent variable $\boldsymbol{x}$. As earlier, the independent variables $\boldsymbol{x}$ and t will be omitted frequently. Watch out!

Now, plug Fourier's law

$$\mathbf{j}(\boldsymbol{x}) = -\kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\,, \quad \boldsymbol{x}\in\Omega\,, \tag{2.6.5}$$

into the local form of the energy balance law (6.1.4).

$$\boxed{\frac{\partial}{\partial t}(\rho u) - \operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in} \quad \widetilde{\Omega} := \Omega\times]0,T[} \,. \tag{6.1.5}$$

(6.1.6) Evolution problem for heat conduction

A pointed out in § 6.0.5 the PDE (6.1.5) has to be supplemented with initial conditions for $(\boldsymbol{x},t) \in \Omega \times \{0\}$ and boundary conditions for $(\boldsymbol{x},t) \in \partial\Omega\times]0,T[$. A simple and intuitive choice is

Dirichlet boundary conditions (fixed surface temperatur) on $\partial\Omega\times]0,T[$:

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \text{for} \quad (\boldsymbol{x},t) \in \partial\Omega\times]0,T[\,. \tag{6.1.7}$$

+ initial conditions for $t = 0$:

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \text{for all} \quad \boldsymbol{x} \in \Omega\,. \tag{6.1.8}$$

Terminology: (6.1.5) & (6.1.7) & (6.1.8) is a specimen of a

2nd-order parabolic initial-boundary value problem

(6.1.9) (Spatial) boundary conditions for 2nd-order parabolic IBVPs

As in Section 2.7 we appeal to physical intuition about heat conduction to justify that all of the following spatial boundary conditions make sense for the heat equation (6.1.5).

On $\partial\Omega]0,T[$ we can impose any of the boundary conditions discussed in Section 2.7:
- Dirichlet boundary conditions $u(\boldsymbol{x},t) = g(\boldsymbol{x},t)$, see (6.1.7) (fixed surface temperature),
- Neumann boundary conditions $\mathbf{j}(\boldsymbol{x},t)\cdot\boldsymbol{n} = -h(\boldsymbol{x},t)$ (fixed heat flux through surface),
- radiation boundary conditions $\mathbf{j}(\boldsymbol{x},t)\cdot\boldsymbol{n} = \Psi(u(\boldsymbol{x},t))$,

and any combination of these as discussed in Ex. 2.7.8, yet, *only one* of them at any part of $\partial\Omega\times]0,T[$, see Rem. 2.7.7.

For second order parabolic evolutions we can/must use the *same* spatial boundary conditions as for stationary second order elliptic boundary value problems.

Remark 6.1.10 (Compatible boundary and initial data)

We consider spatial Dirichlet boundary conditions (6.1.7) for the heat equation (6.1.5).

Natural regularity requirements for the temperature u and the Dirichlet data g:

u and g are continuous in time and space

➢ Natural compatibility requirement at initial time for $u_0 \in C^0(\overline{\Omega})$

$$g(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \partial\Omega \,.$$

6.1.2 Spatial variational formulation

(6.1.11) Model second-order linear parabolic evolution problem

Now we study the linear 2nd-order parabolic initial-boundary value problem with pure Dirichlet boundary conditions, introduced in the preceding section:

$$\frac{\partial}{\partial t}(\rho(\boldsymbol{x})u) - \operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u) = f \quad \text{in} \quad \widetilde{\Omega} := \Omega\times]0,T[\,, \tag{6.1.5}$$

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \text{for} \quad (\boldsymbol{x},t) \in \partial\Omega\times]0,T[\,, \tag{6.1.7}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \text{for all} \quad \boldsymbol{x} \in \Omega\,. \tag{6.1.8}$$

Here ρ and κ are uniformly positive (→ Def. 2.2.18) and bounded integrable functions on Ω. The source function $f = f(\boldsymbol{x},t)$ may depend on space an time and fulfills $f(\cdot,t) \in L^2(\Omega)$.

Imposed: Homogeneous Dirichlet boundary conditions $g \equiv 0$

The general case can be reduced to this by using the offset function trick, see Section 3.6.6, and solve the parabolic initial-boundary value problem for $w(\boldsymbol{x},t) := u(\boldsymbol{x},t) - \widetilde{g}(\boldsymbol{x},t)$, where $\widetilde{g}(\cdot,t)$ is an extension of the Dirichlet data g to $\widetilde{\Omega}$. Then w will satisfy homogeneous Dirichlet boundary conditions and solve an evolution equation with a modified source function $\widetilde{f}(\boldsymbol{x},t)$.

(6.1.12) Derivation of spatial variational formulation

Now we pursue the formal derivation of the *spatial* variational formulation of (6.1.5)–(6.1.7).

The steps completely mirror those discussed in Section 2.9, § 2.9.1. This paragraph should be reviewed again.

STEP 1:	*test PDE with functions* $v \in H^1_0(\Omega)$
	(Rule: do not test, where the solution is known, that is, on the boundary $\partial\Omega$)

Note: test function does *not depend on time*: $v = v(\boldsymbol{x})$!

STEP 2:	*integrate over domain* Ω

$$\int_\Omega \left(\frac{d}{dt}(\rho u) - \operatorname{div}(\kappa(\boldsymbol{x})\,\mathbf{grad}\,u)\right) v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v : \Omega \to \mathbb{R}\,, \quad v|_{\partial\Omega} = 0\,.$$

STEP 3:	*perform integration by parts in space* (by using Green's first formula, Thm. 2.5.9)

$$\int_\Omega \frac{d}{dt}(\rho u)(\boldsymbol{x})\, v(\boldsymbol{x}) + \kappa(\boldsymbol{x})\mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} -$$
$$\int_{\partial\Omega} \kappa(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\, \underbrace{v(\boldsymbol{x})}_{=0}\, \mathrm{d}S(\boldsymbol{x}) = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v : \Omega \to \mathbb{R}\,, \quad v|_{\partial\Omega} = 0\,.$$

Supplement 6.1.13.

For the concrete PDE (6.1.5) and boundary conditions (6.1.7) refer to Ex. 2.9.2 for a discussion of these steps in the stationary context. For more general boundary conditions study Ex. 2.9.6 to refresh yourself on how to obtain variational formulations. The derivation will include another STEP 4, which recasts boundary terms using the spatial boundary conditions.

△

The final step is the selection of an appropriate Sobolev space with respect to the dependence on the spatial variable. Following the guideline from Section 2.3.1 we pick the largest space, for which both left and right hand side of the formal variational problem are still well defined for every time t. With the arguments from Section 2.3.4 we find the space $H_0^1(\Omega)$.

Since the coefficient ρ must not depend on time, we arrive at the following variational problem:

Spatial variational form of (6.1.5)–(6.1.7): seek $t \in]0,T[\mapsto u(t) \in H_0^1(\Omega)$

$$\int_\Omega \rho(\boldsymbol{x})\dot{u}(t) v\, \mathrm{d}\boldsymbol{x} + \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u(t) \cdot \mathbf{grad}\, v\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x},t) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,, \tag{6.1.14}$$

$$u(0) = u_0 \in H_0^1(\Omega)\,. \tag{6.1.15}$$

Remark 6.1.16 (Function space valued functions)

What does it mean, when we write $u(t)$? Be aware that $t \mapsto u(t)$ describes a function space valued function on $]0,T[$, here assigning to every instance of time a function in $H_0^1(\Omega)$:

$$u :]0,T[\to H_0^1(\Omega)\,.$$

Also note that $\mathbf{grad} = \mathbf{grad}_{\boldsymbol{x}}$ acts on the spatial independent variables that are suppressed in the notation $u(t)$. Hence $t \mapsto \mathbf{grad}_{\boldsymbol{x}}\, u(t)$ is a function space valued function, too, with values in $(L^2(\Omega))^d$.

✎ Notation: $\dot{u}(t) = \frac{\partial u}{\partial t}(t) \mathrel{\hat{=}}$ (partial) derivative w.r.t. time: $\frac{\partial u}{\partial t}(t) \in H_0^1(\Omega)$

(6.1.17) Abstract linear parabolic evolution problems

Shorthand abstract notation for (6.1.14) (with obvious correspondences):

$$t \in]0,T[\mapsto u(t) \in V_0 \quad : \quad \begin{cases} \mathsf{m}(\dot{u}(t),v) + \mathsf{a}(u(t),v) = \ell(t)(v) & \forall v \in V_0 \,, \\ u(0) = u_0 \in V_0 \,. \end{cases} \tag{6.1.18}$$

Again, here $\ell(t) \hat{=}$ linear form valued function on $]0,T[$.

Concretely for evolution problem (6.1.5), (6.1.7), (6.1.8):

$$\begin{aligned} \mathsf{m}(\dot{u},v) &:= \int_\Omega \rho(\boldsymbol{x})\dot{u}(t)v\,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H_0^1(\Omega)\,, \\ \mathsf{a}(u,v) &:= \int_\Omega \kappa(\boldsymbol{x})\,\mathbf{grad}\,u(t)\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x}\,, \quad u,v \in H_0^1(\Omega)\,, \\ \ell(t)(v) &:= \int_\Omega f(\boldsymbol{x},t)v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad v \in H_0^1(\Omega)\,. \end{aligned}$$

Note that both m and a are *symmetric, positive definite* bilinear forms (→ Def. 2.2.40).

➢ Both m and a induce related energy norms $\|\cdot\|_a$ and $\|\cdot\|_m$ (→ Def. 2.2.43)

Since the bilinear formm does not depend on time, we conclude

$$\mathsf{m}(\dot{u},v) = \int_\Omega \rho(\boldsymbol{x})\dot{u}(t)v\,\mathrm{d}\boldsymbol{x} = \frac{d}{dt}\int_\Omega \rho(\boldsymbol{x})u(t)v\,\mathrm{d}\boldsymbol{x} = \frac{d}{dt}\mathsf{m}(u,v)\,,$$

and we can rewrite (6.1.18) equivalently as follows:

$$\boxed{t \in]0,T[\mapsto u(t) \in V_0 \quad : \quad \begin{cases} \frac{d}{dt}\mathsf{m}(u(t),v) + \mathsf{a}(u(t),v) = \ell(t)(v) & \forall v \in V_0 \,, \\ u(0) = u_0 \in V_0 \,. \end{cases}} \tag{6.1.19}$$

This is a linear evolution problem in the sense that the mapping that associates the solution $u = u(\boldsymbol{x},t)$ to the data (ℓ, u_0) is linear.

6.1.3 Stability of parabolic evolution problems

Now we are concerned with the stability of linear parabolic evolution problems, also known as well-posedness (→ Def. 2.4.13), more precisely, whether

1. solutions of (6.1.19) exist and are unique,
2. relevant norms of the solution can be bounded by suitable norms of the data u_0, f (and g for Dirichlet boundary conditions (6.1.7)).

Similar considerations for (stationary) abstract variational problems can be found in Section 2.4.2.

We investigate only whether $\|u(t)\|_{H^1(\Omega)}$ depends continuously on u_0 for all times t in the case $f \equiv 0$.

For the sake of simplicity we restrict ourselves to constant coefficients $\rho \equiv 1$ and $\kappa \equiv 1$. (The general case is not more difficult, because both ρ and κ are bounded and uniformly positive, see (2.6.6).)

We use that by the first Poincaré-Friedrichs inequality from Thm. 2.3.31

$$\exists \gamma > 0: \quad |v|^2_{H^1(\Omega)} \geq \gamma \|v\|^2_{L^2(\Omega)} \quad \forall v \in H^1_0(\Omega) \ . \tag{6.1.20}$$

In fact, Thm. 2.3.31 reveals $\gamma = \operatorname{diam}(\Omega)^{-2}$, but the numerical value of γ is not important for our considerations.

Remark 6.1.21 (Differentiating bilinear forms with time-dependent arguments)

Consider (temporally) smooth $u : [0,T] \mapsto V_0$, $v : [0,T] \mapsto V_0$ and a *symmetric* bilinear form $\mathsf{b} : V_0 \times V_0 \mapsto \mathbb{R}$. We are concerned with computing the temporal derivative $\frac{d}{dt}\mathsf{b}(u(t),v(t))$, because this will be a key step in the proof of stability estimates.

Perform formal Taylor expansion:

$$\begin{aligned}
\mathsf{b}(u(t+\tau),v(t+\tau)) &= \mathsf{b}(u(t)+\dot{u}(t)\tau + O(\tau^2), v(t)+\dot{v}(t)\tau + O(\tau^2)) \\
&= \mathsf{b}(u(t),v(t)) + \tau(\mathsf{b}(\dot{u}(t),v(t)) + \mathsf{b}(u(t),\dot{v}(t))) + O(\tau^2) \ . \\
\blacktriangleright \quad \frac{d}{dt}\mathsf{b}(u(t),v(t)) &= \lim_{\tau\to 0} \frac{\mathsf{b}(u(t+\tau),v(t+\tau)) - \mathsf{b}(u(t),v(t))}{\tau} \\
&= \mathsf{b}(\dot{u}(t),v(t)) + \mathsf{b}(u(t),\dot{v}(t)) \ .
\end{aligned}$$

This is a general product rule, see [4, Eq. (2.4.9)].

Lemma 6.1.22. Decay of solutions of parabolic evolutions

For $f \equiv 0$ the solution $u(t)$ of (6.1.14) *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 (6.1.20)*, and $\|\cdot\|_a$, $\|\cdot\|_m$ stand for the energy norms induced by $\mathsf{a}(\cdot,\cdot)$ and $\mathsf{m}(\cdot,\cdot)$, respectively.*

Proof. Multiply the solution of the parabolic IBVP with an exponential weight function:

$$w(t) := \exp(\gamma t)u(t) \in H^1_0(\Omega) \quad \Rightarrow \quad \dot{w} := \tfrac{dw}{dt}(t) = \gamma w(t) + \exp(\gamma t)\tfrac{du}{dt}(t) \ , \tag{6.1.23}$$

solves the parabolic IBVP

$$\begin{aligned}
\mathsf{m}(\dot{w},v) + \widetilde{\mathsf{a}}(w,v) &= 0 \quad \forall v \in V \ , \\
w(0) &= u_0 \ ,
\end{aligned} \tag{6.1.24}$$

with $\widetilde{\mathsf{a}}(w,v) = \mathsf{a}(w,v) - \gamma \mathsf{m}(w,v)$, γ from (6.1.20). To see this, use that $u(t)$ solves (6.1.19) with $f \equiv 0$ (elementary calculation).

Note: $$(6.1.20) \quad \Rightarrow \quad \widetilde{\mathsf{a}}(v,v) \geq 0 \quad \forall v \in V$$

We show the exponential decay of $\|\cdot\|_m$-norm of solution:

$$\frac{d}{dt}\frac{1}{2}\|w\|_m^2 = \frac{d}{dt}\frac{1}{2}\mathsf{m}(w,w) \overset{\text{Rem. 6.1.21}}{=} \mathsf{m}(\dot{w},w) = -\widetilde{\mathsf{a}}(w,w) \leq 0 \tag{6.1.25}$$

This confirms that $t \mapsto \|w(t)\|_m$ is a decreasing function, which involves

$$(6.1.25) \quad \Rightarrow \quad \|w(t)\|_m \leq \|w(0)\|_m \,,$$

and the first assertion of the Lemma is evident. Next, we verify the exponential decay of $|\cdot|_{H^1(\Omega)}$-norm of solution by a similar trick:

$$\frac{1}{2}\frac{d}{dt}\|w\|_{\widetilde{\mathsf{a}}}^2 \overset{\text{Rem. 6.1.21}}{=} \widetilde{\mathsf{a}}(\tfrac{d}{dt}w,w) = -\mathsf{m}(\tfrac{d}{dt}w,\tfrac{d}{dt}w) \leq 0 \quad \Rightarrow \quad \|w(t)\|_{\widetilde{\mathsf{a}}} \leq \|w(0)\|_{\widetilde{\mathsf{a}}} \,,$$

$$\blacktriangleright \quad \|w(t)\|_a^2 \leq \|w(0)\|_a^2 - \gamma(\underbrace{\|w(0)\|_m^2 - \|w(t)\|_m^2}_{\geq 0 \text{ by (6.1.25)}}) \,.$$

Reverting the transformation (6.1.23) gives the estimates for $|u|_{H^1(\Omega)}$. □

Dissipation of energy in parabolic evolutions

$\blacktriangleright$ Exponential decay of energy during parabolic evolution without excitation
("Parabolic evolutions dissipate energy")

Note that if the source term f does not depend on time, then the lemma asserts exponential convergence (in time) of $u = u(t)$ solution of (6.1.14) to the solution $u^* = u^*(\boldsymbol{x}) \in$ of the stationary boundary value problem

$$\int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u^*(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} = \int_\Omega f(\boldsymbol{x}) v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,.$$

$\blacktriangleright$ Exponential convergence (in time) to "equilibrium solution" in the case of time-independent excitation

6.1.4 Method of lines

Idea: Apply Galerkin discretization (→ Section 3.2) to abstract linear parabolic variational problem (6.1.19).

Recall from Section 3.2 that the fundamental ideas behind Galerkin discretization are

(I) the use of finite dimensional subspaces of the function spaces as trial and test spaces
➢ discrete variational problem,

(II) the choice of ordered bases in order to convert the discrete variational problem into a system of equations for unknown expansion coefficients.

We pursue this steps for the following abstract linear parabolic evolution problem posed over a vector space V_0:

$$t \in]0,T[\mapsto u(t) \in V_0 \quad : \quad \begin{cases} \mathsf{m}(\dot{u}(t),v) + \mathsf{a}(u(t),v) = \ell(t)(v) & \forall v \in V_0 \,, \\ u(0) = u_0 \in V_0 \,. \end{cases} \tag{6.1.19}$$

1st step: replace V_0 with a finite dimensional subspace $V_{0,N}$, $N := \dim V_{0,N} < \infty$

▶ (Spatially) discrete parabolic evolution problem

$$t \in]0,T[\mapsto u(t) \in V_{0,N} \quad : \quad \begin{cases} \mathsf{m}(\dot{u}_N(t),v_N) + \mathsf{a}(u_N(t),v_N) = \ell(t)(v_N) & \forall v_N \in V_{0,N} \,, \\ u_N(0) = \text{projection/interpolant of } u_0 \text{ in } V_{0,N} \,. \end{cases} \tag{6.1.27}$$

2nd step: introduce (ordered) basis $\mathfrak{B}_N := \{b_N^1, \dots, b_N^N\}$ of $V_{0,N}$

Next plug in basis expansion of $u_N(t)$ with *time-dependent* coefficients μ_i:

$$u_N(t) = \sum_{i=1}^{N} \mu_i(t) b_N^i \,. \tag{6.1.28}$$

Note that the basis functions themselves do not depend on time, of course.

Method-of-lines ordinary differential equation

Combining (6.1.27) and (6.1.28) we obtain

$$(6.1.27) \quad \Rightarrow \quad \begin{cases} \mathbf{M}\left\{\frac{d}{dt}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu}(t) = \vec{\varphi}(t) & \text{for } 0 < t < T \,, \\ \vec{\mu}(0) = \vec{\mu}_0 \,. \end{cases} \tag{6.1.30}$$

with

▷ s.p.d. stiffness matrix $\mathbf{A} \in \mathbb{R}^{N,N}$, $(\mathbf{A})_{ij} := \mathsf{a}(b_N^j, b_N^i)$ (independent of time),

▷ s.p.d. mass matrix $\mathbf{M} \in \mathbb{R}^{N,N}$, $(\mathbf{M})_{ij} := \mathsf{m}(b_N^j, b_N^i)$ (independent of time),

▷ source (load) vector $\vec{\varphi}(t) \in \mathbb{R}^N$, $(\vec{\varphi}(t))_i := \ell(t)(b_N^i)$ (time-dependent),

▷ $\vec{\mu}_0 \hat{=}$ coefficient vector of a projection of u_0 onto $V_{0,N}$.

Note:

(6.1.30) is an ordinary differential equation (ODE) for $t \mapsto \vec{\mu}(t) \in \mathbb{R}^N$

Conversion (6.1.19) ➙ (6.1.30) through Galerkin discretization *in space only* is known as method of lines.

(6.1.30) $\hat{=}$ a semi-discrete evolution problem

Discretized in space $\longleftrightarrow$ but still continuous in time

(6.1.31) Galerkin matrices in the method of lines ODE

For the concrete linear parabolic evolution problem (6.1.14)–(6.1.15) and spatial finite element discretization based on a finite element trial/test space $V_{0,N} \subset H^1(\Omega)$ we can compute

- the mass matrix $\mathbf{M}$ as the Galerkin matrix for the bilinear form $(u,v) \mapsto \int_\Omega \rho(\boldsymbol{x}) u(\boldsymbol{x}) v(\boldsymbol{x}) \,\mathrm{d}\boldsymbol{x}$, $u,v \in L^2(\Omega)$,

- the stiffness matrix $\mathbf{A}$ as Galerkin matrix arising from the bilinear form $(u,v) \mapsto \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u(\boldsymbol{x}) \cdot \mathbf{grad}\, v(\boldsymbol{x})\, \mathrm{d}\boldsymbol{x}$, $u,v \in H^1(\Omega)$.

The calculations are explained in Section 3.6.4 and Section 3.6.5 and may involve numerical quadrature.

Remark 6.1.32 (Spatial discretization options)

Beside the Galerkin approach any other method for spatial discretization of 2nd-order elliptic BVPs can be used in the context of the method of lines: the matrices $\mathbf{A}$, $\mathbf{M}$ may also be generated by finite differences ($\rightarrow$ Section 4.1), finite volume methods ($\rightarrow$ Section 4.2), or collocation methods ($\rightarrow$ Section 1.5.3).

6.1.5 Timestepping

For implementation we need a fully discrete evolution problem. This requires additional discretization in time:

semi-discrete evolution problem (6.1.30) + timestepping ▶ fully discrete evolution problem

Benefit of method of lines: we can apply already known integrators for initial value problems for ODEs to (6.1.30).

(6.1.33) Numerical integration of ordinary differential equations

First, refresh central concepts from numerical integration of initial value problems for ODEs, see [4, Chapter 11], [4, Chapter 12]:

- single step methods of order p, see [4, Def. 11.3.5] and [4, Section 11.3.2], defined as recursions in state space based on discrete evolution operators.
- explicit and implicit Runge-Kutta single step methods, see [4, Section 11.4], [4, Section 12.3], encoded by Butcher scheme [4, Eq. (11.4.11)], [4, Eq. (12.3.20)].
- the notion of a stiff initial value problem ($\rightarrow$ [4, Notion 12.2.9]),
- the definition of the stability function of a single step method, see [4, Thm. 12.3.27],
- the concept of L-stability [4, Def. 12.3.38] and how to verify it for Runge-Kutta methods.

6.1.5.1 Single step methods

(6.1.34) Fundamentals of single step methods

Recall: single step methods ($\rightarrow$ [4, Def. 11.3.5]) for ODE $\frac{d}{dt}\vec{\boldsymbol{\mu}} = F(t,\vec{\boldsymbol{\mu}})$

✦ are based on a temporal mesh $\{0 = t_0 < t_1 < \ldots < t_{M-1} < t_M := T\}$ (with local timestep size $\tau_j = t_j - t_{j-1}$),

- compute sequence $\left(\vec{\mu}^{(j)}\right)_{j=0}^{M}$ of approximations $\vec{\mu}^{(j)} \approx \mu(t_j)$ to the solution of (6.1.30) at the nodes of the temporal mesh according to

$$\vec{\mu}^{(j)} := \mathbf{\Psi}^{t_{j-1},t_j}\vec{\mu}^{(j-1)} := \mathbf{\Psi}(t_{j-1},t_j,\vec{\mu}^{(j-1)}) \;, \quad j = 1,\dots,M \;,$$

where $\mathbf{\Psi}$ is the discrete evolution defining the single step method, see [4, Def. 11.3.5]. Usually, we will have formulas for $\mathbf{\Psi}$ involving only evaluations of F at a few points in time.

Example 6.1.35 (Euler timestepping $\rightarrow$ [4, Section 11.2])

The Euler method is the simplest conceivable timestepping scheme. Here, we target the abstract variational initial value problem

$$\begin{aligned} &\mathbf{M}\left\{\tfrac{d}{dt}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu}(t) = \vec{\varphi}(t) \qquad \text{for } 0 < t < T \;, \\ &\vec{\mu}(0) = \vec{\mu}_0 \;. \end{aligned} \tag{6.1.30}$$

Explicit Euler method [4, Eq. (11.2.7)] $\hat{=}$ replace $\frac{d}{dt}$ in (6.1.30) with forward difference quotient, see [4, Rem. 11.2.8]:

$$(6.1.30) \;\blacktriangleright\; \mathbf{M}\vec{\mu}^{(j)} = \mathbf{M}\vec{\mu}^{(j-1)} - \tau_j\left(\mathbf{A}\vec{\mu}^{(j-1)} - \vec{\varphi}(t_{j-1})\right) \;, \quad j = 1,\dots,M-1 \;. \tag{6.1.36}$$

Implicit Euler method [4, Eq. (11.2.13)]: replace $\frac{d}{dt}$ in (6.1.30) with backward difference quotient

$$(6.1.30) \;\blacktriangleright\; \mathbf{M}\vec{\mu}^{(j)} = \mathbf{M}\vec{\mu}^{(j-1)} - \tau_j\left(\mathbf{A}\vec{\mu}^{(j)} - \vec{\varphi}(t_j)\right) \;, \quad j = 1,\dots,M-1 \;. \tag{6.1.37}$$

Note that both (6.1.36) and (6.1.37) require the solution of a linear system of equations in each step

$$\begin{aligned} &(6.1.36): \quad \vec{\mu}^{(j)} = \vec{\mu}^{(j-1)} + \tau_j\mathbf{M}^{-1}(\vec{\varphi}(t_{j-1}) - \mathbf{A}\vec{\mu}^{(j-1)}) \;, \\ &(6.1.37): \quad \vec{\mu}^{(j)} = (\tau_j\mathbf{A} + \mathbf{M})^{-1}\left(\mathbf{M}\vec{\mu}^{(j-1)} + \tau_j\vec{\varphi}(t_j)\right) \;. \end{aligned}$$

Recall [4, Section 11.3.2]: both Euler method are of first order.

Example 6.1.38 (Crank-Nicolson timestepping)

Crank-Nicolson method = implicit midpoint rule: replace $\frac{d}{dt}$ in (6.1.30) with symmetric difference quotient and average right hand side:

$$\mathbf{M}\left\{\tfrac{d}{dt}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu}(t) = \vec{\varphi}(t)$$
$$\Downarrow$$
$$\mathbf{M}\frac{\vec{\mu}^{(j)} - \vec{\mu}^{(j-1)}}{\tau} = -\tfrac{1}{2}\mathbf{A}\left(\vec{\mu}^{(j)} + \vec{\mu}^{(j-1)}\right) + \tfrac{1}{2}(\vec{\varphi}(t_j) + \vec{\varphi}(t_{j-1})) \;. \tag{6.1.39}$$

This yields a method that is 2nd-order consistent.

Both the Euler method from Ex. 6.1.35 and the Crank-Nicolson timestepping from Ex. 6.1.38 belong to a famous class of single step methods, the

Runge-Kutta single step methods $\rightarrow$ [4, Section 11.4], [4, Section 12.3]

Definition 6.1.40. General Runge-Kutta method $\rightarrow$ [4, Def. 12.3.18]

For coefficients $b_i, a_{ij} \in \mathbb{R}$, $c_i := \sum_{j=1}^{s} a_{ij}$, $i,j = 1,\dots,s$, $s \in \mathbb{N}$, the discrete evolution $\mathbf{\Psi}^{s,t}$ of an s-stage Runge-Kutta single step method (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, is defined by

$$\mathbf{k}_i := \mathbf{f}(t + c_i\tau, \mathbf{y} + \tau \sum_{j=1}^{s} a_{ij}\mathbf{k}_j) \;, \quad i = 1,\dots,s \quad , \quad \mathbf{\Psi}^{t,t+\tau}\mathbf{y} := \mathbf{y} + \tau \sum_{i=1}^{s} b_i \mathbf{k}_i \;.$$

The $\mathbf{k}_i \in \mathbb{R}^d$ are called increments.

(6.1.41) Butcher scheme

Shorthand notation for s-stage Runge-Kutta methods: Butcher scheme $\rightarrow$ [4, Eq. (12.3.20)]

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} \quad \hat{=} \quad \begin{array}{c|ccccc} c_1 & a_{11} & a_{12} & \dots & \dots & a_{1s} \\ c_2 & a_{21} & \ddots & & & a_{2s} \\ \vdots & \vdots & & \ddots & & \vdots \\ c_s & a_{s1} & \vdots & & & a_{ss} \\ \hline & b_1 & b_2 & \dots & \dots & b_s \end{array} \quad , \quad \mathbf{c}, \mathbf{b} \in \mathbb{R}^s, \quad \mathfrak{A} \in \mathbb{R}^{s,s} \;. \tag{6.1.42}$$

(6.1.43) Application of general Runge-Kutta timestepping to method of lines ODE

Concretely for linear parabolic evolution after spatial semi-discretization: Application of s-stage Runge-Kutta method to the method of lines ODE

$$\mathbf{M}\left\{\frac{d}{dt}\vec{\boldsymbol{\mu}}(t)\right\} + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{\varphi}}(t) \quad \Leftrightarrow \quad \dot{\vec{\boldsymbol{\mu}}} = \underbrace{\mathbf{M}^{-1}(\vec{\boldsymbol{\varphi}}(t) - \mathbf{A}\vec{\boldsymbol{\mu}}(t))}_{=\mathbf{f}(t,\vec{\boldsymbol{\mu}})} \;. \tag{6.1.30}$$

Then simply plug this into the formulas of Def. 6.1.40.

Timestepping scheme for (6.1.30): compute $\vec{\boldsymbol{\mu}}^{(j+1)}$ from $\vec{\boldsymbol{\mu}}^{(j)}$ through

$$\vec{\boldsymbol{\kappa}}_i \in \mathbb{R}^N\colon \quad \mathbf{M}\vec{\boldsymbol{\kappa}}_i + \sum_{m=1}^{s} \tau a_{im}\mathbf{A}\vec{\boldsymbol{\kappa}}_m = \vec{\varphi}(t_j + c_i\tau) - \mathbf{A}\vec{\boldsymbol{\mu}}^{(j)} \;, \quad i = 1,\dots,s \;, \tag{6.1.44}$$

$$\vec{\boldsymbol{\mu}}^{(j+1)} = \vec{\boldsymbol{\mu}}^{(j)} + \tau \sum_{m=1}^{s} \vec{\boldsymbol{\kappa}}_m b_m \;. \tag{6.1.45}$$

Note: For an implicit RK-method (6.1.44) is a linear system of equations of size Ns. Using the Kronecker product of matrices for $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$, $m,n,l,k \in \mathbb{N}$, defined as ($\rightarrow$ [4, Def. 1.4.16])

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{nl,nk} \;,$$

(6.1.44) can be recast into the following form

$$(6.1.44) \quad \Leftrightarrow \quad (\mathbf{I}_s \otimes \mathbf{M} + \tau \mathfrak{A} \otimes \mathbf{A}) \begin{bmatrix} \vec{\kappa}_1 \\ \vdots \\ \vec{\kappa}_s \end{bmatrix} = \begin{bmatrix} \vec{\varphi}(t_j + c_1\tau) - \mathbf{A}\vec{\mu}^{(j)} \\ \vdots \\ \vec{\varphi}(t_j + c_s\tau) - \mathbf{A}\vec{\mu}^{(j)} \end{bmatrix} . \tag{6.1.46}$$

6.1.5.2 Stability

In Section 6.1.3 we have seen that the energy norm and L^2-norm of solutions of linear parabolic evolution problems remain bounded for all times. The same arguments confirm that this remains true for the solution $\vec{\mu}(t)$ of the semi-discrete evolution (6.1.30). However, some well-established single step methods applied to (6.1.30) may not enjoy this stability.

Experiment 6.1.47 (Convergence of Euler timestepping for M.O.L. ODE)

Parabolic evolution problem in one spatial dimension (IBVP):

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} \quad \text{in } [0,1]\times]0,1[\,, \tag{6.1.48}$$

$$u(t,0) = u(t,1) = 0 \quad \text{for } 0 \le t \le 1\,, \quad u(0,x) = \sin(\pi x) \quad \text{for } 0 < x < 1\,. \tag{6.1.49}$$

$$\blacktriangleright \quad \text{exact solution} \qquad u(t,x) = \exp(-\pi^2 t)\sin(\pi x)\,. \tag{6.1.50}$$

- Spatial finite element Galerkin discretization by means of linear finite elements ($V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M})$) on equidistant mesh $\mathcal{M}$ with meshwidth $h := \frac{1}{N+1}$ $\rightarrow$ Section 1.5.2.2.
- $u_{N,0} := \mathsf{I}_1 u_0$ by linear interpolation on $\mathcal{M}$, see Section 5.3.1.
- Timestepping by explicit and implicit Euler method (6.1.36), (6.1.37) with uniform timestep $\tau := \frac{1}{M}$.

We obtain tridiagonal $N \times N$ Galerkin matrices, see (1.5.77):

$$\mathbf{A} = \frac{1}{h}\begin{bmatrix} 2 & -1 & 0 & & & & 0 \\ -1 & 2 & -1 & & & & \\ 0 & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & 0 \\ & & & & -1 & 2 & -1 \\ 0 & & & & 0 & -1 & 2 \end{bmatrix}, \quad \mathbf{M} = \frac{h}{6}\begin{bmatrix} 4 & 1 & 0 & & & & 0 \\ 1 & 4 & 1 & & & & \\ 0 & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & 0 \\ & & & & 1 & 4 & 1 \\ 0 & & & & 0 & 1 & 4 \end{bmatrix}.$$

MATLAB code 6.1.51: Euler timestepping for (6.1.48)

```
function [errex,errimp] = sinevl(N,M,u)
% Solve fully discrete two-point parabolic evolution problem (6.1.48)
```

```
% in [0,1]×]0,1[.  Use both explicit and implicit Euler method for timestepping
% N: number of spatial grid cells
% M: number of timesteps
% u:  handle of type @(t,x) to exact solution

if (nargin < 3), u = @(t,x) (exp(-(pi^2)*t).*sin(pi*x)); end % Exact solution

h = 1/N; tau = 1/M; % Spatial and temporal meshwidth
x = h:h:1-h;     % Spatial grid, interior points

% Finite element stiffness and mass matrix
Amat = gallery('tridiag',N-1,-1,2,-1)/h;
Mmat = h/6*gallery('tridiag',N-1,1,4,1);
Xmat = Mmat+tau*Amat;

mu0 = u(0,x)'; % Discrete initial value
mui = mu0; mue = mu0;

%Timestepping
erre = 0; erri = 0;
for k=1:M
  mue = mue - tau*(Mmat\(Amat*mue)); % explicit Euler step
  mui = Xmat\(Mmat*mui);             % implicit Euler step
  utk = u(k*tau,x)';
  erre = erre + norm(mue-utk)^2;     % Computation of error norm
  erri = erri + norm(mui-utk)^2;
end

errex = sqrt(erre*h*tau);
errimp = sqrt(erri*h*tau);
```

C++11 code 6.1.52: Euler timestepping for (6.1.48) ➙ **GITLAB**

```
// arguments:
// int N number of spatial grid cells
// int M number of timesteps
//
// returns:
// tuple containing
// the error from explicit Euler timestepping
// the error from implict Euler timestepping
//
// Solve fully discrete two-point parabolic evolution problem (6.1.48)
// in [0,1]×]0,1[.  Use both explicit and implicit Euler method for timestepping
std::tuple<double, double> sinevl(int N, int M)
{
  //exact solution
```

```
  const auto u = [] (double t, Eigen::ArrayXd& x) {
    return std::exp(-pi*pi*t)*(pi*x).sin();
  };
  const double h = 1.0/N; //spatial meshwidth
  const double tau = 1.0/M; //temporal meshwidth
  Eigen::ArrayXd x = Eigen::ArrayXd::LinSpaced(N - 1, h, 1.0 - h);
      //spatial grid, interior
      points

  //finite element stiffness and mass matrix
  Eigen::SparseMatrix<double> Amat = NPDE::tridiagonal(N - 1, -1.0,
      2.0, -1.0)/h;
  Eigen::SparseMatrix<double> Mmat = NPDE::tridiagonal(N - 1, 1.0,
      4.0, 1.0)*h/6.0;
  Eigen::SparseMatrix<double> Xmat = Mmat + tau*Amat;

  Eigen::VectorXd mu0 = u(0, x); //discrete initial values

  Eigen::VectorXd mui = mu0;
  Eigen::VectorXd mue = mu0;

  //timestepping
  double erre = 0.0;
  double erri = 0.0;
  for (int k = 1; k <= M; k++) {
    mue = mue - tau*(Mmat/(Amat*mue)); //explicit Euler timestep
    mui = Xmat/(Mmat*mui);             //implicit Euler timestep
    Eigen::VectorXd utk = u(k*tau, x);
    //computation of error norms
    double norme = (mue - utk).norm();
    double normi = (mui - utk).norm();
    erre = erre + norme*norme;
    erri = erri + normi*normi;
  }

  return std::make_tuple(std::sqrt(erre*h*tau),
      std::sqrt(erri*h*tau));
}
```

Evaluation of approximate space-time L^2-norm of the discretization error:

$$\mathtt{err}^2 := h\tau \cdot \sum_{j=1}^{M}\sum_{i=1}^{N} |u(t_j,x_i) - \mu_i^{(j)}|^2 \,. \tag{6.1.53}$$

($N \mathrel{\hat{=}}$ no. of grid points in space, $M \mathrel{\hat{=}}$ no. of timesteps.)

Space-time (discrete) L^2-norm of error for explicit Euler timestepping:

$N\backslash M$	50	100	200	400	800	1600	3200
5	Inf	0.009479	0.006523	0.005080	0.004366	0.004011	0.003834
10	Inf	Inf	Inf	Inf	0.001623	0.001272	0.001097
20	Inf	Inf	Inf	Inf	Inf	Inf	0.000405
40	Inf	Inf	Inf	Inf	Inf	Inf	Inf
80	Inf	Inf	Inf	Inf	Inf	Inf	Inf
160	Inf	Inf	Inf	Inf	Inf	Inf	Inf
320	Inf	Inf	Inf	Inf	Inf	Inf	Inf

Space-time (discrete) L^2-norm of error for implicit Euler timestepping:

$N\backslash M$	50	100	200	400	800	1600	3200
5	0.007025	0.001828	0.000876	0.002257	0.002955	0.003306	0.003482
10	0.009641	0.004500	0.001826	0.000461	0.000228	0.000575	0.000749
20	0.010303	0.005175	0.002509	0.001149	0.000461	0.000116	0.000058
40	0.010469	0.005345	0.002681	0.001321	0.000634	0.000289	0.000116
80	0.010511	0.005387	0.002724	0.001364	0.000677	0.000332	0.000159
160	0.010521	0.005398	0.002734	0.001375	0.000688	0.000343	0.000170
320	0.010524	0.005400	0.002737	0.001378	0.000691	0.000346	0.000172

For explicit Euler timestepping we observe a glaring instability (exponential blow-up) in case of *large timestep combined with fine mesh*.

Implicit Euler timestepping incurs no blow-up for any combination of spatial and temporal mesh width.

Experiment 6.1.54 (MATLAB `ode45` for discrete parabolic evolution)

Same IBVP and spatial discretization as in Exp. 6.1.47.

Timestepping by means of adaptive *explicit* Runge-Kutta timestepping using MATLAB's standard integrator `ode45`, see [4, Rem. 11.5.23].

Monitored:
- ✦ Number of timesteps as a function of spatial meshwidth h,
- ✦ discrete L^2-error (6.1.53).

C++11 code 6.1.55: ode45 applied semi-discrete (6.1.48)

```
function [Nsteps,err] = peode45(N,tol,u)
% Solving fully discrete two-point parabolic evolution problem (6.1.48)
% in [0,1]×]0,1[ by means of adaptiv MATLAB standard Runge-Kutta integrator.
if (nargin < 3), u = @(t,x) (exp(-(pi^2)*t).*sin(pi*x)); end % Exact solution

% Finite element stiffness and mass matrix, see Sect. 1.5.2.2
h = 1/N; % spatial meshwidth
Amat = gallery('tridiag',N-1,-1,2,-1)/h;
Mmat = h/6*gallery('tridiag',N-1,1,4,1);
x = h:h:1-h;     % Spatial grid, interior points
```

```
mu0 = u(0,x)'; % Discrete initial value
fun = @(t,muv) -(Mmat\(Amat*muv)); % right hand side of ODE

opts = odeset('reltol',tol,'abstol',0.01*tol);
[t,mu] = ode45(fun,[0,1],mu0,opts);

Nsteps = length(t);
[T,X] = meshgrid(t,x); err = norm(mu'-u(T,X),'fro');
```

C++11 code 6.1.56: ode45 applied semi-discrete (6.1.48) ➔ **GITLAB**

```
// arguments:
// int N number of spatial grid cells
// double tol tolerance value for ode45 integrator
//
// returns:
// tuple containing
// the number of steps taken
// the error
//
// Solve fully discrete two-point parabolic evolution problem (6.1.48)
// in [0,1]×]0,1[ by means of an adaptive Runge-Kutta integrator.
std::tuple<int, double> pode(int N, double tol)
{

  // Lambda function providing xact solution
  const auto u = [] (const Eigen::ArrayXXd& T, const Eigen::ArrayXXd&
      X) {
    return (-pi*pi*T).exp()*(pi*X).sin();
  };

  const double h = 1.0/N; //spatial meshwidth
  Eigen::ArrayXd x = Eigen::ArrayXd::LinSpaced(N - 1, h, 1.0 - h);
      //spatial grid, interior
      points

  //finite element stiffness and mass matrix
  Eigen::SparseMatrix<double> Amat = NPDE::tridiagonal(N - 1, -1.0,
      2.0, -1.0)/h;
  Eigen::SparseMatrix<double> Mmat = NPDE::tridiagonal(N - 1, 1.0,
      4.0, 1.0)*h/6.0;

  Eigen::VectorXd mu0 = u(Eigen::ArrayXd::Constant(N - 1, 0.0), x);
      //discrete initial
      values

  //right hand side of ODE
  auto odefun = [&Amat, &Mmat](const Eigen::VectorXd& x,
      Eigen::VectorXd& dxdt, const double t) {
```

```
    dxdt = -Mmat/(Amat*x);
  };

  //solve the system
  Eigen::ArrayXXd mu;
  Eigen::ArrayXd ts;
  std::tie(ts, mu) = NPDE::ode45(odefun, 0.0, 1.0, mu0, 0.01*tol,
      tol);

  int Nsteps = ts.size();
  Eigen::ArrayXXd T;
  Eigen::ArrayXXd X;
  std::tie(T, X) = NPDE::meshgrid(ts, x);

  double err = (mu - u(T, X)).matrix().norm();

  return std::make_tuple(Nsteps, err);
}
```

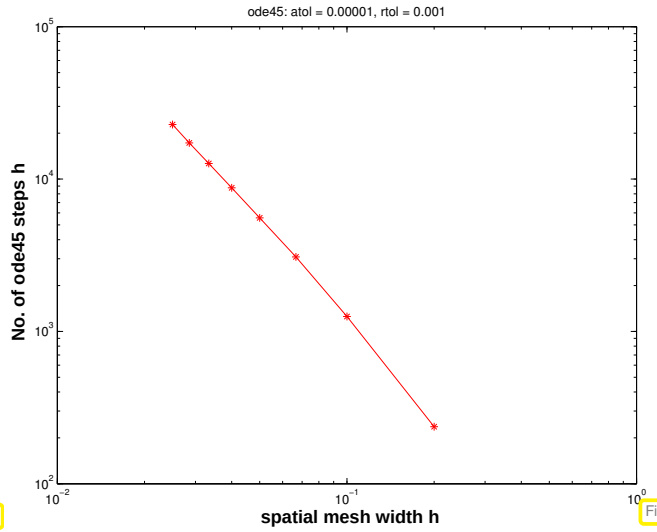


Fig. 270

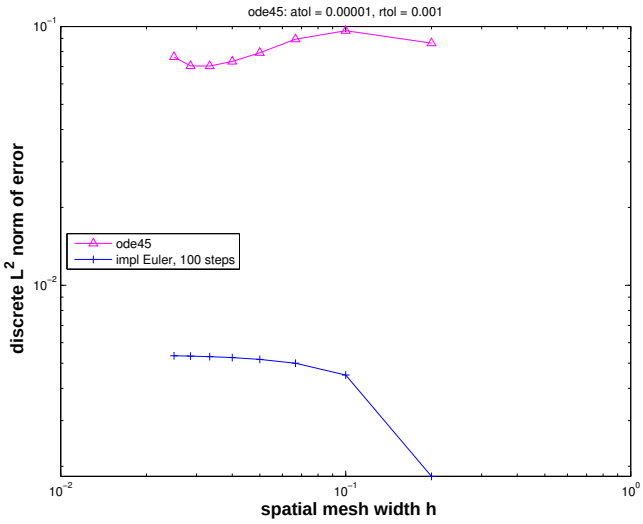


Fig. 271

Observations:

- `ode45`: dramatic increase of no. of timesteps for $h_{\mathcal{M}} \to 0$ without gain in accuracy.
- Implicit Euler achieves better accuracy with only 100 equidistant timesteps!

This reminds us of the stiff initial value problems studied in [4, Section 12.2]:

Notion 6.1.57. Stiff IVP $\to$ **[4, Notion 12.2.9]**

An initial value problem for an ODE is called stiff, if stability imposes much tighter timestep constraints on *explicit single step methods* than the accuracy requirements.

Admittedly, this is a fuzzy notion. Yet, it cannot be fleshed out on the abstract level, but has to be discussed for concrete evolution problem, which is done next.

Let us try to understand, why semi-discrete parabolic evolutions (6.1.30) arising from the method of lines lead to stiff initial value problems.

(6.1.58) Diagonalization of method-of-lines ODE

Analysis technique: **Diagonalization**, *cf.* [4, Eq. (12.1.37)]

Diagonalization (also called spectral decomposition) is a very versatile technique for decomposing a big problem into *decoupled* small problems. Here we discuss it for the method-of-lines ODE (6.1.30):

$$\mathbf{M}\left\{\tfrac{d}{dt}\vec{\boldsymbol{\mu}}(t)\right\} + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{\varphi}}(t) \; . \tag{6.1.30}$$

Let $\vec{\varphi}_1, \dots, \vec{\varphi}_N \in \mathbb{R}^N$ denote the N linearly independent generalized eigenvectors satisfying

$$\mathbf{A}\vec{\boldsymbol{\psi}}_i = \lambda_i \mathbf{M}\vec{\boldsymbol{\psi}}_i \quad , \quad \vec{\boldsymbol{\psi}}_j^\top \mathbf{M}\vec{\boldsymbol{\psi}}_i = \delta_{ij} \; , \quad 1 \le i,j \le N \; . \tag{6.1.59}$$

with positive eigenvalues $\lambda_i > 0$. Introducing the regular square matrices

$$\mathbf{T} = (\vec{\boldsymbol{\psi}}_1, \dots, \vec{\boldsymbol{\psi}}_N) \in \mathbb{R}^{N,N} \; , \tag{6.1.60}$$

$$\mathbf{D} := \operatorname{diag}(\lambda_1 \dots, \lambda_N) \in \mathbb{R}^{N,N} \; , \tag{6.1.61}$$

we can rewrite (6.1.59) as

$$\mathbf{AT} = \mathbf{MTD} \quad , \quad \mathbf{T}^\top \mathbf{MT} = \mathbf{I} \; . \tag{6.1.62}$$

Supplement 6.1.63.

The existence of eigenvectors $\vec{\varphi}_i$ with positive associated eigenvalues is guaranteed, since both $\mathbf{A}$ and $\mathbf{M}$ are positive definite: Thus, the generalized eigenvalue problem (6.1.59) can be transformed to a standard eigenvalue problem for a symmetric matrix by multiplying from left and right with the inverse of the "square root" $\mathbf{M}^{1/2}$ of $\mathbf{M}$, see [4, Section 8.3]. Then apply the result that every symmetric matrix can be diagonalized by means of an orthogonal transformation [4, Cor. 7.1.9].

△

Diagonalization approach ❶: Expand $\vec{\boldsymbol{\mu}}(t)$ in the eigenvectors $\vec{\boldsymbol{\psi}}_i$ (with time-dependent expansion coefficients)

$$\vec{\boldsymbol{\mu}}(t) = \sum_{k=1}^{N} \eta_k(t) \vec{\boldsymbol{\psi}}_k \; , \tag{6.1.64}$$

and plug this expansion into

$$\mathbf{M}\left\{\tfrac{d}{dt}\vec{\boldsymbol{\mu}}(t)\right\} + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{\varphi}}(t) \; . \tag{6.1.30}$$

Using (6.1.59) this yields

$$\sum_{k=1}^{N} \tfrac{d}{dt}\eta_k(t)\mathbf{M}\vec{\boldsymbol{\psi}}_k + \eta_k(t)\lambda_k \mathbf{M}\vec{\boldsymbol{\psi}}_k = \vec{\boldsymbol{\varphi}}(t) \; .$$

Multiply from left with $\vec{\boldsymbol{\psi}}_i^\top$, $i = 1, \dots, N$, and use (6.1.59) again:

$$\blacktriangleright \quad \tfrac{d}{dt}\eta_i(t) + \lambda_i \eta_i(t) = \vec{\boldsymbol{\psi}}_i^\top \vec{\boldsymbol{\varphi}}(t) \; .$$

We have ended up with N decoupled scalar linear ODEs.

Diagonalization approach ❷: Using compact matrix notations, set

$$\vec{\mu}(t) = \mathbf{T}\vec{\eta}(t) \quad \Leftrightarrow \quad \mathbf{T}^\top \mathbf{M}\vec{\mu}(t) = \vec{\eta}(t) \ .$$

Substitute this in (6.1.30) and invoke (6.1.62):

$$\blacktriangleright \qquad \mathbf{MT}\tfrac{d}{dt}\vec{\eta}(t) + \mathbf{MTD}\vec{\eta}(t) = \vec{\varphi}(t) \ .$$

Then multiply this equation from left with $\mathbf{T}^\top$ and use (6.1.62) again:

$$\blacktriangleright \qquad \tfrac{d}{dt}\vec{\eta}(t) + \mathbf{D}\vec{\eta}(t) = \mathbf{T}^\top\vec{\varphi}(t) \ .$$

Through both approaches, setting $\vec{\eta} = (\eta_1, \dots, \eta_N)^\top \in \mathbb{R}^N$, we have thus arrived at the transformed ODE

$$(6.1.30) \quad \overset{\vec{\eta}:=\mathbf{T}^\top\mathbf{M}\vec{\mu}}{\Longrightarrow} \quad \tfrac{d}{dt}\vec{\eta}(t) + \mathbf{D}\vec{\eta} = \mathbf{T}^\top\vec{\varphi}(t) \ . \tag{6.1.65}$$

(Note that, thanks to the $\mathbf{M}$-orthogonality of the $\vec{\psi}_i$ stated in (6.1.59), (6.1.64) is equivalent to $\vec{\eta} = \mathbf{T}^\top\mathbf{M}\vec{\mu}$.)

➤ Since $\mathbf{D}$ is *diagonal*, (6.1.65) amounts to N decoupled scalar ODEs (for eigencomponents η_i of $\vec{\mu}$).

Note: for $\vec{\varphi} \equiv 0$, $\lambda > 0$: $\eta_i(t) = \exp(-\lambda_i t)\eta_i(0) \to 0$ for $t \to \infty$

(6.1.66) Diagonalization applied to explicit Euler timestepping

As in [4, Eq. (12.1.40)] the above diagonalizing transformation can be applied to the explicit Euler timestepping (6.1.36) (for $\vec{\varphi} \equiv 0$, uniform timestep $\tau > 0$)

$$\vec{\mu}^{(j)} = \vec{\mu}^{(j-1)} - \tau\mathbf{M}^{-1}\mathbf{A}\vec{\mu}^{(j-1)} \quad \overset{\vec{\eta}:=\mathbf{T}^\top\mathbf{M}\vec{\mu}}{\blacktriangleright} \quad \vec{\eta}^{(j)} = \vec{\eta}^{(j-1)} - \tau\mathbf{D}\vec{\eta}^{(j-1)} \ ,$$

that is, the decoupling of eigencomponents carries over to the explicit Euler method: for $i = 1, \dots, N$

$$\eta_i^{(j)} = \eta_i^{(j-1)} - \tau\lambda_i\eta^{(j-1)} \quad \Rightarrow \quad \boxed{\eta_i^{(j)} = (1-\tau\lambda_i)^j\eta_i^{(0)}} \ . \tag{6.1.67}$$

$$\Downarrow$$

$$|1-\tau\lambda_i| < 1 \quad \Leftrightarrow \quad \lim_{j\to\infty}\eta_i^{(j)} = 0 \ .$$

The condition $|1-\tau\lambda_i| < 1$ enforces a

timestep size constraint: $\quad \tau < \dfrac{2}{\lambda_i}$

in order to achieve the qualitatively correct behavior $\lim_{j\to\infty}\eta_i^{(j)} = 0$ and to avoid blow-up $\lim_{j\to\infty}|\eta_i^{(j)}| = \infty$: the timestep size constraint (6.1.66) is necessary *only* for the sake of stability (not in order to guarantee a prescribed accuracy).

This accounts to the observed blow-ups in Exp. 6.1.47. On the other hand, adaptive stepsize control [4, Section 11.5] manages to ensure the timestep constraint, but the expense of prohibitively small timesteps that render the method *grossly inefficient*, *if some of the* λ_i *are large*.

Remark 6.1.68 (von Neumann stability analysis)

The diagonalization approach to the stability analysis of timestepping methods for fully discrete linear evolution problems is a generalization of the classical von Neumann stability analysis, which applies to cases, where the eigenfunctions of the generalized eigenvalue problem (6.1.59) are Fourier harmonics (sines/cosines).

This special version of stability analysis will be covered in Section 8.4.2.

The next numerical demonstrations and Lemma show that $\lambda_{\max} := \max_i \lambda_i$ will inevitably become huge for finite element discretization on fine meshes.

Experiment 6.1.69 (Behavior of generalized eigenvalues of $\mathbf{A}\vec{\mu} = \lambda\mathbf{M}\vec{\mu}$)

Bilinear forms associated with parabolic IBVP and homogeneous Dirichlet boundary conditions

$$\mathsf{a}(u,v) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}x \quad , \quad \mathsf{m}(u,v) = \int_\Omega u(x)v(x)\,\mathrm{d}x \,, \quad u,v \in H_0^1(\Omega) \,.$$

Linear finite element Galerkin discretization, see Section 1.5.2.2 for 1D, and Section 3.3 for 2D.

Numerical experiments in 1D & 2D:

- $\Omega =]0,1[$, equidistant meshes $\rightarrow$ Exp. 6.1.47
- "disk domain" $\Omega = \{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\| < 1\}$, sequence of regularly refined meshes.

Monitored: largest and smallest generalized eigenvalue

MATLAB LehrFEM [1] code 6.1.70: Computation of extremal generalized eigenvalues

```
% LehrFEM MATLAB script for computing Dirichlet eigenvalues of Laplacian
% on a unit disc domain.

GD_HANDLE = @(x,varargin)zeros(size(x,1),1); % Zero Dirichlet data
H0 =[ .25 .2 .1 .05 .02 .01 0.005]'; % target mesh widths
NRef = length(H0); % Number of refinement steps

% Variables for mesh widths and eigenvalues
M_W = zeros(NRef,1); lmax = M_W; lmin = M_W;

% Main refinement loop
for iter = 1:NRef

```

```
% Set parameters for mesh
  C = [0 0];                % Center of circle
  R = 1;                    % Radius of circle
  BBOX = [-1 -1; 1 1];      % Bounding box
  DHANDLE = @dist_circ;     % Signed distance function
  HHANDLE = @h_uniform;     % Element size function
  FIXEDPOS = [];            % Fixed boundary vertices of the mesh
  DISP = 0;                 % Display flag

% Mesh generation
  Mesh =
     init_Mesh(BBOX,H0(iter),DHANDLE,HHANDLE,FIXEDPOS,DISP,C,R);
  Mesh = add_Edges(Mesh);   % Provide edge information
  Loc = get_BdEdges(Mesh);  % Obtain indices of edges on ∂Ω
  Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
  Mesh.BdFlags(Loc) = -1;   % Flag boundary edges
  Mesh.ElemFlag = zeros(size(Mesh.Elements,1),1);
  M_W(iter) = get_MeshWidth(Mesh); % Get mesh width

  fprintf('Mesh on level %i: %i elements, h =
     %f\n',iter,size(Mesh,1),M_W(iter));
% Assemble stiffness matrix and mass matrix
  A = assemMat_LFE(Mesh,@STIMA_Lapl_LFE,P7O6());
  M = assemMat_LFE(Mesh,@MASS_LFE,P7O6());
% Incorporate Dirichlet boundary data (nothing to do here)
  [U,FreeNodes] = assemDir_LFE(Mesh,-1,GD_HANDLE);
  A = A(FreeNodes,FreeNodes);
  M = M(FreeNodes,FreeNodes);

% Use MATLAB's built-in eigs-function to compute the
% extremal eigenvalues, see [4, Section 7.4].
  NEigen = 6;
  d = eigs(A,M,NEigen,'sm'); lmin(iter) = min(d);
  d = eigs(A,M,NEigen,'lm'); lmax(iter) = max(d);
end

figure; plot(M_W,lmin,'b-+',M_W,lmax,'r-*'); grid on;
set(gca,'XScale','log','YScale','log','XDir','reverse');
title('\bf Eigenvalues of Laplacian on unit disc');
xlabel('{\bf mesh width h}','fontsize',14);
ylabel('{\bf generalized eigenvalues}','fontsize',14);
legend('\lambda_{min}','\lambda_{max}','Location','NorthWest')
p = polyfit(log(M_W),log(lmax),1);
add_Slope(gca,'east',p(1));

print -depsc2 '../../../Slides/NPDEPics/geneigdisklfe.eps';
```

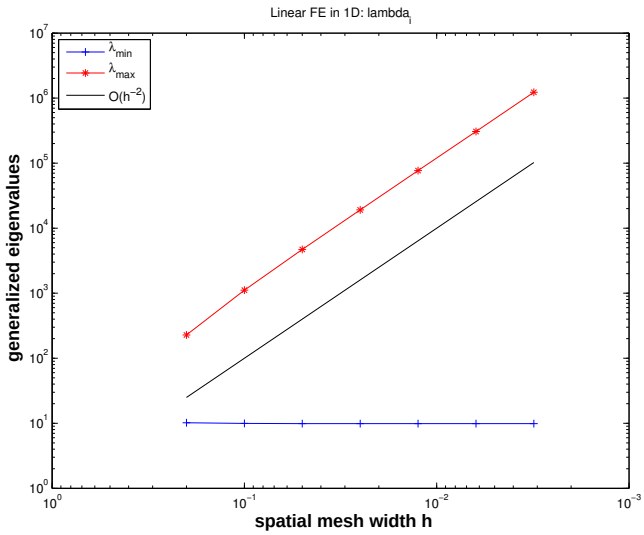


Fig. 272

$\Omega =]0,1[$

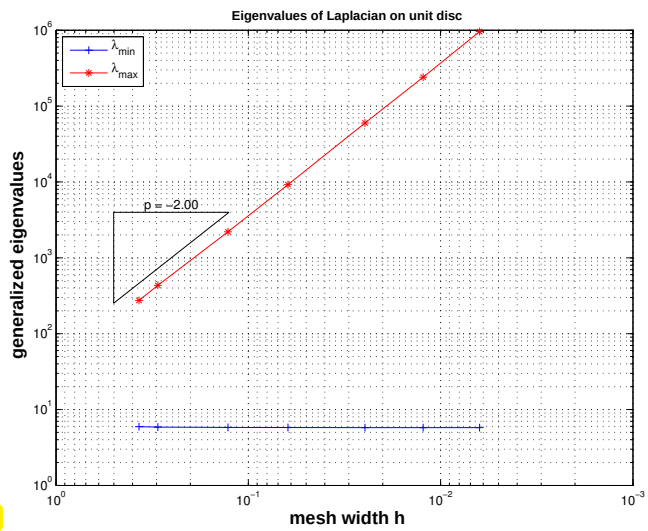


Fig. 273

$\Omega = \{\boldsymbol{x} \in \mathbb{R}^2 : \|\boldsymbol{x}\| < 1\}$

Observation:

- $\lambda_{\min} := \min_i \lambda_i$ does hardly depend on the mesh width.
- $\lambda_{\max} := \max_i \lambda_i$ displays a $O(h_{\mathcal{M}}^{-2})$ growth as $h_{\mathcal{M}} \to 0$

Remark 6.1.71 (Spectrum of elliptic operators)

The observation made in Exp. 6.1.69 is not surprising! Now we establish them as general property of finite element Galerkin matrices for second-order linear scalar variational problems.

To do so, let us translate the generalized eigenproblem "back to the ODE/PDE level":

$$\mathbf{A}\vec{\mu} = \lambda \mathbf{M}\vec{\mu} \tag{6.1.72}$$

$$\Updownarrow$$

$$u_N \in V_{0,N}: \quad \mathsf{a}(u_N, v_N) = \lambda \mathsf{m}(u_N, v_N) \quad \forall v_N \in V_{0,N} \,.$$

← "undo Galerkin discretization"

$$u \in H_0^1(\Omega): \quad \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \lambda \int_\Omega u \cdot v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega) \,.$$

$$\Downarrow$$

$$-\Delta u = \lambda u \quad \text{in } \Omega \quad , \quad u = 0 \quad \text{on } \partial\Omega \,, \tag{6.1.73}$$

which is a so-called elliptic eigenvalue problem.

It is easily solved in 1D on $\Omega =]0,1[$:

$$(6.1.73) \mathrel{\hat{=}} \quad \frac{d^2u}{dx^2}(x) = \lambda u(x) \,, \quad 0 < x < 1 \,, \quad u(0) = u(1) = 0 \,.$$

$$\Rightarrow \quad u_k(x) = \sin(k\pi x) \quad \leftrightarrow \quad \lambda_k = (\pi k)^2 \,, \quad k \in \mathbb{N} \,.$$

Note that we find an infinite number of eigenfunctions and eigenvalues, parameterized by $k \in \mathbb{N}$. Assuming that the λ_k are sorted, the eigenvalues tend to ∞ for $k \to \infty$:

$$\lambda_k = O(k^2) \quad \text{for} \quad k \to \infty \,.$$

Of course, the matrix eigenvalue problem (6.1.72) can have a finite number of eigenvectors only. Crudely speaking, they correspond to those eigenfunctions $u_k(x) = \sin(k\pi x)$ that can be resolved by the mesh (if u_k "oscillates too much", then it cannot be represented on a grid). These are the first N so that we find in 1D for an equidistant mesh

$$\lambda_{\max} = O(N^2) = O(h_{\mathcal{M}}^{-2}) \,.$$

This is heuristics, but the following Lemma will a precise statement.

Lemma 6.1.74. Behavior of of generalized eigenvalues

Let $\mathcal{M}$ be a simplicial mesh and $\mathbf{A}$, $\mathbf{M}$ denote the Galerkin matrices for the bilinear forms $\mathsf{a}(u,v) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}$ and $\mathsf{m}(u,v) = \int_\Omega u(x)v(x)\,\mathrm{d}\boldsymbol{x}$, respectively, and $V_{0,N} := \mathcal{S}_{p,0}^0(\mathcal{M})$. Then the smallest and largest generalized eigenvalues of $\mathbf{A}\vec{\mu} = \lambda\mathbf{M}\vec{\mu}$, denoted by $\lambda_{\min}$ and $\lambda_{\max}$, satisfy

$$\frac{1}{\operatorname{diam}(\Omega)^2} \leq \lambda_{\min} \leq C \quad , \quad \lambda_{\max} \geq C h_{\mathcal{M}}^{-2} \,,$$

where the "generic constants" ($\rightarrow$ Rem. 5.3.58) depend only on the polybomial degree p, the domain Ω, and the shape regularity measure $\rho_{\mathcal{M}}$.

Proof. (partial) We rely on the Courant-Fischer min-max theorem [4, Thm. 7.3.41] that, among other consequencees, expresses the boundaries of the spectrum of a symmetric matrix through the extrema of its Rayleigh quotient

$$\mathbf{T} = \mathbf{T}^T \in \mathbb{R}^{N,N} \quad \Rightarrow \quad \lambda_{\min}(\mathbf{T}) = \min_{\vec{\xi}\in\mathbb{R}^N\setminus\{0\}} \frac{\vec{\xi}^T\mathbf{T}\vec{\xi}}{\vec{\xi}^T\vec{\xi}} \,, \quad \lambda_{\max}(\mathbf{T}) = \max_{\vec{\xi}\in\mathbb{R}^N\setminus\{0\}} \frac{\vec{\xi}^T\mathbf{T}\vec{\xi}}{\vec{\xi}^T\vec{\xi}} \,.$$

Apply this to the generalized eigenvalue problem (Recall the concept of a "square root" $\mathbf{M}^{1/2}$ of an s.p.d. matrix $\mathbf{M}$, see [4, Section 8.3])

$$\mathbf{A}\vec{\mu} = \lambda\mathbf{M}\vec{\mu} \quad \overset{\vec{\zeta}:=\mathbf{M}^{1/2}\vec{\mu}}{\Leftrightarrow} \quad \underbrace{\mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2}}_{=:\mathbf{T}}\vec{\zeta} = \lambda\vec{\zeta} \,.$$

$$\blacktriangleright \quad \boxed{\lambda_{\min} = \min_{\vec{\mu}\neq 0} \frac{\vec{\mu}^T\mathbf{A}\vec{\mu}}{\vec{\mu}^T\mathbf{M}\vec{\mu}} \quad , \quad \lambda_{\max} = \max_{\vec{\mu}\neq 0} \frac{\vec{\mu}^T\mathbf{A}\vec{\mu}}{\vec{\mu}^T\mathbf{M}\vec{\mu}}} \,. \tag{6.1.75}$$

As a consequence we only have to find bounds for the extrema of a generalized Rayleigh quotient, *cf.* [4, Eq. (7.3.37)]. This generalized Rayleigh quotient can be expressed as

$$\frac{\vec{\mu}^T\mathbf{A}\vec{\mu}}{\vec{\mu}^T\mathbf{M}\vec{\mu}} = \frac{\mathsf{a}(u_N,u_N)}{\mathsf{m}(u_N,u_N)} \,, \quad \vec{\mu} \hat{=} \text{ coefficient vector for } u_N \,. \tag{6.1.76}$$

Now we discuss a lower bound for $\lambda_{\max}$, which can be obtained by inserting a suitable *candidate function* into (6.1.76).

Discussion for special setting: $V_{0,N} = \mathcal{S}_1^0(\mathcal{M})$ on triangular mesh $\mathcal{M}$

Candidate function: "tent function" $u_N = b_N^i$ ($\rightarrow$ Section 3.3.3) for some node $\boldsymbol{x}^i \in \mathcal{V}(\mathcal{M})$ of the mesh!

By elementary computations as in Section 3.3.5 we find

$$\mathsf{a}(b_N^i, b_N^i) \approx C \quad , \quad \mathsf{m}(b_N^i, b_N^i) \leq C \max_{K \in \mathcal{U}(\boldsymbol{x}^i)} h_K^2 \,, \tag{6.1.77}$$

where the generic constants $C > 0$ depend on the shape regularity measure $\rho_{\mathcal{M}}$ only.

$$\text{(6.1.75) \& (6.1.77)} \quad \Rightarrow \quad \lambda_{\max} \geq C h_{\mathcal{M}}^{-2} \,.$$

This provides the estimate (from below) for the largest eigenvalue. □

Lemma 6.1.74 & (6.1.66) imply concrete timestep constraint for explicit Euler method in the case of spatial Galerkin discretization by means of Lagrangian finite elements

$$\tau < C h_{\mathcal{M}}^2 \,, \tag{6.1.78}$$

with $C > 0$ depending only on polynomial degree and the shape regularity measure $\rho_{\mathcal{M}}$.

From [4, Section 12.3] we already know that some *implicit* single step methods are not affected by stability induced timestep constraints. This can be confirmed by rigorous analysis.

(6.1.79) Diagonlization applied to implicit Euler timestepping

Recall [4, § 12.3.2]: apply diagonalization technique, see (6.1.65), to implicit Euler timestepping with uniform timestep $\tau > 0$

$$\vec{\boldsymbol{\mu}}^{(j)} = \vec{\boldsymbol{\mu}}^{(j-1)} - \tau \mathbf{M}^{-1} \mathbf{A} \vec{\boldsymbol{\mu}}^{(j)} \quad \xrightarrow{\vec{\boldsymbol{\eta}} := \mathbf{T}^\top \mathbf{M} \vec{\boldsymbol{\mu}}} \quad \vec{\boldsymbol{\eta}}^{(j)} = \vec{\boldsymbol{\eta}}^{(j-1)} - \tau \mathbf{D} \vec{\boldsymbol{\eta}}^{(j)} \,,$$

that is, the decoupling of eigencomponents carries over to the implicit Euler method: for $i = 1, \dots, N$

$$\eta_i^{(j)} = \eta_i^{(j-1)} - \tau \lambda_i \eta^{(j)} \quad \Rightarrow \quad \boxed{\eta_i^{(j)} = \left(\tfrac{1}{1+\tau\lambda_i} \right)^j \eta_i^{(0)}} \,. \tag{6.1.80}$$

$$\left[\left| \frac{1}{1+\tau\lambda_i} \right| < 1 \quad \text{and} \quad \lambda_i > 0 \quad \Rightarrow \quad \right] \quad \lim_{j \to \infty} \eta_i^{(j)} = 0 \quad \forall \tau > 0 \,. \tag{6.1.81}$$

☞ The implicit Euler method for (6.1.30) will never suffer blow-up regardless of timestep size; it is unconditionally stable.

(6.1.82) Diagonalization applied to general Runge-Kutta timestepping

The diagonalization trick from § 6.1.58 can be applied to general Runge-Kutta single step methods (RKSSM, → Def. 6.1.40). We can start from the increment equations

$$\vec{\boldsymbol{\kappa}}_i \in \mathbb{R}^N : \quad \mathbf{M} \vec{\boldsymbol{\kappa}}_i + \sum_{m=1}^{s} \tau a_{im} \mathbf{A} \vec{\boldsymbol{\kappa}}_m = \vec{\boldsymbol{\varphi}}(t_j + c_i \tau) - \mathbf{A} \vec{\boldsymbol{\mu}}^{(j)} \,, \quad i = 1, \dots, s \,, \tag{6.1.44}$$

$$\vec{\boldsymbol{\mu}}^{(j+1)} = \vec{\boldsymbol{\mu}}^{(j)} + \tau \sum_{m=1}^{s} \vec{\boldsymbol{\kappa}}_m b_m \,. \tag{6.1.45}$$

and apply diagonalization using

$$\mathbf{AT} = \mathbf{MTD}\,, \quad \mathbf{D} = \begin{bmatrix} \lambda_1 & & & \\ & \ddots & & \\ & & \ddots & \\ & & & \lambda_N \end{bmatrix}\,, \quad \mathbf{T}^\top\mathbf{MT} = \mathbf{I}\,, \tag{6.1.62}$$

and the transformed coefficient and increment vectors:

$$\begin{aligned} \vec{\mu}^{(j)} &= \mathbf{T}\vec{\eta}^{(j)}\,, \\ \vec{\kappa}_i &= \mathbf{T}\vec{\zeta}_i \end{aligned} \quad \Leftrightarrow \quad \begin{aligned} \mathbf{T}^\top\mathbf{M}\vec{\mu}^{(j)} &= \vec{\eta}^{(j)}\,, \\ \mathbf{T}^\top\mathbf{M}\vec{\kappa} &= \vec{\zeta}\,. \end{aligned}$$

We multiply the increment equations (6.1.44) with $\mathbf{T}^\top$ from left and rewrite them in terms of $\vec{\zeta}_i$ and $\vec{\eta}^{(j)}$:

$$\underbrace{\mathbf{T}^\top\mathbf{MT}}_{=\mathbf{I}}\vec{\zeta}_i + \sum_{m=1}^{s} \tau a_{im} \underbrace{\mathbf{T}^\top\mathbf{AT}}_{=\mathbf{D}}\vec{\zeta}_m = \mathbf{T}^\top\vec{\varphi}(t_j + c_i\tau) - \underbrace{\mathbf{T}^\top\mathbf{AT}}_{=\mathbf{D}}\vec{\eta}^{(j)}\,, \quad i = 1,\dots,s\,,$$

$$\vec{\eta}^{(j+1)} = \vec{\eta}^{(j)} + \tau\sum_{m=1}^{s}\vec{\zeta}_m b_m\,.$$

We can write these equations in components taking into account that $\mathbf{D}$ is diagonal with diagonal entries λ_j, $j = 1,\dots,N$.

$$\left(\vec{\zeta}_i\right)_k + \sum_{m=1}^{s} \tau a_{im}\lambda_k\left(\vec{\zeta}_m\right)_k = \left(\mathbf{T}^\top\vec{\varphi}(t_j + c_i\tau)\right)_k - \lambda_k\left(\vec{\eta}^{(j)}\right)_k\,, \quad i = 1,\dots,s\,, k = 1,\dots,N\,, \tag{6.1.83}$$

$$\left(\vec{\eta}^{(j+1)}\right)_k = \left(\vec{\eta}^{(j)}\right)_k + \tau\sum_{m=1}^{s}\left(\vec{\zeta}_m\right)_k b_m\,. \tag{6.1.84}$$

Compare this with the formulas arising when applying the same Runge-Kutta single step method to the scalar ODE $\dot{\eta} = -\lambda\eta + \psi(t)$:

$$\kappa_i = -\lambda(\eta^{(j)} + \tau\sum_{m=1}^{s} a_{im}\kappa_m) + \psi(t_j + c_i\tau) \quad i = 1,\dots,s\,,$$

$$\eta^{(j+1)} = \eta^{(j)} + \tau\sum_{m=1}^{s} b_m\kappa_m\,.$$

Obviously, (6.1.83) for fixed k and $\lambda_k = \lambda$ and (6.1.82) describe the same recursion. Summing up, we have found that the following diagram commutes

$$\begin{array}{ccc} \mathbf{M}\frac{d}{dt}\vec{\mu} + \mathbf{A}\mu = 0 & \xrightarrow{\text{transformation } \vec{\eta} = \mathbf{T}^T\mathbf{M}\vec{\mu}} & \frac{d}{dt}\eta_i = -\lambda_i\eta_i\,, i = 1,\dots,N \\ \text{RK-SSM}\Big\downarrow & & \Big\downarrow\text{RK-SSM} \\ \vec{\mu}^{(j)} = \mathbf{\Psi}^\tau\vec{\mu}^{(j-1)} & \xrightarrow{\text{transformation } \vec{\eta} = \mathbf{T}^T\mathbf{M}\vec{\mu}} & \vec{\eta}_i^{(j)} = \widetilde{\Psi}_i^\tau\vec{\eta}_i^{(j-1)}\,, i = 1,\dots,N\,. \end{array} \tag{6.1.85}$$

The bottom line is

> that we have to study the behavior of the RK-SSM *only* for linear scalar ODEs $\dot{y} = -\lambda y$, $\lambda > 0$.

This is the gist of the model problem analysis discussed in [4, Section 12.3].

There we saw that everything boils down to inspecting the modulus of a rational stability function on $\mathbb{C}$, see [4, Thm. 12.3.27]. This gave rise to the concept of L-stability, see [4, Def. 12.3.38]. Here, we will not delve into a study of stability functions.

Unconditional stability of single step methods

Necessary condition for *unconditional stability* of a single step method for semi-discrete parabolic evolution problem (6.1.30)("method of lines"):

The discrete evolution $\Psi_\lambda^\tau : \mathbb{R} \mapsto \mathbb{R}$ of the single step method applied to the scalar ODE $\dot{y} = -\lambda y$ satisfies

$$\lambda > 0 \quad \Rightarrow \quad \lim_{j\to\infty} (\Psi_\lambda^\tau)^j y_0 = 0 \quad \forall y_0 \in \mathbb{R}, \quad \forall \tau > 0 \,. \tag{6.1.87}$$

Definition 6.1.88. L(π)-stability

A single step method satisfying (6.1.87) is called L(π)-stable.

Example 6.1.89 (L(π)-stable Runge-Kutta single step methods)

Simplest L(π)-stable Runge-Kutta single step method = implicit Euler timestepping (6.1.37).

Next we list two commonly used higher order L(π)-stable Runge-Kutta methods, specified through their Butcher schemes, see (6.1.42):

$$\begin{array}{c|cc} \frac{1}{3} & \frac{5}{12} & -\frac{1}{12} \\ 1 & \frac{3}{4} & \frac{1}{4} \\ \hline & \frac{3}{4} & \frac{1}{4} \end{array} \tag{6.1.90}$$

RADAU-3 scheme (order 3)

$$\begin{array}{c|cc} \lambda & \lambda & 0 \\ 1 & 1-\lambda & \lambda \\ \hline & 1-\lambda & \lambda \end{array}, \quad \lambda := 1 - \tfrac{1}{2}\sqrt{2}\,, \tag{6.1.91}$$

SDIRK-2 scheme (order 2)

More examples $\rightarrow$ [4, Ex. 12.3.44]. The class of RADAU methods provides L(π)-stable Runge-Kutta methods up to arbitrary order.

6.1.6 Convergence

Now we investigate the asymptotic algebraic convergence for fully discretized second-order linear parabolic evolution problem, when Lagrangian finite elements in space are used together with some Runge-Kutta single step method. Here we have two natural discretization parameters, namely the mesh width ($\rightarrow$ Def. 5.2.3) of the finite element mesh, and the size τ of the (uniform) timestep.

For general considerations about asymptotic convergence and its meaning refer to § 5.3.59 and § 5.3.62.

We start with a question: Why should one prefer complicated implicit L(π)-stable Runge-Kutta single step methods ($\rightarrow$ Ex. 6.1.89) to the simple implicit Euler method?

Silly question! Because these methods deliver "better accuracy"!

However, we need some clearer idea of what is meant by this. To this end, we now study the dependence of (a norm of) the discretization error for a parabolic IBVP on the parameters of the spatial and temporal discretization.

Experiment 6.1.92 (Convergence of fully discrete timestepping in one spatial dimension)

- ✦ 1D parabolic evolution problem: $\frac{d}{dt}u - u'' = f(t,x)$ on $]0,1[\times]0,1[$
- ✦ exact solution $u(x,t) = (1+t^2)e^{-\pi^2 t}\sin(\pi x)$, source term accordingly
- ✦ Linear finite element Galerkin discretization equidistant mesh, see Section 1.5.2.2, $V_{0,N} = \mathcal{S}^0_{1,0}(\mathcal{M})$,
- ✦ piecewise linear spatial approximation of source term $f(x,t)$
- ✦ implicit Euler timestepping ($\rightarrow$ Ex. 6.1.35) with uniform timestep $\tau > 0$

Monitored: error norm $\left(\tau \sum_{j=1}^{M} |u - u_N(\tau j)|^2_{H^1(\Omega)}\right)^{\frac{1}{2}}$.

The norms $|u - u_N(\tau j)|_{H^1(\Omega)}$ were approximated by high order local quadrature rules, whose impact can be neglected.

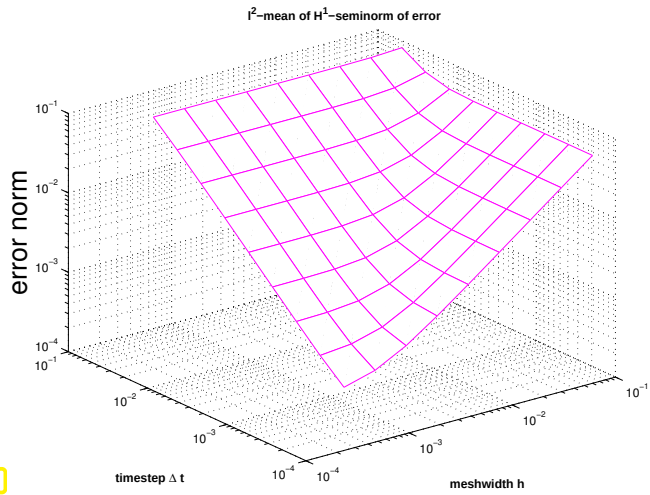


Fig. 274

◁ $h_{\mathcal{M}}$- and τ-dependence of error norm

Obervation:

τ small: error norm $\approx h_{\mathcal{M}}$
$h_{\mathcal{M}}$ small: error norm $\approx \tau$

The error seems to behave like

$$\text{error norm} \approx C_1 h_{\mathcal{M}} + C_2 \tau \,. \tag{6.1.93}$$

Recall from Section 5.3.5, Thm. 5.1.15, Thm. 5.3.56:

energy norm of spatial finite element discretization error $O(h_{\mathcal{M}})$ for $h_{\mathcal{M}} \rightarrow 0$

Since the implicit Euler method is *first order consistent* we expect

temporal timestepping error $O(\tau)$

(6.1.93) ➢ conjecture: total error is sum of spatial and temporal discretization error.

From Fig. 274 we draw the compelling conclusion:

- for big mesh width $h_{\mathcal{M}}$ (spatial error dominates) further reduction of timestep size τ is useless,
- if timestep τ is large (temporal error dominates), refinement of the finite element space does not yield a reduction of the total error.

Experiment 6.1.94 (Higher order timestepping for 1D heat equation)

- ✦ same IBVP as in Exp. 6.1.92,
- ✦ spatial discretization on equidistant grid, *very small meshwidth* $h = 0.5 \cdot 10^{-4}$, $V_N = \mathcal{S}^0_{1,0}(\mathcal{M})$.

Various timestepping methods
(➢ different orders of consistency)

- implicit Euler timestepping (6.1.37), first order
- Crank-Nicolson-method (6.1.39), order 2
- SDIRK-2 timestepping (→ Ex. 6.1.89), order 2
- Gauss-Radau-Runge-Kutta collocation methods with s stages, order $2s-1$

Note: all methods $L(\pi)$-stable (→ Def. 6.1.88), except for Crank-Nicolson-method.

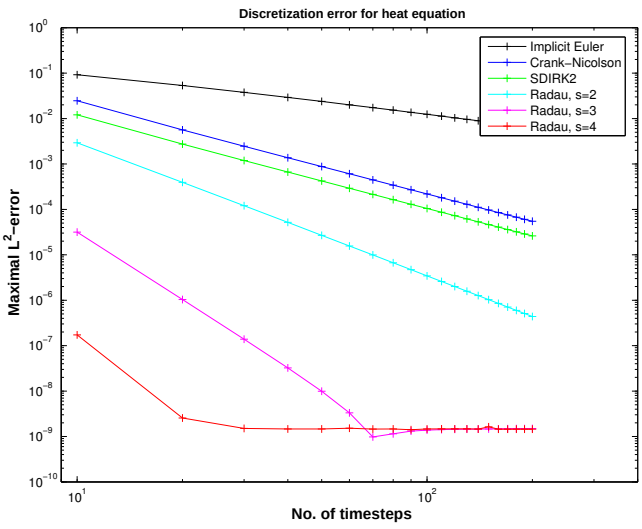


Fig. 275

Monitored: $\max_j \left\| u(t_j) - u_N^{(j)} \right\|_{L^2(]0,1[)}$ (evaluated by high order quadrature)

We observe that higher-order L(π)-stable Runge-Kutta timestepping leads to a faster algebraic decay of the temporal discretization error, the rate matching the theoretical order of the methods. This can be observed until we reach the spatial discretization error which is $\approx 10^{-9}$ in Fig. 275.

(6.1.95) Spatial and temporal error contributions

Theoretical results confirm the conjecture suggested from observation (6.1.93) in Exp. 6.1.92:

"Meta-theorem" 6.1.96. Convergence of solutions of fully discrete parabolic evolution problems

Assume that

- *the solution of the parabolic IBVP* (6.1.5)–(6.1.8) *is "sufficiently smooth" (both in space and time),*
- *its spatial Galerkin finite element discretization relies on degree p Lagrangian finite elements (→ Section 3.5) on uniformly shape-regular families of meshes,*
- *timestepping is based on an $L(\pi)$-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

$$\left(\tau \sum_{j=1}^{M} |u - u_N(\tau j)|^2_{H^1(\Omega)}\right)^{\frac{1}{2}} \leq C\left(h_{\mathcal{M}}^p + \tau^q\right), \tag{6.1.97}$$

where $C > 0$ must not depend on $h_{\mathcal{M}}$, τ.

This has been dubbed a "meta-theorem", because quite a few technical assumptions on the exact solution and the methods have been omitted in its statement. Therefore it is not a mathematically rigorous statement of facts. More details in [5].

A message contained in (6.1.97):

total discretization error = spatial error + temporal error

§ 5.3.59 still applies: (6.1.97) does not give information about actual error, but only about the trend of the error, when discretization parameters $h_{\mathcal{M}}$ and τ are varied.

► Nevertheless, as in the case of the a priori error estimates of Section 5.3.5, we can draw conclusions about optimal refinement strategies in order to achieve prescribed *error reduction*.

As in Section 5.3.5 we make the **assumption** that the estimates (6.1.97) are sharp for all contributions to the total error and that the constants are the same (!)

$$\begin{aligned} \text{contribution of spatial error} &\approx Ch_{\mathcal{M}}^{p} \quad , h_{\mathcal{M}} \hat{=} \text{ mesh width } (\rightarrow \text{Def. 5.2.3}) \ , \\ \text{contribution of temporal error} &\approx C\tau^{q} \quad , \tau \hat{=} \text{ timestep size} \ . \end{aligned} \tag{6.1.98}$$

This suggests the following change of $h_{\mathcal{M}}$, τ in order to achieve *error reduction* by a factor of $\rho > 1$:

$$\begin{aligned} \text{reduce mesh width by factor} \quad & \rho^{1/p} \\ \text{reduce timestep by factor} \quad & \rho^{1/q} \end{aligned} \overset{(6.1.98)}{\Longrightarrow} \text{error reduction by } \rho > 1 \ . \tag{6.1.99}$$

Refinement for fully discrete parabolic evolution problems	
Guideline:	spatial and temporal resolution have to be adjusted in tandem

---

Remark 6.1.101 (Potential inefficiency of conditionally stable single step methods)

Terminology: A timestepping scheme is labelled conditionally stable, if blow-up can be avoided by using sufficient small timesteps (timestep constraint). Examples: all explicit Runge-Kutta methods

Now we can answer the question, why a stability induced timestep constraint like (6.1.78), that is,

$$\tau \leq O(h_{\mathcal{M}}^{-2}) \tag{6.1.102}$$

can render a single step method grossly inefficient for integrating semi-discrete parabolic IBVPs.

(6.1.99) ➢ in order to reduce the error by a fixed factor ρ one has to reduce both timestep and mesh-width by some other fixed factors (asymptotically). More concretely, for the timestep τ:

(6.1.99) ➢ *accuracy* requires reduction of τ by a factor $\rho^{1/q}$

(6.1.102) ➢ *stability* entails reduction of τ by a factor $(\rho^{1/p})^2 = \rho^{2/p}$.

$$\frac{1}{q} < \frac{2}{p} \quad \Rightarrow \quad \text{stability enforces smaller timestep than required by accuracy}$$

$$\Rightarrow \quad \text{timestepping is } \textit{inefficient!}$$

► When faced with conditional stability (6.1.102), for the sake of efficiency

use *high-order spatial discretization* combined with *low order timestepping*.

However, this may not be easy to achieve

- because high-order timestepping is much simpler than high-order spatial discretization,
- because limited spatial smoothness of exact solution ($\rightarrow$ results of Section 5.4 apply!) may impose a limit on q in (6.1.97).

Concretely: 5th-order `ode45` timestepping ($q = 5$) $\overset{\frac{1}{q}=\frac{2}{p}}{\rightsquigarrow}$ use degree-10 Lagrangian FEM!

Moreover, high-order convergence of spatial discretization error is conditional on sufficient smoothness of the solution $u(t)$ for all times, remember (5.3.69).

Remark 6.1.103 (Guessing timestep constraint)

Even if the timestep constraint $\tau < O(h_{\mathcal{M}}^{-1})$ does not thwart the efficiency of the full discretization (finite elements in space & Runge-Kutta timestepping), the actual stability threshold for τ may not be easy to guess, because the estimates for the spectrum of the generalized eigen

---

Experiment 6.1.104 (Convergence for conditionally stable Runge-Kutta timestepping)

Parabolic IBVP of Exp. 6.1.92:

- $\frac{d}{dt}u - u'' = f(t,x)$ on $]0,1[\times]0,1[$
- exact solution $u(x,t) = (1+t^2)e^{-\pi^2 t}\sin(\pi x)$, source term accordingly
- Linear finite element Galerkin discretization equidistant mesh, see Section 1.5.2.2, $V_{0,N} = \mathcal{S}_{1,0}^0(\mathcal{M})$,
- piecewise linear spatial approximation of source term $f(x,t)$
- *explicit* Euler timestepping (6.1.36) with uniform timestep $\tau \sim h^2$ close to the stability limit.

Monitored: error norms $\left(\tau\sum_{j=1}^{M}|u-u_N(\tau j)|^2_{H^1(]0,1[)}\right)^{\frac{1}{2}}$, $\left(\tau\sum_{j=1}^{M}\|u-u_N(\tau j)\|^2_{L^2(]0,1[)}\right)^{\frac{1}{2}}$.

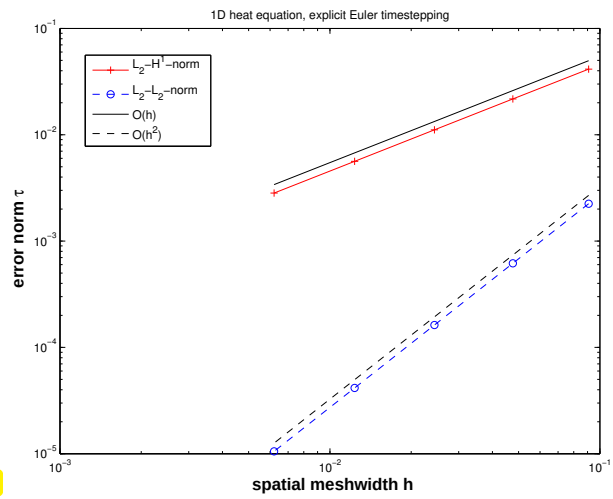


Fig. 276

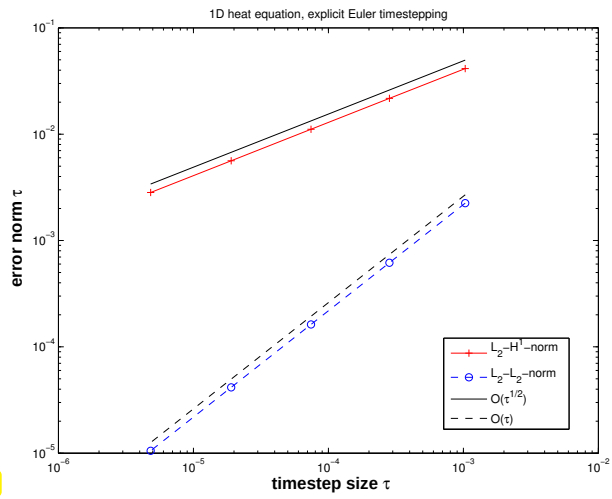


Fig. 277

In comparison with Exp. 6.1.92: degraded rate of convergence $O(\sqrt{\tau})$ for L^2-H^1 space-time norm, because conditional stability prevents us from employing sufficient refinement in space.

?! *Review question(s) 6.1.105.* (Parabolic evolution problems)

1. How will the assertion of Thm. 6.1.96 will probably have to be altered in case we face $u(t) \in H^m(\Omega)$, $m \geq 2$, but $u(t) \notin H^{m+1}(\Omega)$ for all times t.

2. The spatial Galerkin semi-discretization of the evolution problem

$$t \in]0,T[\mapsto u(t) \in V_0 \quad : \quad \begin{cases} \frac{d}{dt}\mathsf{m}(u(t),v) + \mathsf{a}(u(t),v)\ell(t)(v) & \forall v \in V_0 \,, \\ u(0) = u_0 \in V_0 \,. \end{cases}$$

   leads to an ordinary differential equation, which can be written in the form $\frac{d}{dt}\vec{\boldsymbol{\mu}} = F(\vec{\boldsymbol{\mu}})$. Give an expression for F with detailed formulas for all components.

3. Consider the evolution problem

$$t \in]0,T[\mapsto u(t) \in H^1(\Omega) : \quad \frac{d}{dt}\int_\Omega u(t)v \,\mathrm{d}S + \int_\Omega \mathbf{grad}\, u(t) \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H^1(\Omega) \,.$$

   We perform spatial finite element Galerkin semi-discretization based on $\mathcal{S}_1^0(\mathcal{M})$ in the spirit of the method of lines.

   (a) Which problem does the application of explicit Runge-Kutta timestepping face?

   (b) Show that implicit Euler timestepping is feasible.

4. Show that Crank-Nicolson timestepping (6.1.39) for a standard parabolic evolution problem with s.p.d. bilinear forms $\mathsf{m}(\cdot,\cdot)$ and $\mathsf{a}(\cdot,\cdot)$ is *unconditionally stable*.

6.2 Wave equations

This section is dedicated to a class of initial-boundary value problems (IBVP) that have the same structure as (abstract) parabolic IBVP (→ § 6.1.17) except for the occurrence of *second derivatives in time*. This will have profound consequences as regards properties of solutions and choice of timestepping schemes.

(6.2.1) A conservative evolution

Lemma 6.1.22 teaches that in the absence of time-dependent sources the rate of change of temperature will decay exponentially in the case of heat conduction.

Now we will encounter a class of evolution problems where temporal and spatial fluctuations will not be damped and will persist for good:

This will be the class of linear conservative wave propagation problems

As before these initial-boundary value problems (IBVP) will be posed on a space time cylinder $\widetilde{\Omega} := \Omega \times]0,T[\subset \mathbb{R}^{d+1}$ (→ Fig. 269), where $\Omega \subset \mathbb{R}^d$, $d = 2,3$, is a bounded spatial domain as introduced in the context of elliptic boundary value problems, see Section 2.2.1.

The unknown will be a function $u = (x,t) : \widetilde{\Omega} \mapsto \mathbb{R}$.

6.2.1 Vibrating membrane

(6.2.2) Repetition: linear elastic string and membrane models

Recall the stationary simplfied (linearized) models for taut string (1D) and membrane (2D):

- Tense string model (→ Section 1.4), shape of string described by continuous displacement function $u : \Omega := [a,b] \mapsto \mathbb{R}$, $u \in H^1([a,b])$.
- Taut membrane model (→ Section 2.2.1), shape of membrane given by displacement function $u : \Omega \mapsto \mathbb{R}$, $u \in H^1(\Omega)$, over base domain $\Omega \subset \mathbb{R}^2$.

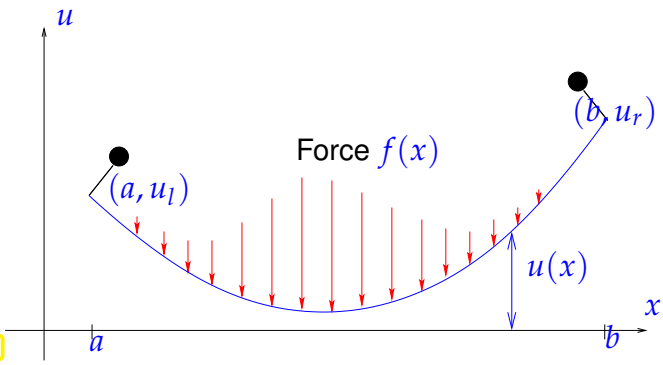


Fig. 278

Tense string ↔ $u : [a,b] \mapsto \mathbb{R}$

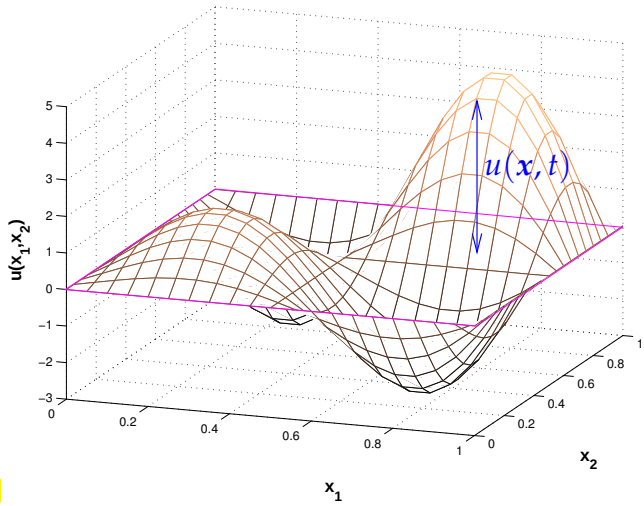


Fig. 279

Taut membrane ↔ $u : \Omega \mapsto \mathbb{R}$

In Section 2.2.3 we introduced the general variational formulation: with Dirichlet data (elevation of frame/pinning conditions) given by $g \in C^0(\partial\Omega)$,

$$V := \{v \in H^1(\Omega) : v_{|\partial\Omega} = g\}$$

we seek

$$u \in V : \quad \int_\Omega \sigma(x)\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}x = \int_\Omega f(x) v(x) \,\mathrm{d}x \,, \quad \forall v \in H_0^1(\Omega) \,, \tag{6.2.3}$$

where $f : \Omega \mapsto \mathbb{R}$ $\hat{=}$ density of vertical force,

$\sigma : \Omega \mapsto \mathbb{R}$ $\hat{=}$ uniformly positive stiffness coefficient (characteristic of material of the membrane).

(6.2.4) Transient membrane model with inertial forces

Now we switch to a *dynamic setting*: we allow variation of displacement with time, $u = u(\boldsymbol{x},t)$, the membrane is allowed to vibrate.

Recall (secondary school): Newton's second law of motion (law of inertia)

$$F = m\,a \tag{6.2.5}$$

$$\text{force} \quad = \quad \text{mass} \cdot \text{acceleration} \tag{6.2.6}$$

Apply this in a local version (stated for densities) to membrane

$$\text{force density} \quad f(\boldsymbol{x},t) = \rho(\boldsymbol{x}) \cdot \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\,, \tag{6.2.7}$$

where

- ✦ $\rho : \Omega \mapsto \mathbb{R}^+ \mathrel{\hat{=}}$ uniformly positive mass density of membrane, $[\rho] = \mathrm{kg\,m^{-2}}$,
- ✦ $\ddot{u} := \frac{\partial^2 u}{\partial t^2} \mathrel{\hat{=}}$ vertical aceleration (second temporal derivative of position).

Now, we assume that the force f in (2.4.4) is due to inertia forces only and express these using (6.2.7):

$$(2.4.4) \overset{(6.2.7)}{\blacktriangleright} \quad \int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x},t) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = -\int_\Omega \rho(\boldsymbol{x}) \cdot \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} \quad \forall v \in H_0^1(\Omega)\,.$$

Why the "$-$"-sign? Because, here the inertia force enters as a *reaction* force.

▶ Homogeneous linear wave equation in variational form (Dirichlet boundary conditions):

$$u \in V(t): \quad \underbrace{\int_\Omega \rho(\boldsymbol{x}) \cdot \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t)\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{\text{mass density}} + \underbrace{\int_\Omega \sigma(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x},t) \cdot \mathbf{grad}\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{\text{stiffness}} = 0 \quad \forall v \in H_0^1(\Omega) \tag{6.2.8}$$

$$\updownarrow$$

$$\boxed{u \in V(t): \quad \mathsf{m}(\ddot{u},v) + \mathsf{a}(u,v) = 0 \quad \forall v \in V_0}\,. \tag{6.2.9}$$

where $V(t) := \{v :]0,T[\mapsto H^1(\Omega) : v(\boldsymbol{x},t) = g(\boldsymbol{x},t) \text{ for } \boldsymbol{x} \in \partial\Omega,\ 0 < t < T\}$
(with continuous time-dependent Dirichlet data $g : \partial\Omega \times]0,T[\mapsto \mathbb{R}$.)

The bilinear forms a and m ($\rightarrow$ Def. 2.2.40) in (6.2.9) are the same as those in (6.1.18), § 6.1.17 (except for the notation for the coefficient σ). In particular, both a and m are *symmetric and positive definite* ($\rightarrow$ Def. 2.2.40). Thus they induce energy norms $\|\cdot\|_a$ and $\|\cdot\|_m$ ($\rightarrow$ Def. 2.2.43).

(6.2.10) Wave equation

Undo integration by parts by reverse application of Green's first formula Thm. 2.5.9:

$$(6.2.8) \quad \Rightarrow \quad \int_\Omega \left\{ \rho(\boldsymbol{x}) \frac{\partial^2 u}{\partial t^2}(\boldsymbol{x},t) - \mathrm{div}_{\boldsymbol{x}}(\sigma(\boldsymbol{x})(\mathbf{grad}_{\boldsymbol{x}}\,u)(\boldsymbol{x},t)) \right\} v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H_0^1(\Omega)\,. \tag{6.2.11}$$

Here it is indicated that the differential operators $\mathbf{grad}$ and div act on the spatial independent variable $\boldsymbol{x}$ only. As in the case of the heat equation ($\rightarrow$ § 6.1.2) this will tacitly be assumed below.

Now appeal to the fundamental lemma of calculus of variations in higher dimensions Lemma 2.5.12. This gives a PDE on the space-time cylinder $\widetilde{\Omega}$, see § 6.0.3.

$$\text{(6.2.11)} \quad \boxed{\overset{\text{Lemma 2.5.12}}{\Longrightarrow} \quad \rho(\boldsymbol{x})\frac{\partial^2 u}{\partial t^2} - \operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) = 0 \quad \text{in} \quad \widetilde{\Omega}} \,. \tag{6.2.12}$$

(6.2.12) is called a (homogeneous) wave equation. A general wave equation is obtained, when an additional exciting vertical force density $f = f(\boldsymbol{x}, t)$ comes into play:

$$\rho(\boldsymbol{x})\frac{\partial^2 u}{\partial t^2} - \operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) = f(\boldsymbol{x}, t) \quad \text{in} \quad \widetilde{\Omega}\,. \tag{6.2.13}$$

(6.2.14) Initial and boundary conditions

The wave equations (6.2.12), (6.2.13) have to be supplemented by

- spatial Dirichlet boundary conditions: $v(\boldsymbol{x}, t) = g(\boldsymbol{x}, t)$ for $\boldsymbol{x} \in \partial\Omega$, $0 < t < T$,
- two initial conditions

$$u(\boldsymbol{x}, 0) = u_0(\boldsymbol{x}) \quad , \quad \frac{\partial u}{\partial t}(\boldsymbol{x}, 0) = v_0 \quad \text{for} \quad \boldsymbol{x} \in \Omega\,,$$

with initial data $u_0, v_0 \in H^1(\Omega)$, satisfying the compatibility conditions $u_0(\boldsymbol{x}) = g(\boldsymbol{x}, 0)$ for $\boldsymbol{x} \in \partial\Omega$.

(6.2.12) & boundary conditions & initial conditions = hyperbolic evolution problem

Excuse me, why do we need **two** initial conditions in contrast to the heat equation**?**

Remember that

- (6.2.12) is a *second-order equation* also in time (whereas the heat equation is merely first-order),
- for second order ODEs $\ddot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ we need two initial conditions

$$\mathbf{y}(0) = \mathbf{y}_0 \quad \text{and} \quad \dot{\mathbf{y}}(0) = \mathbf{v}_0\,, \tag{6.2.15}$$

in order to get a well-posed initial value problem, see [4, Rem. 11.1.23].

The physical meaning of the initial conditions (6.2.15) in the case of the membrane model is

- $u_0 \mathrel{\hat{=}}$ initial displacement of membrane, $u_0 \in H^1(\Omega)$ "continuous",
- $v_0 \mathrel{\hat{=}}$ initial vertical velocity of membrane.

Remark 6.2.16 (Boundary conditions for wave equation)

The message of § 6.1.9 also applies to the wave equation (6.2.12):

On $\partial\Omega\times]0,T[$ we can impose any of the boundary conditions discussed in Section 2.7:
- Dirichlet boundary conditions $u(\boldsymbol{x},t) = g(\boldsymbol{x},t)$ (membrane attached to frame),
- Neumann boundary conditions $\mathbf{j}(\boldsymbol{x},t)\cdot\boldsymbol{n} = 0$ (free boundary, Ex. 2.5.18)
- radiation boundary conditions $\mathbf{j}(\boldsymbol{x},t)\cdot\boldsymbol{n} = \Psi(u(\boldsymbol{x},t))$,

and any combination of these as discussed in Ex. 2.7.8, yet, *only one* of them at any part of $\partial\Omega\times]0,T[$, see Rem. 2.7.7.

(6.2.17) Wave equation as first order system in time

Usual procedure [4, Rem. 11.1.23]: higher-order ODE can be converted into first-order ODEs by introducing derivatives as additional solution components. This approach also works for the second-order (in time) wave equation (6.2.12):

Additional unknown: velocity $v(\boldsymbol{x},t) = \dfrac{\partial u}{\partial t}(\boldsymbol{x},t)$

$$\rho(\boldsymbol{x})\frac{\partial^2 u}{\partial t^2} - \operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) = 0 \quad \blacktriangleright \quad \boxed{\begin{cases} \dot{u} = v\,, \\ \rho(\boldsymbol{x})\dot{v} = \operatorname{div}(\sigma(\boldsymbol{x})\,\mathbf{grad}\,u) \end{cases} \text{in } \widetilde{\Omega}} \tag{6.2.18}$$

with initial conditions

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad , \quad v(\boldsymbol{x},0) = v_0(\boldsymbol{x}) \quad \text{for} \quad \boldsymbol{x}\in\Omega\,. \tag{6.2.19}$$

6.2.2 Wave propagation

Now we study properties of solutions of solutions of IBVPs for the wave equations (6.2.8)/(6.2.12).

(6.2.20) Cauchy problem

Constant coefficient wave equation ($\rho \equiv 1$) for $d = 1$, $\Omega = \mathbb{R}$ (so-called Cauchy problem for the wave equation):

$$c > 0\colon \quad \frac{\partial^2 u}{\partial t^2} - c^2\frac{\partial^2 u}{\partial x^2} = 0 \quad , \quad u(x,0) = u_0(x)\,, \quad \frac{\partial u}{\partial t}(x,0) = v_0(x)\,, \quad x\in\mathbb{R}\,. \tag{6.2.21}$$

Change of variables: $\xi = x + ct$, $\tau = x - ct$: $\widetilde{u}(\xi,\tau) := u(\frac{\xi+\tau}{2}, \frac{\xi-\tau}{2c})$.

Applying the chain rule we immediately see

$$u \text{ satisfies (6.2.21)} \quad \blacktriangleright \quad \frac{\partial^2\widetilde{u}}{\partial\xi\partial\tau} = 0 \quad \Rightarrow \quad \widetilde{u}(\xi,\tau) = F(\xi) + G(\tau)\,,$$

for **any** $F, G \in C^2(\mathbb{R})$!

The initial conditions from (6.2.21) fix the functions F and G: for all $x \in \mathbb{R}$

$$u(x,0) = F(x) + G(x) = u_0(x) \,,$$
$$\frac{\partial u}{\partial t} = c\frac{\partial \widetilde{u}}{\partial \xi}(x,x) - c\frac{\partial \widetilde{u}}{\partial \tau}(x,x) = cF'(x) - cG'(x) = v_0(x) \,.$$

$$\boxed{u(x,t) = \tfrac{1}{2}(u_0(x+ct) + u_0(x-ct)) + \tfrac{1}{2c}\int_{x-ct}^{x+ct} v_0(s)\,\mathrm{d}s} \,. \qquad (6.2.22)$$

(6.2.22) = d'Alembert solution of Cauchy problem (6.2.21).

(6.2.23) Finite speed of propagation

A simple consequence of the solution formula (6.2.22) for the Cauchy problem for the wave equation with constant coefficients:

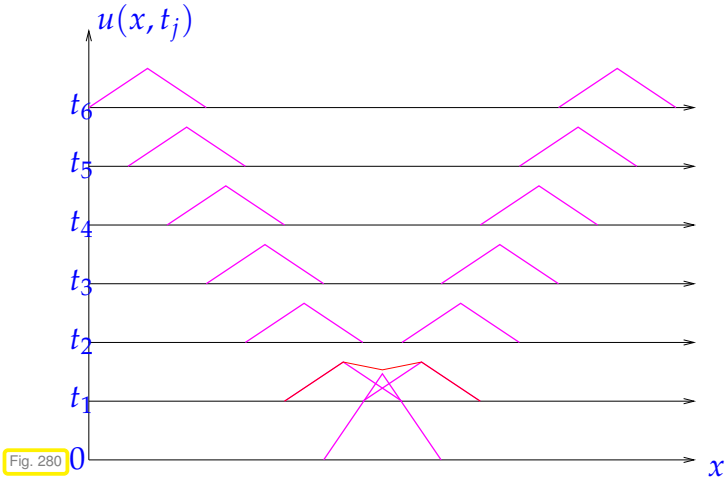


Fig. 280

$v_0 = 0$ ➤ initial data u_0 travel with speed c in opposite directions

finite speed of propagation is typical feature of solutions of wave equations

Note: (6.2.22) meaningful even for discontinuous u_0, v_0 !

➡ "generalized solutions" !

finite speed of propagation ➤ "point value" $u(\bar{x},\bar{t})$, $(\bar{x},\bar{t}) \in \widetilde{\Omega}$, may not depend on initial values outside proper subdomain of Ω !

Example 6.2.24 (Domain of dependence/influence for 1D wave equation, constant coefficient case)

Consider $d = 1$, initial-boundary value problem (6.2.21) for wave equation:

$$c > 0: \quad \frac{\partial^2 u}{\partial t^2} - c^2\frac{\partial^2 u}{\partial x^2} = 0 \quad , \quad u(x,0) = u_0(x) \,, \quad \frac{\partial u}{\partial t}(x,0) = v_0(x) \,, \quad x \in \mathbb{R} \,. \qquad (6.2.21)$$

Intuitive: from D'Alembert formula (6.2.22)

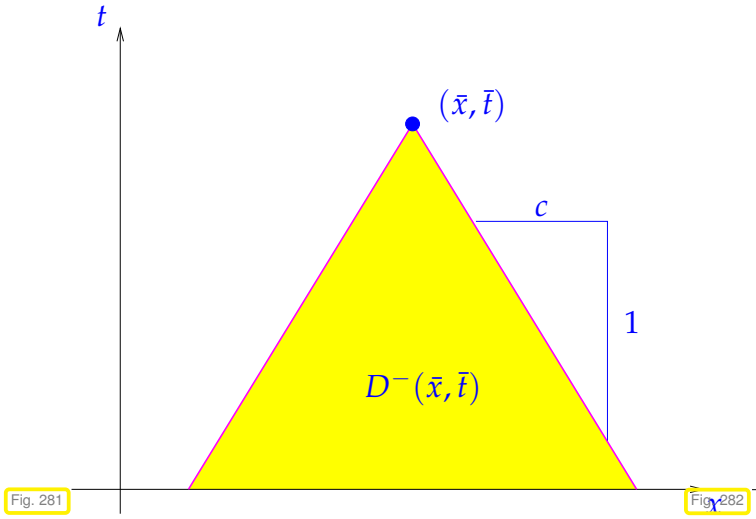


Fig. 281

domain of dependence of $(\bar{x},\bar{t}) \in \widetilde{\Omega}$

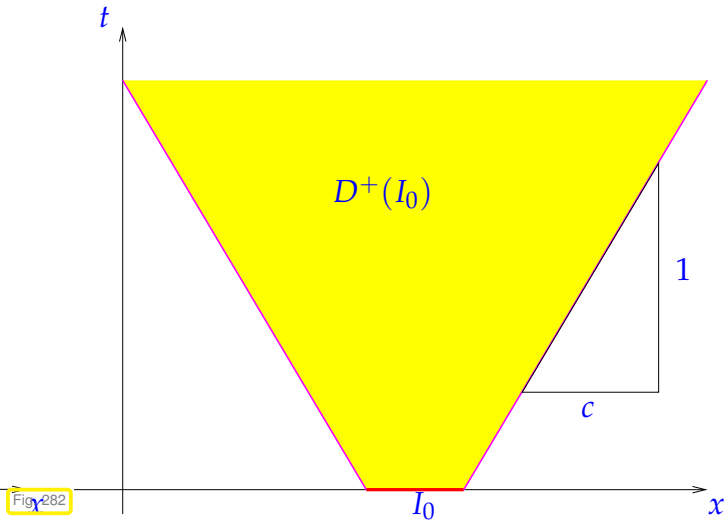


Fig. 282

domain of influence of $I_0 \subset \mathbb{R}$

Domain of dependence: the value of the solution in $(\bar{x},\bar{t})$ (•) will depend only on data in the yellow triangle in Fig. 281.

Domain of influence: intial data in I_0 will be relevant for the solution only in the yellow triangle in Fig. 282.

Theorem 6.2.25. Domain of dependence for isotropic wave equation →**[2, 2.5, Thm. 6]**

Let $u : \widetilde{\Omega} \mapsto \mathbb{R}$ be a (classical) solution of $\frac{\partial^2 u}{\partial t^2} - c\Delta u = 0$. Then

$$\left(|\boldsymbol{x} - \boldsymbol{x}_0| \geq R \quad \Rightarrow \quad \begin{array}{c} u(\boldsymbol{x},0) = 0\,, \\ \frac{\partial u}{\partial t}(\boldsymbol{x},0) = 0 \end{array} \right) \quad \Rightarrow \quad u(x,t) = 0 \quad \text{, if} \quad |\boldsymbol{x} - \boldsymbol{x}_0| \geq R + ct\,.$$

(6.2.26) Wave propagation: conservation of energy

The solution formula (6.2.22) clearly indicates that in 1D and in the absence of boundary conditions the solution of the wave equation will persist undamped for all times.

This absence of damping corresponds to a *conservation of total energy*, which is a distinguishing feature of conservative wave propagation phenomena.

Now, we examine this for the model problem

$$u \in H_0^1(\Omega): \quad \int_\Omega \rho(\boldsymbol{x}) \cdot \frac{\partial^2 u}{\partial t^2}\, v \,\mathrm{d}\boldsymbol{x} + \int_\Omega \sigma(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = 0 \quad \forall v \in H_0^1(\Omega) \tag{6.2.27}$$

$$\updownarrow$$

$$\boxed{u \in V_0: \quad \mathsf{m}(\ddot{u}, v) + \mathsf{a}(u,v) = 0 \quad \forall v \in V_0}\,. \tag{6.2.28}$$

Here we do not include the case of non-homogeneous spatial Dirichlet boundary conditions through an affine trial space. This can always be taken into account by offset functions, see the remark after (6.1.8).

Theorem 6.2.29. Energy conservation in wave propagation

If $u : \widetilde{\Omega} \mapsto \mathbb{R}$ *solves* (6.2.28), *then*

$$t \mapsto \quad \underbrace{\tfrac{1}{2}\mathsf{m}(\frac{\partial u}{\partial t}, \frac{\partial u}{\partial t})}_{\text{kinetic energy}} + \underbrace{\tfrac{1}{2}\mathsf{a}(u,u)}_{\text{elastic (potential) energy, see (2.2.7)}} \equiv \text{const} \; .$$

Proof. The "formal proof" boils down to a straightforward application of the product rule (→ Rem. 6.1.21) together with the symmetry of the bilinear forms m and a.

Introduce the total energy and apply the product rule from Rem. 6.1.21

$$E(t) := \tfrac{1}{2}\mathsf{m}(\frac{\partial u}{\partial t}, \frac{\partial u}{\partial t}) + \tfrac{1}{2}\mathsf{a}(u,u) \; .$$

$$\blacktriangleright \quad \frac{dE}{dt}(t) = \mathsf{m}(\ddot{u}, \dot{u}) + \mathsf{a}(\dot{u}, u) = 0 \quad \text{for solution } u \text{ of (6.2.28)} \; ,$$

because this is what we conclude from (6.2.28) for the special test function $v(\boldsymbol{x}) = \dot{u}(\boldsymbol{x}, t)$ for any $t \in]0, T[$.

□

6.2.3 Method of lines

(6.2.30) Spatial Galerkin semi-discretization

The method of lines approach to the wave equation (6.2.27), (6.2.28) is exactly the same as for the heat equation, see Section 6.1.4.

Idea: Apply Galerkin discretization (→ Section 3.2) in space to abstract linear hyperbolic variational problem (6.1.19).

$$t \in]0, T[\mapsto u(t) \in V_0 \quad : \quad \begin{cases} \mathsf{m}(\frac{d^2u}{dt^2}(t), v) + \mathsf{a}(u(t), v) = 0 \quad \forall v \in V_0 \; , \\ u(0) = u_0 \in V_0 \quad , \quad \frac{du}{dt}(0) = v_0 \in V_0 \; . \end{cases} \tag{6.2.31}$$

1st step: replace V_0 with a finite dimensional subspace $V_{0,N}$, $N := \dim V_{0,N} < \infty$

$\blacktriangleright$ Spatially discrete linear wave equation/hyperbolic evolution problem

$$t \in]0, T[\mapsto u(t) \in V_{0,N} \quad : \quad \begin{cases} \mathsf{m}(\frac{d^2u_N}{dt^2}(t), v_N) + \mathsf{a}(u_N(t), v_N) = 0 \quad \forall v_N \in V_{0,N} \; , \\ u_N(0) = \text{projection/interpolant of } u_0 \text{ in } V_{0,N} \; , \\ \frac{du_N}{dt}(0) = \text{projection/interpolant of } v_0 \text{ in } V_{0,N} \; . \end{cases} \tag{6.2.32}$$

2nd step: introduce (ordered) basis $\mathfrak{B}_N := \{b_N^1, \dots, b_N^N\}$ of trial/test space $V_{0,N}$

$$(6.2.32) \quad \Rightarrow \quad \begin{cases} \mathbf{M}\left\{\frac{d^2}{dt^2}\vec{\boldsymbol{\mu}}(t)\right\} + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = 0 \qquad \text{for } 0 < t < T \; , \\ \vec{\boldsymbol{\mu}}(0) = \vec{\boldsymbol{\mu}}_0 \quad , \quad \frac{d\vec{\boldsymbol{\mu}}}{dt}(0) = \vec{\boldsymbol{v}}_0 \; . \end{cases} \tag{6.2.33}$$

- ▷ s.p.d. stiffness matrix $\mathbf{A} \in \mathbb{R}^{N,N}$, $(\mathbf{A})_{ij} := \mathsf{a}(b_N^j, b_N^i)$ (independent of time),
- ▷ s.p.d. mass matrix $\mathbf{M} \in \mathbb{R}^{N,N}$, $(\mathbf{M})_{ij} := \mathsf{m}(b_N^j, b_N^i)$ (independent of time),
- ▷ source (load) vector $\vec{\boldsymbol{\varphi}}(t) \in \mathbb{R}^N$, $(\vec{\boldsymbol{\varphi}}(t))_i := \ell(t)(b_N^i)$ (time-dependent),
- ▷ $\vec{\boldsymbol{\mu}}_0 \hat{=}$ coefficient vector of a projection of u_0 onto $V_{0,N}$.
- ▷ $\vec{\boldsymbol{v}}_0 \hat{=}$ coefficient vector of a projection of v_0 onto $V_{0,N}$.

Note: (6.2.33) is a 2nd-order ordinary differential equation (ODE) for $t \mapsto \vec{\boldsymbol{\mu}}(t) \in \mathbb{R}^N$

Remark 6.2.34 (First-order semidiscrete hyperbolic evolution problem)

Completely analoguous to § 6.2.17, introduce separate unknown function for the velocity:

$$\mathbf{M}\left\{\frac{d^2}{dt^2}\vec{\boldsymbol{\mu}}(t)\right\} + \mathbf{A}\vec{\boldsymbol{\mu}}(t) = 0$$

$\leftarrow$ auxiliary unknown $\vec{\boldsymbol{v}} = \dot{\vec{\boldsymbol{\mu}}}$

$$\left\{\begin{array}{l} \dfrac{d}{dt}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{v}}(t)\,, \\ \mathbf{M}\dfrac{d}{dt}\vec{\boldsymbol{v}}(t) = -\mathbf{A}\vec{\boldsymbol{\mu}}(t)\,, \end{array}\right. \quad , \quad 0 < t < T\,. \tag{6.2.35}$$

with intial conditions

$$\vec{\boldsymbol{\mu}}(0) = \vec{\boldsymbol{\mu}}_0 \quad , \quad \vec{\boldsymbol{v}}(0) = \vec{\boldsymbol{v}}_0\,. \tag{6.2.36}$$

6.2.4 Timestepping

(6.2.37) Method-of-lines ODE

The method of lines approach gives us the semi-discrete hyperbolic evolution problem = 2nd-order ODE:

$$\mathbf{M}\left\{\frac{d^2}{dt^2}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu}(t) = 0 \quad , \quad \vec{\mu}(0) = \vec{\mu}_0\,, \quad \frac{d\vec{\mu}}{dt}(0) = \vec{\eta}_0\,. \tag{6.2.38}$$

Key features of (6.2.38) ➡ to be respected "approximately" by timestepping:

- ✦ reversibility: (6.2.38) invariant under time-reversal $t \leftarrow -t$
- ✦ energy conservation, *cf.* Thm. 6.2.29: $E_N(t) := \frac{1}{2}\frac{d\vec{\mu}}{dt} \cdot \mathbf{M}\frac{d\vec{\mu}}{dt} + \frac{1}{2}\vec{\mu} \cdot \mathbf{A}\vec{\mu} = \mathrm{const}$

Experiment 6.2.39 (Euler timestepping for 1st-order form of semi-discrete wave equation)

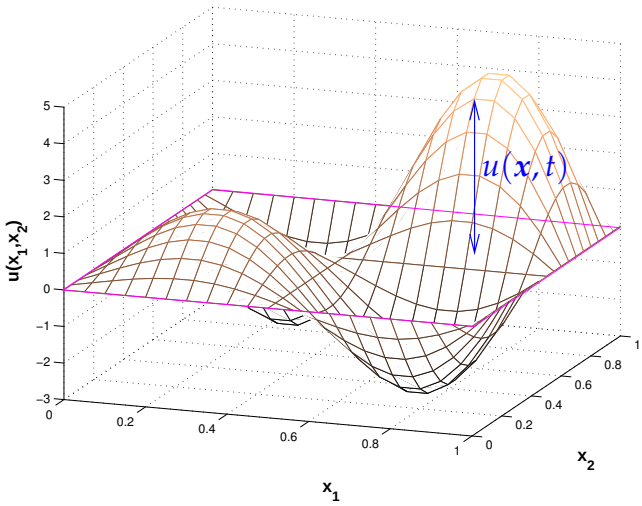


Fig. 283

Model problem: wave propagation on a square membrane

$$\frac{\partial^2 u}{\partial t^2} - \Delta u = 0 \quad \text{on} \quad]0,1[^2 \times]0,1[\,,$$
$$u(\boldsymbol{x},t) = 0 \quad \text{on} \quad \partial\Omega \times]0,T[\,,$$
$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad , \quad \frac{\partial u}{\partial t}(\boldsymbol{x},0) = 0\,.$$

- Initial data $u_0(\boldsymbol{x}) = \max\{0, \frac{1}{5} - \|\boldsymbol{x}\|\}$, $v_0(\boldsymbol{x}) = 0$,
- $\mathcal{M} \hat{=}$ "structured triangular tensor product mesh", see Fig. 178, n squares in each direction,
- linear finite element space $V_{N,0} = \mathcal{S}^0_{1,0}(\mathcal{M})$, $N := \dim \mathcal{S}^0_{1,0}(\mathcal{M}) = (n-1)^2$,
- All local computations ($\rightarrow$ Section 3.6.5) rely on 3-point vertex based local quadrature formula "2D trapezoidal rule" (3.3.49). More explanations will be given in Rem. 6.2.45 below.

- $\mathbf{A}$ = $N \times N$ Poisson matrix, see (4.1.5), scaled with $h := n^{-1}$,
- mass matrix $\mathbf{M} = h\mathbf{I}$, thanks to quadrature formula, see Rem. 6.2.45.

Timestepping: implicit and explicit Euler method ($\rightarrow$ Ex. 6.1.35, [4, Section 11.2]) for 1st-order ODE (6.2.35), timestep $\tau > 0$:

$$\vec{\boldsymbol{\mu}}^{(j)} - \vec{\boldsymbol{\mu}}^{(j-1)} = \tau \vec{\boldsymbol{v}}^{(j-1)}\,,$$
$$\mathbf{M}(\vec{\boldsymbol{v}}^{(j)} - \vec{\boldsymbol{v}}^{(j-1)}) = -\tau \mathbf{A} \vec{\boldsymbol{\mu}}^{(j-1)}\,.$$

explicit Euler

$$\vec{\boldsymbol{\mu}}^{(j)} - \vec{\boldsymbol{\mu}}^{(j-1)} = \tau \vec{\boldsymbol{v}}^{(j)}\,,$$
$$\mathbf{M}(\vec{\boldsymbol{v}}^{(j)} - \vec{\boldsymbol{v}}^{(j-1)}) = -\tau \mathbf{A} \vec{\boldsymbol{\mu}}^{(j)}\,.$$

implicit Euler

Monitored: behavior of (discrete) kinetic, potential, and total energy

$$E^{(j)}_{\text{kin}} = (\vec{\boldsymbol{v}}^{(j)})^T \mathbf{M} \vec{\boldsymbol{v}}^{(j)} \quad , \quad E^{(j)}_{\text{pot}} = (\vec{\boldsymbol{\mu}}^{(j)})^T \mathbf{A} \vec{\boldsymbol{\mu}}^{(j)}\,, \quad j = 0,1,\ldots\,.$$

Explicit Euler timestepping:

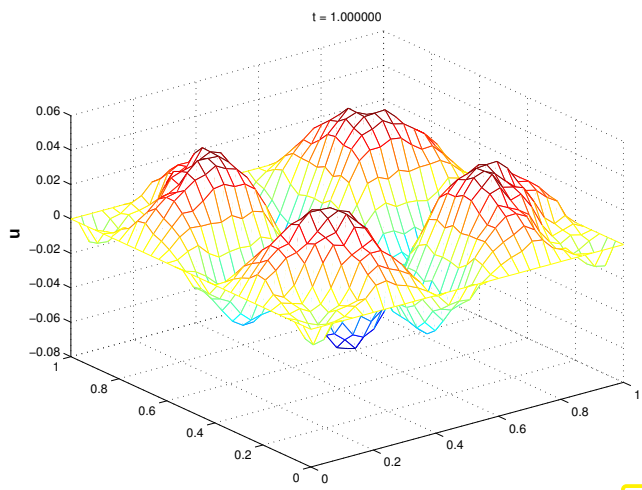


Fig. 284

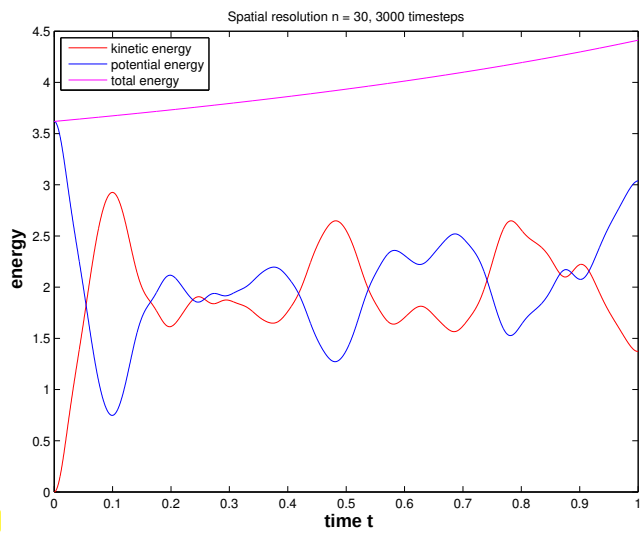


Fig. 285

Implcit Euler timestepping:

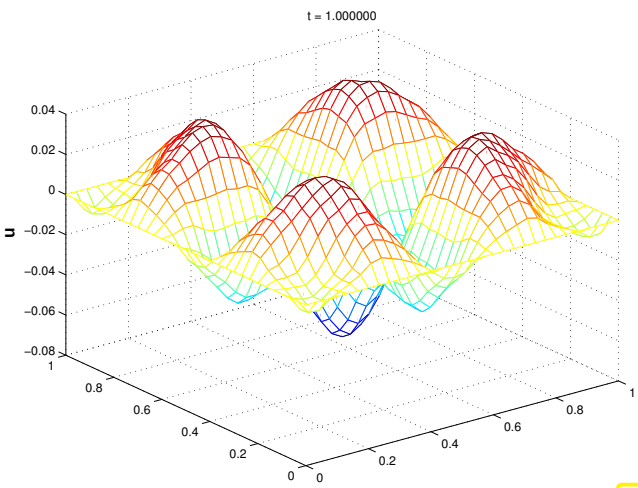


Fig. 286

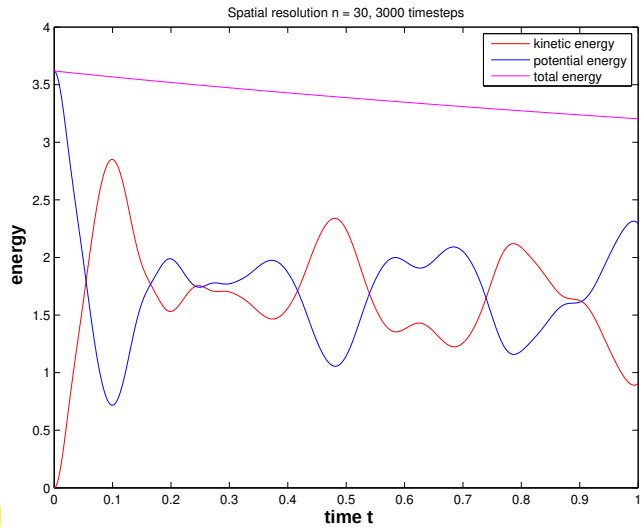


Fig. 287

Observation: neither method conserves energy,

☞ explicit Euler timestepping ➢ steady increase of total energy
☞ implicit Euler timestepping ➢ steady decrease of total energy

---

(6.2.40) Störmer-Verlet timestepping

Exp. 6.2.39 ➢ Euler methods violate energy conservation!

(The same is true of all explicit Runge-Kutta methods, which lead to an increase of the total energy over time, and L(π)-stable implicit Runge-Kutta method, which make the total energy decay.)

Let us try another simple idea for the 2nd-order ODE (6.2.33):

Replace $\frac{d^2}{dt^2}\vec{\mu}$ with symmetric difference quotient (1.5.138)

$$\mathbf{M}\left\{\frac{d^2}{dt^2}\vec{\mu}(t)\right\} + \mathbf{A}\vec{\mu}(t) = 0 \tag{6.2.38}$$

$$\boxed{\mathbf{M}\,\frac{\vec{\mu}^{(j+1)} - 2\vec{\mu}^{(j)} + \vec{\mu}^{(j-1)}}{\tau^2} = -\mathbf{A}\vec{\mu}^{(j)}\,, \quad j = 0,1,\dots}\,. \tag{6.2.41}$$

This is a two-step method, the Störmer scheme/explicit trapezoidal rule

By Taylor expansion: Störmer scheme is a 2nd-order method

However, from where do we get $\vec{\mu}^{(-1)}$? Two-step methods need to be kick-started by a *special initial step*: This is constructed by approximating the second initial condition by a symmetric difference quotient:

$$\frac{d}{dt}\vec{\mu}(0) = \vec{v}_0 \quad \blacktriangleright \quad \frac{\vec{\mu}^{(1)} - \vec{\mu}^{(-1)}}{2\tau} = \vec{v}_0\,. \tag{6.2.42}$$

(6.2.43) Leapfrog timestepping

For the semi-discrete wave equation we again consider the explicit trapezoidal rule (Störmer scheme):

$$\mathbf{M}\,\frac{\vec{\mu}^{(j+1)} - 2\vec{\mu}^{(j)} + \vec{\mu}^{(j-1)}}{\tau^2} = -\mathbf{A}\vec{\mu}^{(j)}\,, \quad j = 1,\dots\,. \tag{6.2.41}$$

Inspired by Rem. 6.2.34 we introduce the auxiliary variable

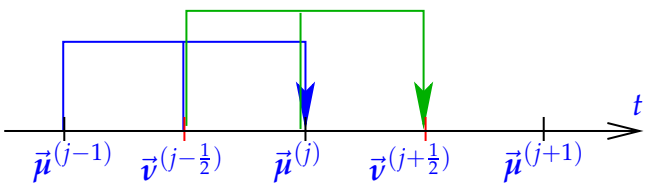
$$\vec{v}^{(j+1/2)} := \frac{\vec{\mu}^{(j+1)} - \vec{\mu}^{(j)}}{\tau}\,,$$

which can be read as an approximation of the velocity $v := \dot{u}$.

This leads to a timestepping scheme, which is *algebraically equivalent* to the explicit trapezoidal rule: leapfrog timestepping (with uniform timestep $\tau > 0$):

$$\boxed{\begin{aligned} \mathbf{M}\,\frac{\vec{v}^{(j+\frac{1}{2})} - \vec{v}^{(j-\frac{1}{2})}}{\tau} &= -\mathbf{A}\vec{\mu}^{(j)}\,, \\ \frac{\vec{\mu}^{(j+1)} - \vec{\mu}^{(j)}}{\tau} &= \vec{v}^{(j+\frac{1}{2})}\,, \end{aligned} \quad j = 0,1,\dots\,,} \tag{6.2.44}$$

$+$ initial step $\vec{v}^{(-\frac{1}{2})} + \vec{v}^{(\frac{1}{2})} = 2\vec{v}_0$.



work per step:

1$\times$ evaluation $\mathbf{A}\times$vector,

1$\times$ solution of linear system for $\mathbf{M}$

Remark 6.2.45 (Mass lumping)

Required in each step of leapfrog timestepping: solution of linear system of equations with (large sparse) system matrix $\mathbf{M} \in \mathbb{R}^{N,N}$ ➢ expensive!

Trick for (bi-)linear finite element Galerkin discretization: $V_{0,N} \subset \mathcal{S}_1^0(\mathcal{M})$:

use *vertex based local quadrature rule*

(e.g. "2D trapezoidal rule" (3.3.49) on triangular mesh)

$$\int_K f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{\sharp\mathcal{V}(K)} \sum_{\boldsymbol{p}\in\mathcal{V}(K)} f(\boldsymbol{p})\,, \quad \mathcal{V}(K) := \text{set of vertices of } K\,.$$

(For a comprehensive discussion of local quadrature rules see Section 3.6.5)

Mass matrix $\mathbf{M}$ will become a *diagonal* matrix (due to defining equation (3.3.13) for nodal basis functions, which are associated with nodes of the mesh).

This so-called mass lumping trick was was used in the finite element discretization of Exp. 6.2.39.

Experiment 6.2.46 (Energy conservation for leapfrog)

Model problem and discretization as in Exp. 6.2.39.

Leapfrog timestepping with constant timestep size $\tau = 0.01$

MATLAB code 6.2.47: Computing behavior of energies for Störmer timestepping

```
function lfen(n,m)
% leapfrom timestepping for 2D wave equation, computation of energies
% n:  spatial resolution (no.  of cells in one direction)
% m:  number of timesteps

% Assemble stiffness matrix, see Sect. 4.1, (4.1.5)
N = (n-1)^2; h = 1/n; A = gallery('poisson',n-1)/(h*h);

% initial displacement u_0(x) = max{0, 1/5 - ||x||}
[X,Y] = meshgrid(0:h:1,0:h:1);
U0 = 0.2-sqrt((X-0.5).^2+(Y-0.5).^2);
U0(find(U0 < 0)) = 0.0;
u0 = reshape(U0(2:end-1,2:end-1),N,1);
v0 = zeros(N,1);   % initial velocity

% loop for Störmer timestepping, see (6.2.41)
tau = 1/m;                        % uniform timestep size
u = u0+tau*v0-0.5*tau^2*A*u0;  % special initial step
u_old = u0;
[pen,ken] = geten(A,tau,u0,u); % compute potential and kinetic energy
E = [0.5*tau,pen,ken,pen+ken];
for k=1:m-1
  u_new = -(tau^2)*(A*u) + 2*u - u_old;
  [pen,ken] = geten(A,tau,u,u_new);
  E = [E;(k+0.5)*tau,pen,ken,pen+ken];
  u_old = u; u = u_new;
end

figure('name','Leapfrog energies');
plot(E(:,1),E(:,3),'r-',E(:,1),E(:,2),'b-',E(:,1),E(:,4),'m-');
xlabel('{\bf time t}','fontsize',14);
ylabel('{\bf energies}','fontsize',14);
legend('kinetic energy','potential energy','total energy','location','south');
```

```
title(sprintf('Spatial resolution n = %i, %i timesteps',n,m));

print('-depsc',sprintf('../../../../rw/Slides/NPDEPics/leapfrogen%d.eps',n
```

MATLAB code 6.2.48: Computing potential and kinetic energiy for Störmer timestepping

```
function [pen,ken] = geten(A,ts,u_old,u_new)
% Compute the current approximate potential and kinetic energies for u_old
% and u_new from Störmer timestepping
%E_kin^(j) = τ^-2(μ^(j) − μ^(j−1))^T M(μ^(j) − μ^(j−1)) , E_pot^(j) = 1/4(μ^(j) + μ^(j−1))^T A(μ^(j) + μ^(j−1)), j = 0,1,... .
meanv = 0.5*(u_old+u_new); pen = dot(meanv,A*meanv); % potential energy
dtemp = (u_new-u_old)/ts; ken = dot(dtemp,dtemp); % kinetic energy
```

MATLAB code 6.2.49: Computing behavior of energies for Störmer timestepping ↪ GITLAB

```
// arguments:
// integer n Spatial resolution (no. of cells in one direction)
// integer m Number of timesteps
void lfen(int n, int m) {
  // Leapfrog timestepping for 2D wave equation, computation of energies

  //assemble stiffness matrix, see Sect. 4.1, (4.1.5)
  int N = (n − 1)*(n − 1);
  double h = 1.0/n;
  Eigen::MatrixXd A = NPDE::poisson(n − 1)/(h*h);

  //initial displacement u_0(x) = max{0, 1/5 − ||x||}
  Eigen::ArrayXXd X;
  Eigen::ArrayXXd Y;
  Eigen::ArrayXd gridcoords = Eigen::ArrayXd::LinSpaced(n − 1, h, 1.0 − h);
  std::tie(X, Y) = NPDE::meshgrid(gridcoords, gridcoords);
  Eigen::ArrayXXd U0 = 0.2 − ((X − 0.5).square() + (Y − 0.5).square()).sqrt();
  U0 = U0.max(0.0);
  Eigen::Map<Eigen::VectorXd> u0(U0.data(), U0.size());

  //initial velocity
  Eigen::VectorXd v0 = Eigen::VectorXd::Zero(N);

  //loop for Störmer timestepping, see (6.2.41)
  double tau = 1.0/m; //uniform timestep size
  Eigen::VectorXd u = u0 + tau*v0 − 0.5*tau*tau*A*u0; //special initial step
  Eigen::VectorXd u_old = u0;
  double pen;
```

```
  double ken;
  std::tie(pen, ken) = geten(A, tau, u0, u); //compute potential and kinetic energy
  Eigen::MatrixXd E(m, 4);
  E.row(0) << 0.5*tau, pen, ken, pen + ken;
  for (int k = 1; k < m; k++) {
    Eigen::VectorXd u_new = -(tau*tau)*(A*u) + 2*u - u_old;
    std::tie(pen, ken) = geten(A, tau, u, u_new);
    E.row(k) << (k + 0.5)*tau, pen, ken, pen + ken;
    u_old = u;
    u = u_new;
  }

  mgl::Figure fig;
  fig.plot(E.col(0), E.col(2), "r-").label("kinetic energy");
  fig.plot(E.col(0), E.col(1), "b-").label("potential energy");
  fig.plot(E.col(0), E.col(3), "m-").label("total energy");
  fig.legend();
  fig.xlabel("{\b time t}");
  fig.ylabel("{\b energies}");
  std::stringstream title;
  title << "Spatial resolution n = " << n << ", " << m << " timesteps";
  fig.title(title.str());
  std::stringstream filename;
  filename << "leapfrogen" << m << ".eps";
  fig.save(filename.str());
}
```

MATLAB code 6.2.50: Computing potential and kinetic energiy for Störmer timestepping
➙ GITLAB

```
// arguments:
// Matrix A stiffnes matrix
// double ts timestep size
// Vector u_old previous solution
// Vectur u_new current solution
// returns:
// tuple containing
// the potential energy
// the kinetic energy
std::tuple<double, double> geten(const Eigen::MatrixXd& A, double ts, const Eigen::VectorXd& u_old, const Eigen::VectorXd& u_new) {
  // Compute the current approximate potential and kinetic energies for u_old
  // and u_new from Sörmer timestepping
  // E_kin^(j) = τ^-2 (μ^(j) - μ^(j-1))^T M (μ^(j) - μ^(j-1)) , E_pot^(j) = 1/4 (μ^(j) + μ^(j-1))^T A (μ^(j) + μ^(j-1)) , j = 0,1,... .
  Eigen::VectorXd meanv = 0.5*(u_old + u_new);
  Eigen::VectorXd dtemp = (u_new - u_old)/ts;
```

```
  double pen = meanv.dot(A*meanv);
  double ken = dtemp.dot(dtemp);
  return std::make_tuple(pen, ken);
}
#pragma endgeten

// 6.2.49
//
/*LSTBEGIN2*/
// arguments:
// integer n Spatial resolution (no. of cells in one direction)
// integer m Number of timesteps
void lfen(int n, int m) {
  // Leapfrog timestepping for 2D wave equation, computation of energies

  //assemble stiffness matrix, see Sect. 4.1, (4.1.5)
  int N = (n - 1)*(n - 1);
  double h = 1.0/n;
  Eigen::MatrixXd A = NPDE::poisson(n - 1)/(h*h);

  //initial displacement u_0(x) = max{0, 1/5 - ||x||}
  Eigen::ArrayXXd X;
  Eigen::ArrayXXd Y;
  Eigen::ArrayXd gridcoords = Eigen::ArrayXd::LinSpaced(n - 1, h, 1.0 - h);
  std::tie(X, Y) = NPDE::meshgrid(gridcoords, gridcoords);
  Eigen::ArrayXXd U0 = 0.2 - ((X - 0.5).square() + (Y - 0.5).square()).sqrt();
  U0 = U0.max(0.0);
  Eigen::Map<Eigen::VectorXd> u0(U0.data(), U0.size());

  //initial velocity
  Eigen::VectorXd v0 = Eigen::VectorXd::Zero(N);

  //loop for Störmer timestepping, see (6.2.41)
  double tau = 1.0/m; //uniform timestep size
  Eigen::VectorXd u = u0 + tau*v0 - 0.5*tau*tau*A*u0; //special initial step
  Eigen::VectorXd u_old = u0;
  double pen;
  double ken;
  std::tie(pen, ken) = geten(A, tau, u0, u); //compute potential and kinetic energy
  Eigen::MatrixXd E(m, 4);
  E.row(0) << 0.5*tau, pen, ken, pen + ken;
  for (int k = 1; k < m; k++) {
    Eigen::VectorXd u_new = -(tau*tau)*(A*u) + 2*u - u_old;
    std::tie(pen, ken) = geten(A, tau, u, u_new);
    E.row(k) << (k + 0.5)*tau, pen, ken, pen + ken;
    u_old = u;
```

```
    u = u_new;
  }

  mgl::Figure fig;
  fig.plot(E.col(0), E.col(2), "r-").label("kinetic energy");
  fig.plot(E.col(0), E.col(1), "b-").label("potential energy");
  fig.plot(E.col(0), E.col(3), "m-").label("total energy");
  fig.legend();
  fig.xlabel("{\b time t}");
  fig.ylabel("{\b energies}");
  std::stringstream title;
  title << "Spatial resolution n = " << n << ", " << m << "
     timesteps";
  fig.title(title.str());
  std::stringstream filename;
  filename << "leapfrogen" << m << ".eps";
  fig.save(filename.str());
}
/*LSTEND2*/

int main(int argc, char* args[])
{
  lfen(30, 100);
}
```

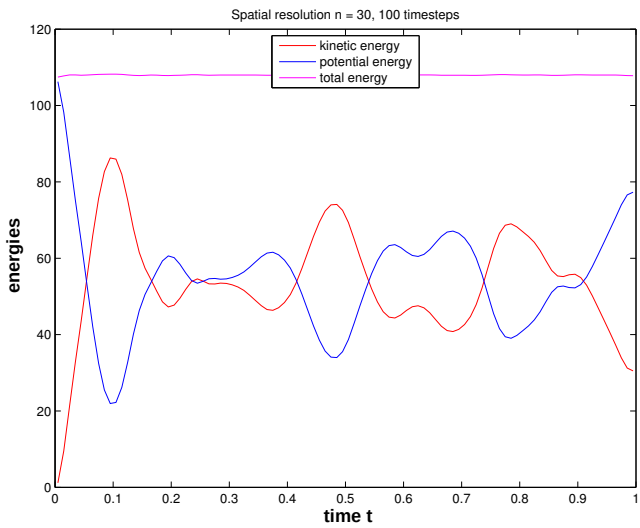


Fig. 288

Leapfrog is (nearly) energy conserving

(no energy drift, only small oscillations)

This behavior is explained by the deep mathemtical theorie of symplectic integrators, see [3].

6.2.5 CFL-condition

Experiment 6.2.51 (Blow-up for leapfrog timestepping)

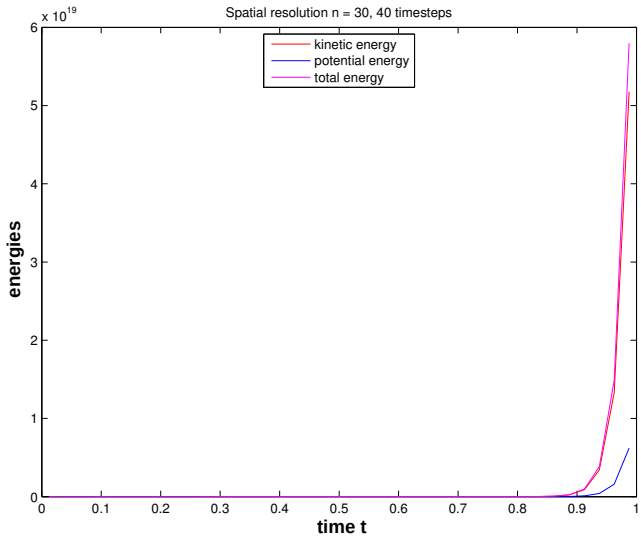


Fig. 289

◁ Exp. 6.2.46 repeated with $\tau = 0.04$
Observation:
Leapfrog suffers a blow-up: exponential increase of energies!
A similar behavior is observed with the explicit Euler scheme for the semi-discrete heat equation, in case the timestep constraint is violated, see Section 6.1.5.2.

(6.2.52) Diagonalization of method-of-lines ODE $\rightarrow$ **§ 6.1.58**

➢ (as in Section 6.1.5.2) Stability analysis of leapfrog timestepping based on **diagonalization**:

$$\exists \text{ orthogonal } \mathbf{T} \in \mathbb{R}^{N,N}: \quad \mathbf{T}^\top \mathbf{M}^{-1/2}\mathbf{A}\mathbf{M}^{-1/2}\mathbf{T} = \mathbf{D} := \operatorname{diag}(\lambda_1, \dots, \lambda_N) \,.$$

where the $\lambda_i > 0$ are *generalized eigenvalues* for $\mathbf{A}\vec{\xi} = \lambda \mathbf{M}\vec{\xi}$ ➤ $\lambda_i \geq \gamma$ for all i (γ is the constant introduced in (6.1.20)).

Next, apply transformation $\vec{\eta} := \mathbf{T}^T\mathbf{M}^{1/2}\vec{\mu}$ to the 2-step formulation (6.2.41)

$$\text{(6.2.41)} \quad \overset{\vec{\eta}:=\mathbf{T}^\top\mathbf{M}^{1/2}\vec{\mu}}{\Longrightarrow} \quad \vec{\eta}^{(j+1)} - 2\vec{\eta}^{(j)} + \vec{\eta}^{(j-1)} = -\tau^2\mathbf{D}\vec{\eta}^{(j)} \,.$$

Again, we have achieved a complete decoupling of the timestepping for the eigencomponents.

$$\eta_i^{(j+1)} - 2\eta_i^{(j)} + \eta_i^{(j-1)} = -\tau^2\lambda_i\eta_i^{(j)} \,, \quad i = 1, \dots, N \,, \quad j = 1, 2, \dots \,. \tag{6.2.53}$$

In fact, (6.2.53) is what we end up with then applying Störmers scheme to the *scalar* linear 2nd-order ODE $\ddot{\eta}_i = -\lambda_i\eta_i$. In a sense, the commuting diagram (6.1.85) remains true for 2-step methods and second-order ODEs.

(6.2.53) is a linear two-step recurrence formula for the sequences $(\eta_i^{(j)})_j$.

Try: $$\eta_i^{(j)} = \xi^j \quad \text{for some} \quad \xi \in \mathbb{C} \setminus \{0\}$$

Plug this into (6.2.53)

$$\blacktriangleright \quad \xi^2 - 2\xi + 1 = -\tau^2\lambda_i\xi \quad \Leftrightarrow \quad \xi^2 - (2 - \tau^2\lambda_i)\xi + 1 = 0 \,.$$

$$\Rightarrow \quad \text{two solutions} \quad \xi_\pm = \tfrac{1}{2}\left(2 - \tau^2\lambda_i \pm \sqrt{(2-\tau^2\lambda_i)^2 - 4}\right) \,.$$

We can get a blow-up of some solutions of (6.2.53), if $|\xi_+| > 1$ of $|\xi_-| > 1$. From secondary school we know Vieta's formula

$$\xi_+ \cdot \xi_- = 1 \quad \Rightarrow \quad \begin{array}{l} \left\{\xi_\pm \in \mathbb{R} \quad \text{and} \quad \xi_+ \neq \xi_- \quad \Rightarrow \quad |\xi_+| > 1 \text{ or } |\xi_-| > 1\right\} \,, \\ \left\{\xi_- = \xi_+^* \quad \Rightarrow \quad |\xi_-| = |\xi_+| = 1\right\} \,, \end{array}$$

where $\bar{\zeta}_+^*$ designates complex conjugation. So the recurrence (6.2.53) has only bounded solution, if and only if

$$\text{discriminant} \quad D := (2-\tau^2\lambda_i)^2 - 4 \leq 0 \quad \Leftrightarrow \quad \boxed{\tau \leq \frac{2}{\sqrt{\lambda_i}}} \,. \tag{6.2.54}$$

$\longleftrightarrow$ *stability induced timestep constraint for leapfrog timestepping*

(6.2.55) The CFL-condition

Special setting: spatial finite element Galerkin discretization based on fixed degree Lagrangian finite element spaces ($\rightarrow$ Section 3.5), meshes created by uniform regular refinement.

Under these conditions a generalization of Lemma 6.1.74 shows

Stability of leapfrog timestepping entails $\tau \leq O(h_{\mathcal{M}})$ for $h_{\mathcal{M}} \to 0$

This is known as Courant-Friedrichs-Lewy (CFL) condition

Remark 6.2.56 (Geometric interpretation of CFL condition in 1D)

Setting:

- 1D wave equation, (spatial) boundary conditions ignored ("Cauchy problem"),

$$c > 0: \quad \frac{\partial^2 u}{\partial t^2} - c^2 \frac{\partial^2 u}{\partial x^2} = 0 \quad , \quad u(x,0) = u_0(x) \,, \quad \frac{\partial u}{\partial t}(x,0) = v_0(x) \,, \quad x \in \mathbb{R} \,. \tag{6.2.21}$$

- Linear finite element Galerkin discretization on equidistant spatial mesh $\mathcal{M} := \{[x_{j-1}, x_j] : j \in \mathbb{Z}\}$, $x_j := hj$ (meshwidth h), see Section 1.5.2.2.
- Mass lumping for computation of mass matrix, which will become $h \cdot \mathbf{I}$, see Rem. 6.2.45.
- Timestepping by Sörmer scheme (6.2.41) with constant timestep $\tau > 0$.

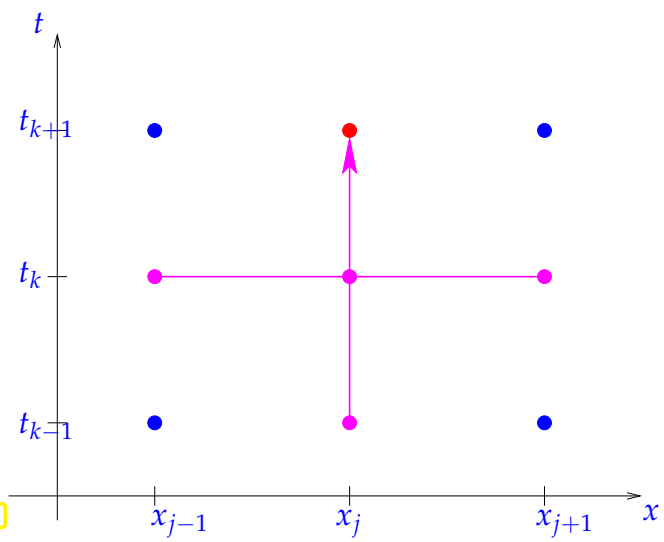


Fig. 290

$\triangleleft$ flow of information in one step of Störmer scheme
Since the method is a two-step method, information from time-slices t_k and t_{k-1} is needed.

Below: yellow region $\hat{=}$ domain of dependence (d.o.d.) of $(\bar{x},\bar{t})$

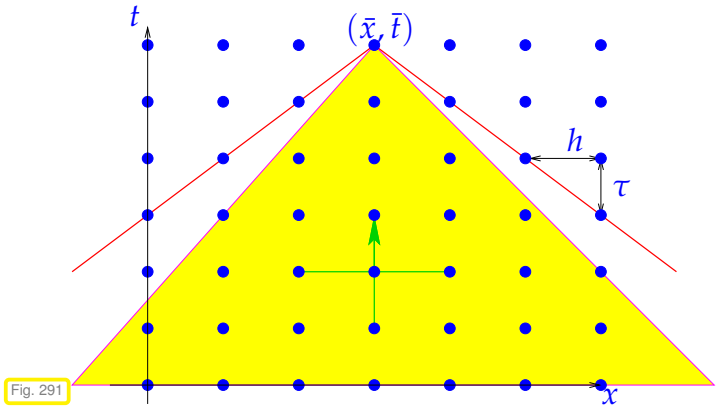


Fig. 291

$c\tau < h$: numerical domain of dependence (marked with —) contained in d.o.d.

➪ CFL-condition met

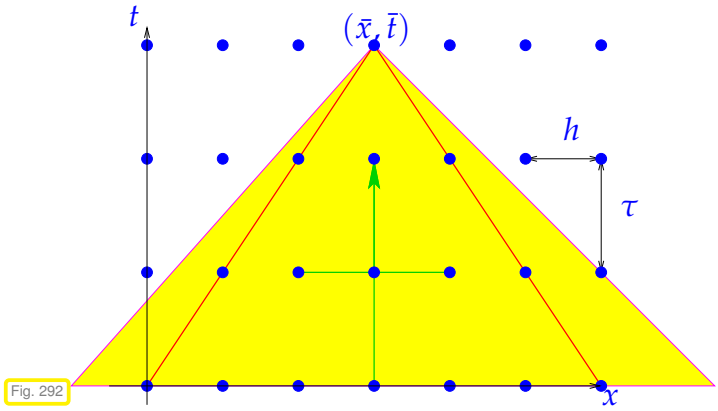


Fig. 292

$c\tau > h$: numerical domain of dependence (marked with —) **not** contained in d.o.d.

➪ CFL-condition violated

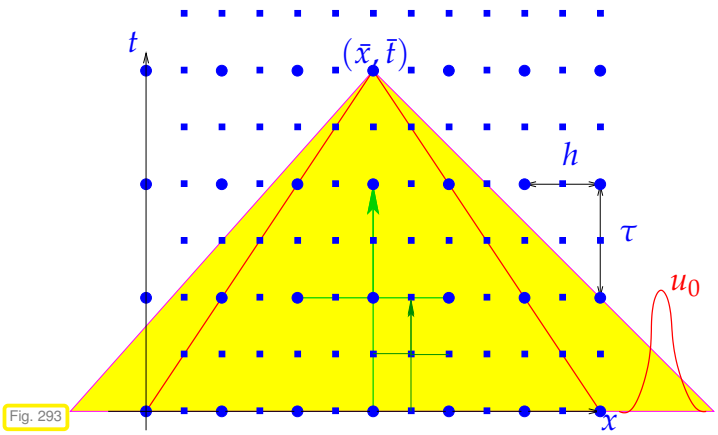


Fig. 293

(• $\hat{=}$ coarse grid, ■ $\hat{=}$ fine grid, □ $\hat{=}$ d.o.d)

◁ 1D consideration:

sequence of equidistant space-time grids of $\widetilde{\Omega}$ with $\tau = \gamma h$ (τ/h = meshwidth in time/space)

If $\gamma >$ CFL-constraint (here $\gamma > c^{-1}$), then

analytical domain of dependence $\not\subset$ numerical domain of dependence

▲ initial data u_0 outside numerical domain of dependence cannot influence approximation at grid point $(\bar{x},\bar{t})$ on *any* mesh ► no convergence !

CFL-condition $\Leftrightarrow$ analytical domain of dependence $\subset$ numerical domain of dependence

Will the CFL-condition thwart the efficient use of leapfrog, see Rem. 6.1.101 **?**

To this end we need an idea about the convergence of the solutions of the fully discrete method:

"Meta-theorem" 6.2.57. Convergene of fully discrete solutions of the wave equation

Assume that

- *the solution of the IBVP for the wave equation* (6.2.27) *is "sufficiently smooth",*
- *its spatial Galerkin finite element discretization relies on degree p Lagrangian finite elements ($\rightarrow$ Section 3.5) on uniformly shape-regular families of meshes,*
- *timestepping is based on the leapfrog method* (6.2.44) *with uniform timestep $\tau > 0$ satisfying* (6.2.54).

Then we can expect an asymptotic behavior of the total discretization error according to

$$\left(\tau \sum_{j=1}^{M} \|u - u_N(\tau j)\|^2_{H^1(\Omega)}\right)^{\frac{1}{2}} \leq C\left(h_{\mathcal{M}}^p + \tau^2\right), \tag{6.2.58}$$

$$\left(\tau \sum_{j=1}^{M} \|u - u_N(\tau j)\|^2_{L^2(\Omega)}\right)^{\frac{1}{2}} \leq C\left(h_{\mathcal{M}}^{p+1} + \tau^2\right), \tag{6.2.59}$$

where $C > 0$ must not depend on $h_{\mathcal{M}}$, τ. *L.F. is 2nd-order !*

"expect": unless lack of regularity of the solution u interferes, *cf.* Section 5.4, § 5.4.11.

As in the case of Thm. 6.1.96 (☞ nothing new!) we find:

total discretization error = spatial error + temporal error

§ 5.3.59 still applies: (6.2.58) does not give information about actual error, but only about the trend of the error, when discretization parameters $h_{\mathcal{M}}$ and τ are varied.

Nevertheless, as in the case of the a priori error estimates of Section 5.3.5, we can draw conclusions about optimal y refinement strategies in order to achieve prescribed *error reduction*.

As in Section 5.3.5 we make the **assumption** that the estimates (6.2.58) are sharp for all contributions to the total error and that the constants are the same (!)

$$\begin{aligned} \text{contribution of spatial (energy) error} &\approx Ch_{\mathcal{M}}^p \quad , h_{\mathcal{M}} \mathrel{\hat{=}} \text{mesh width } (\rightarrow \text{Def. 5.2.3}) \,, \\ \text{contribution of temporal error} &\approx C\tau^2 \quad , \tau \mathrel{\hat{=}} \text{timestep size} \,. \end{aligned} \tag{6.2.60}$$

This suggests the following change of $h_{\mathcal{M}}$, τ in order to achieve *error reduction* by a factor of $\rho > 1$:

$$\begin{aligned} \text{reduce mesh width by factor } & \rho^{1/p} \\ \text{reduce timestep by factor } & \rho^{1/2} \end{aligned} \overset{(6.1.98)}{\Longrightarrow} \text{(energy) error reduction by } \rho > 1 \,. \tag{6.2.61}$$

Guideline:	spatial and temporal resolution have to be adjusted in tandem

Parallel zu Rem. 6.1.101 we may wonder whether the timestep constraint $\tau < O(h_{\mathcal{M}})$ (asymptotically) enforces small timesteps not required for accuracy:

When interested in error in *energy norm* ($\leftrightarrow H^1(\Omega)$-norm):

Only for $p = 1$ (linear Lagrangian finite elements) the requirement $\tau < O(h_{\mathcal{M}})$ stipulates the use of a smaller timestep than accuracy balancing according to (6.2.61).

When interested in $L^2(\Omega)$-norm:

> No undue timestep constraint enforced by CFL-conditon for any (h-version) of Lagrangian finite element Galerkin discretization.

The leapfrog timestep constraint $\tau \leq O(h_{\mathcal{M}})$ does not compromise (asymptotic) efficiency, if $p \geq 2$ ($p \hat{=}$ degree of spatial Lagrangian finite elements).

Learning Outcomes

Learning outcomes

After having studied this section you should

- know the transient heat equation along with suitable initial and boundary conditions for it.
- know the wave equation governing the movement of a taut elastic membrane.
- be able to state the spatial variational formulation of given second-order linear parabolic and hyperbolic IBVPs.
- understand the principle of the method of lines and how to apply it to convert a second-order linear parabolic/hyperbolic IBVP into an ordinary differential equation.
- be able to apply a Runge-Kutta timestepping scheme to a spatially semi-discrete linear IBVP.
- why semi-discrete second-order linear parabolic IBVP are "stiff" and why this calls for implicit timestepping based on efficiency considerations.
- be able to predict the convergence of a full discretization of a second-order linear parabolic/hyperbolic IBVP.
- know Störmer timestepping scheme for the linear wave equation.

Bibliography

[1] A. Burtscher, E. Fonn, P. Meury, and C. Wiesmayr. *LehrFEM - A 2D Finite Element Toolbox*. SAM, ETH Zürich, Zürich, Switzerland, 2010. http://www.sam.math.ethz.ch/ hiptmair/tmp/LehrFEMManual.pdf.

[2] L.C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1998.

[3] E. Hairer, C. Lubich, and G. Wanner. *Geometric numerical integration*, volume 31 of *Springer Series in Computational Mathematics*. Springer, Heidelberg, 2 edition, 2006.

[4] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[5] P. Knabner and L. Angermann. *Numerical Methods for Elliptic and Parabolic Partial Differential Equations*, volume 44 of *Texts in Applied Mathematics*. Springer, Heidelberg, 2003.

Chapter 7

Convection-Diffusion Problems

Contents

7.1 Heat conduction in a fluid

$\Omega \subset \mathbb{R}^d \mathrel{\hat{=}}$ bounded computational domain, $d = 1, 2, 3$

To begin with we want to develop a mathematical model for stationary fluid flow, for instance, the steady streaming of water.

7.1.1 Modelling fluid flow

Flow field:

$$\mathbf{v} : \Omega \mapsto \mathbb{R}^d$$

Assumption:

$\mathbf{v}$ is *continuous*, $\mathbf{v} \in (C^0(\overline{\Omega}))^d$

In fact, we will require that $\mathbf{v}$ is uniformly Lipschitz continuous, but this is a mere technical assumption.

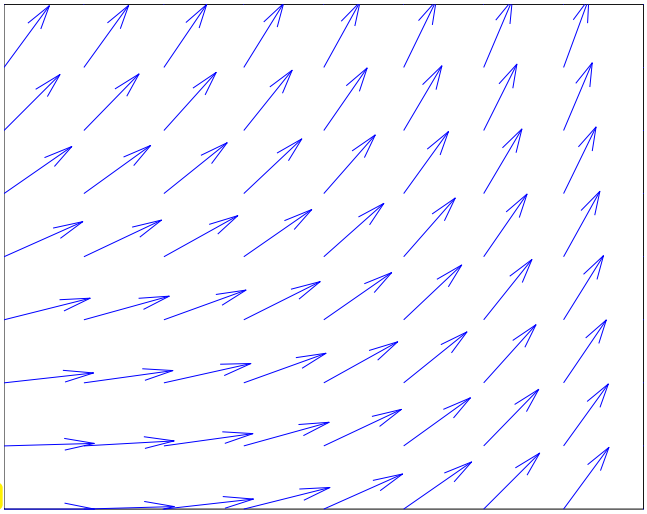


Fig. 294

Clearly: $\boxed{\mathbf{v}(\boldsymbol{x}) \mathrel{\hat{=}} \text{fluid velocity at point } \boldsymbol{x} \in \Omega}$

➢ $\mathbf{v}$ corresponds to a velocity field!

Given a flow field $\mathbf{v} \in (C^0(\overline{\Omega}))^d$ we can consider the autonomous initial value problems

$$\frac{d}{dt}\mathbf{y} = \mathbf{v}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \boldsymbol{x}_0 \,. \tag{7.1.1}$$

Its solution $t \mapsto \mathbf{y}(t)$ defines the path travelled by a particle carried along by the fluid, a particle trajectory, also called a streamline.

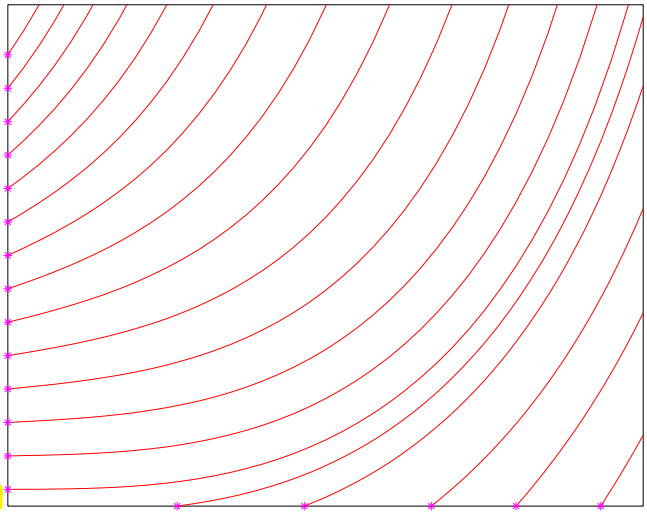


Fig. 295

◁ particle trajectories (streamlines) in flow field of Fig. 294.

($*$ $\mathrel{\hat{=}}$ initial particle positions)

A flow field induces a transformation (mapping) of space! to explain this, let us temporarily make the assumption that

the flow does neither enter nor leave Ω,
(this applies to fluid flow in a closed container)

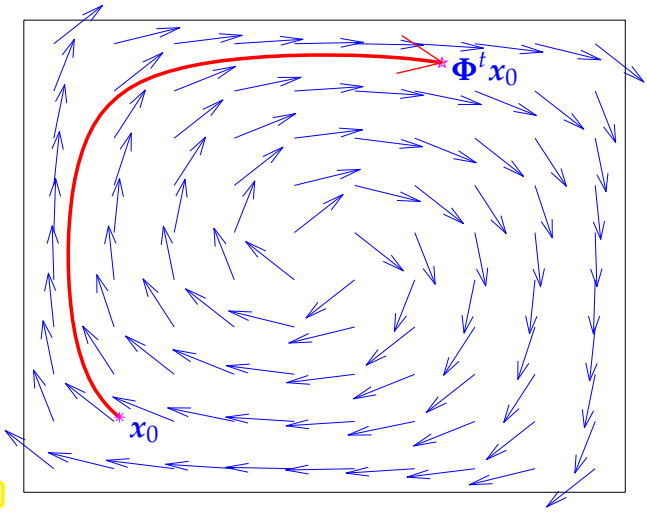


Fig. 296

which can be modelled by

$$\mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x} \in \partial\Omega \,, \tag{7.1.2}$$

that is, the flow is always parallel to the boundary of Ω: all particle trajectories stay inside Ω.

Now we fix some "time of interest" $t > 0$.

$$\rightarrow \quad \text{mapping} \quad \boldsymbol{\Phi}^t : \left\{ \begin{array}{ccc} \Omega & \mapsto & \Omega \\ \boldsymbol{x}_0 & \mapsto & \mathbf{y}(t) \end{array} \right. , \quad t \mapsto \mathbf{y}(t) \text{ solution of IVP (7.1.1)} \,, \tag{7.1.3}$$

is well-defined mapping of Ω to itself, the flow map. Obviously, it satisfies

$$\boldsymbol{\Phi}^0 \boldsymbol{x}_0 = \boldsymbol{x}_0 \quad \forall \boldsymbol{x}_0 \in \Omega \,. \tag{7.1.4}$$

In [3, Def. 11.1.39] the more general concept of an evolution operator was introduced, which agrees with the flow map in the current setting.

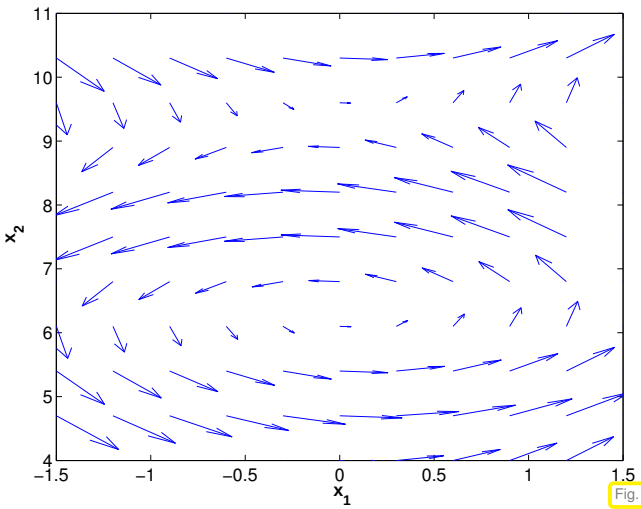


Fig. 297

flow field $\mathbf{v} : \Omega \mapsto \mathbb{R}^2$

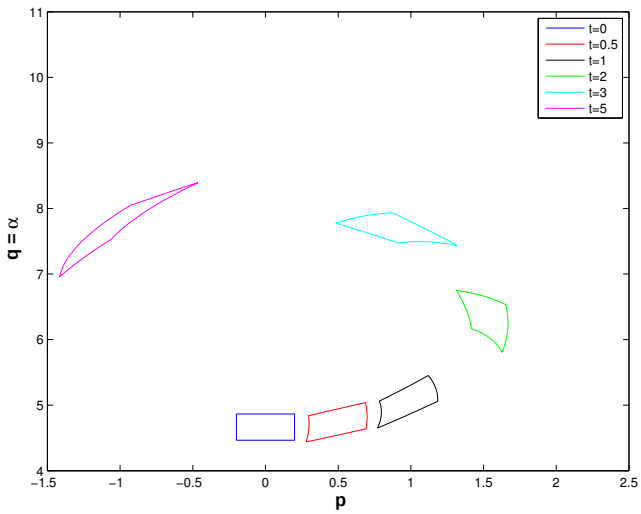


Fig. 298

snapshots of $\boldsymbol{\Phi}^t(V)$ for control volume V

$\boldsymbol{\Phi}^\tau(V) \hat{=}$ volume occupied at time $t = \tau$ by particles that occupied $V \subset \Omega$ at time $t = 0$.

7.1.2 Heat convection and diffusion

$u : \Omega \mapsto \mathbb{R} \hat{=}$ stationary temperature distribution in fluid *moving* according to a stationary flow field $\mathbf{v} : \Omega \mapsto \mathbb{R}^d$

We adapt the considerations of Sect. 2.6 that led to the stationary heat equation. Recall

Conservation of energy

$$\int_{\partial V} \mathbf{j} \cdot \boldsymbol{n} \, \mathrm{d}S = \int_V f \, \mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes" } V \,. \tag{2.6.3}$$

power flux through surface of V heat production inside V

From 2.6.3 by Gauss' theorem Thm. 2.5.7

$$\int_V \operatorname{div} \mathbf{j}(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} = \int_V f(\boldsymbol{x}) \, \mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes"} \quad V \subset \Omega \,.$$

Now appeal to another version of the fundamental lemma of the calculus of variations, see Lemma 2.5.12, this time sporting piecewise constant test functions.

▶ local form of energy conservation:

$$\operatorname{div} \mathbf{j} = f \quad \text{in } \Omega \,. \tag{2.6.8}$$

However, in a moving fluid a power flux through a fixed surface is already caused by the sheer fluid flow carrying along thermal energy. This is reflected in a modified Fourier's law (2.6.5):

Fourier's law in moving fluid

$$\mathbf{j}(\boldsymbol{x}) = -\kappa \, \mathbf{grad}\, u(\boldsymbol{x}) + \mathbf{v}(\boldsymbol{x}) \rho u(\boldsymbol{x}) \,, \quad \boldsymbol{x} \in \Omega \,. \tag{7.1.5}$$

diffusive heat flux (due to spatial variation of temperature) convective heat flux (due to fluid flow)

$\kappa > 0 \mathrel{\hat{=}}$ heat conductivity ($[\kappa] = 1 \frac{\mathrm{W}}{\mathrm{Km}}$), $\rho > 0 \mathrel{\hat{=}}$ heat capavity ($[\rho] = \frac{\mathrm{J}}{\mathrm{K\,m^3}}$), both assumed to be constant (in contrast to the models of Sect. 2.6 and Sect. 6.1.1).

Combine equations (2.6.8) & (7.1.5):

$$\operatorname{div} \mathbf{j} = f \quad + \quad \mathbf{j}(\boldsymbol{x}) = -\kappa \, \mathbf{grad}\, u(\boldsymbol{x}) + \mathbf{v}(\mathbf{x}) \rho u(\boldsymbol{x})$$

$$-\operatorname{div}(\kappa \, \mathbf{grad}\, u) + \operatorname{div}(\rho \mathbf{v}(\boldsymbol{x}) u) = f \quad \text{in} \quad \Omega \,. \tag{7.1.6}$$

▶ *Linear* scalar convection-diffusion equation (for unknown temperature u)

Terminology :

$$\underbrace{-\operatorname{div}(\kappa \, \mathbf{grad}\, u)}_{\substack{\downarrow \\ \text{diffusive term} \\ \text{(2nd-order)}}} + \underbrace{\operatorname{div}(\rho \mathbf{v}(\boldsymbol{x}) u)}_{\substack{\downarrow \\ \text{convective term} \\ \text{(1st-order)}}} = f \,.$$

The 2nd-order elliptic PDE (7.1.6) has to be supplemented with exactly one *boundary condition* on any part of $\partial\Omega$, see Sect. 2.7, Ex. 2.7.8. This can be any of the ("elliptic") boundary conditions introduced in Sect. 2.7:

- Dirichlet boundary conditions: $u = g \in C^0(\partial\Omega)$ on $\partial\Omega$ (fixed surface temperatur),
- Neumann boundary conditions: $\mathbf{j} \cdot \boldsymbol{n} = -h$ on $\partial\Omega$ (fixed heat flux),
- (non-linear) radiation boundary conditions: $\mathbf{j} \cdot \boldsymbol{n} = \Psi(u)$ on $\partial\Omega$ (temperature dependent heat flux, radiative heat flux).

Guideline: Required boundary conditions determined by highest-order term

7.1.3 Incompressible fluids

For the sake of simplicity we will mainly consider incompressible fluids.

Definition 7.1.7. Incompressible flow field

A fluid flow is called incompressible, if the associated flow map $\boldsymbol{\Phi}^t$ is volume preserving,

$$|\boldsymbol{\Phi}^t(V)| = |\boldsymbol{\Phi}^0(V)| \quad \text{for all sufficiently small } t > 0, \text{ for all control volumes } V\ .$$

Can incompressibility be read off the velocity field $\mathbf{v}$ of the flow?

To investigate this issue, again assume the "no flow through the boundary condition" (7.1.2) and recall that the flowmap $\boldsymbol{\Phi}^t$ from (7.1.3) satisfies

$$\frac{\partial}{\partial t}\boldsymbol{\Phi}(t,\boldsymbol{x}) = \mathbf{v}(\boldsymbol{\Phi}(t,\boldsymbol{x}))\ , \quad \boldsymbol{x} \in \Omega\,, t > 0\ . \tag{7.1.8}$$

Here, in order to make clear the dependence on independent variables, time occurs as an argument of $\boldsymbol{\Phi}$ in brackets, on par with $\boldsymbol{x}$.

Next, formal differentiation w.r.t. $\boldsymbol{x}$ and change of order of differentiation yields a differential equation for the Jacobian $D_{\boldsymbol{x}}\boldsymbol{\Phi}^t$,

$$(7.1.8) \quad \Rightarrow \quad \frac{\partial}{\partial t}\underbrace{(D_{\boldsymbol{x}}\boldsymbol{\Phi})}_{\text{Jacobian } \in \mathbb{R}^{d,d}}(t,\boldsymbol{x}) = D\mathbf{v}(\boldsymbol{\Phi}(t,\boldsymbol{x}))\underbrace{(D_{\boldsymbol{x}}\boldsymbol{\Phi})}_{\text{Jacobian } \in \mathbb{R}^{d,d}}(t,\boldsymbol{x})\ . \tag{7.1.9}$$

Second strand of thought: apply transformation formula for integrals (3.6.150), [6, Satz 8.5.2]: for fixed $t > 0$

$$|\boldsymbol{\Phi}(t,V)| = \int_{\boldsymbol{\Phi}(t,V)} 1\,\mathrm{d}\boldsymbol{x} = \int_V |\det(D_{\boldsymbol{x}}\boldsymbol{\Phi})(t,\widehat{\boldsymbol{x}})|\,\mathrm{d}\widehat{\boldsymbol{x}}\ . \tag{7.1.10}$$

Volume preservation by the flow map is equivalent to

$$t \mapsto |\boldsymbol{\Phi}(t,V)| = \text{const.} \quad \Longleftrightarrow \quad \frac{d}{dt}|\boldsymbol{\Phi}(t,V)| = 0\ ,$$

for any control volume $V \subset \Omega$.

$$(7.1.10) \quad \Rightarrow \quad \frac{d}{dt}|\boldsymbol{\Phi}(t,V)| = \int_V \frac{\partial}{\partial t}|\det(D_{\boldsymbol{x}}\boldsymbol{\Phi})(t,\widehat{\boldsymbol{x}})|\,\mathrm{d}\widehat{\boldsymbol{x}}\ .$$

Theorem 7.1.11. Differentiation formula for determinants

Let $\mathbf{S} : I \subset \mathbb{R} \mapsto \mathbb{R}^{n,n}$ be a smooth matrix-valued function. If $\mathbf{S}(t_0)$ is regular for some $t_0 \in I$, then

$$\frac{d}{dt}(\det \circ \mathbf{S})(t_0) = \det(\mathbf{S}(t_0))\ \operatorname{tr}(\frac{d\mathbf{S}}{dt}(t_0)\mathbf{S}^{-1}(t_0))\ .$$

$$\frac{\partial}{\partial t}\det(D_x\mathbf{\Phi})(t,\widehat{\boldsymbol{x}}) \overset{(7.1.9)}{=} \det(D_x\mathbf{\Phi})(t,\widehat{\boldsymbol{x}})\ \operatorname{tr}(D\mathbf{v}(\mathbf{\Phi}(t,\widehat{\boldsymbol{x}}))\ \underbrace{(D_x\mathbf{\Phi})(t,\widehat{\boldsymbol{x}})(D_x\mathbf{\Phi})^{-1}(t,\widehat{\boldsymbol{x}})}_{=\mathbf{I}})$$
$$= \det(D_x\mathbf{\Phi})(t,\widehat{\boldsymbol{x}})\ \operatorname{div}\mathbf{v}(\mathbf{\Phi}(t,\widehat{\boldsymbol{x}}))\ ,$$

because the divergence of a vector field $\mathbf{v}$ is just the trace of its Jacobian $D\mathbf{v}$! From (7.1.4) we know that for small $t > 0$ the Jacobian $D_x\mathbf{\Phi}(t,\widehat{\boldsymbol{x}})$ will be close to $\mathbf{I}$ and, therefore, $\det(D_x\mathbf{\Phi})(t,\widehat{\boldsymbol{x}}) \neq 0$ for $t \approx 0$. Thus, for small $t > 0$ we conclude

$$\frac{d}{dt}|\mathbf{\Phi}(t,V)| = 0 \quad \Leftrightarrow \quad \operatorname{div}\mathbf{v}(\mathbf{\Phi}(t,\widehat{\boldsymbol{x}})) = 0 \quad \forall \widehat{\boldsymbol{x}} \in V\ .$$

Since this is to hold for **any** control volume V, the final equivalence is

$$\frac{d}{dt}|\mathbf{\Phi}(t,V)| = 0 \quad \forall \text{ control volumes } V \quad \Leftrightarrow \quad \operatorname{div}\mathbf{v} = 0 \quad \text{in } \Omega\ .$$

Theorem 7.1.12. Divergence-free velocity fields for incompressible flows

A stationary fluid flow in Ω is incompressible ($\rightarrow$ Def. 7.1.7), if and only if its associated velocity field $\mathbf{v}$ satisfies $\operatorname{div}\mathbf{v} = 0$ everywhere in Ω.

In the sequel we make the **assumption**:

$$\boxed{\operatorname{div}\mathbf{v} = \sum_{j=1}^{d}\frac{\partial v_j}{\partial x_j} = 0}\ .$$

(Note: for $d = 1$ this boils down to $\frac{dv}{dx} = 0$ and implies $v = \text{const}$.)

Then we can use the product rule in higher dimensions of Lemma 2.5.4:

$$\operatorname{div}(\rho\mathbf{v}\,u) \overset{\text{Lemma 2.5.4}}{=} \rho(u\ \operatorname{div}\mathbf{v} + \mathbf{v}\cdot\mathbf{grad}\,u) \overset{\operatorname{div}\mathbf{v}=0}{=} \rho\mathbf{v}\cdot\mathbf{grad}\,u\ . \tag{7.1.13}$$

Thus, we can rewrite the scalar convection-diffusion equation (7.1.6) for an incompressible flow field

$$-\operatorname{div}(\kappa\,\mathbf{grad}\,u) + \operatorname{div}(\rho\mathbf{v}(\boldsymbol{x})u) = f \quad \text{in} \quad \Omega$$

$\leftarrow \operatorname{div}\mathbf{v} = 0$

$$\boxed{-\kappa\Delta u + \rho\mathbf{v}\cdot\mathbf{grad}\,u = f \quad \text{in} \quad \Omega}\ . \tag{7.1.14}$$

When carried along by the flow of an incompressible fluid, the temperature cannot be increased by local compression, the effect that you can witness when pumping air. Hence, only sources/sinks can lead to local extrema of the temperature.

Now recall the discussion of the physical intuition behind the maximum principle of Thm. 5.7.2. These considerations still apply to stationary heat flow in a moving incompressible fluid.

Theorem 7.1.15. Maximum principle for scalar 2nd-order convection diffusion equations → [2, 6.4.1, Thm. I]

Let $\mathbf{v} : \Omega \mapsto \mathbb{R}^d$ be a continuously differentiable vector field. Then there holds the maximum principle

$$\begin{aligned} -\Delta u + \mathbf{v} \cdot \mathbf{grad}\, u \geq 0 \quad &\Longrightarrow \quad \min_{x \in \partial\Omega} u(x) = \min_{x \in \Omega} u(x) \,, \\ -\Delta u + \mathbf{v} \cdot \mathbf{grad}\, u \leq 0 \quad &\Longrightarrow \quad \max_{x \in \partial\Omega} u(x) = \max_{x \in \Omega} u(x) \,. \end{aligned}$$

7.1.4 Transient heat conduction

In Sect. 6.1.1 we generalized the laws of stationary heat conduction derived in Sect. 2.6 to time-dependent temperature distributions $u = u(x,t)$ sought on a space-time cylinder $\widetilde{\Omega} := \Omega \times]0,T[$. The same ideas apply to heat conduction in a fluid:

- Start from energy balance law (6.1.3) and convert it into local form (6.1.4).
- Combine it with the extended Fourier's law (7.1.5).

$$\frac{\partial}{\partial t}(\rho u) - \operatorname{div}(\kappa\, \mathbf{grad}\, u) + \operatorname{div}(\rho \mathbf{v}(x,t) u) = f(x,t) \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,. \tag{7.1.16}$$

For details and notations refer to Sect. 6.1.1.

This PDE has to be supplemented with

- boundary conditions (as in the stationary case, see Sect. 2.7),
- initial conditions (same as for pure diffusion, see Sect. 6.1.1).

Under the assumption $\operatorname{div}_x \mathbf{v}(x,t) = 0$ of incompressibilty (→ Def. 7.1.7 and Thm. 7.1.12) and in the case of constant (in space) coefficients (7.1.16) is equivalent to, *cf.* (7.1.13),

$$\boxed{\tfrac{\partial}{\partial t}(\rho u) - \kappa \Delta u + \rho \mathbf{v}(x,t) \cdot \mathbf{grad}\, u = f(x,t) \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[} \,. \tag{7.1.17}$$

Remark 7.1.18 (Conversion into non-dimensional form by scaling → Rem. 1.2.10)

Let us elaborate how to cast (7.1.17) into non-dimensional form, a procedure known as scaling. The first step consists of fixing *reference quantities*:

- reference length l_0, $[l_0] = 1\mathrm{m}$,
- reference time span t_0, $[t_0] = 1\mathrm{s}$,
- reference temperature T_0, $[T_0] = 1\mathrm{K}$,
- reference heat capacity ρ_0, $[\rho_0] = \frac{\mathrm{J}}{\mathrm{Km}^3}$

Here we choose l_0 and t_0 such that $\quad v_{\max} = \dfrac{l_0}{t_0}, \quad v_{\max} := \max\limits_{x,t} \|\mathbf{v}(x,t)\|$.

A hint on how many reference quantities are at our disposal is offered by considering the number of different basic SI units relevant for the model. Here those are $1\mathrm{K}$, $1\mathrm{m}$, $1\mathrm{s}$, $1\mathrm{J}$.

Then we introduce the dimensionless temperature

$$\widetilde{u}(\boldsymbol{\xi},\tau) := u(l_0\boldsymbol{\xi}, t_0\tau)\ , \quad \boldsymbol{\xi} \in \mathbb{R}^3,\ \tau \in \mathbb{R}\ .$$

By the chain rule we obtain

$$\begin{aligned}
\mathbf{grad}_{\xi}\, \widetilde{u} &= \frac{l_0}{u_0}\, \mathbf{grad}_x\, u(x,t)\ , \\
\Delta_{\xi}\widetilde{u} &= \frac{l_0^2}{u_0}\Delta_x u(x,t)\ , \\
\frac{\partial}{\partial\tau}\widetilde{u} &= \frac{t_0}{u_0}\frac{\partial}{\partial t}u(x,t)\ .
\end{aligned}$$

These expressions can be inserted into (7.1.17). In the case of constant coefficients ρ, κ, and $\rho_0 := \rho$, after division by ρ_0 and u_0 we arrive at

$$\frac{\partial}{\partial\tau}\widetilde{u} - \frac{t_0\kappa}{l_0^2\rho_0}\Delta_{\xi}\widetilde{u} + \frac{\mathbf{v}}{v_{\max}}\,\mathbf{grad}_{\xi}\,\widetilde{u} = \frac{t_0}{l_0\rho_0}f\ .$$

Check that $\frac{t_0\kappa}{l_0^2\rho_0}$ and $\frac{t_0}{l_0\rho_0}f$ really are dimensionless!

7.2 Stationary convection-diffusion problems

Model problem, *cf.* (7.1.14), modelling stationary heat flow in an incompressible fluid with prescribed temperature at "walls of the container" ($\leftrightarrow$ Dirichlet boundary conditions).

$$-\kappa\Delta u + \rho\mathbf{v}(x)\cdot\mathbf{grad}\, u = f \quad \text{in } \Omega\ , \quad u = 0 \quad \text{on } \partial\Omega\ .$$

Perform scaling $\hat{=}$ choice of physical units: makes equation non-dimensional by fixing "reference length", "reference time interval". "reference temperature", "reference power", see Rem. 7.1.18.

Remark 7.2.1 (Scaling of convection-diffusion equation)

As elaborated in Remark 7.1.18, page 512, scaling produces the following boundary value problem for non-dimensional quantities, where $\|\mathbf{v}\|_{L^\infty(\Omega)} \leq 1$, and $\epsilon := \frac{t_0\kappa}{l_0^2\rho_0}$, see Remark 7.1.18 for the choice of reference quantities t_0, l_0, ρ_0.

$$-\epsilon\Delta u + \mathbf{v}(\boldsymbol{x})\cdot\mathbf{grad}\,u = f \quad \text{in } \Omega\,, \quad u = 0 \quad \text{on } \partial\Omega\,, \tag{7.2.2}$$

diffusive term (2nd-order term)

convective term (1st-order term)

with $\epsilon > 0$, $\|\mathbf{v}\|_{L^\infty(\Omega)} = 1$, $\boxed{\operatorname{div}\mathbf{v} = 0 \quad \to \text{incompressible fluid, see Def. 7.1.7}}$.

Remark 7.2.3 (Variational formulation for convection-diffusion BVP)

Standard "4-step approach" of Sect. 2.9 can be directly applied to BVP (7.2.2) with one new twist:

Do not use integration by parts (Green's formula, Thm. 2.5.9) on convective terms!

▶ variational formulation for BVP (7.2.2):

$$u \in H_0^1(\Omega): \quad \underbrace{\epsilon\int_\Omega \mathbf{grad}\,u\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x} + \int_\Omega (\mathbf{v}\cdot\mathbf{grad}\,u)\,v\,\mathrm{d}\boldsymbol{x}}_{\text{bilinear form } \mathsf{a}(u,v)} = \underbrace{\int_\Omega f(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}}_{\text{linear form } \ell(v)} \quad \forall v \in H_0^1(\Omega)\,.$$

$\hat{=}$ a linear variational problem, see Sect. 2.4.1.

Obvious: a is **not** symmetric, see (2.2.29).

➥ a does not induce an energy norm ($\to$ Def. 2.2.43)

As replacement for the energy norm use $H^1(\Omega)$-(semi)norm ($\to$ Def. 2.3.23)

In this case we have to make sure that a fits the chosen norm in the sense that

$$\exists C > 0: \quad |\mathsf{a}(u,v)| \le C|u|_{H^1(\Omega)}|v|_{H^1(\Omega)} \quad \forall u,v \in H_0^1(\Omega)\,. \tag{7.2.4}$$

$\longleftrightarrow$ Terminology: (7.2.4) $\hat{=}$ a is continuous on $H^1(\Omega)$, *cf.* (3.2.4).

By Cauchy-Schwarz inequality for integrals (2.2.44): for all $u,v \in H_0^1(\Omega)$

$$|\mathsf{a}(u,v)| \le \|v\|_{L^\infty(\Omega)}|u|_{H^1(\Omega)}\|v\|_{L^2(\Omega)} \overset{\text{Thm. 2.3.31}}{\le} \operatorname{diam}(\Omega)\|v\|_{L^\infty(\Omega)}|u|_{H^1(\Omega)}|v|_{H^1(\Omega)}\,,$$

which confirms (7.2.4)

Surprise: a is *positive definite* ($\to$ Def. 2.2.40), because

$$\begin{aligned}\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\,u)\,u\,\mathrm{d}\boldsymbol{x} &= \int_\Omega (\mathbf{v}\,u)\cdot\mathbf{grad}\,u\,\mathrm{d}\boldsymbol{x}\\ &\overset{\text{Green's formula}}{=} -\int_\Omega \operatorname{div}(\mathbf{v}\,u)\,u\,\mathrm{d}\boldsymbol{x} + \int_{\partial\Omega}\underbrace{u^2}_{=0}\,\mathbf{v}\cdot\boldsymbol{n}\,\mathrm{d}S\\ &\overset{(2.5.5)\ \&\ \operatorname{div}\mathbf{v}=0}{=} -\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\,u)\,u\,\mathrm{d}\boldsymbol{x}\,.\end{aligned}$$

$$\blacktriangleright \quad \mathsf{a}(u,u) = \epsilon \int_\Omega \|\mathbf{grad}\, u\|^2 \,\mathrm{d}\boldsymbol{x} > 0 \quad \forall u \in H_0^1(\Omega) \setminus \{0\} \,. \tag{7.2.5}$$

From this and (7.2.4) we conclude existence and uniqueness of solutions of the BVP (7.2.2) in the Sobolev space $H_0^1(\Omega)$.

7.2.1 Singular perturbation

Setting: fast-moving fluid $\leftrightarrow$ convection dominates diffusion $\leftrightarrow$ $\epsilon \ll 1$ in (7.2.2).

Example 7.2.6 (1D convection-diffusion boundary value problem)

$$-\epsilon \frac{d^2 u_\epsilon}{dx^2} + \frac{du_\epsilon}{dx} = 1 \quad \text{in } \Omega \,, \tag{7.2.7}$$

$$u_\epsilon(0) = 0 \quad , \quad u_\epsilon(1) = 0 \,,$$

$$\blacktriangleright \quad u_\epsilon(x) = x + \frac{\exp(-x/\epsilon) - 1}{1 - \exp(-1/\epsilon)} \,. \tag{7.2.8}$$

For $\epsilon \ll 1$:

boundary layer at $x = 1$

Pointwise limit:

$$\lim_{\epsilon \to 0} u_\epsilon(x) \to x \quad \forall 0 < x < 1 \,.$$

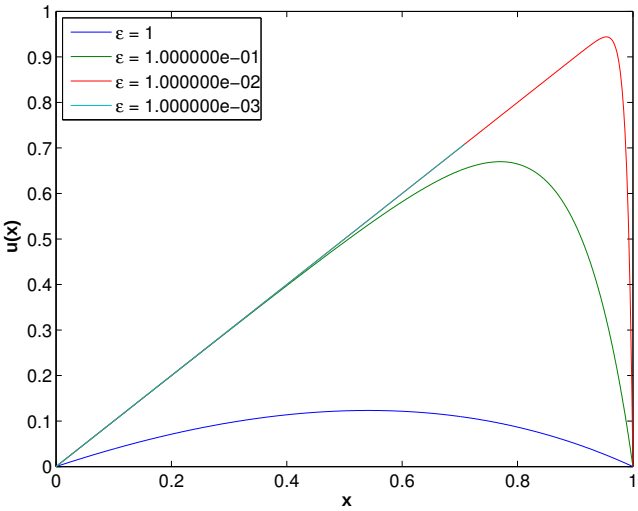


Fig. 299

Obviously, the pointwise limit $u_0(x) = x$ for $\epsilon \to 0$ solves the differential equation $\frac{du_0}{dx} = 1$ ("limit equation"), which can be obtained from (7.2.7) by simply setting $\epsilon := 0$.

"Limit problem" for (7.2.2): ignore diffusion $\rightarrow$ set $\epsilon = 0$

$$(7.2.2) \quad \overset{\epsilon=0}{\blacktriangleright} \quad \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u = f(\boldsymbol{x}) \quad \text{in } \Omega \,. \tag{7.2.9}$$

Case $d = 1$ ($\Omega =]0,1[$, $v = \pm 1$)

$$(7.2.9) \quad \overset{d=1}{\blacktriangleright} \quad \pm \frac{du}{dx}(x) = f(x) \quad \Rightarrow \quad u(x) = \int f \,\mathrm{d}x + C \,. \tag{7.2.10}$$

What about this constant C?

If $v = 1 \leftrightarrow$ fluid flows "from left to right", so we should integrate the source from 0 to x:

$$u(x) = u(0) + \int_0^x f(s)\,\mathrm{d}s = \int_0^x f(s)\,\mathrm{d}s \,, \tag{7.2.11}$$

because $u(0) = 0$ by the boundary condition $u = 0$ on $\partial\Omega$. If $v = -1$ we start the integration at $x = 1$. Note that this makes the maximum principle of Thm. 7.1.15 hold.

For $d > 1$ we can solve (7.2.9) by the method of characteristics:

To motivate it, be aware that (7.2.9) describes pure transport of a temperature distribution in the velocity field $\mathbf{v}$, that is, the heat/temperature is just carried along particle trajectories and changes only under the influence of heat sources/sinks along that trajectory.

Denote by u the solution of (7.2.9) and recall the differential equation (7.1.1) for a particle trajectory

$$\frac{d\mathbf{y}}{dt}(t) = \mathbf{v}(\mathbf{y}(t)) \quad , \quad \mathbf{y}(0) = \boldsymbol{x}_0 \; . \tag{7.1.1}$$

$$\blacktriangleright \quad \frac{d}{dt}u(\mathbf{y}(t)) = \mathbf{grad}\, u(\mathbf{y}(t)) \cdot \frac{d}{dt}\mathbf{y}(t) = \mathbf{grad}\, u \cdot \mathbf{v}(\mathbf{y}(t)) \overset{(7.2.9)}{=} f(\mathbf{y}(t)) \; .$$

➢ Compute $\mathbf{u}(\mathbf{y}(t))$ by integrating source f along particle trajectory!

$$u(\mathbf{y}(t)) = u(\boldsymbol{x}_0) + \int_0^t f(\mathbf{y}(s))\,\mathrm{d}s \tag{7.2.12}$$

Taking the cue from $d = 1$ we choose $\boldsymbol{x}_0$ as "the point on the boundary where the particle enters Ω". These points form the part of the boundary through which the flow enters Ω, the inflow boundary

$$\Gamma_{\mathrm{in}} := \{\boldsymbol{x} \in \partial\Omega \colon \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\} \; . \tag{7.2.13}$$

Its complement in $\partial\Omega$ contains the outflow boundary

$$\Gamma_{\mathrm{out}} := \{\boldsymbol{x} \in \partial\Omega \colon \mathbf{v}(\boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) > 0\} \; . \tag{7.2.14}$$

Remark 7.2.15 (Streamlines)

→ velocity field

—: Streamline connecting Γ_{in} and Γ_{out}

—: Closed streamline (recirculating flow)

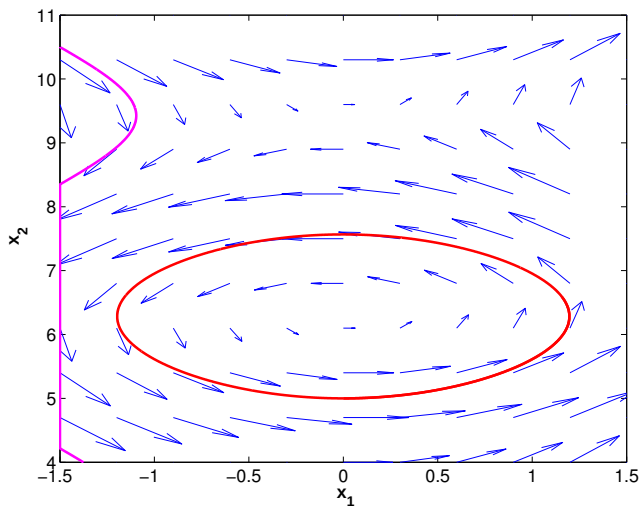


Fig. 300

In the case of closed streamlines the stationary pure transport problem fails to have a unique solution: on a closed streamline u can attain "any" value, because there is no boundary value to fix u.

Return to case $d = 1$. In general solution $u(x)$ from (7.2.10) will **not** satisfy the boundary condition $u(1) = 0$**!** Also for $u(\boldsymbol{x})$ from (7.2.12) the homogeneos boundary conditions may be violated where the particle trajectory leaves Ω**!**

In the limit case $\epsilon = 0$ not all boundary conditions of (7.2.2) can be satisfied.

Notion 7.2.16. Singularly perturbed problem

A boundary value problem depending on parameter $\epsilon \approx \epsilon_0$ is called singularly perturbed, if the limit problem for $\epsilon \to \epsilon_0$ is not compatible with the boundary conditions.

Especially in the case of 2nd-order elliptic boundary value problems:

Singular perturbation = 1st-order terms become dominant for $\epsilon \to \epsilon_0$

In mathematical terms, singular perturbation for boundary values for PDEs is defined as a *change of type* of the PDE for $\epsilon = 0$: in the case of (7.2.2) the type changes from elliptic to hyperbolic, see Rem. 2.1.2.

7.2.2 Upwinding

Focus: linear finite element Galerkin discretization for 1D model problem, *cf.* Ex. 7.2.6

$$-\epsilon \frac{d^2u}{dx^2} + \frac{du}{dx} = f(x) \quad \text{in } \Omega\,, \quad u(0) = 0 \quad , \quad u(1) = 0\,. \tag{7.2.17}$$

Variational formulation, see Rem. 7.2.3:

$$u \in H_0^1(]0,1[): \quad \underbrace{\epsilon \int_0^1 \frac{du}{dx}(x)\,\frac{dv}{dx}(x)\,\mathrm{d}x + \int_0^1 \frac{du}{dx}(x)\,v(x)\,\mathrm{d}x}_{=:\mathsf{a}(u,v)} = \underbrace{\int_0^1 f(x)v(x)\,\mathrm{d}x}_{=:\ell(v)} \quad \forall v \in H_0^1(]0,1[)\,.$$

As in Sect. 1.5.2.2: use equidistant mesh $\mathcal{M}$ (mesh width $h > 0$), composite trapezoidal rule (1.5.80) for right hand side linear form, standard "tent function basis", see (1.5.70).

▶ linear system of equations for coefficients μ_i, $i = 1,\dots,M-1$, providing approximations for point values $u(ih)$ of exact solution u.

$$\left(-\frac{\epsilon}{h} - \frac{1}{2}\right)\mu_{i-1} + \frac{2\epsilon}{h}\mu_i + \left(-\frac{\epsilon}{h} + \frac{1}{2}\right)\mu_{i+1} = hf(ih)\,, \quad i = 1,\dots,M-1\,, \tag{7.2.18}$$

where the homogeneous Dirichlet boundary conditions are taken into account by setting $\mu_0 = \mu_M = 0$.

Remark 7.2.19 (Finite differences for convection-diffusion equation in 1D)

As in Sect. 1.5.4 on the finite difference in 1D, we can also obtain (7.2.18) by replacing the derivatives by suitable difference quotients:

$$\begin{array}{ccccc} -\epsilon\dfrac{d^2u}{dx^2} & + & \dfrac{du}{dx} & = & f(x) \\ \updownarrow & & \updownarrow & & \updownarrow \\ \epsilon\underbrace{\dfrac{-\mu_{i+1}+2\mu_i-\mu_{i-1}}{h^2}}_{\text{difference quotient for } \frac{d^2u}{dx^2}} & + & \underbrace{\dfrac{\mu_{i+1}-\mu_{i-1}}{2h}}_{\text{symmetric d.q. for } \frac{du}{dx}} & = & f(ih)\,. \end{array} \tag{7.2.18}$$

Example 7.2.20 (Linear FE discretization of 1D convection-diffusion problem)

- Model boundary value problem (7.2.17)
- linear finite element Galerkin discretization as described above
- As in Ex. 7.2.6: $f \equiv 1$

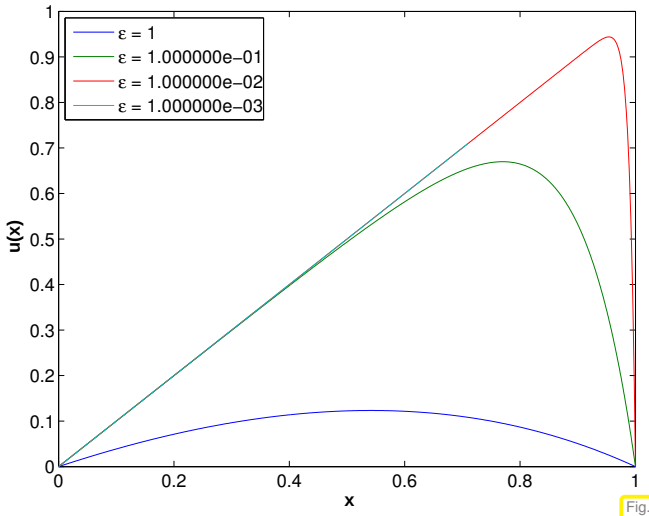


Fig. 301

exact solutions

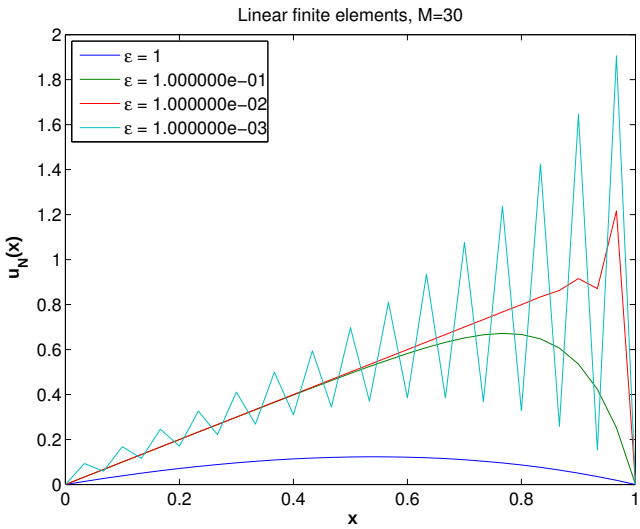


Fig. 302

FE solutions

For very small ϵ: spurious *oscillations* of linear FE Galerkin solution.

In order to understand this observation, study the linear finite element Galerkin discretization in the limit case $\epsilon = 0$

$$(7.2.18) \xrightarrow{\epsilon=0} \mu_{i+1} - \mu_{i-1} = 2hf(ih)\,, \quad i = 1,\dots,M\,. \tag{7.2.21}$$

(7.2.21) $\hat{=}$ Linear system of equations with, for even M, *singular* system matrix!

Explanation: the difference equations (7.2.21) do not couple grid nodes with even and odd indices. Hence, for even M, an arbitrary constant can be added to μ_i, i odd, whereas the linear systems for the μ_j, j even, is overdetermined. The "even-odd decoupling" inherent in (7.2.21) causes the glaring spurious oscillations in the numerical solutions in Ex. 7.2.20 for very small ϵ.

For $\epsilon > 0$ the Galerkin matrix will always be regular due to (7.2.5), but the linear relationship (7.2.21) will become more and more dominant as $\epsilon > 0$ becomes smaller and smaller. In particular, (7.2.21) sends the message that values at even and odd numbered nodes will become decoupled, which accounts for the oscillations.

Desired: robust discretization of (7.2.17)

= discretization that produces qualitatively correct $(*)$ solutions for **any** $\epsilon > 0$

$(*)$: "qualitatively correct", e.g., satisfaction of maximum principle, Thm. 7.1.15]

Guideline:

Numerical methods for singularly perturbed problems must "work" for the limit problem

What is a meaningful scheme for limit problem $u' = f$ on an equidistant mesh of $\Omega :=]0,1[$?

$$\text{Explicit Euler method:} \quad \mu_{i+1} - \mu_i = hf(\xi_i) \quad i = 0,\dots,N\,,$$
$$\text{Implicit Euler method:} \quad \mu_{i+1} - \mu_i = hf(\xi_{i+1}) \quad i = 0,\dots,N\,.$$

Both Euler methods can be regarded as finite difference discretizations of $u' = f$ based on one-sided difference quotients:

$$\text{Explicit Euler:} \quad \frac{du}{dx}(x_i) \approx \frac{u(x_{i+1}) - u(x_i)}{h} \quad , \quad \text{Implicit Euler:} \quad \frac{du}{dx}(x_i) \approx \frac{u(x_i) - u(x_{i-1})}{h}\,.$$

Conversely, (7.2.21) can be obtained by relying on a symmetric difference quotient:

$$\text{(7.2.21):} \quad \frac{du}{dx}(x_i) \approx \frac{u(x_{i+1}) - u(x_{i-1})}{2h}\,.$$

Apparently, the use of a symmetric difference quotient for discretizing the convective term incurs spurious oscillations, see Ex. 7.2.20.

Conclusion: use one-sided difference quotients for discretization of convective term!

Which type **?** (Explicit or implicit Euler ?)

Linear system arising from *use of backward difference quotient* $\frac{du}{dx}_{|x=x_i} = \frac{\mu_i - \mu_{i-1}}{h}$:

$$\left(-\frac{\epsilon}{h} - 1\right)\mu_{i-1} + \left(\frac{2\epsilon}{h} + 1\right)\mu_i + -\frac{\epsilon}{h}\mu_{i+1} = hf(ih)\,, \quad i = 1,\dots,M-1\,, \tag{7.2.22}$$

Linear system arising from *use of forward difference quotient* $\frac{du}{dx}_{|x=x_i} = \frac{\mu_{i+1} - \mu_i}{h}$:

$$-\frac{\epsilon}{h}\mu_{i-1} + \left(\frac{2\epsilon}{h} - 1\right)\mu_i + \left(-\frac{\epsilon}{h} + 1\right)\mu_{i+1} = hf(ih)\,, \quad i = 1,\dots,M-1\,, \tag{7.2.23}$$

Example 7.2.24 (One-sided difference approximation of convective terms)

Model problem of Ex. 7.2.20, discretizations (7.2.22) and (7.2.23).

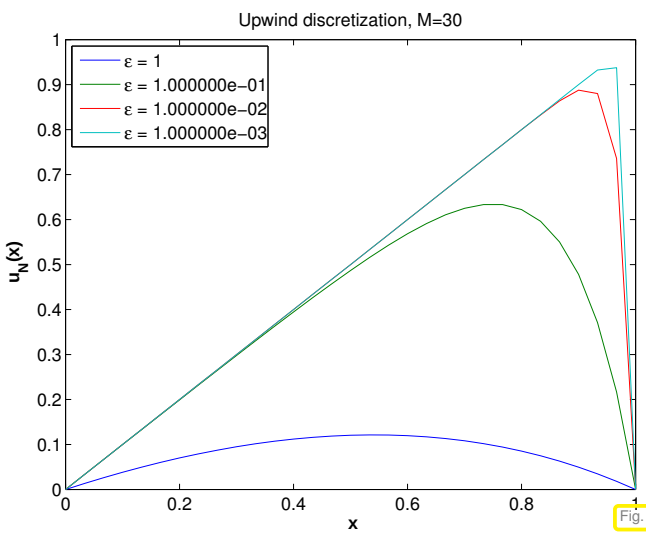


Fig. 303

backward difference quotient

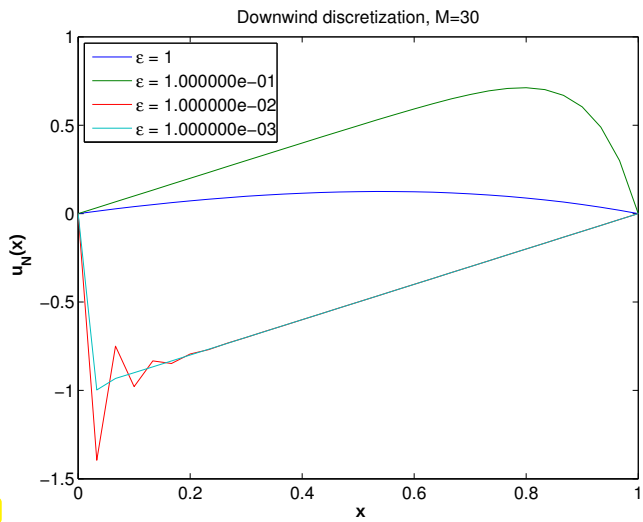


Fig. 304

forward difference quotient

Only the discretization of $\frac{du}{dx}$ based on the backward difference quotient generates qualitatively correct (piecewise linear) discrete solutions (a "good method").

If the forward difference quotient is used, the discrete solutions may violate the maximum principle of Thm. 7.1.15 (a "bad method").

How can we tell a good method from a bad method by merely examining the system matrix?

Heuristic criterion for $\epsilon \to 0$-robust stability of nodal finite element Galerkin discretization/finite difference discretization of *singularly perturbed* scalar linear convection-diffusion BVP (7.2.2) (with Dirichlet b.c.):

(Linearly interpolated) discrete solution satisfies maximum principle (5.7.4).

$$\Updownarrow$$

System matrix complies with sign-conditions (5.7.10)–(5.7.12).

Nodal finite element Galerkin discretization $\hat{=}$ basis expansion coefficients μ_i of Galerkin solution $u_N \in V_N$ double as point values of u_N at interpolation nodes. This is satisfied for Lagrangian finite element methods ($\to$ Sect. 3.5) when standard nodal basis functions according to (3.5.4) are used.

Recall the sign-conditions (5.7.10)–(5.7.12) for the system matrix $\mathbf{A}$ arising from nodal finite element Galerkin discretization or finite difference discretization:

- (5.7.10): positive diagonal entries, $\boxed{(\mathbf{A})_{ii} > 0}$,
- (5.7.11): non-positive off-diagonal entries, $\boxed{(\mathbf{A})_{ij} \leq 0, \text{ if } i \neq j}$,
- "(5.7.12)": diagonal dominance, $\boxed{\sum_j (\mathbf{A})_{ij} \geq 0}$.

These conditions are met for *equidistant meshes in 1D*

- for the standard $\mathcal{S}_1^0(\mathcal{M})$-Galerkin discretization (7.2.18), **provided that** $|\epsilon h^{-1}| \geq \frac{1}{2}$,
- when using *backward* difference quotients for the convective term (7.2.22) for **any** choice of $\epsilon \geq 0$, $h > 0$,
- when using *forward* difference quotients for the convective term (7.2.23), **provided that** $|\epsilon h^{-1}| \geq 1$.

Only the use of a *backward* difference quotient for the convective term guarantees the (discrete) maximum principle in an $\epsilon \to 0$-robust fashion!

Terminology: Approximation of $\frac{du}{dx}$ by *backward* difference quotients $\hat{=}$ upwinding

Example 7.2.25 (Spurious Galerkin solution for 2D convection-diffusion BVP)

- Triangle domain $\Omega = \{(x,y) : 0 \leq x \leq 1, -x \leq y \leq x\}$.
- Velocity $\mathbf{v}(\boldsymbol{x}) = \binom{1}{0}$ ➢ (7.2.2) becomes $-\epsilon\Delta u + u_x = 1$.
- Exact solution: $u_\epsilon(x_1,x_2) = x - \frac{1}{1-e^{-1/\epsilon}}(e^{-(1-x_1)/\epsilon} - e^{-1/\epsilon})$, Dirichlet boundary conditions set accordingly

✦ Standard Galerkin discretization by means of linear finite elements on sequence of triangular mesh created by regular refinement.

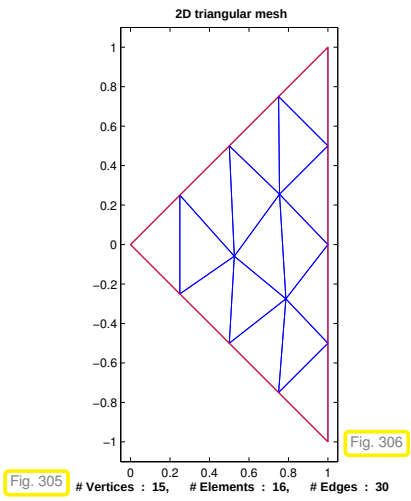
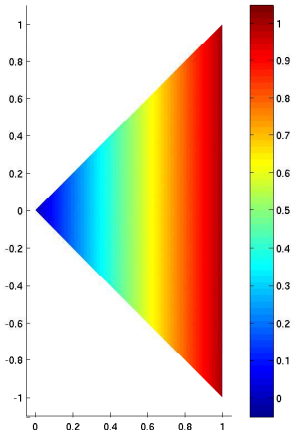


Fig. 305 Coarse initial mesh

Fig. 306



Exact solution ($\epsilon = 10^{-10}$)

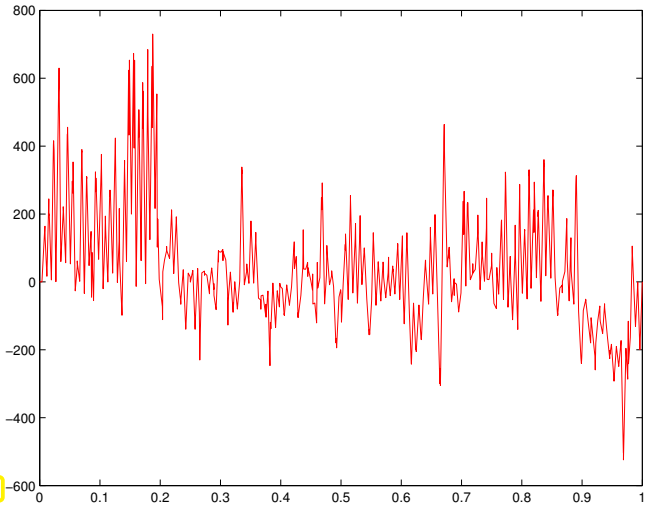


Fig. 307 Standard Galerkin solution on $x_2 = 0$-line

As expected: spurious oscillations mar Galerkin solution

➢ Difficulty observed in 1D also haunts discretization in higher dimensions.

Issue: extension of upwinding idea to $d > 1$

7.2.2.1 Upwind quadrature

Revisit 1D model problem

$$-\epsilon \frac{d^2u}{dx^2} + \frac{du}{dx} = f(x) \quad \text{in } \Omega\,, \quad u(0) = 0 \quad , \quad u(1) = 0\,, \tag{7.2.17}$$

with variational formulation, see Rem. 7.2.3: convective term

$$u \in H_0^1(]0,1[): \quad \underbrace{\epsilon \int_0^1 \frac{du}{dx}(x)\,\frac{dw}{dx}(x)\,\mathrm{d}x + \int_0^1 \frac{du}{dx}(x)\,w(x)\,\mathrm{d}x}_{=:\mathsf{a}(u,w)} = \underbrace{\int_0^1 f(x)w(x)\,\mathrm{d}x}_{=:\ell(w)} \quad \forall w \in H_0^1(]0,1[)\,.$$

Linear finite element Galerkin discretization on equidistant mesh $\mathcal{M}$ with M cells, meshwidth $h = \frac{1}{M}$, *cf.* Sect. 1.5.2.2.

We opt for the *global* composite trapezoidal rule

$$\int_0^1 \psi(x)\,\mathrm{d}x \approx h \sum_{j=1}^{M-1} \psi(jh)\,, \quad \text{for} \quad \psi \in C^0(]0,1[), \psi(0) = \psi(1) = 0\,,$$

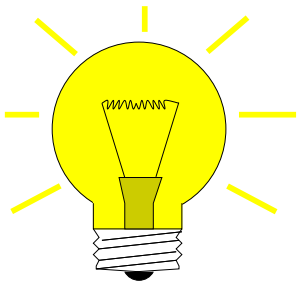
for evaluation of convective term in bilinear form a:

$$\int_0^1 \frac{du_N}{dx}(x) w_N(x)\,\mathrm{d}x \approx h \sum_{j=1}^{M-1} \frac{du_N}{dx}(jh)\,w_N(hj)\,, \quad w_N \in \mathcal{S}_{1,0}^0(\mathcal{M})\,. \tag{7.2.26}$$

Note: this is not a valid formula, because $\frac{du_N}{dx}(jh)$ is *ambiguous*, since $\frac{du_N}{dx}$ is discontinuous at nodes of the mesh for $u_N \in \mathcal{S}^0_{1,0}(\mathcal{M})$!

Up to now we resolved this ambiguity by the policy of *local* quadrature, see Sect. 3.6.5: quadrature rule applied locally on each cell with all information taken from that cell.

However: Convection transports information in the direction of $\mathbf{v}$!



Idea:

Use upstream/upwind information to evaluate $\frac{du_N}{dx}(jh)$ in (7.2.26)

$$\frac{du_N}{dx}(jh) := \lim_{\delta\to 0}\frac{du_N}{dx}(jh-\delta) = \frac{du_N}{dx}_{|]x_{j-1},x_j[}\,.$$

$\hat{=}$ upwind quadrature

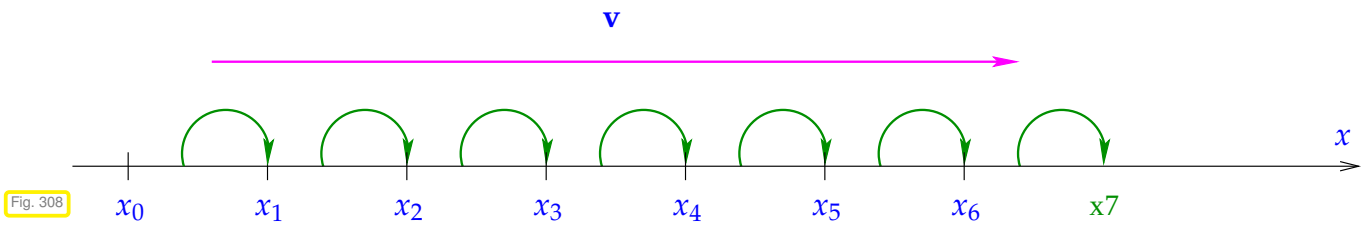


Fig. 308

Upwind quadrature yields the following contribution of the discretized convective term to the linear system using the basis expansion $u_N = \sum_{l=1}^{M-1}\mu_l b_N^l$ into *locally supported* nodal basis functions ("tent functions")

$$\int_0^1 \sum_{l=1}^{M-1}\mu_l \frac{db_N^l}{dx}(x)b_N^i(x)\,\mathrm{d}x \overset{(7.2.26)}{\approx} h\,\frac{\mu_i-\mu_{i-1}}{h}\,,$$

where we used

- $b_N^i(jh) = \delta_{ij}$, see (1.5.71),
- $\frac{du_N}{dx}_{|]x_{j-1},x_j[} = \frac{\mu_i-\mu_{i-1}}{h}$ from (1.5.72).

► Linear system from upwind quadrature:

$$\left(-\frac{\epsilon}{h}-1\right)\mu_{i-1} + \left(\frac{2\epsilon}{h}+1\right)\mu_i + -\frac{\epsilon}{h}\mu_{i+1} = hf(ih)\,,\quad i=1,\dots,M-1\,, \tag{7.2.22}$$

which is the **same** as that obtained from a backward finite difference discretization of $\frac{du}{dx}$!

The idea of upwind quadrature can be generalized to $d>1$: we consider $d=2$ and linear Lagrangian finite element Galerkin discretization on triangular meshes, see Sect. 3.3.

❶ Approximation of contribution of convective terms to bilinear form by means of *global trapezoidal rule*:

$$\int_\Omega (\mathbf{v}\cdot\mathbf{grad}\,u_N)v_N\,\mathrm{d}\boldsymbol{x} \approx \sum_{\boldsymbol{p}\in\mathcal{V}(\mathcal{M})}\left(\tfrac{1}{3}\sum_{K\in\mathcal{U}_{\boldsymbol{p}}}|K|\right)\cdot\mathbf{v}(\boldsymbol{p})\cdot\mathbf{grad}\,u_N(\boldsymbol{p})\,v_N(\boldsymbol{p})\,. \tag{7.2.27}$$

ambiguous for $u \in \mathcal{S}^0_1(\mathcal{M})$!

✎ notation: $\mathcal{U}_{\boldsymbol{p}} := \{K\in\mathcal{M}\colon \boldsymbol{p}\in\overline{K}\}$, $\boldsymbol{p}\in\mathcal{V}(\mathcal{M})$

For a continuous function $\psi : \Omega \mapsto \mathbb{R}$ the trapezoidal rule can easily be derived from the 2D *composite* trapezoidal rule based on

$$\int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \frac{|K|}{3}(\psi(\boldsymbol{a}^1) + \psi(\boldsymbol{a}^2) + \psi(\boldsymbol{a}^3))\,, \tag{3.3.49}$$

where the $\boldsymbol{a}^i$, $i = 1,2,3$, are the vertices of the triangle K.

$$\begin{aligned} \int_\Omega \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \sum_{K\in\mathcal{M}} \int_K \psi(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \approx \sum_{K\in\mathcal{M}} \frac{|K|}{3}(\psi(\boldsymbol{a}_K^1) + \psi(\boldsymbol{a}_K^2) + \psi(\boldsymbol{a}_K^3)) \\ &\approx \sum_{\boldsymbol{p}\in\mathcal{V}(\mathcal{M})} \left(\tfrac{1}{3}\sum_{K\in\mathcal{U}_{\boldsymbol{p}}} |K|\right) \psi(\boldsymbol{p})\,, \end{aligned} \tag{7.2.28}$$

by changing the order of summation. This formula is the *global* trapezoidal rule in 2D on a triangular mesh.

❷ Fix the ambiguous value of $\mathbf{v}(\boldsymbol{p}) \cdot \mathbf{grad}\, u_N(\boldsymbol{p})$, $u_N \in \mathcal{S}_1^0(\mathcal{M})$, by taking the gradient from the triangle upstream to the node $\boldsymbol{p}$:

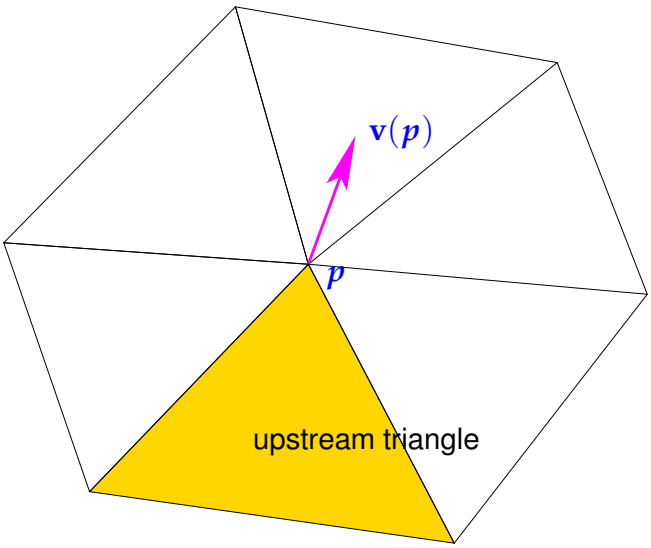
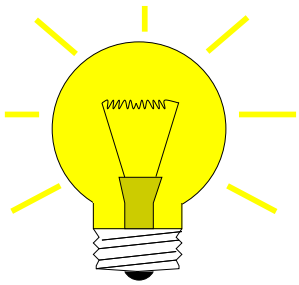


Fig. 309



Idea: Use upstream/upwind information to evaluate $\mathbf{grad}\, u_N(\boldsymbol{p})$ in (7.2.27)

$$\mathbf{v}(\boldsymbol{p}) \cdot \mathbf{grad}\, u_N(\boldsymbol{p}) := \lim_{\delta\to 0} \mathbf{v}(\boldsymbol{p}) \cdot \mathbf{grad}\, u_N(\boldsymbol{p} - \delta\mathbf{v}(\boldsymbol{p}))\,. \tag{7.2.29}$$

$\hat{=}$ general upwind quadrature

Note: By (7.1.1) the vector $\mathbf{v}(\boldsymbol{p})$ supplies the direction of the streamline through $\boldsymbol{p}$. Hence, $-\mathbf{v}(\boldsymbol{p})$ is the direction from which information is "carried into $\boldsymbol{p}$" by the flow.

Contribution of convective term to the i-th row of the final linear system of equations (test function = tent function b_N^i)

$$\underbrace{\left(\frac{1}{3}\sum_{K\in\mathcal{U}_i}|K|\right)}_{=:U_i}\mathbf{v}(\boldsymbol{x}^i)\cdot\mathbf{grad}\,u_{N\,|K_u}\,,$$

where K_u is the upstream triangle of $\boldsymbol{p}$. ▷

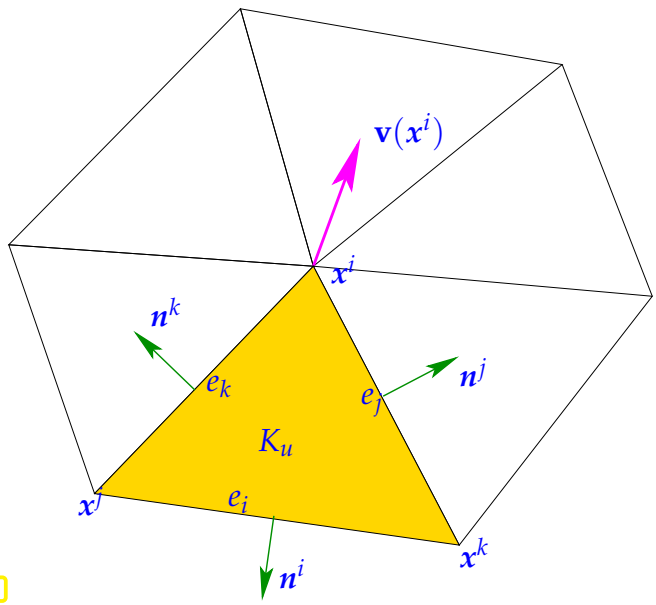


Fig. 310

Using the expressions for the gradients of barycentric coordinate functions from Sect. 3.3.5

$$\mathbf{grad}\,\lambda_* = -\frac{|e_i|}{2|K|}\boldsymbol{n}^*\,,\quad *=i,j,k\,,\quad \text{see Fig. 310}\,,$$

and the nodal basis expansion of u_N, we obtain for the convective contribution to the i-th line of the final linear system

$$\frac{U_i}{2|K_u|}\left(\underbrace{-\left\|\mathbf{x}^j-\mathbf{x}^k\right\|\boldsymbol{n}^i\cdot\mathbf{v}(\boldsymbol{x}^i)}_{\leftrightarrow\text{ diagonal entry}}\mu_i-\left\|\mathbf{x}^i-\mathbf{x}^j\right\|\boldsymbol{n}^k\cdot\mathbf{v}(\boldsymbol{x}^i)\mu_k-\left\|\mathbf{x}^i-\mathbf{x}^k\right\|\boldsymbol{n}^j\cdot\mathbf{v}(\boldsymbol{x}^i)\mu_j\right)$$

By the very definition of the upstream triangle K_u we find

$$\boldsymbol{n}^i\cdot\mathbf{v}(\boldsymbol{x}^i)\le 0\quad,\quad \boldsymbol{n}^k\cdot\mathbf{v}(\boldsymbol{x}^i)\ge 0\quad,\quad \boldsymbol{n}^j\cdot\mathbf{v}(\boldsymbol{x}^i)\ge 0\,.$$

➢ sign conditions (5.7.10), (5.7.11) are satisfied for the discretized convective term, (5.7.12) is obvious from $\lambda_i+\lambda_j+\lambda_k=1$, which means $\mathbf{grad}(\lambda_i+\lambda_j+\lambda_k)=0$.

Usually, the upwind quadrature discretization of the convective term will be combined with a standard finite element Galerkin discretization of the diffusive term. In this case the finite element solution of (7.2.2) will satisfy the maximum principle, if this is true for the discretization of the diffusive term. Criteria for this have been established in Section 5.7, see Theorem 5.7.13 and Remark 5.7.14, page 443.

Example 7.2.30 (Upwind quadrature discretization)

- $\Omega=[0,1]^2$
- $-\epsilon\Delta u+\binom{1}{1}\cdot\mathbf{grad}\,u=0$
- Dirichlet boundary conditions: $u(x,y)=1$ for $x>y$ and $u(x,y)=0$ for $x\le y$
- Limiting case ($\epsilon\to 0$): $u(x,y)=1$ for $x>y$ and $u(x,y)=0$ for $x\le y$
- layer along the diagonal from $\binom{0}{1}$ to $\binom{1}{0}$ in the limit $\epsilon\to 0$
- 2D triangular Delaunay triangulation, see Rem. 4.2.5
- linear finite element upwind quadrature discretization

Monitored: discrete solutions along diagonal from $\binom{0}{1}$ to $\binom{1}{0}$ for $\epsilon=10^{-10}$.

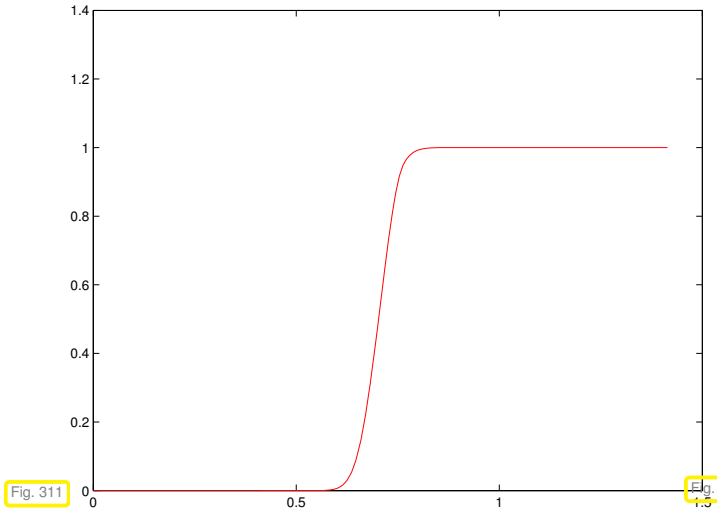


Fig. 311

upwind quadrature solution

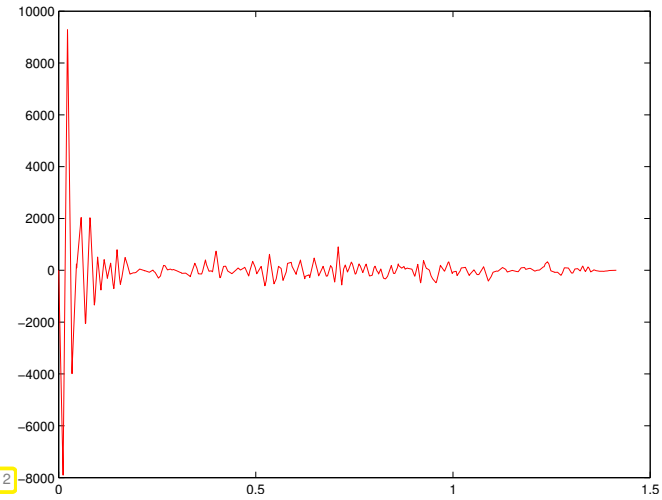


Fig. 312

standard Galerkin solution

Upwind quadrature scheme respects maximum principle, whereas the standard Galerkin solution is rendered useless by spurious oscillations.

7.2.2.2 Streamline diffusion

We take another look at the 1D upwind discretization of (7.2.17) and view it from a different perspective.

1D upwind (finite difference) discretization of (7.2.17):

$$\left(-\frac{\epsilon}{h}-1\right)\mu_{i-1}+\left(\frac{2\epsilon}{h}+1\right)\mu_i+-\frac{\epsilon}{h}\mu_{i+1}=hf(ih)\ .i=1,\dots,M-1\ . \tag{7.2.22}$$

$$\Updownarrow$$

$$(\epsilon+h/2)\underbrace{\frac{-\mu_{i-1}+2\mu_i-\mu_{i+1}}{h^2}}_{\hat{=}\text{ difference quotient for }\frac{d^2u}{dx^2}}+\underbrace{\frac{-\mu_{i-1}+\mu_{i+1}}{2h}}_{\hat{=}\text{ difference quotient for }\frac{du}{dx}}=f(ih)\ ,$$

for $i=1,\dots,M-1$.

Upwinding = h-dependent enhancement of diffusive term

artificial diffusion/viscosity

We also observe that the upwinding strategy just ads the *minimal amount of diffusion* to make the resulting system matrix comply with the conditions (5.7.10)–(5.7.12), which ensure that the discrete solution satisfies the maximum principle.

Issue: How to extend the trick of adding artificial diffusion to $d>1$?

Well, just add an extra h-dependent multiple of $-\Delta$! Let's try.

Example 7.2.31 (Effect of added diffusion)

Convection-diffusion boundary value problem ((7.2.2) with $\mathbf{v} = \binom{1}{0}$)

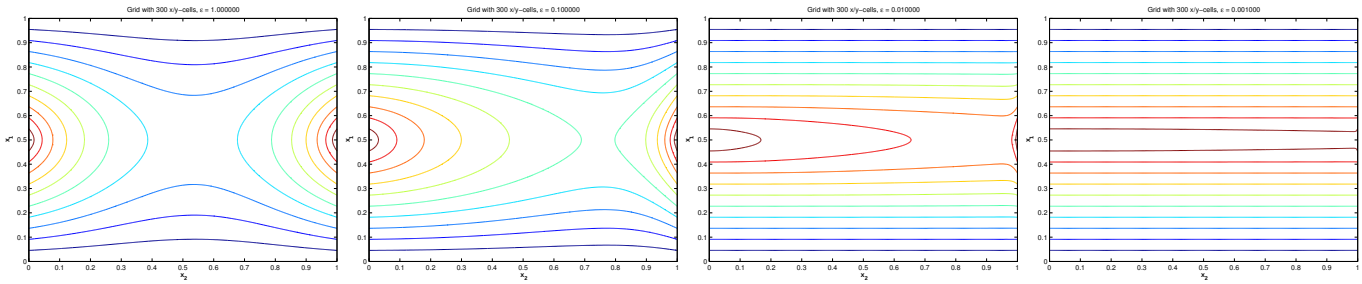
$$-\epsilon\Delta u + \frac{\partial u}{\partial x_1} = 0 \quad \text{in } \Omega =]0,1[^2 \,, \quad u = g \quad \text{on } \partial\Omega \,.$$

Here, Dirichlet data $g(\boldsymbol{x}) = 1 - 2|x_2 - \frac{1}{2}|$.

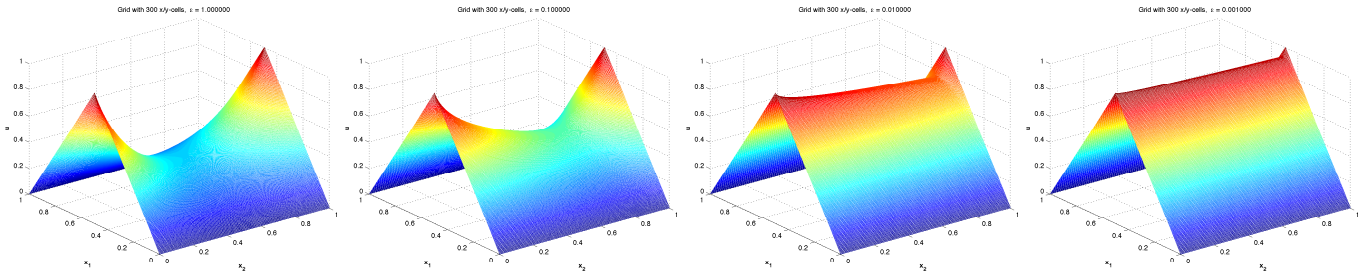
Thus, for $\epsilon \approx 0$ we expect $u \approx g$, because the Dirichlet data are just transported in x_1-direction and there are no boundary layers.

MATLAB Code 7.2.32: Upwind finite difference solution of 2D convection-diffusion problem

```
function diffeffect(epsilon)
% MATLAB function for solving simplex convection diffusion problem from
% Example 7.2.31 by means of upwind finite differences
v = 1;      % constant velocity
if (nargin < 1), epsilon = 0.1; end % strength of diffusion
M = 300; h = 1/M; % Number of mesh cells in one direction
n = (M-1)*(M-1);
g = @(x) (1-2*abs(x(2)-0.5)); % Dirichlet data

% Assemble finite difference matrix using MATLAB's kron command.  Since
% the velocity is aligned with the x1-axis, the finite difference
% equations agree with (7.2.22) in x1-direction, and boil
% down to the simple second difference quotient (1.5.138) in
% x2-direction.  Lexikographic ordering of unknowns is assumed.
I = speye(M-1,M-1);
A =
   kron(I,gallery('tridiag',M-1,-epsilon-h*v,4*epsilon+h*v,-epsilon))
   + ...
    epsilon*kron(gallery('tridiag',M-1,-1,0,-1),I);
% Boundary conditions enter through right hand side
f = zeros(n,1);
lowbd = zeros(1,M-1); upbd = zeros(1,M-1);
left = zeros(1,M-1); right = zeros(1,M-1);
for j=1:M-1
  x = [h*j;0]; lowbd(j) = g(x); f(j) = f(j) + lowbd(j);
  x = [h*j;1]; upbd(j) = g(x); f((M-2)*(M-1)+j) =
     f((M-2)*(M-1)+j) + upbd(j);
  x = [0;h*j]; left(j) = g(x); f((M-1)*(j-1)+1) =
     f((M-1)*(j-1)+1) + (epsilon+h*v)*left(j);
  x = [1;h*j]; right(j) = g(x); f((M-1)*j) = f((M-1)*j) +
     epsilon*right(j);
end

% Finally, solve linear system.
u = A\f;
```



$\epsilon = 1$ $\epsilon = 0.1$ $\epsilon = 0.01$ $\epsilon = 0.001$



$\epsilon = 1$ $\epsilon = 0.1$ $\epsilon = 0.01$ $\epsilon = 0.001$

Stronger diffusion leads to "smearing" of features that the flow field transports into the interior of the domain.

(Too much) artificial diffusion ➢ smearing of internal layers

(We are no longer solving the right problem!)

Remark 7.2.33 (Internal layers)

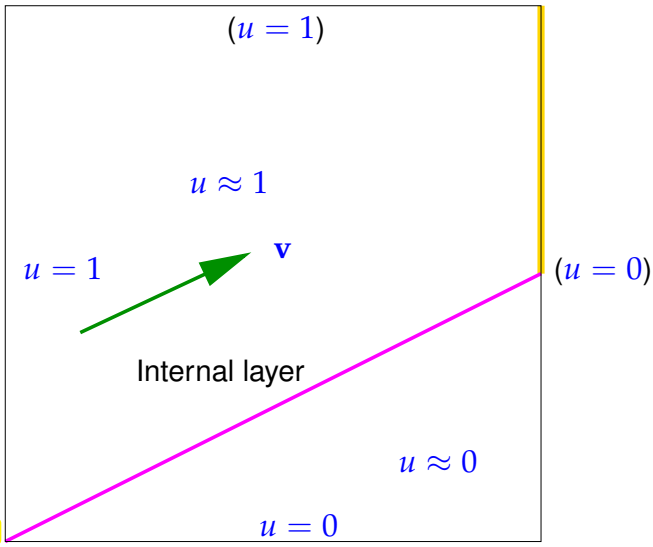


Fig. 313

Pure transport problem:

$$\mathbf{v} \cdot \mathbf{grad}\, u = 0 \quad \text{in } \Omega\,,$$

where $\Omega =]0,1[^2$, $\mathbf{v} = \binom{2}{1}$, $\epsilon = 10^{-4}$,
Dirchlet b.c. that can only partly be fulfilled on inflow boundary: $u = 1$ on $\{x_1 = 0\} \cup \{x_2 = 1\}$, $u = 0$ on $\{x_1 = 1\} \cup \{x_2 = 0\}$.
◁ Boundary conditions in brackets cannot be imposed for the limit problem.

Solution of pure transport problem with discontinuous boundary data

- displays a discontinuity across the streamline emanating from the point of discontinuity on $\partial\Omega$,
- is *smooth along streamlines*.

Qualitative solution of

$$-\delta\Delta + \mathbf{v}\cdot\mathbf{grad}\,u = 0 \quad \text{in } \Omega\,,$$

with $\delta > 0$, the same boundary data ▷

➢ Smearing of internal layer !
As in Ex. 7.2.6, we would also find a boundary layer which is marked in gold in the figure. Inside this boundary layer the solution drops to zero abruptly.

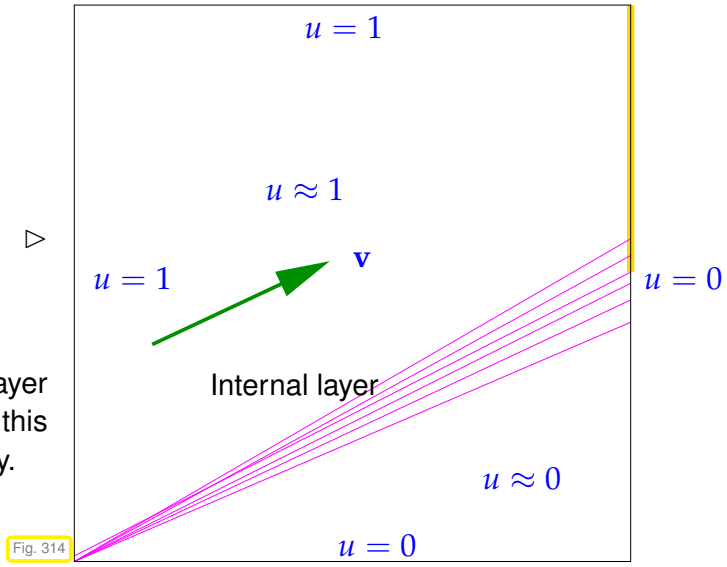


Fig. 314

Note: the above boundary conditions actually do not supply valid Dirichlet data for a second-order elliptic boundary value problem, because they jump at the corners, *cf.* Remark 2.10.6, page 173. However, they make sense for the limit problem and a finite element discretization can also be applied in this case.

Heuristics: If the solution is smooth along streamlines, then adding diffusion in the direction of streamlines cannot do much harm.

What does "diffusion in a direction" mean**?**

☞ Think of a generalized Fourier's law (2.6.5) for $d = 2$, e.g.,

$$\mathbf{j}(\boldsymbol{x}) = -\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \mathbf{grad}\,u(\boldsymbol{x})\,.$$

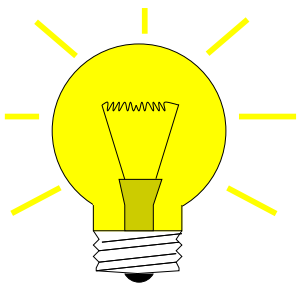
This means, only a temperature variation in x_1-direction triggers a heat flow.

➥ diffusion in a direction $\mathbf{v} \in \mathbb{R}^2$

$$\mathbf{j}(\boldsymbol{x}) = -\mathbf{v}\mathbf{v}^T\,\mathbf{grad}\,u(\boldsymbol{x}) \tag{7.2.34}$$

Such an extended Fourier's law is an example of anisotropic diffusion.

Anisotropic diffusion can simply be taken into account in variational formulations and Galerkin discretization by replacing the heat conductivity κ/stiffness σ with a symmetric, positive (semi-)definite matrix, the diffusion tensor.



Idea: Anisotropic artificial diffusion in streamline direction

On cell K replace: $\epsilon \leftarrow \underbrace{\epsilon\mathbf{I} + \delta_K \mathbf{v}_K \mathbf{v}_K^T}_{\text{new diffusion tensor}} \in \mathbb{R}^{2,2}$.

$\mathbf{v}_K \mathrel{\hat{=}}$ local velocity (e.g., obtained by averaging)
$\delta_K > 0 \mathrel{\hat{=}}$ method parameter controlling the strength of anisotropic diffusion

This idea underlies the so-called streamline diffusion method.

Thus, (for the model problem) Galerkin discretization may target the variational problem

$$\int_\Omega \left(\epsilon \mathbf{I} + \delta_K \mathbf{v}_K \mathbf{v}_K^T\right) \mathbf{grad}\, u \cdot \mathbf{grad}\, w + \mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u\, w \,\mathrm{d}\boldsymbol{x} = \int_\Omega f w \,\mathrm{d}\boldsymbol{x} \quad \forall w \in H_0^1(\Omega)\,. \tag{7.2.35}$$

This tampering affects the solution u
(solution of (7.2.35) $\neq$ solution of (7.2.2))

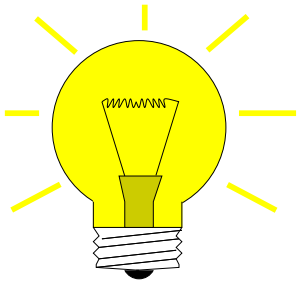
Desirable: Maintain *consistency* of variational problem!

Definition 7.2.36. Consistent modifications of variational problems

A variational problem is called a consistent modification of another, if both possess the same (unique) solution(s).

Note: the variational crimes investigated in Sect. 5.5 represent non-consistent modifications.

Ensuring consistency for streamline upwind variational problem:



Idea: Add anistropic diffusion through a residual term that vanishes for the exact solution u

streamline upwind variational problem: given mesh $\mathcal{M}$ seek $u \in H_0^1(\Omega) \cap H^2(\mathcal{M})$

$$\int_\Omega \epsilon\, \mathbf{grad}\, u \cdot \mathbf{grad}\, w + (\mathbf{v}(\boldsymbol{x}) \cdot \mathbf{grad}\, u)\, w \,\mathrm{d}\boldsymbol{x}$$
$$+ \underbrace{\sum_{K\in\mathcal{M}} \delta_K \int_K (-\epsilon\Delta u + \mathbf{v}\cdot\mathbf{grad}\, u - f)\cdot(\mathbf{v}\cdot\mathbf{grad}\, w)\,\mathrm{d}\boldsymbol{x}}_{\text{stabilization term}} = \int_\Omega f w \,\mathrm{d}\boldsymbol{x} \quad \forall w \in H_0^1(\Omega)\,. \tag{7.2.37}$$

☞ Note that enhanced smoothness of u, namely in addition $u \in H^2(K)$ for all $K \in \mathcal{M}$, is required to render (7.2.37) meaningful ($\rightarrow$ Sobolev space $H^2(\mathcal{M})$).

Note: in the case of Galerkin discretization based on $V_{N,0} = \mathcal{S}_1^0(\mathcal{M})$, we find $\Delta u_N = 0$ in each $K \in \mathcal{M}$.

For Galerkin discretization of (7.2.37) by means of linear Lagrangian finite elements, the local control parameters δ_K are usually chosen according to the rule

$$\delta_K := \begin{cases} \epsilon^{-1} h_K^2 & \text{, if } \dfrac{\|\mathbf{v}\|_{K,\infty} h_K}{2\epsilon} \leq 1\,, \\ h_K & \text{, if } \dfrac{\|\mathbf{v}\|_{K,\infty} h_K}{2\epsilon} > 1\,. \end{cases}$$

which is suggested by theoretical investigations and practical experience, *cf.* 1D artificial diffusion (7.2.22) for a reason why to choose $\delta_K \sim h_K$ for small ϵ.

Example 7.2.38 (Streamline-diffusion discretization)

Exactly the same setting as in Ex. 7.2.30 with the upwind quadrature approach replaced with the streamline diffusion method.

MATLAB Code 7.2.39: Assembling SUPG stabilization part of element matrix in LehrFEM

```
function Aloc = STIMASUPGLFE(Vertices, flag, QuadRule, VHandle,
    a,d1,d2, varargin)
% ALOC = STIMA_SUPG_LFE(VERTICES) provides the extra terms for SUPG
    stabilization to be
% added to the Galerkin element matrix for linear finite elements
%
% VERTICES is 3-by-2 matrix specifying the vertices of the current
    element
% in a row wise orientation.
%
% a:  diffusivity
% d1 d2:  apriori chosen constants for SUPG-modification
%
% Flag not used, needed for interface to assemMat_LFE
%
% QUADRULE is a struct, which specifies the Gauss qaudrature that is
    used
% to do the numerical integration:
% W Weights of the Gauss quadrature.
% X Abscissae of the Gauss quadrature.e:
%
% VHANDLE is function handle for velocity field

% Preallocate memory for element matrix
Aloc = zeros(3,3);

% Analytic computation of entries of element matrix using barycentric
% coordinates, see Sect. 3.3.5
l1x = Vertices(2,2)-Vertices(3,2);
l1y = Vertices(3,1)-Vertices(2,1);
l2x = Vertices(3,2)-Vertices(1,2);
l2y = Vertices(1,1)-Vertices(3,1);
l3x = Vertices(1,2)-Vertices(2,2);
l3y = Vertices(2,1)-Vertices(1,1);

% Compute element mapping

P1 = Vertices(1,:);
P2 = Vertices(2,:);
P3 = Vertices(3,:);

BK = [ P2 - P1 ; P3 - P1 ];          % transpose of transformation
```

```
    matrix
det_BK = abs(det(BK));                    % twice the area of the triagle

nPoints = size(QuadRule.w,1);

% Quadrature points in actual element stored as rows of a matrix
x = QuadRule.x*BK + ones(nPoints,1)*P1;

% Evaluate coefficient function (velocity) at quadrature nodes
c =VHandle(x,varargin{:});
% Entries of anisotropic diffusion tensor
FHandle=[c(:,1).*c(:,1) c(:,1).*c(:,2) c(:,2).*c(:,1)
   c(:,2).*c(:,2)];

% Compute local PecletNumber for SUPG control parameter
hK=max([norm(P2-P1),norm(P3-P1),norm(P2-P3)]);
v_infK=max(abs(c(:))); PK=v_infK*hK/(2*a);
% Apply quadrature rule and fix constant part
w = QuadRule.w; e = sum((FHandle.*[w w w w]), 1);
te = (reshape(e,2,2)')/det_BK;

% Compute Aloc values
Aloc(1,1) = (te*[l1x l1y]')'*[l1x l1y]';
Aloc(1,2) = (te*[l1x l1y]')'*[l2x l2y]';
Aloc(1,3) = (te*[l1x l1y]')'*[l3x l3y]';
Aloc(2,2) = (te*[l2x l2y]')'*[l2x l2y]';
Aloc(2,3) = (te*[l2x l2y]')'*[l3x l3y]';
Aloc(3,3) = (te*[l3x l3y]')'*[l3x l3y]';
Aloc(2,1) = (te*[l2x l2y]')'*[l1x l1y]';
Aloc(3,1) = (te*[l3x l3y]')'*[l1x l1y]';
Aloc(3,2) = (te*[l3x l3y]')'*[l2x l2y]';

if (PK<=1), Aloc=d1*hK^2/a*Aloc;
else Aloc=d2*hK*Aloc; end

return
```

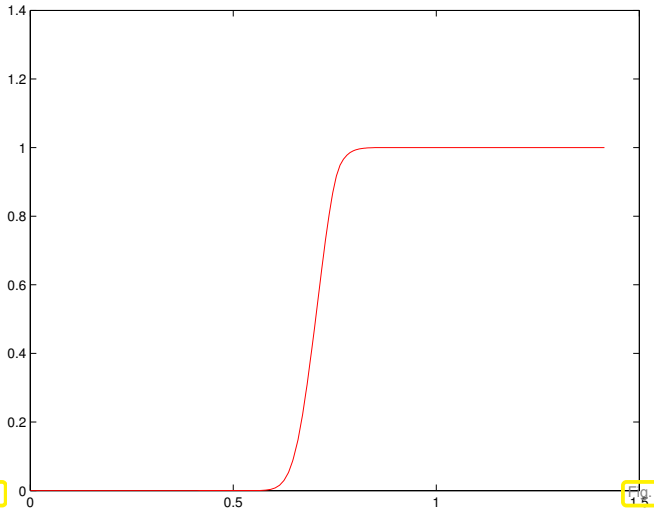


Fig. 315

upwind quadrature solution

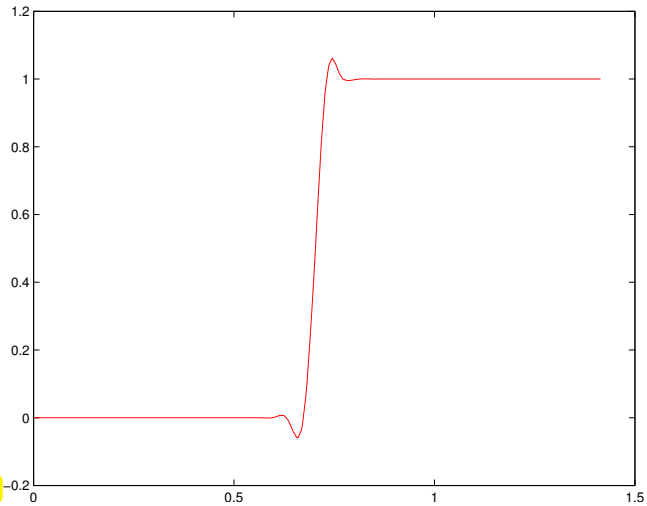


Fig. 316

streamline-diffusion solution

Observations:

- The streamline upwind method does not exactly respect the maximum principle, but offers a better resolution of the internal layer compared with upwind quadrature (Parlance: streamline diffusion method is "less diffusive").

Example 7.2.40 (Convergence of SUPG and upwind quadrature FEM)

- $\Omega =]0,1[^2$, model problem (7.2.2), $\mathbf{v}(\boldsymbol{x}) = \binom{2}{3}$, right hand side f such that

$$u_\varepsilon(x,y) = xy^2 - y^2 e^{2\frac{x-1}{\epsilon}} - xe^{3\frac{y-1}{\epsilon}} + e^{2\frac{x-1}{\epsilon}+3\frac{y-1}{\epsilon}} \ .$$

- Finite element discretization, $V_{0,N} = \mathcal{S}_1^0(\mathcal{M})$ und sequence of unstructured triangular "uniform" meshes, with
  - upwind quadrature stabilization from Sect. 7.2.2.1,
  - SUPG stabilization according to (7.2.37).
- Monitored: (Approximate) $L^2(\Omega)$-norm of discretization error (computed with high-order local quadrature)

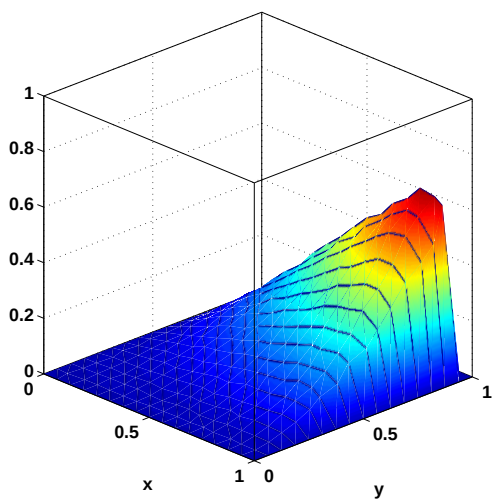


Fig. 317

u_ϵ for $\epsilon = 1$

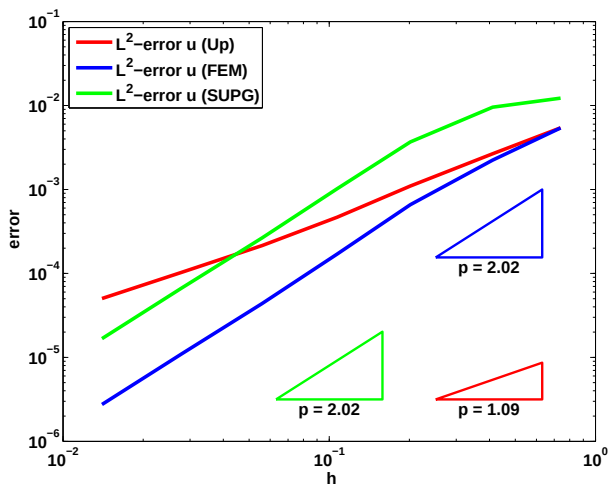


Fig. 318

Convergence for $\epsilon = 1$

Observation: SUPG stabilization does not affect $O(h_{\mathcal{M}}^2)$-convergence of $\|u - u_N\|_{L^2(\Omega)}$ for h-refinement and $h_{\mathcal{M}} \to 0$, whereas upwind quadrature leads to worse $O(h_{\mathcal{M}})$ convergence of the L^2-error norm.

7.3 Transient convection-diffusion BVP

Sect. 7.1.4 introduced the transient heat conduction model in a fluid, whose motion is described by a non-stationary velocity field (→ Sect. 7.1.1) $\mathbf{v} : \Omega \times]0,T[\mapsto \mathbb{R}^d$

$$\frac{\partial}{\partial t}(\rho u) - \operatorname{div}(\kappa\, \mathbf{grad}\, u) + \operatorname{div}(\rho \mathbf{v}(\boldsymbol{x},t)u) = f(\boldsymbol{x},t) \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,, \tag{7.1.16}$$

where $u = u(\boldsymbol{x},t) : \widetilde{\Omega} \mapsto \mathbb{R}$ is the unknown temperature.

Assuming $\operatorname{div} \mathbf{v}(\boldsymbol{x},t) = 0$, as in Sect. 7.2, by scaling we arrive at the model equation for transient convection-diffusion

$$\frac{\partial u}{\partial t} - \epsilon \Delta u + \mathbf{v}(\boldsymbol{x},t) \cdot \mathbf{grad}\, u = f \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,, \tag{7.3.1}$$

supplemented with

- ✦ Dirichlet boundary conditions: $u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \forall \boldsymbol{x} \in \partial\Omega\,, \quad 0 < t < T$,
- ✦ initial conditions: $u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Omega$.

7.3.1 Method of lines

For the solution of IBVP (7.3.1) follow the general policy introduced in Sect. 6.1.4:

❶ Discretization in space on a *fixed* mesh ➢ initial value problem for ODE

❷ Discretization in time (by suitable numerical integrator = timestepping)

For instance, in the case of Dirichlet boundary conditions,

$$\begin{cases} \dfrac{\partial u}{\partial t} - \epsilon \Delta u + \mathbf{v}(\boldsymbol{x},t) \cdot \mathbf{grad}\, u = f \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,, \\ u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \forall \boldsymbol{x} \in \partial\Omega,\, 0 < t < T \quad , \quad u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Omega\,. \end{cases} \tag{7.3.2}$$

← spatial discretization

$$\mathbf{M}\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) + \epsilon \mathbf{A}\vec{\boldsymbol{\mu}}(t) + \mathbf{B}\vec{\boldsymbol{\mu}}(t) = \vec{\boldsymbol{\varphi}}(t)\,, \tag{7.3.3}$$

where

- ✦ $\vec{\boldsymbol{\mu}} = \vec{\boldsymbol{\mu}}(t) :]0,T[\mapsto \mathbb{R}^N \mathrel{\hat{=}}$ coefficient vector describing approximation $u_N(t)$ of $u(\cdot,t)$,
- ✦ $\mathbf{A} \in \mathbb{R}^{N,N} \mathrel{\hat{=}}$ s.p.d. matrix of discretized $-\Delta$, e.g., (finite element) Galerkin matrix,
- ✦ $\mathbf{M} \in \mathbb{R}^{N,N} \mathrel{\hat{=}}$ (lumped → Rem. 6.2.45) mass matrix
- ✦ $\mathbf{B} \in \mathbb{R}^{N,N} \mathrel{\hat{=}}$ matrix for discretized convective term, e.g., Galerkin matrix, upwind quadrature matrix (→ Sect. 7.2.2.1), streamline diffusion matrix (→ Sect. 7.2.2.2).

Example 7.3.4 (Implicit Euler method of lines for transient convection-diffusion)

1D convection-diffusion IBVP:

$$\frac{\partial u}{\partial t} - \epsilon \frac{\partial^2 u}{\partial x^2} + \frac{\partial u}{\partial x} = 0 \,, \quad u(x,0) = \max(1 - 3|x - \tfrac{1}{3}|, 0) \,, \quad u(0) = u(1) = 0 \,. \tag{7.3.5}$$

- Spatial discretization on equidistant mesh with meshwidth $h = 1/N$:
  1. central finite difference scheme, see (7.2.18) ($\leftrightarrow$ linear FE Galerkin discretization),
  2. upwind finite difference discretization, see (7.2.22),
- $\mathbf{M} = h\mathbf{I}$ ("lumped" mass matrix, see Rem. 6.2.45),
- Temporal discretization with uniform timestep $\tau > 0$:
  1. implicit Euler method, see (6.1.37),
  2. explicit Euler method, see (6.1.36),

Computations with $\epsilon = 10^{-5}$, implicit Euler discretization, $h = 0.01$, $\tau = 0.00125$:

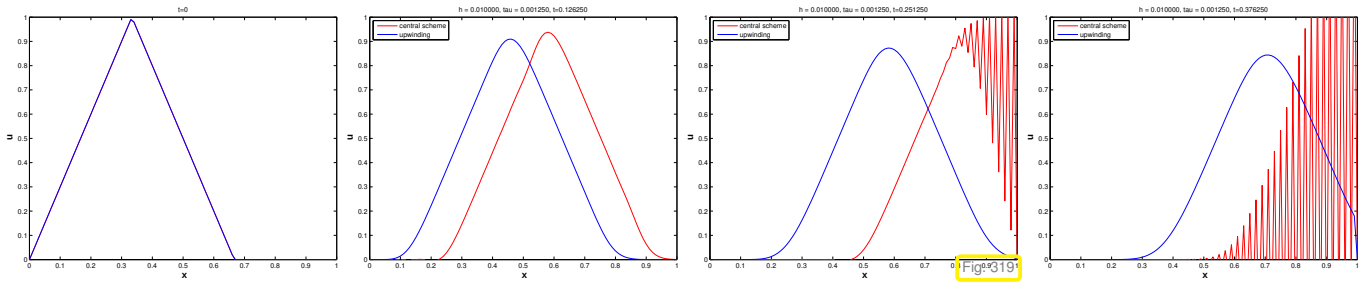


Fig. 319

Observation:

- Central finite differences display spurious oscillations as in Ex. 7.2.20.
- Upwinding suppresses spurious oscillations, but introduces *spurious damping*.

Computations with $\epsilon = 10^{-5}$, spatial upwind discretization, $h = 0.01$, $\tau = 0.005$:

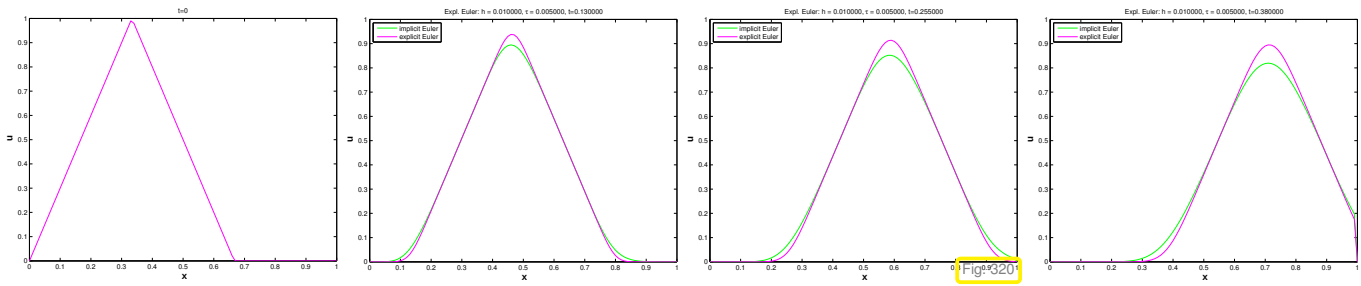


Fig. 320

Observation: implicit Euler timestepping causes stronger spurious damping than explicit Euler timestepping.

However, explicit Euler subject to tight stability induced timestep constraint for larger values of ϵ, see Sect. 6.1.5.2.

Advice for spatial discretization for method of lines approach

Use ϵ-robustly stable spatial discretization of convective term.

Remark 7.3.6 (Choice of timestepping for m.o.l. for transient convection-diffusion)

If ϵ-robustness *for all* $\epsilon > 0$ (including $\epsilon > 1$) desired ➢ Arguments of Sect. 6.1.5.2 stipulate use of L(π)-stable ($\rightarrow$ Def. 6.1.88) timestepping methods (implicit Euler (6.1.37), RADAU-3 (6.1.90), SDIRK-2 (6.1.91))

In the *singularly perturbed case* $0 < \epsilon \ll 1$ conditionally stable explicit timestepping is an option, due to a timestep constraint of the form "$\tau < O(h_{\mathcal{M}})$", which does not interfere with efficiency, *cf.* the discussion in Sect. 6.1.6.

7.3.2 Transport equation

Focus on the situation of singular perturbation ($\rightarrow$ Def. 7.2.16): $0 < \epsilon \ll 1$

➢ study limit problem (as in Sect. 7.2.1)

$$\frac{\partial u}{\partial t} - \epsilon \Delta u + \mathbf{v}(\boldsymbol{x},t) \cdot \mathbf{grad}\, u = f \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,,$$

$\leftarrow \epsilon = 0$

$$\frac{\partial u}{\partial t} + \mathbf{v}(\boldsymbol{x},t) \cdot \mathbf{grad}\, u = f \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,. \tag{7.3.7}$$

= transport equation

Now: focus on case $f \equiv 0$ (no sources)

Let $u = u(\boldsymbol{x},t)$ be a C^1-solution of

$$\frac{\partial u}{\partial t} + \mathbf{v}(\boldsymbol{x},t) \cdot \mathbf{grad}\, u = 0 \quad \text{in } \widetilde{\Omega} := \Omega \times]0,T[\,. \tag{7.3.8}$$

Recall: for the stationary pure tranport problem (7.2.9) we found solutions by integrating the source term along streamlines (following the flow direction).

➢ study the behavior of u "as seen from a moving fluid particle"

$$t \mapsto u(\mathbf{y}(t),t)\,, \quad \text{where} \quad \mathbf{y}(t) \quad \text{solves} \quad \frac{d\mathbf{y}}{dt}(t) = \mathbf{v}(\mathbf{y}(t),t)\,, \quad \text{see (7.1.1)}\,.$$

By the chain rule

$$\begin{aligned} \frac{d}{dt} u(\mathbf{y}(t),t) &= \mathbf{grad}\, u(\mathbf{y}(t),t) \cdot \frac{d\mathbf{y}}{dt}(t) + \frac{\partial u}{\partial t}(\mathbf{y}(t),t) \\ &= \mathbf{grad}\, u(\mathbf{y}(t),t) \cdot \mathbf{v}(\mathbf{y}(t),t) + \frac{\partial u}{\partial t}(\mathbf{y}(t),t) \overset{(7.3.8)}{=} 0\,. \end{aligned} \tag{7.3.9}$$

A fluid particle "sees" a constant temperature!

Remark 7.3.10 (Solution formula for sourceless transport)

Situation: *no inflow/outflow* (e.g., fluid in a container)

$$\mathbf{v}(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x}\in\partial\Omega\,,\, 0<t<T\,. \tag{7.1.2}$$

➢ all streamlines will "stay inside Ω", flow map $\boldsymbol{\Phi}^t$ (7.1.3) defined for all times $t\in\mathbb{R}$.

Initial value problem:

$$\mathbf{v}(\boldsymbol{x},t)\cdot\mathbf{grad}\,u = 0 \quad \text{in } \widetilde{\Omega} \quad , \quad u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x}\in\Omega\,.$$

Exact solution

$$u(\boldsymbol{x},t) = u_0(\boldsymbol{x}_0(\boldsymbol{x},t))\,, \tag{7.3.11}$$

where $\boldsymbol{x}_0(\boldsymbol{x},t)$ is the position at time 0 of the fluid particle that is located at $\boldsymbol{x}$ at time t.

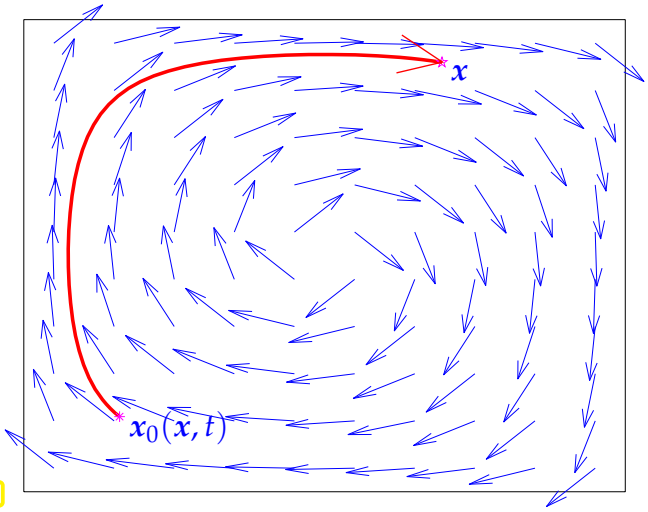


Fig. 321

This solution formula can be generalized to any divergence free velocity field $\mathbf{v}:\Omega\mapsto\mathbb{R}^d$ and $f\neq 0$. The new aspect is that streamlines can *enter* and *leave* the domain Ω. In the former case the solution value is given by a "transported boundary value":

$$\frac{d}{dt}u(\mathbf{y}(t)) = f(\mathbf{y}(t),t)$$

$$u(\boldsymbol{x},t) = \begin{cases} u_0(\boldsymbol{x}_0) + \int_0^t f(\mathbf{y}(s),s)\,\mathrm{d}s & \text{, if } \boldsymbol{y}(s)\in\Omega \quad \forall 0<s<t\,, \\ g(\mathbf{y}(s_0),s_0) + \int_{s_0}^t f(\mathbf{y}(s),s)\,\mathrm{d}s & \text{, if } \boldsymbol{y}(s_0)\in\partial\Omega,\, \boldsymbol{y}(s)\in\Omega \quad \forall s_0<s<t\,, \end{cases} \tag{7.3.12}$$

where we have assumed Dirichlet boundary conditions on the inflow boundary

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \text{for} \quad \boldsymbol{x}\in\Gamma_{\text{in}} := \{\boldsymbol{x}\in\partial\Omega:\ \mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x})<0\} \quad \text{, cf. (7.2.13).}$$

7.3.3 Lagrangian split-step method

Lagrangian discretization schemes for the IBVP (7.3.2) are inspired by insight into the traits of solutions of pure transport problems.

The variant that we are going to study separates the transient convection-diffusion problem into a pure diffusion problem (heat equation → Sect. 6.1.1) and a pure transport problem (7.3.7). This is achieved by means of a particular approach to timestepping.

7.3.3.1 Split-step timestepping

Abstract perspective: consider ODE, whose right hand side is the sum of two (smooth) functions

$$\dot{\mathbf{y}} = \mathbf{g}(t,\mathbf{y}) + \mathbf{h}(t,\mathbf{y}) \;, \quad \mathbf{g},\mathbf{h} : \mathbb{R}^m \mapsto \mathbb{R}^m \;. \tag{7.3.13}$$

There is an abstract timestepping scheme that offers great benefits if one commands efficient methods to solve initial value problems for both $\dot{\mathbf{z}} = \mathbf{g}(\mathbf{z})$ and $\dot{\mathbf{w}} = \mathbf{h}(\mathbf{w})$.

Strang splitting single step method for (7.3.13), timestep $\tau := t_j - t_{j-1} > 0$: compute $\mathbf{y}^{(j)} \approx \mathbf{y}(t_j)$ from $\mathbf{y}^{(j-1)} \approx \mathbf{y}(t_{j-1})$ according to

$$\widetilde{\mathbf{y}} := \mathbf{z}(t_{j-1} + \tfrac{1}{2}\tau) \;, \quad \text{where} \quad \mathbf{z}(t) \quad \text{solves} \quad \dot{\mathbf{z}} = \mathbf{g}(t,\mathbf{z}) \;, \quad \mathbf{z}(t_{j-1}) = \mathbf{y}^{(j-1)} \;, \tag{7.3.14}$$

$$\widehat{\mathbf{y}} := \mathbf{w}(t_j) \quad \text{where} \quad \mathbf{w}(t) \quad \text{solves} \quad \dot{\mathbf{w}} = \mathbf{h}(t,\mathbf{w}) \;, \quad \mathbf{w}(t_{j-1}) = \widetilde{\mathbf{y}} \;, \tag{7.3.15}$$

$$\mathbf{y}^{(j)} := \mathbf{z}(t_j) \;, \quad \text{where} \quad \mathbf{z}(t) \quad \text{solves} \quad \dot{\mathbf{z}} = \mathbf{g}(t,\mathbf{z}) \;, \quad \mathbf{z}(t_{j-1} + \tfrac{1}{2}\tau) = \widehat{\mathbf{y}} \;. \tag{7.3.16}$$

One timestep involves three sub-steps:

① Solve $\dot{\mathbf{z}} = \mathbf{g}(t,\mathbf{z})$ over time $[t_{j-1}, t_{j-1} + \frac{1}{2}\tau]$ using the result of the previous timestep as initial value ↔ (7.3.14).

② Solve $\dot{\mathbf{w}} = \mathbf{h}(t,\mathbf{w})$ over time τ using the result of ① as initial value ↔ (7.3.15).

③ Solve $\dot{\mathbf{z}} = \mathbf{g}(t,\mathbf{z})$ over time $[t_{j-1} + \frac{1}{2}\tau, t_j]$ using the result of ② as initial value ↔ (7.3.16).

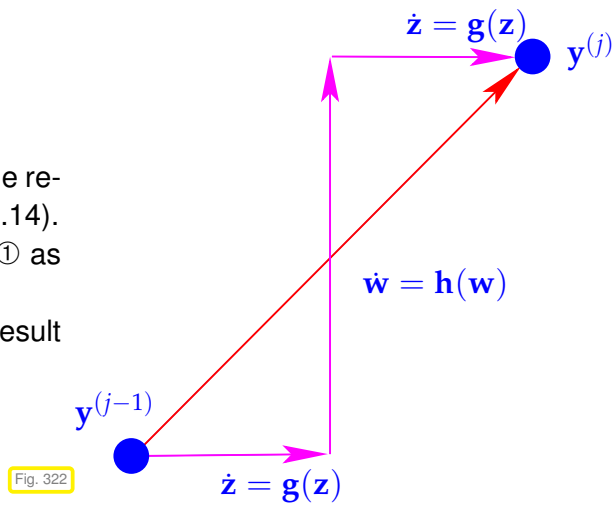


Fig. 322

Theorem 7.3.17. Order of Strang splitting single step method

Assuming exact solution of the initial value problems of the sub-steps, the Strang splitting single step method for (7.3.13) *is of second order.*

This applies to Strang splitting timestepping for initial value problems for ODEs. Now we boldly regard (7.3.2) as an "*ODE in function space*" for the unknown "function space valued function" $u = u(t) : [0,T] \mapsto H^1(\Omega)$.

$$\begin{array}{ccccc} \dfrac{du}{dt} & = & \epsilon\Delta u & + & f - \mathbf{v}\cdot\mathbf{grad}\, u \\ \updownarrow & & \updownarrow & & \updownarrow \\ \dot{\mathbf{y}} & = & \mathbf{g}(\mathbf{y}) & + & \mathbf{h}(\mathbf{y}) \end{array}$$

Formally, we arrive at the following "timestepping scheme in function space" on a temporal mesh $0 = t_0 < t_1 < \cdots < t_M := T$ for (7.3.1):

Given approximation $u^{(j-1)} \approx u(t_{j-1})$,

① Solve (autonomous) parabolic IBVP for *pure diffusion* from t_{j-1} to $t_{j-1} + \frac{1}{2}\tau$

$$\text{(7.3.14)} \quad \leftrightarrow \quad \begin{aligned} \frac{\partial w}{\partial t} - \epsilon \Delta w &= 0 \quad \text{in } \Omega \times]t_{j-1}, t_{j-1} + \tfrac{1}{2}\tau[\,, \\ w(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t_{j-1}) \quad \forall \boldsymbol{x} \in \partial\Omega,\ t_{j-1} < t < t_{j-1} + \tfrac{1}{2}\tau\,, \\ w(\boldsymbol{x}, t_{j-1}) &= u^{(j-1)}(\boldsymbol{x}) \quad \forall \boldsymbol{x} \in \Omega\,. \end{aligned} \tag{7.3.18}$$

② Solve IBVP for *pure transport* (= advection), see Sect. 7.3.2,

$$\text{(7.3.15)} \quad \leftrightarrow \quad \begin{aligned} \frac{\partial z}{\partial t} + \mathbf{v}(\boldsymbol{x}, t) \cdot \mathbf{grad}\, z &= f(\boldsymbol{x}, t) \quad \text{in } \Omega \times]t_{j-1}, t_j[\,, \\ z(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t) \quad \text{on inflow boundary } \Gamma_{\text{in}}\,,\ t_{j-1} < t < t_j\,, \\ z(\boldsymbol{x}, t_{j-1}) &= w(\boldsymbol{x}, t_{j-1} + \tfrac{1}{2}\tau) \quad \forall \boldsymbol{x} \in \Omega\,. \end{aligned} \tag{7.3.19}$$

③ Solve (autonomous) parabolic IBVP for *pure diffusion* from $t_{j-1} + \frac{1}{2}\tau$ to t_j

$$\text{(7.3.16)} \quad \leftrightarrow \quad \begin{aligned} \frac{\partial w}{\partial t} - \epsilon \Delta w &= 0 \quad \text{in } \Omega \times]t_{j-1} + \tfrac{1}{2}\tau, t_j[\,, \\ w(\boldsymbol{x}, t) &= g(\boldsymbol{x}, t_j) \quad \forall \boldsymbol{x} \in \partial\Omega\,,\ t_{j-1} + \tfrac{1}{2}\tau < t < t_j\,, \\ w(\boldsymbol{x}, t_{j-1} + \tfrac{1}{2}\tau) &= z(\boldsymbol{x}, t_j) \quad \forall \boldsymbol{x} \in \Omega\,. \end{aligned} \tag{7.3.20}$$

Then set $u^{(j)}(\boldsymbol{x}) := w(\boldsymbol{x}, t_j)$, $\boldsymbol{x} \in \Omega$.

Efficient "implementation" of Strang splitting timestepping, if $\mathbf{g} = \mathbf{g}(\mathbf{y})$:

combine last sub-step ③ with first sub-step ① of the next timestep

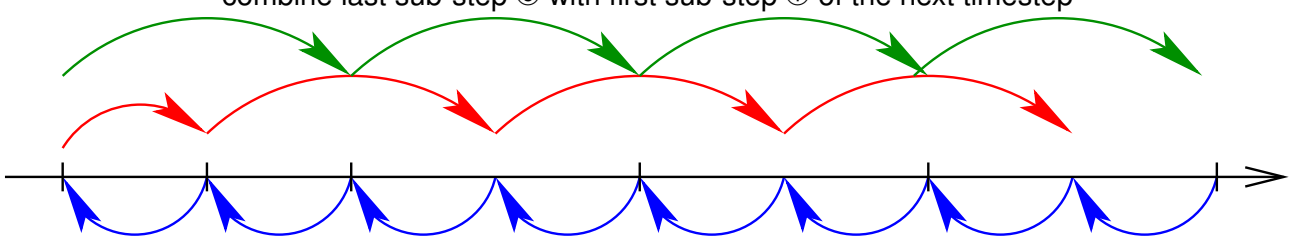


Fig. 323 t_0 t_1 t_2 t_3 t_4

—: sub-step ②, —: sub-step ① & ③

Remark 7.3.21 (Approximate sub-steps for Strang splitting time)

The solutions of the initial value problems in the sub-steps of Strang splitting timestepping may be computed *only approximately*.

If this is done by one step of a 2nd-order timestepping method in each case, then the resulting approximate Strang splitting timestepping will still be of second order, *cf.* Thm. 7.3.3.1.

7.3.3.2 Particle method for advection

Recall the discussion of the IBVP for the pure transport (= advection) equation from Sect. 7.3.2

$$\begin{aligned} \frac{\partial u}{\partial t} + \mathbf{v}(\boldsymbol{x},t)\cdot \mathbf{grad}\, u = f \quad &\text{in } \widetilde{\Omega} := \Omega\times]0,T[\,, \\ u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad &\text{on } \quad \Gamma_{\text{in}}\times]0,T[\,, \\ u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad &\text{in } \Omega\,, \end{aligned} \tag{7.3.22}$$

with inflow boundary

$$\Gamma_{\text{in}} := \{\boldsymbol{x}\in\partial\Omega\colon \mathbf{v}(\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x}) < 0\}\,. \tag{7.2.13}$$

Case $f \equiv 0$: a travelling fluid particle sees a constant solution, see (7.3.9)

$$\blacktriangleright \quad u(\boldsymbol{x},t) = \begin{cases} u_0(\boldsymbol{x}_0) & \text{, if } \quad \boldsymbol{y}(s)\in\Omega \quad \forall 0<s<t\,, \\ g(\mathbf{y}(s_0),s_0) & \text{, if } \quad \boldsymbol{y}(s_0)\in\partial\Omega,\, \boldsymbol{y}(s)\in\Omega \quad \forall s_0<s<t\,, \end{cases} \tag{7.3.23}$$

where $s \mapsto \mathbf{y}(s)$ solves the initial value problem $\frac{d\mathbf{y}}{ds}(s) = \mathbf{v}(\mathbf{y}(s),s)$, $\mathbf{y}(t) = \boldsymbol{x}$ ("backward particle trajectory"). Case of general f, see Rem. 7.3.10: Since $\frac{d}{dt}u(\mathbf{y}(t)) = f(\mathbf{y}(t),t)$

$$\blacktriangleright \quad u(\boldsymbol{x},t) = \begin{cases} u_0(\boldsymbol{x}_0) + \int\limits_0^t f(\mathbf{y}(s),s)\,\mathrm{d}s & \text{, if } \quad \boldsymbol{y}(s)\in\Omega \quad \forall 0<s<t\,, \\ g(\mathbf{y}(s_0),s_0) + \int\limits_{s_0}^t f(\mathbf{y}(s),s)\,\mathrm{d}s & \text{, if } \quad \boldsymbol{y}(s_0)\in\partial\Omega,\, \boldsymbol{y}(s)\in\Omega \quad \forall s_0<s<t\,. \end{cases} \tag{7.3.12}$$

The solution formula (7.3.12) suggests an approach for solving (7.3.22) approximately.

We first consider the simple situation of no inflow/outflow (e.g., fluid in a container, see Rem. 7.3.10)

$$\mathbf{v}(\boldsymbol{x},t)\cdot\boldsymbol{n}(\boldsymbol{x}) = 0 \quad \forall \boldsymbol{x}\in\partial\Omega\,,\, 0<t<T\,. \tag{7.1.2}$$

① Pick suitable interpolation nodes $\{\boldsymbol{p}_i\}_{i=1}^N \subset \Omega$ (initial 'particle positions")

② "Particle pushing": Solve initial value problems (*cf.* ODE (7.1.1) for particle trajectories)

$$\dot{\mathbf{y}}(t) = \mathbf{v}(\mathbf{y}(t),t) \quad , \quad \mathbf{y}(0) = \boldsymbol{p}_i\,, \quad i = 1,\dots,N\,,$$

by means of a suitable single-step method with uniform timestep $\tau := T/M$, $M\in\mathbb{N}$.

➢ sequencies of solution points $\boldsymbol{p}_i^{(j)}$, $j = 0,\dots,M$, $i = 1,\dots,N$

③ Reconstruct approximation $u_N^{(j)} \approx u(\cdot,t_j)$, $t_j := j\tau$, by interpolation:

$$u_N^{(j)}(\boldsymbol{p}_i^{(j)}) := u_0(\boldsymbol{p}_i) + \tau\sum_{l=1}^{j-1} f(\tfrac{1}{2}(\boldsymbol{p}_i^{(l)} + \boldsymbol{p}_i^{(l-1)}), \tfrac{1}{2}(t_l + t_{l-1}))\,, \quad i = 1,\dots,N$$

where the composite midpoint quadrature rule was used to approximate the source integral in (7.3.12).

This method falls into the class of

- particle methods, because the interpolation nodes can be regarded fluid particles tracked by the method,
- Lagrangian methods, which treat the IBVP in coordinate systems moving with the flow,
- characteristic methods, which reconstruct the solution from knowledge about its behavior along streamlines.

For general velocity field $\mathbf{v} : \Omega \mapsto \mathbb{R}^d$:

- Stop tracking i-th trajectory as soon as an interpolation nodes $\boldsymbol{p}_i^{(j)}$ lies outside spatial domain Ω.
- In each timestep start new trajectories from fixed locations on inflow boundary Γ_{in} ("particle injection"). These interpolation nodes will carry the boundary value.

Example 7.3.24 (Point particle method for pure advection)

- IBVP (7.3.22) on $\Omega =]0,1[^2$, $T = 2$, with $f \equiv 0$, $g \equiv 0$.
- Initial locally supported bump $u_0(\boldsymbol{x}) = \max\{0, 1 - 4\left\|\boldsymbol{x} - \binom{1/2}{1/4}\right\|\}$.
- Two stationary divergence-free velocity fields
  - $\mathbf{v}_1(\boldsymbol{x}) = \begin{pmatrix} -\sin(\pi x_1)\cos(\pi x_2) \\ \cos(\pi x_1)\sin(\pi x_2) \end{pmatrix}$ satisfying (7.1.2),
  - $\mathbf{v}_2(\boldsymbol{x}) = \begin{pmatrix} -x_2 \\ x_1 \end{pmatrix}$.
- Initial positions of interpolation points on regular tensor product grid with meshwidth $h = \frac{1}{40}$.
- Approximation of trajectories by means of explicit trapezoidal rule [3, Eq. (11.4.6)] (method of Heun).

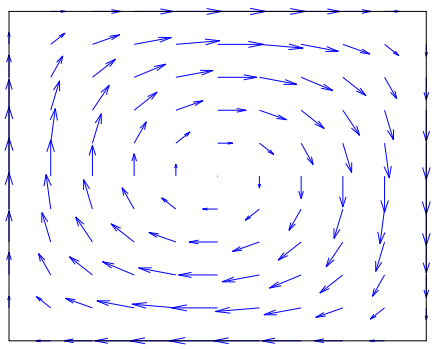


Fig. 324

velocity field $\mathbf{v}_1$ (`circvel`)

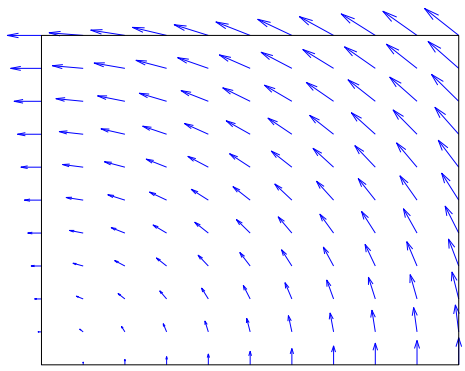


Fig. 325

velocity field $\mathbf{v}_2$ (`rotvel`)

MATLAB Code 7.3.25: Confined velocity field

```
function V = circvel(P)
% Circular velocity (divergence free, zero normal component on unit
    square).
% P: 2xN matrix of point coordinates
% return value:  velocity vectors at points in P
```

```
v = @(p) [-sin(pi*p(1))*cos(pi*p(2));sin(pi*p(2))*cos(pi*p(1))];

V = [];
for p=P
  V = [V, v(p)];
end
```

MATLAB Code 7.3.26: Pass-through velocity field

```
function V = rotvel(P)
% Circular velocity

v = @(p) [-p(2); p(1)];

V = [];
for p=P
  V = [V, v(p)];
end
```

MATLAB Code 7.3.27: Point particle method for pure advection

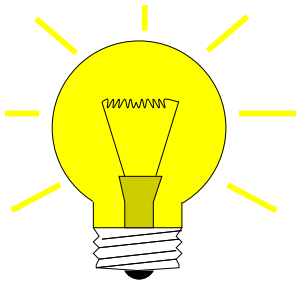
```
function partadv(v,u0,g,n,tau,m)
% Point particle method for pure advection problem
% on the unit sqaure
% v:  handle to a function returning the velocity field for (an array) of points
% u0:  handle to a function returning the initial value u_0 for (an array)
% of points
% g:  handle to a function g = g(x) returning the Dirichlet boundary values
% n:  h = 1/n is the grid spacing of the inintial point distribution
% tau:  timestep size, m:  number of timesteps, that is, T = mτ

% Initialize points
h = 1/n; [Xp,Yp] = meshgrid(0:h:1,0:h:1);
P = [reshape(Xp,1,(n+1)^2);reshape(Yp,1,(n+1)^2)];
% Initialize points on the boundary
BP = [[(0:h:1);zeros(1,n+1)],[ones(1,n+1);(0:h:1)],...
      [(0:h:1);ones(1,n+1)],[zeros(1,n+1);(0:h:1)]];
U = u0(P); % Initial values

% Plot velocity field
hp = 1/10; [Xp,Yp] = meshgrid(0:hp:1,0:hp:1);
Up = zeros(size(Xp)); Vp = zeros(size(Xp));
for i=0:10, for j=0:10
  x = v([Xp(i+1,j+1);Yp(i+1,j+1)]);
  Up(i+1,j+1) = x(1); Vp(i+1,j+1) = x(2);
end; end
```

```
figure('name','velocity field','renderer','painters');
quiver(Xp,Yp,Up,Vp,'b-'); set(gca,'fontsize',14); hold on;
plot([0 1 1 0 0],[0 0 1 1 0],'k-');
axis([-0.1 1.1 -0.1 1.1]);
xlabel('{\bf x_1}'); ylabel('{\bf x_2}');
axis off;

fp = figure('name','particles','renderer','painters');
fs = figure('name','solution','renderer','painter');

% Visualize points (interior points in red, boundary points in blue)
figure(fp); plot(P(1,:),P(2,:),'r+',BP(1,:),BP(2,:),'b*');
title(sprintf('n = %i, t = %f, \\tau = %f, %i points',n,0,tau,size(P,2)));
drawnow; pause;

% Visualize solution
figure(fs); plotpartsol(P,U); drawnow;

t = 0;
for l=1:m
% Advect points (explicit trapezoidal rule)
  P1 = P + tau/2*v(P); P  = P + tau*v(P1);

% Remove points on the boundary or outside the domain
  Pnew = []; Unew = []; l = 1;
  for p=P
    if ((p(1) > eps)  (p(1) < 1-eps)  (p(2) > eps)  (p(2) < 1-eps))
      Pnew = [Pnew,p]; Unew = [Unew; U(l)];
    end
    l = l+1;
  end

% Add points on the boundary (particle injection)
  P = [Pnew, BP]; U = [Unew; g(BP)];

% Visualize points
  figure(fp); plot(P(1,:),P(2,:),'r+',BP(1,:),BP(2,:),'b*');
  title(sprintf('n = %i, t = %f, \\tau = %f, %i points',n,t,tau,size(P,2)));
  drawnow;
% Visualize solution
  figure(fs); plotpartsol(P,U); drawnow;

  t = t+tau;
end
```

7.3.3.3 Particle mesh method

The method introduced in the previous section, can be used to tackle the pure advection problem (7.3.19) in the 2nd sub-step of the Strang splitting timestepping.

Issue: How to combine Lagrangian advection with a method for the pure diffusion problem (7.3.18) faced in the other sub-steps of the Strang splitting timestepping**?**



Idea: two views

"particle temperatures" $u(\boldsymbol{p}_i^{(j)})$

$\updownarrow$

Nodal values of finite element function $u_N^{(j)} \in \mathcal{S}_1^0(\mathcal{M})$

Outline: algorithm for one step of size $\tau > 0$ of Strang splitting timestepping for transient convection-diffusion problem

$$\begin{cases} \dfrac{\partial u}{\partial t} - \epsilon\Delta u + \mathbf{v}(\boldsymbol{x},t)\cdot\mathbf{grad}\, u = f \quad \text{in } \widetilde{\Omega} := \Omega\times]0,T[\,, \\ u(\boldsymbol{x},t) = 0 \quad \forall \boldsymbol{x}\in\partial\Omega,\, 0<t<T \quad , \quad u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \forall \boldsymbol{x}\in\Omega\,. \end{cases} \tag{7.3.2}$$

❶ Given
- ✦ triangular mesh $\mathcal{M}^{(j-1)}$ of Ω,
- ✦ $u_N^{(j-1)} \in \mathcal{S}_{1,0}^0(\mathcal{M}^{(j-1)}) \leftrightarrow$ coefficient vector $\vec{\boldsymbol{\mu}}^{(j-1)} \in \mathbb{R}^{N_{j-1}}$,

approximately solve (7.3.18) by a single step of implicit Euler (6.1.37) (size $\frac{1}{2}\tau$)

$$\vec{\boldsymbol{\nu}} = (\mathbf{M} + \tfrac{1}{2}\tau\epsilon\mathbf{A})^{-1}\vec{\boldsymbol{\mu}}^{(j-1)}\,,$$

where $\mathbf{A} \in \mathbb{R}^{N_{j-1},N_{j-1}} \mathrel{\hat{=}} \mathcal{S}_{1,0}^0(\mathcal{M})$-Galerkin matrix for $-\Delta$, $\mathbf{M} \mathrel{\hat{=}}$ (possibly lumped) $\mathcal{S}_{1,0}^0(\mathcal{M})$-mass matrix.

More advisable to maintain 2nd-order timestepping: 2nd-order $L(\pi)$-stable single step method,e.g., SDIRK-2 (6.1.91).

❷ Lagrangian advection step (of size τ) for (7.3.19) with
- ✦ initial "particle positions" $\boldsymbol{p}_i$ given by nodes of $\mathcal{M}^{(j-1)}$, $i = 1,\dots,N_j$,
- ✦ initial "particle temperatures" given by corresponding coefficients ν_i.

❸ *Remeshing*: advection step has moved nodes to new positions $\widetilde{\boldsymbol{p}}_i$ (and, maybe, introduced new nodes by "particle injection", deleted nodes by "particle removal").

➢ Create **new** triangular mesh $\mathcal{M}^{(j)}$ with nodes $\widetilde{\boldsymbol{p}}_i$ (+ boundary nodes), $i = 1,\dots,N_j$

❹ Repeat diffusion step ❶ starting with $w_N \in \mathcal{S}_{1,0}^0(\mathcal{M}^{(j)})$ = linear interpolant ($\rightarrow$ Def. 5.3.18) of "particle temperatures" on $\mathcal{M}^{(j)}$.

➢ new approximate solution $u_N^{(j)}$

Example 7.3.28 (Delaunay-remeshing in 2D)

Delaunay algorithm for creating a 2D triangular mesh *with prescribed nodes*:

① Compute Voronoi cells, see (4.2.4) & http://www.qhull.org/.

② Connect two nodes, if their associated Voronoi dual cells have an edge in common.

➥ MATLAB `TRI = delaunay(x,y)`

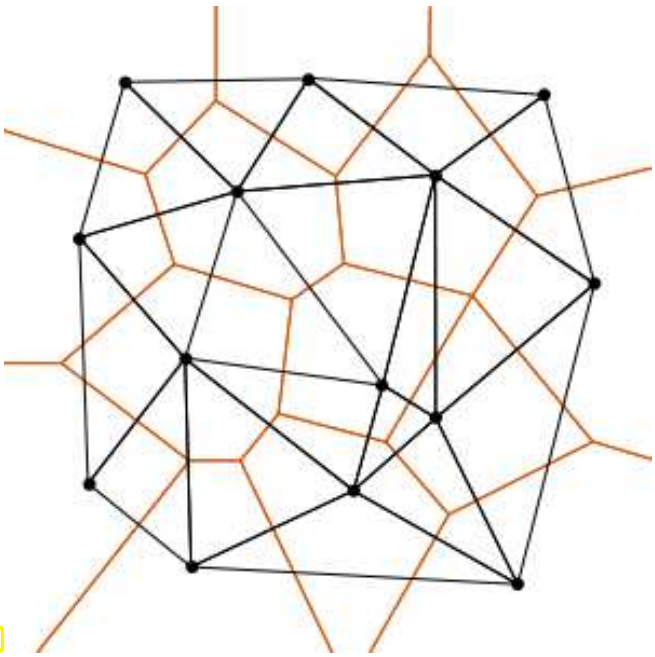


Fig. 326

MATLAB Code 7.3.29: Demonstration of Delaunay-remeshing

```
function meshadv(v,n,tau,m)
% Point advaction and remeshing for Lagrangian method
% v:  handle to a function returning the velocity field for (an array) of points
% n:  h = 1/n is the grid spacing of the inintial point distribution

% Initialize points
h = 1/n; [Xp,Yp] = meshgrid(0:h:1,0:h:1);
P = [reshape(Xp,1,(n+1)^2);reshape(Yp,1,(n+1)^2)];
% Initialize points on the boundary
BP = [[(0:h:1);zeros(1,n+1)],[ones(1,n+1);(0:h:1)],...
      [(0:h:1);ones(1,n+1)],[zeros(1,n+1);(0:h:1)]];

% Plot triangulation
fp = figure('name','evolving meshes','renderer','painters');
TRI = delaunay(P(1,:),P(2,:));
plot(P(1,:),P(2,:),'r+'); hold on; triplot(TRI,P(1,:),P(2,:),'blue'); hold off;
title(sprintf('n = %i, t = %f, \\tau = %f, %i points',n,0,tau,size(P,2)));
drawnow; pause;

t = 0;
for l=1:m
% Advect points (explicit trapezoidal rule)
  P1 = P + tau/2*v(P); P  = P + tau*v(P1);

% Remove points on the boundary or outside the domain
  Pnew = []; l = 1;
  for p=P
    if ((p(1) > eps)  (p(1) < 1-eps)  (p(2) > eps)  (p(2) < 1-eps))
```

```
      Pnew = [Pnew,p];
    end
    l = l+1;
  end

  P = [Pnew, BP]; % Add points on the boundary (particle injection)

% Plot triangulation
  TRI = delaunay(P(1,:),P(2,:));
  plot(P(1,:),P(2,:),'r+'); hold on;
     triplot(TRI,P(1,:),P(2,:),'blue'); hold off;
  title(sprintf('n = %i, t = %f, \\tau = %f, %i
     points',n,t,tau,size(P,2)));
  drawnow;

  t = t+tau;
end
```

$\Omega =]0,1[^2$, velocity fields like in Ex. 7.3.24. Advection of interpolation nodes by means of explicit trapezoidal rule.

Start animations:

```
meshadv(@circvel,20,0.05,40);
meshadv(@rotvel,20,0.05,40);
```

Example 7.3.30 (Lagrangian method for convection-diffusion in 1D)

Same IBVP as in Ex. 7.3.4

- Linear finite element Galerkin discretization with mass lumping in space
- Strang splitting applied to diffusive and convective terms
- Implicit Euler timestepping for diffusive partial timestep

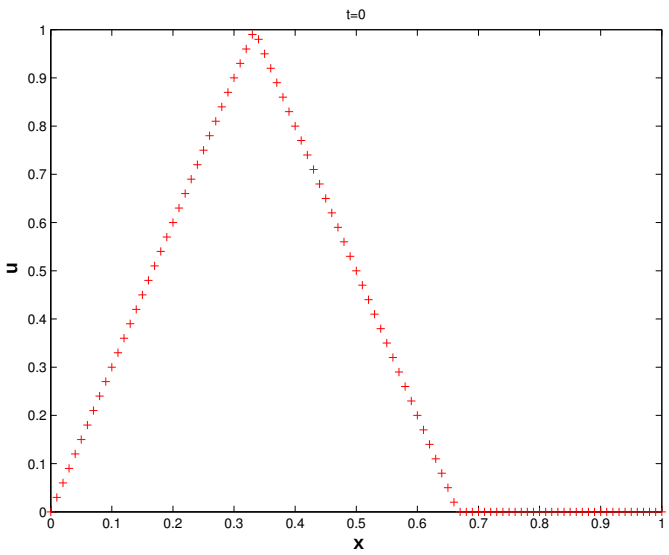


Fig. 327

MATLAB Code 7.3.31: Lagrangian method for (7.3.5)

```
function lagr(epsilon,N,M)
% This function implements a simple Lagrangian advection scheme for the
```

```
% 1D convection-diffusion IBVP −ε d²u/dx² + du/dx = 0, u(x,0) = max(1 − 3|x − 1/3|,0),
% and homogeneous Dirichlet boundary conditions u(0) = u(1) = 0. Timestepping employs Strang splitting
% applied to diffusive and convective spatial operators.
% epsilon:  strength of diffusion
% N: number of cells of spatial mesh
% M: number of timesteps

T = 0.5; tau = T/M;                                          % timestep size
h = 1/N; x = 0:h:1; u = max(1-3*abs(x(2:end-1)-1/3),0)';     % Initial value

[Amat,Mmat] = getdeltamat(x);                 % Obtain stiffness and mass matrix
u = (Mmat+0.5*tau*epsilon*Amat)\(Mmat*u); % Implicit Euler timestep

for j=1:M+1
  % Advection step:  shift meshpoints, drop those travelling out of Ω =]0,1[, insert
  % new meshpoints from the left.  Solution values are just copied.
  xm = x(2:end-1)+tau;      % Transport of meshpoints (here:  explicit Euler)
  idx = find(xm < 1);       % Drop meshpoints beyond x = 1
  x = [0,tau,xm(idx),1];    % Insert new meshpoint at left end of Ω
  u = [0;u(idx)];           % Copy nodal values and feed 0 from left

  % Diffusion partial timestep
  [Amat,Mmat] = getdeltamat(x);           % Obtain stiffness and mass matrix on new mesh
  u = (Mmat+tau*epsilon*Amat)\(Mmat*u); % Implicit Euler step
  end
end
```

$\epsilon = 10^{-5}$:

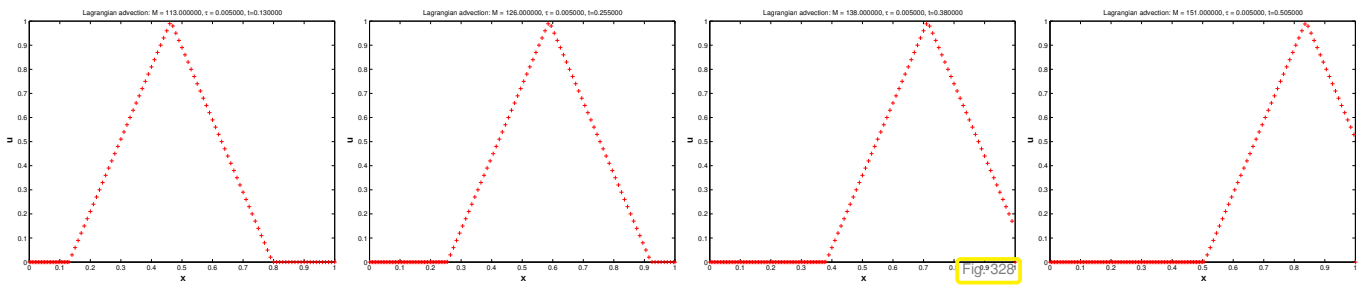


Fig. 328

$\epsilon = 0.1$:

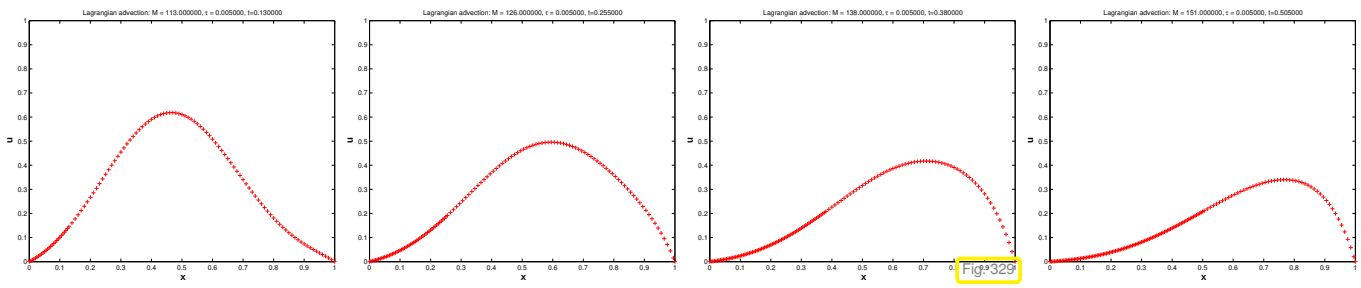


Fig. 329

"Reference solution" computed by method of lines, see Ex. 7.3.4, with $h = 10^{-3}$, $\tau = 5 \cdot 10^{-5}$:

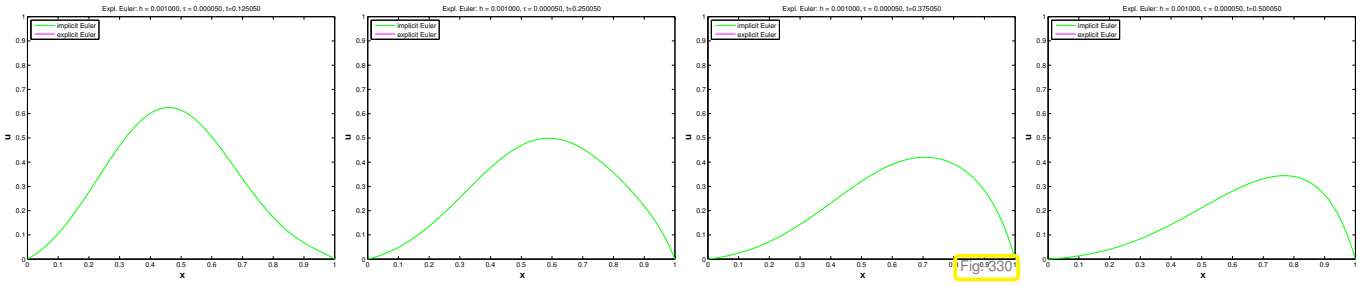


Fig. 330

Example 7.3.32 (Lagrangian method for convection-diffusion in 2D)

- IBVP (7.3.2) on $\Omega =]0,1[^2$, $T = 1$,
- Particle mesh method based on Delaunay remeshing, see Ex. 7.3.28, and linear finite element Galerkin discretizatin for diffusion step.

MATLAB Code 7.3.33: Particle mesh method in 2D

```
function ConvDiffLagr(v,epsilon,u0,n,tau,m)
% Point particle method for convection-diffusion problem on the unit sqaure
% v:  handle to a function returning the velocity field for (an array) of points
% u0:  handle to a function returning the initial value u_0 for (an array)
% of points
% n:  h = 1/n is the grid spacing of the inintial point distribution
% tau:  timestep size, m:  number of timesteps, that is, T = mτ

% Initialize points
h = 1/n; [Xp,Yp] = meshgrid(0:h:1,0:h:1);
P = [reshape(Xp,1,(n+1)^2);reshape(Yp,1,(n+1)^2)];
% Initialize points on the boundary
BP = [[(0:h:1);zeros(1,n+1)],[ones(1,n+1);(0:h:1)],...
      [(0:h:1);ones(1,n+1)],[zeros(1,n+1);(0:h:1)]];
% Construct initial mesh by Delaunay algorithm
TRI = delaunay(P(1,:),P(2,:));

U = u0(P); % Initial values

fp = figure('name','particles','renderer','painters');
fs = figure('name','solution','renderer','painters');

% Visualize mesh, points (interior points in red, boundary points in blue)
% the piecewise linear approximate solution
figure(fp); plot(P(1,:),P(2,:),'r+',BP(1,:),BP(2,:),'m*'); hold on;
triplot(TRI,P(1,:),P(2,:),'blue'); hold off;
title(sprintf('n = %i, t = %f, \\tau = %f, %i
```

```
   points',n,0,tau,size(P,2)));
drawnow;

figure(fs); trisurf(TRI,P(1,:),P(2,:),U');
axis([0 1 0 1 0 1]); xlabel('{\bf x_1}');
ylabel('{\bf x_2}');  zlabel('{\bf u}');
title(sprintf('n = %i, t = %f, \\tau = %f, %i
   points',n,0,tau,size(P,2)));
pause;

% Initial diffusion half step (implicit Euler)
[Amat,Mmat] = getGalerkinMatrices(TRI,P(1,:),P(2,:)); % Compute
   Galerkin matrices
% Isolate indices of interior points
j = 1; intidx = [];
for p=P
  if ((p(1) > eps)  (p(1) < 1-eps)  (p(2) > eps)  (p(2) < 1-eps))
    intidx = [intidx,j];
  end
  j = j+1;
end
Amat = Amat(intidx,intidx); Mmat = Mmat(intidx,intidx);
U(intidx) = (Mmat+0.5*epsilon*tau*Amat)\(Mmat*U(intidx));

% full(Amat), full(Mmat), return;

t = 0;
for l=1:m
% Advect points (explicit trapezoidal rule)
  P1 = P + tau/2*v(P); P  = P + tau*v(P1);

% Remove points on the boundary or outside the domain
  Pnew = []; Unew = []; l = 1; j = 0;
  for p=P
    if ((p(1) > eps)  (p(1) < 1-eps)  (p(2) > eps)  (p(2) <
       1-eps))
      Pnew = [Pnew,p]; Unew = [Unew; U(l)];
      j = j+1; % Counter for interior points
    end
    l = l+1;
  end

% Add points on the boundary (particle injection)
  P = [Pnew, BP];

% Delaunay algorithm for building triangulation
  TRI = delaunay(P(1,:),P(2,:));
  [Amat,Mmat] = getGalerkinMatrices(TRI,P(1,:),P(2,:)); % Compute
     Galerkin matrices
  Amat = Amat(1:j,1:j); Mmat = Mmat(1:j,1:j);
  U = (Mmat+epsilon*tau*Amat)\(Mmat*Unew);  % implicit Euler step
```

```
  U = [U; zeros(size(BP,2),1)]; % zero padding for boundary nodes

% Visualize mesh, points (interior points in red, boundary points in blue)
% the piecewise linear approximate solution
  figure(fp); plot(P(1,:),P(2,:),'r+',BP(1,:),BP(2,:),'m*'); hold on;
  triplot(TRI,P(1,:),P(2,:),'blue'); hold off;
  title(sprintf('n = %i, t = %f, \\tau = %f, %i points',n,t,tau,size(P,2)));
  drawnow;

  figure(fs); trisurf(TRI,P(1,:),P(2,:),U');
  axis([0 1 0 1 0 1]); xlabel('{\bf x_1}');
  ylabel('{\bf x_2}');  zlabel('{\bf u}');
  title(sprintf('n = %i, t = %f, \\tau = %f, %i points',n,t,tau,size(P,2)));
  t = t+tau;
end
```

Invocation: `ConvDiffLagr(@circvel,0.001,@initvals,1/40,0.01,100)`

Advantage of Lagrangian (particle) methods for convection diffusion:
No artificial diffusion required (no "smearing")
No stability induced timestep constraint

Drawback of Lagrangian (particle) methods for convection diffusion:
Remeshing (may be) expensive and difficult.
Point advection may produce "voids" in point set.

7.3.4 Semi-Lagrangian method

Now we study a family of methods for transient convection-diffusion that takes into account transport along streamlines, but, in constrast to genuine Lagrangian methods, relies on a *fixed* mesh.

Definition 7.3.34. Material derivative

Given a velocity field $\mathbf{v} : \Omega\times]0,T[\mapsto \mathbb{R}^d$, the material derivative of a function $f = f(\boldsymbol{x},t)$ at $(\boldsymbol{x},t)$ is

$$\frac{Df}{D\mathbf{v}}(\boldsymbol{x},t_0) = \lim_{\tau\to 0}\frac{f(\boldsymbol{x},t_0) - f(\boldsymbol{\Phi}_{t_0}^{-\tau}\boldsymbol{x},t_0-\tau)}{\tau}\;,\quad \boldsymbol{x}\in\Omega,\ 0<t_0<T\;,$$

with $\boldsymbol{\Phi}_{t_0}^{t}$ the flow map (at time t_0) associated with $\mathbf{v}$, that is, *cf.* (7.1.3), (7.1.4),

$$\frac{d\boldsymbol{\Phi}_{t_0}^{t}\boldsymbol{x}}{dt} = \mathbf{v}(\boldsymbol{\Phi}_{t_0}^{t}\boldsymbol{x},t-t_0)\quad,\quad \boldsymbol{\Phi}^0\boldsymbol{x} = \boldsymbol{x}\;.$$

The material derivative $\frac{Df}{D\mathbf{v}}$ is the

rate of change of f experienced by a particle carried along by the flow

because $\boldsymbol{\Phi}_{t_0}^{t}\boldsymbol{x}$ describes the trajectory of a particle located at $\boldsymbol{x}$ at time t_0 ($\leftrightarrow t=0$).

By a straightforward application of the chain rule for smooth f

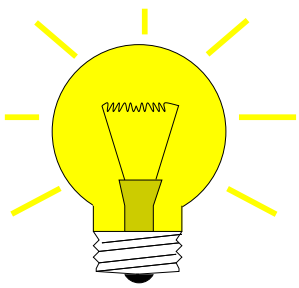
$$\boxed{\frac{Df}{D\mathbf{v}}(\boldsymbol{x},t) = \mathbf{grad}_{\boldsymbol{x}} f(\boldsymbol{x},t)\cdot\mathbf{v}(\boldsymbol{x},t) + \frac{\partial f}{\partial t}(\boldsymbol{x},t)}\;. \tag{7.3.35}$$

➢ The transient convection-diffusion equation can be rewritten as (7.3.1)

$$\frac{\partial u}{\partial t} - \epsilon\Delta u + \mathbf{v}(\boldsymbol{x},t)\cdot\mathbf{grad}\,u = f \quad\text{in}\quad \widetilde{\Omega} := \Omega\times]0,T[\;,$$

$\leftarrow$ (7.3.35)

$$\frac{Du}{D\mathbf{v}} - \epsilon\Delta u = f \quad\text{in}\quad \widetilde{\Omega} := \Omega\times]0,T[\;. \tag{7.3.36}$$



Idea: *Backward difference* ("implicit Euler") discretization of material derivative

$$\frac{Du}{D\mathbf{v}}_{|(\boldsymbol{x},t)=(\overline{\boldsymbol{x}},t_0)} \approx \frac{u(\overline{\boldsymbol{x}},t_0) - u(\boldsymbol{\Phi}_{t_0}^{-\tau}\overline{\boldsymbol{x}},t_0-\tau)}{\tau},$$

with timestep $\tau>0$, where $t\mapsto\boldsymbol{\Phi}^t\overline{\boldsymbol{x}}$ solves the initial value problem

$$\frac{d\boldsymbol{\Phi}_{t_0}^{t}\overline{\boldsymbol{x}}}{dt}(t) = \mathbf{v}(\boldsymbol{\Phi}_{t_0}^{t}\overline{\boldsymbol{x}},t_0+t)\quad,\quad \boldsymbol{\Phi}_{t_0}^{0}\overline{\boldsymbol{x}} = \overline{\boldsymbol{x}}\;.$$

▶ Semi-discretization of (7.3.36) *in time* (with fixed timestep $\tau>0$)

$$\begin{aligned}&\frac{u^{(j)}(\boldsymbol{x}) - u^{(j-1)}(\boldsymbol{\Phi}_{t_j}^{-\tau}\boldsymbol{x})}{\tau} - \epsilon\Delta u^{(j)}(\boldsymbol{x}) = f(\boldsymbol{x},t_j) \quad\text{in}\quad \Omega\;,\\ &+\quad \text{boundary conditions at } t=t_j\;,\end{aligned} \tag{7.3.37}$$

where $u^{(j)} : \Omega\mapsto\mathbb{R}$ is an approximation for $u(\cdot,t_j)$, $t_j := j\tau$, $j\in\mathbb{N}$. Note the difference to the method of lines ($\rightarrow$ Sects. 6.1.4, 6.2.3, 7.3.1): in (7.3.37) semidiscretization in time was carried out first, now followed by discretization in space, which reverses the order adopted in the method of lines.

Cast (7.3.37) into variational form according to the recipe of Sect. 2.9 and apply *Galerkin discretization* (here discussed for linear finite elements, homogeneous Dirichlet boundary conditions $u = 0$ on $\partial\Omega$).

This yields one timestep (size τ) for the semi-Lagrangian method: the approximation $u_N^{(j)}$ for $u(j\tau)$ (equidistant timesteps) is computed from the previous timestep according to

$$u_N^{(j)} \in \mathcal{S}_{1,0}^0(\mathcal{M}): \quad \int_\Omega \frac{u_N^{(j)}(\boldsymbol{x}) - u_N^{(j-1)}(\boldsymbol{\Phi}_{t_j}^{-\tau}\boldsymbol{x})}{\tau} v_N(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \epsilon \int_\Omega \mathbf{grad}\, u_N^{(j)} \cdot \mathbf{grad}\, v_N\,\mathrm{d}\boldsymbol{x}$$
$$= \int_\Omega f(\boldsymbol{x}, t_j) v_N(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v_N \in \mathcal{S}_{1,0}^0(\mathcal{M})\,. \qquad (7.3.38)$$

Here, $\mathcal{M}$ is supposed to be a *fixed* triangular mesh of Ω.

However, (7.3.38) cannot be implemented: $\boldsymbol{x} \mapsto u_N^{(j-1)}(\boldsymbol{\Phi}_{t_j}^{-\tau}\boldsymbol{x})$ is a finite element function that has been "transported with the (reversed) flow" (in the sense of pullback, see Def. 3.7.2)

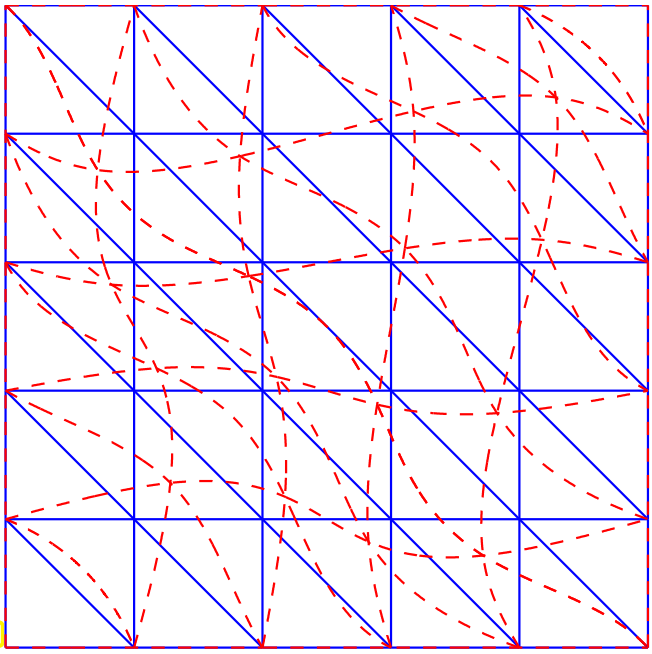


Fig. 331

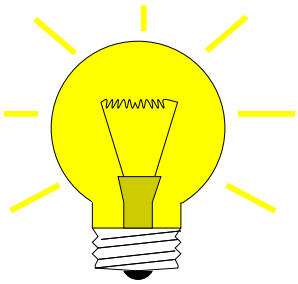
◁ - - - $\hat{=}$ image of $\mathcal{M}$ (—) under $\boldsymbol{\Phi}_{t_j}^{-\tau}$

The pullback $\boldsymbol{x} \mapsto v_N(\boldsymbol{\Phi}_{t_j}^{-\tau}\boldsymbol{x})$ of $v_N \in \mathcal{S}_{1,0}^0(\mathcal{M})$ is piecewise smooth w.r.t. the mapped mesh drawn with - - -. Hence, it is not smooth inside the cells of $\mathcal{M}$.

➢ the transported function may **not** be a finite element function on $\mathcal{M}$,

➢ the transported function may not even be piecewise smooth on $\mathcal{M}$

➤ local quadrature for the approximate computation of the integral in (7.3.38) that involves $u_N^{(j-1)}(\boldsymbol{\Phi}_{t_j}^{-\tau}\boldsymbol{x})$ is not possible, because accurate numerical quadrature requires a (locally) smooth integrand.



Idea:

✦ replace $u_N^{(j-1)}(\mathbf{y}_{\boldsymbol{x}}(-\tau))$ with linear interpolant (→ Def. 5.3.18)

$$\mathsf{I}_1\big(u_N^{(j-1)} \circ \boldsymbol{\Phi}_{t_j}^{-\tau}\big) \in \mathcal{S}_{1,0}^0(\mathcal{M}),$$

✦ approximate $\boldsymbol{\Phi}_{t_j}^{-\tau}\boldsymbol{x}$ by $\boldsymbol{x} - \tau\mathbf{v}(\boldsymbol{x}, t_j)$ (explicit Euler). ("streamline backtracking")

$$u_N^{(j)} \in \mathcal{S}_{1,0}^0(\mathcal{M}): \quad \int_\Omega \frac{u_N^{(j)}(\boldsymbol{x}) - \mathsf{I}_1(u_N^{(j-1)}(\cdot - \tau\mathbf{v}(\cdot,t_j)))(\boldsymbol{x})}{\tau} v_N(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \epsilon \int_\Omega \mathbf{grad}\, u_N^{(j)} \cdot \mathbf{grad}\, v_N\,\mathrm{d}\boldsymbol{x}$$
$$= \int_\Omega f(\boldsymbol{x},t_j) v_N(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \quad \forall v_N \in \mathcal{S}_{1,0}^0(\mathcal{M})\,.$$

Then apply local vertex based numerical quadrature (2D trapezoidal rule (3.3.49) = global trapezoidal rule) to the first integral. This amounts to using mass lumping, see Rem. 6.2.45.

Implementable version of (7.3.38):

$$u_N^{(j)} \in \mathcal{S}_{1,0}^0(\mathcal{M}): \quad \tfrac{1}{3}|U_{\boldsymbol{p}}|(\mu_{\boldsymbol{p}}^{(j)} - u_N^{(j-1)}(\boldsymbol{p} - \tau\mathbf{v}(\boldsymbol{p},t_j)) + \tau \int_\Omega \mathbf{grad}\, u_N^{(j)} \cdot \mathbf{grad}\, b_N^{\boldsymbol{p}}\,\mathrm{d}\boldsymbol{x}$$
$$= \tfrac{1}{3}|U_{\boldsymbol{p}}| f(\boldsymbol{p})\,, \quad \boldsymbol{p} \in \mathcal{N}(\mathcal{M}) \cap \Omega\,, \qquad (7.3.39)$$

where $\mu_{\boldsymbol{p}}^{(j)}$ are the nodal values of $u_N^{(j)} \in \mathcal{S}_{1,0}^0(\mathcal{M})$ associated with the interior nodes of the mesh $\mathcal{M}$, $b_N^{\boldsymbol{p}}$ is the "tent function" belonging to node $\boldsymbol{p}$, $|U_{\boldsymbol{p}}|$ is the sum of the areas of all triangles adjacent to $\boldsymbol{p}$.

Example 7.3.40 (Semi-Lagrangian method for convection-diffusion in 1D)

Same IBVP as in Ex. 7.3.30

- Linear finite element Galerkin discretization with mass lumping in space
- Semi-Lagrangian method: 1D version of (7.3.38)
- Explicit Euler streamline backtracking

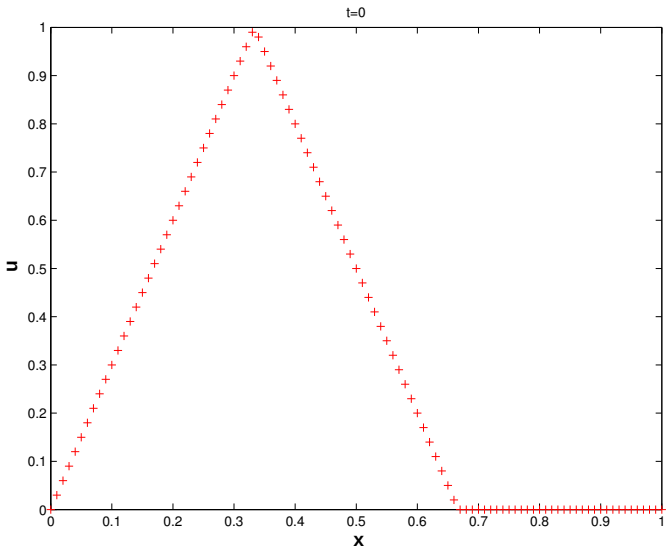


Fig. 332

$\epsilon = 10^{-5}$:

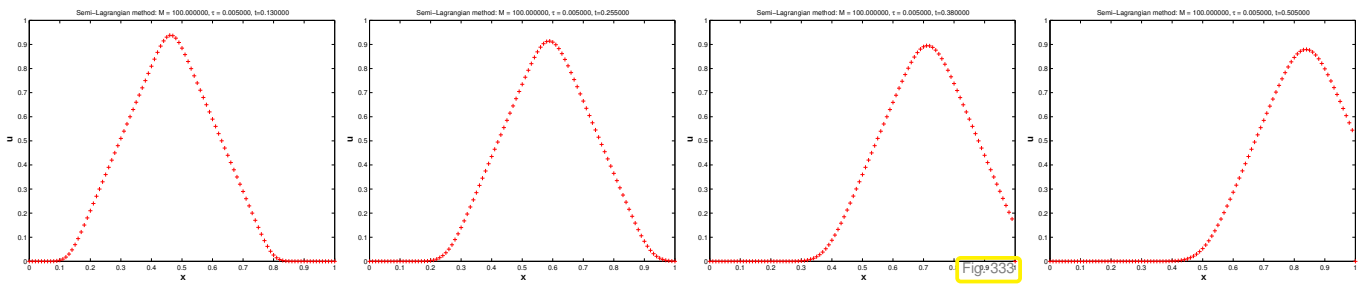


Fig. 333

$\epsilon = 0.1$:

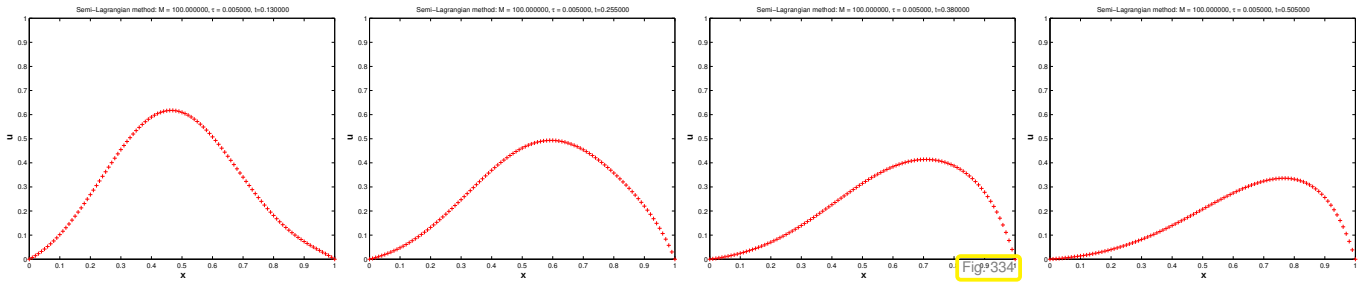


Fig. 334

"Reference solution" computed by method of lines, see Ex. 7.3.4, with $h = 10^{-3}$, $\tau = 5 \cdot 10^{-5}$:

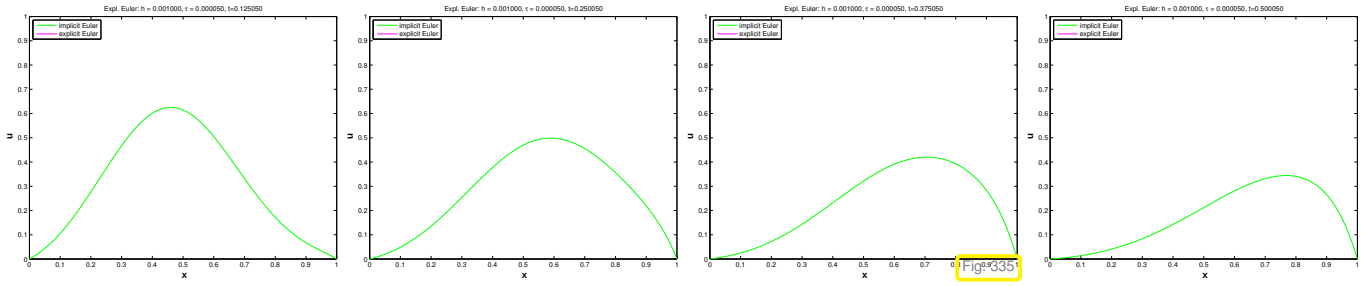


Fig. 335

Example 7.3.41 (Semi-Lagrangian method for convection-diffusion in 2D)

- 2nd-order scalar convection diffusion problem (7.3.2), $\Omega :=]0,1[^2$, $f = 0$, $g = 0$,
- velocity field

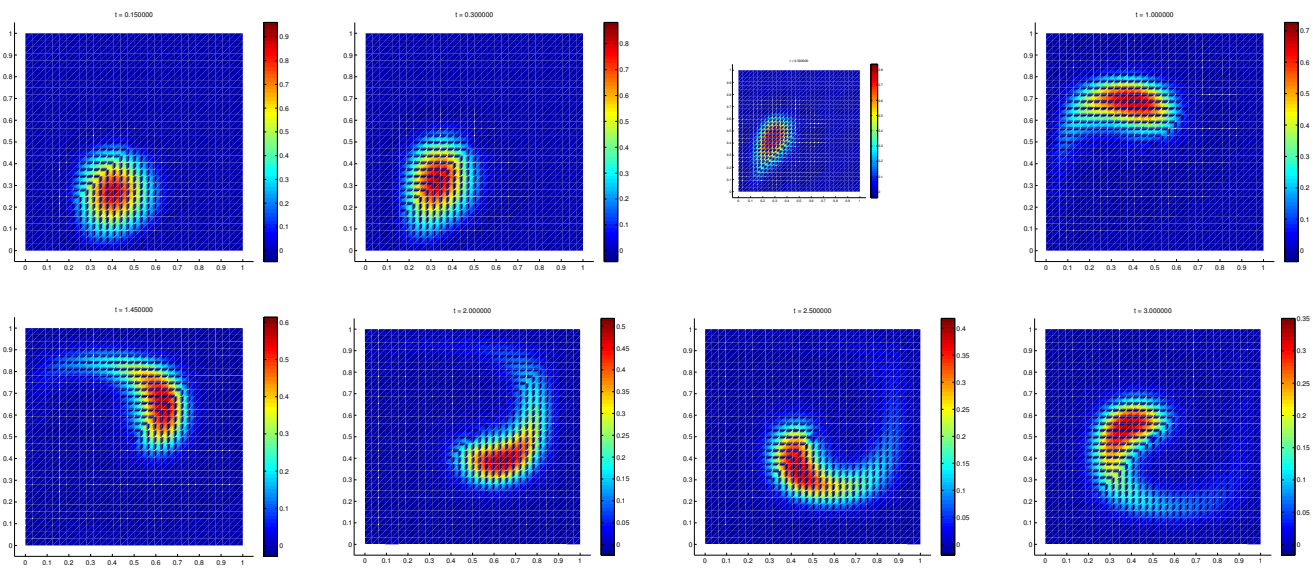
$$\mathbf{v}(\boldsymbol{x}) := \begin{pmatrix} -\sin(\pi x_1)\cos(\pi x_2) \\ \sin(\pi x_2)\cos(\pi x_1) \end{pmatrix} .$$

- Initial condition: "compactly supported cone shape"

```
u0(x) = max(0,1-4*sqrt((x(:,1)-0.5).^2+(x(:,2)-0.25).^2));
```

- semi-Lagrangian finite element Galerkin discretization according to (7.3.38) on regual triangular meshes of square domain Ω, see Fig. 178.

Example with $\epsilon = 0$:



We observe smearing of initial data due to numerical diffusion inherent in the interpolation step of the semi-Lagrangian method.

Learning outcomes

After having digested the contents of this chapter you should

- know the mathematical model ("convection-diffusion equation") for stationary and transient heat conduction in a moving (incompressible) fluid,
- understand the notion of *singular perturbation* and when convection-diffusion boundary value problems are singularly perturbed.
- know that standard Galerkin finite element discretization of convection-diffusion boundary value problem runs risk of spurious oscillations of the numerical solution in the case of singular perturbation.
- be familiar with the idea of *upwind quadrature* for a stable discretization of singularly perturbed convection-diffusion problems.
- know stabilization through*artificial diffusion/viscosity* and how it is used in the streamline diffusion method.
- remember that the method of lines approach for singularly perturbed transient convection-diffusion problems requires a stable discretization in space.
- comprehend the main idea of Lagrangian particle methods for transient advection.
- be familiar with the principle of semi-Lagrangian finite element methods for transient advection-diffusion boundary value problems.

Bibliography

[1] A. Burtscher, E. Fonn, P. Meury, and C. Wiesmayr. *LehrFEM - A 2D Finite Element Toolbox*. SAM, ETH Zürich, Zürich, Switzerland, 2010. http://www.sam.math.ethz.ch/ hiptmair/tmp/LehrFEMManual.pdf.

[2] L.C. Evans. *Partial differential equations*, volume 19 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 1998.

[3] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[4] T. N. Phillips and A. J. Williams. Conservative semi-Lagrangian finite volume schemes. *Numer. Methods Partial Differential Equations*, 17(4):403–425, 2001.

[5] H.-G. Roos, M. Stynes, and L. Tobiska. *Numerical methods for singularly perturbed differential equations. Convection-diffusion-reaction and flow problems*, volume 24 of *Springer Series in Computational Mathematics*. Springer, Berlin, 2nd edition, 2008.

[6] M. Struwe. Analysis für Informatiker. Lecture notes, ETH Zürich, 2009. https://moodle-app1.net.ethz.ch/lms/mod/resource/index.php?id=145.

[7] J. Xu and L. Zikatanov. A monotone finite element scheme for convection diffusion equations. *Math. Comp.*, 68(228):1429–1446, May 1999.

Chapter 8

Numerical Methods for Conservation Laws

Conservation laws describe physical phenomena governed by

- *conservation* laws for certain physical quantities (e.g., mass momentum, energy, etc.),
- *transport* of conserved physical quantities.

We have already examined problems of this type in connection with transient heat conduction in Sect. 7.1.4. There thermal energy was the conserved quantity and a *prescribed* external velocity field $\mathbf{v}$ determined the transport. Familiarity with Chapter 7 is advantageous, but not essential for understanding this chapter.

A new aspect emerging for general conservation laws is that the transport velocity itself may depend on the conserved quantities themselves, which gives rise to *non-linear models*.

Contents

8.1 Conservation laws: Examples

Focus:

Cauchy problems

Spatial domain $\Omega = \mathbb{R}^d$ (unbounded!)

➤ Cauchy problems are pure initial value problems (no boundary values).

Why do we restrict ourselves to Cauchy problems **?**

❶ *Finite speed of propagation* typical of conservation laws $\rightarrow$ Thm. 8.2.43

(Potential spatial boundaries will not affect the solution for some time in the case of compactly supported initial data, *cf.* situation for wave equation, where we also examined the Cauchy problem, see (6.2.21).)

❷ No spatial boundary ➢ need not worry about (spatial) boundary conditions!

(Issue of spatial boundary conditions can be very intricate for conservation laws, *cf.* Rem. 8.2.6)

8.1.1 Linear advection

The simplest case are models where transport governed by a given velocity field (advection/convection $\rightarrow$ Chapter 7).

(8.1.1) Heat transport in a moving fluid

A typical specimen is the following Cauchy problem for the linear transport equation (advection equation) $\rightarrow$ Sect. 7.1.4, (7.1.16):

$$\frac{\partial}{\partial t}(\rho u) + \operatorname{div}(\mathbf{v}(\boldsymbol{x},t)(\rho u)) = f(\boldsymbol{x},t) \quad \text{in} \quad \widetilde{\Omega} := \mathbb{R}^d \times]0,T[\,, \tag{8.1.2}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \text{for all} \quad \boldsymbol{x} \in \mathbb{R}^d \quad \text{(initial conditions)}\,. \tag{8.1.3}$$

$u = u(\boldsymbol{x},t) \mathrel{\hat{=}}$ temperature, $\rho > 0 \mathrel{\hat{=}}$ heat capacity, $\mathbf{v} = \mathbf{v}(\boldsymbol{x},t) \mathrel{\hat{=}}$ prescibed locally Lipschitz-continuous velocity field, $\mathbf{v} : \mathbb{R}^d \times [0,T] \rightarrow \mathbb{R}^d$.

(8.1.2) = linear scalar conservation law

➤ Conserved quantity: thermal energy (density) ρu
(Recall the derivation of (7.1.16) through conservation of energy, *cf.* (6.1.3).)

Simplfied problem: assume constant heat capacity $\rho \equiv 1$, no sources: $f \equiv 0$, stationary velocity field $\mathbf{v} = \mathbf{v}(\boldsymbol{x})$ ➢ rescaled initial value problem *written in conserved variables*

$$\boxed{\begin{aligned} &\frac{\partial u}{\partial t} + \operatorname{div}(\mathbf{v}(\boldsymbol{x})u) = 0 \quad \text{in} \quad \widetilde{\Omega} := \mathbb{R}^d \times]0,T[\,, \\ &u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \text{for all} \quad \boldsymbol{x} \in \mathbb{R}^d \quad \text{(initial conditions)}\,. \end{aligned}} \tag{8.1.4}$$

Convention: differential operator div acts on spatial independent variable only,

$$(\operatorname{div} \mathbf{f})(\boldsymbol{x},t) := \frac{\partial f_1}{\partial x_1} + \cdots + \frac{\partial f_d}{\partial x_d}\,, \quad \mathbf{f}(\boldsymbol{x},t) = \begin{bmatrix} f_1(\boldsymbol{x},t) \\ \vdots \\ f_d(\boldsymbol{x},t) \end{bmatrix}.$$

A general solution formula exists for (8.1.4), based on the notion of the flow map induced by the velocity field $\mathbf{v} = \mathbf{v}(\boldsymbol{x})$, see also (7.1.3). The flow map $\boldsymbol{\Phi} = \boldsymbol{\Phi}(\boldsymbol{x},t)$, $\boldsymbol{x} \in \mathbb{R}^d$, $t \in \mathbb{R}$ is a mapping

$$\boldsymbol{\Phi} : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d \quad \text{defined by} \quad \begin{aligned} &\tfrac{\partial \boldsymbol{\Phi}}{\partial t}(\boldsymbol{x},t) = \mathbf{v}(\boldsymbol{\Phi}(\boldsymbol{x},t)) \text{ in } \mathbb{R}^d \times \mathbb{R}\,, \\ &\boldsymbol{\Phi}(\boldsymbol{x},0) = \boldsymbol{x} \quad \text{for all } \boldsymbol{x} \in \mathbb{R}^d\,. \end{aligned} \tag{8.1.5}$$

See Fig. 296, Fig. 297, and Fig. 298 for visualizations. By existence and uniqueness theorems for initial value problems for ordinary differential equations [33, Thm. 11.1.32] the flow map $\boldsymbol{\Phi}$ is well defined by (8.1.5), if $\mathbf{v}$ is (locally) Lipschitz continuous, which we take or granted. The flow map satisfies

$$\boldsymbol{\Phi}(\boldsymbol{\Phi}(\boldsymbol{x},t),-t) = \boldsymbol{x} \quad \text{for all } \boldsymbol{x} \in \mathbb{R}^d\,. \tag{8.1.6}$$

Theorem 8.1.7. Solution of linear advection problem

The solution of (8.1.4) *is given by*

$$u(\boldsymbol{x},t) = |\det(\mathrm{D}_{\boldsymbol{x}}\boldsymbol{\Phi})(\boldsymbol{x},t)|^{-1} u_0(\boldsymbol{\Phi}(\boldsymbol{x},-t))\,, \quad (\boldsymbol{x},t) \in \mathbb{R}^d \times \mathbb{R}\,, \tag{8.1.8}$$

where $\mathrm{D}_{\boldsymbol{x}}\boldsymbol{\Phi}$ *is the Jacobian of the flow map.*

Example 8.1.9 (Constant advection in 1D)

Special case: constant coefficient linear advection in 1D

- ✦ $d = 1$ ➢ (Cauchy problem) spatial domain $\Omega = \mathbb{R}$,
- ✦ constant velocity $v = \text{const.}$.

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(vu) = 0 \quad \text{in} \quad \widetilde{\Omega} = \mathbb{R} \times]0,T[\,, \quad u(x,0) = u_0(x) \quad \forall x \in \mathbb{R}\,. \tag{8.1.10}$$

This is the 1D version of the transport equation (7.3.7) and its solution is given by formula (7.3.11), which is a special case of the result stated in Thm. 8.1.7.

$$\overset{(7.3.11)}{\blacktriangleright} \quad u(x,t) = u_0(x - vt)\,, \quad x \in \mathbb{R}\,, \quad 0 \le t < T\,. \tag{8.1.11}$$

Solution $u = u(x,t)$ = initial data "travelling" with velocity v. For differentiable u_0 the solution property of $u(x,t)$ from (8.1.11) can be verified by direct computation.

Remark 8.1.12 (Discontinuous solutions of advection equations)

To verify that (8.1.11) solves (8.1.10) in the sense of classical calculus we need $u_0 \in C^1(\mathbb{R})$. However, (8.1.10) remains meaningful even without this smoothness assumption.

The solution formula from Thm. 8.1.7 makes perfect sense even for *discontinuous* initial data u_0!

➥ We should not expect $u = u(x,t)$ to be differentiable in space or time.
A "weaker" concept of solution is required, see Section 8.2.3 below.

This consideration should be familiar: for second order elliptic boundary value problems, for which classical solutions are to be twice continuously differentiable, the concept of a variational solution made it possible to give a meaning to solutions $\in H^1(\Omega)$ that are merely continuous and piecwise differentiable, see Rem. 1.3.47.

Related to (8.1.11): d'Alembert solution formula (6.2.22) for 1D wave equation (6.2.21).

Remark 8.1.13 (Boundary conditions for linear advection)

Recall the discussion in Sects. 7.2.1, 7.3.2, *cf.* solution formula (7.3.12):

For the scalar linear advection initial boundary value problem

$$\frac{\partial u}{\partial t} + \operatorname{div}(\mathbf{v}(\boldsymbol{x},t)u) = f(\boldsymbol{x},t) \quad \text{in} \quad \widetilde{\Omega} := \Omega \times]0,T[\,, \tag{8.1.14}$$

$$u(\boldsymbol{x},0) = u_0(\boldsymbol{x}) \quad \text{for all} \quad \boldsymbol{x} \in \Omega\,, \tag{8.1.15}$$

on a bounded domain $\Omega \subset \mathbb{R}^d$, boundary conditions (e.g., prescribed temperature)

$$u(\boldsymbol{x},t) = g(\boldsymbol{x},t) \quad \text{on} \quad \Gamma_{\text{in}}(t) \times]0,T[\,,$$

can be imposed on the inflow boundary

$$\Gamma_{\text{in}}(t) := \{\boldsymbol{x} \in \partial\Omega\colon \mathbf{v}(\boldsymbol{x},t) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\}\,, \quad 0 < t < T\,. \tag{8.1.16}$$

Note: Γ_{in} can change with time!

Bottom line:

Knowledge of local and current direction of transport
needed to impose meaningful boundary conditions!

8.1.2 Traffic modeling [7]

We design simple mathematical models for non-stationary traffic flow on a *single long highway lane*. This situation often occurs, for instance, at bypasses of long highway construction sites.

We make simplifying *modeling assumptions* (not quite matching reality):

- Identical cars and behavior of drivers (8.1.17)
- Uniformity of road conditions (8.1.18)
- Speed of a car determined only by (its distance from) the car in front (8.1.19)

8.1.2.1 Particle model

The gist of a particle model or agent based model for traffic flow is to track a finite number of individual cars over a period of time $[0,T]$. Hence, the particle model is semi-discrete (still continuous in time). The key state parameter of a car is its position on the road:

$x_i(t) \hat{=}$ position of i-th car at time t, $i = 1,\dots,N$ ($N \hat{=}$ total number of cars), hence the configuration space is $\mathbb{R}^N$.

We will always take for granted ordering: $x_i(t) < x_{i+1}(t)$

The curve $t \mapsto x_i(t)$ in the $x-t$-plane is the trajectory of the i-th car.

(8.1.20) Velocity model

In order to describe the dynamics of the moving cars we need a *velocity model*.

Here: optimal velocity model

$$\boxed{\dot{x}_i(t) = v_{\text{opt}}(\Delta x_i) \quad , \quad \Delta x_i(t) = x_{i+1}(t) - x_i(t) > 0} \quad , \quad i = 1,\dots,N-1 \; . \tag{8.1.21}$$

$\leftrightarrow$ relies on Assumptions (8.1.17)–(8.1.19) above, in particular (8.1.19).

The function $\Delta x \mapsto v_{\text{opt}}(\Delta x)$ is deduced from the assumption that

each car drives as fast as possible under safety constraints.
(drive more slowly if the you are close to the car in front)

$$v_{\text{opt}}(\Delta x) = v_{\max}(1 - \frac{\Delta_0}{\Delta x}) \; , \tag{8.1.22}$$

with $\Delta_0 \hat{=}$ length of a car = distance of cars in bumper to bumper traffic jam.

(8.1.21) + (8.1.22): ordinary differential equation (ODE) on state space $\mathbb{R}^N$

In order to get a well-posed initial value problem, the ODE has to be supplemented with initial conditions

$$x_i(0) = x_{i,0} \in \mathbb{R} \quad , \quad x_{i,0} \leq x_{i+1,0} - \Delta_0 \; . \tag{8.1.23}$$

Obviously (why?): the solution of (8.1.21), (8.1.22), (8.1.23) satisfies $x_i(t) \leq x_{i+1}(t) - \Delta_0$.

Remark 8.1.24 (Acceleration based traffic modeling)

The speed of a car is a consequence of drivers accelerating and breaking.
➢ acceleration based modeling of car dynamics under Assumptions (8.1.17)–(8.1.19)

$$\ddot{x}_i(t) = F(\Delta x_i(t), \Delta v_i(t)) \quad , \quad \Delta v_i(t) = \dot{x}_{i+1} - \dot{x}_i \; . \tag{8.1.25}$$

Models of this type are popular in practice.

Experiment 8.1.26 (Particle simulation of traffic flow)

Usually one sets $v_{\max} = 1$ by rescaling of spatial/temporal units, *cf.* Rem. 1.2.10.

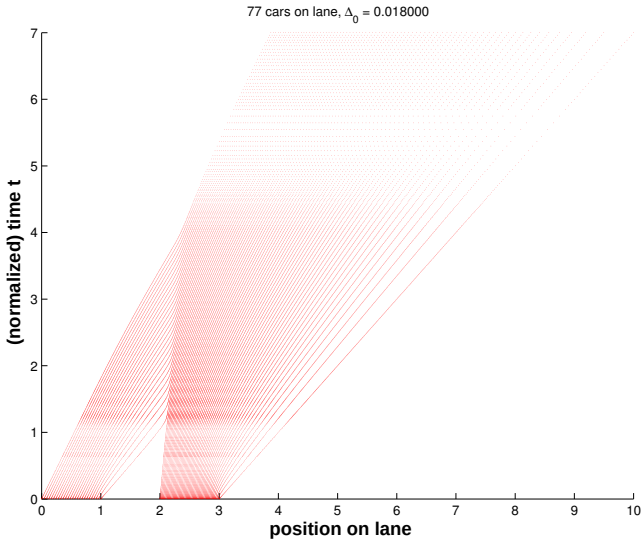


Fig. 336

We chose the following initial positions of cars (row vector in MATLAB syntax)

```
RowVectorXd x0(77);
x0 <<
   RowVectorXd::LinSpaced(26,0.0,1
      RowVectorXd::LinSpaced(51,2
```

This corresponds to two clusters of evenly spaced cars at different sections of the road.

◁ Simulation based on optimal velocity model (8.1.22) with (dimensionless) $\Delta_0 = 0.0180$.

MATLAB code 8.1.27: Particle simulation of cars based on optimal velocity model

```
function [times,Y,fig] = carsim(x0,T,xL,xR,d0)
% Particle simulation of single lane traffic flow using the normalization
% v_max = 1 and d_0 := Δ_0 = 1/N, where N is the
% total number of cars.  x0 passes the intial positions of the cars.
% This vector is assumed to be sorted with the last component providing the
% position of the rightmost car.

% Total number N of cars
N = length(x0); x0 = reshape(x0,N,1);
% Bumper to bumper distance d_0 of the cars
if (nargin < 5), d0 = (xR-xL)/(5*N); end
u0 = 1/d0; % Maximal number density of cars in a bumper to bumper jam

% Check validity of initial positions
dist0 = diff(x0);  % compute Δx_i
if (min(dist0) < 0.99*d0), d0, min(dist0), error('Cars too close'); end

```

```
% right hand side of the numerical integrator according to (8.1.21) and
% (8.1.22) with v_max = 1.  Note that x has to be a row
% vector.  The rightmost car travels at speed v_max.
rhs = @(t,x) [1-d0*1./diff(x);1];

% perform numerical integration using MATLAB's standard integrator
options = odeset('abstol',1E-8,'reltol',1E-7);
[times,X] = ode45(rhs,[0.0 T],x0,options);

% Compute density of cars normalized with the maximal density (Δ_0)^{-1},
% based on averages over N/5 equally long sections of the lane, that
% is δ = 5|x_R-x_L|/N in (8.1.30).
Y = []; M = floor(N/6);
for k=1:length(times)
  Y = [Y;cardensity(X(k,:),xL,xR,M)/u0];
end

% Plot positions of cars as a function of time ("fan plot")
fig = figure('name','positions of cars');
axis([xL xR 0 T]); hold on;
k = 1;
for t=times'
  plot(X(k,:),t*ones(N,1),'r.','markersize',1);
  k = k+1;
end
xlabel('{\bf position on lane}','fontsize',14);
ylabel('{\bf (normalized) time t}','fontsize',14);
title(sprintf('%d cars on lane, \\Delta_{0} = %f',N,d0));
hold off;

% (Animated) plot of normalized density of cars.  The times for the frames are
% stored in the vector times, the density data in the matrix Y.
figure('name','Animation of densities');
for k=1:length(times)
  stairs((xL:(xR-xL)/M:xR),Y(k,:),'m');
  axis([xL xR 0 1]);
  xlabel('{\bf position on lane}','fontsize',14);
  ylabel('{\bf density of cars}','fontsize',14);
  title(sprintf('%d cars on lane, time = %f',N,times(k)));
  drawnow;
end
```

C++ EIGEN code 8.1.28: Particle simulation of cars based on optimal velocity model
➙ GITLAB

```
// arguments:
// Vector x0 with initial positions of cars which is assumed to be sorted with the last component providing the position of the rightmost car.
```

```

// double T endtime of simulation
// doubles [x_L,x_R] to select spatial domain
// double d_0 := Δ_0 = 1/N, where d_0 is the bumper to bumper distance of the cars
//
// returns:
// Vector with timesteps
// Matrix with all carpositions at all timesteps
// Matrix with cardensities at all timesteps
// double d0 bumper to bumper distance of the cars
//
//
// Particle simulation of single lane traffic flow using the normalization
// v_max = 1
//
std::tuple<Eigen::VectorXd, Eigen::MatrixXd, Eigen::MatrixXd, double> carsim(const Eigen::VectorXd& x0, double T, double xL , double xR, double d0) {

  // Total number N of cars
  const unsigned N = x0.size();

  const double u0 = 1/d0; // Maximal number density of cars in a bumper to bumper jam

  // Check validity of initial positions
  const Eigen::VectorXd diff0 = NPDE::diff(x0); // compute Δx_i
  assert(diff0.minCoeff() > 0.99*d0); //Cars too close

  // right hand side of the numerical integrator according to (8.1.21) and
  // (8.1.22) with v_max = 1.  The rightmost car travels at speed v_max.
  auto rhs = [d0] (const Eigen::VectorXd& x, Eigen::VectorXd& dxdt, const double t) {
    const unsigned N = x.size();
    //Eigen::VectorXd dxdt(N);
    dxdt.resize(N);
    Eigen::VectorXd diffx = NPDE::diff(x);
    for (unsigned i = 0; i < N-1; ++i) {
      dxdt[i] = 1-d0/diffx[i];
    }
    dxdt[N-1] = 1;
  };

  // perform numerical integration using Boost's integrator
  double abstol = 1E-8;
  double reltol = 1E-7;

  //solve the system
  Eigen::VectorXd times;
```

```
  Eigen::MatrixXd X;
  std::tie(times, X) = NPDE::ode45(rhs, 0, T, x0, abstol, reltol);

  // Compute density of cars normalized with the maximal density (Δ0)^-1,
  // based on averages over N/5 equally long sections of the lane, that
  // is δ = 5|xR−xL|/N in (8.1.30).

  unsigned timesteps = times.size();
  unsigned M = N/6;
  Eigen::MatrixXd Y(M, timesteps);
  for (unsigned k = 0; k < timesteps; ++k) {
    Y.col(k) = cardensity(X, k, xL, xR, M)/u0;
  }

  return std::make_tuple(times, X, Y, d0);
}
```

When we launch the simulation we observe that the two clusters merge and dissolve as cars "escape" to the right. *Fan-shaped* patterns emerge, see Fig. 336.

(8.1.29) Extraction of macroscopic quantities

Our goal is to pass from the semi-discrete particle model to a *continuum model*, where the state of traffic is described by functions.

These correspond to "macroscopic quantities" $\hat{=}$ quantities describing the traffic flow detached from the existence of individual cars.

Macroscopic quantities can be obtained by *averaging* from the microscopic particle description.

Key macroscopic quantity: (normalized) density of cars

$$u_\delta(x,t) := \frac{\Delta_0}{2\delta}\,\sharp\{i \in \{1,\dots,N\} : \; x-\delta \le x_i(t) < x+\delta\}\,, \tag{8.1.30}$$

where $\delta > 0$ is the spatial averaging length. (The density defined in (8.1.30) is "normalized" because it is the ratio of the number density of cars and the maximal density Δ_0^{-1}. Hence, invariably, $0 \le u_\delta(x,t) \le 1$.)

Note: u_δ will crucially depend on δ

Experiment 8.1.31 (Particle simulation of traffic flow, cnt'd $\rightarrow$ Exp. 8.1.26)

We use initial car distribution

```
x0 = [0:2/k:1,2:1/k:3]a
```

(MATLAB syntax) with `k=50,200,800`, $\Delta_0 = 0.9/k$, see (8.1.22), $\delta = 3.33/k$ in (8.1.30). Simulation based on Code 8.1.27.

Initial density $x \mapsto u_\delta(x,0)$ ▷

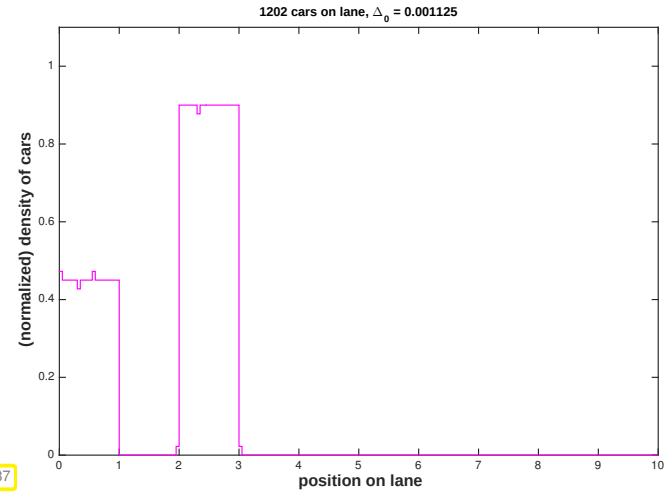
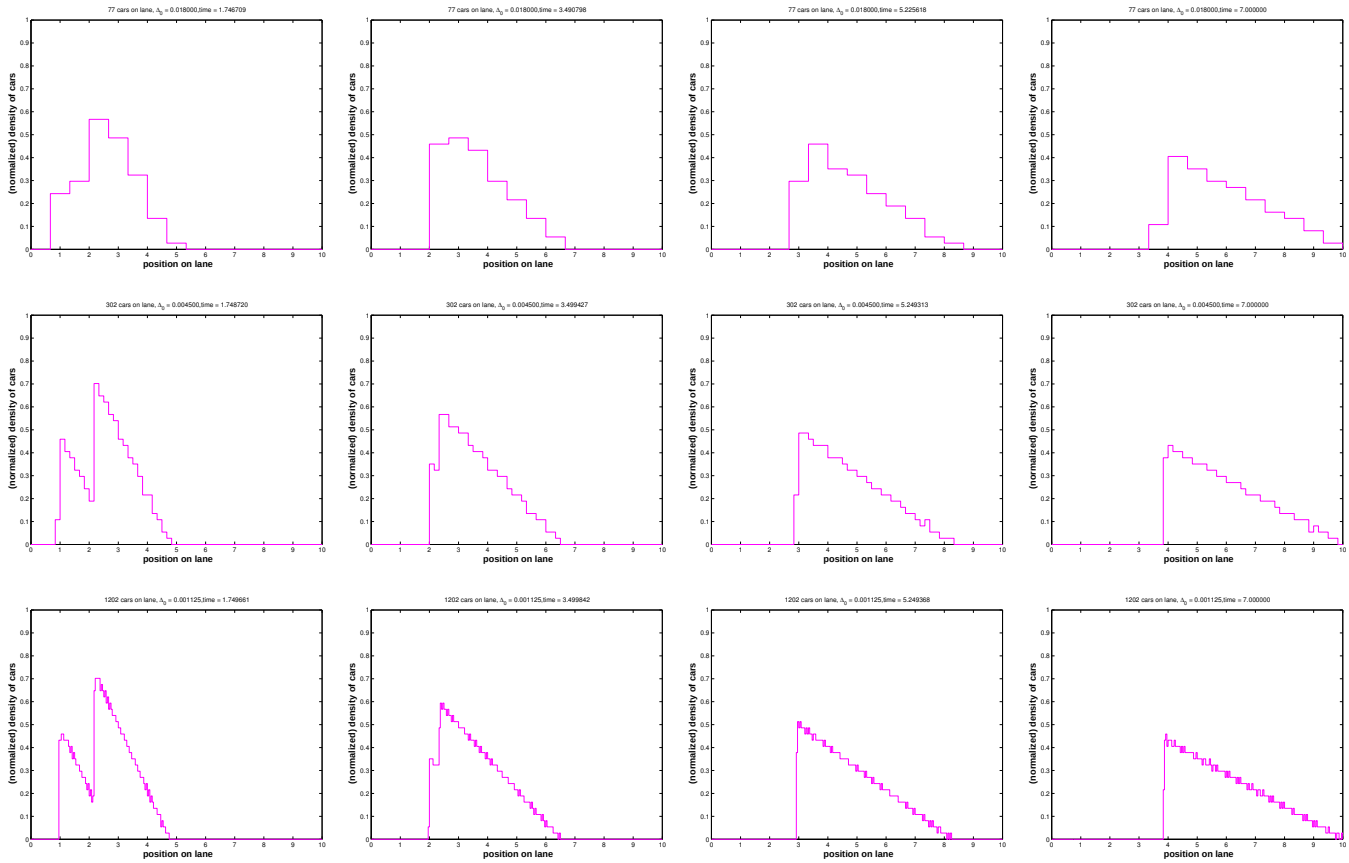


Fig. 337



Striking observation:

For $N \to \infty$, $\Delta_0 \sim N^{-1}$, $\delta \sim N^{-1}$ the normalized car densities $u_\delta(x,t)$ seem to approach a *limit density*. What is it? Can it be obtained as a solution of a "limit model". These issues will be addressed next.

Note: We have made similar observation in the case of the mass-spring model of Section 1.2.2 in the limit $n \to \infty$.

8.1.2.2 Continuum traffic model

In Exp. 8.1.31 we observed the emergence of a stable limit density in the microscopic particle model of traffic flow according to (8.1.21) and (8.1.22), when the number of cars and their maximum density tended to ∞ in tandem, while the spatial averaging length tends to zero.

Now we derive a macroscopic continuum model describing this limit. This macroscopic model will be stated in terms of macroscopic quantities, which are functions of position along the road x and time t.

Note: There are many parallels with derivation of continuum elastic string model in Section 1.2.

Remark 8.1.32 (Suitability of macroscopic models for traffic flow)

The limit $N \to \infty$ in traffic modeling is commonly denounced as dubious, because the number of cars on a road is way too small to render the limit a good approximation of actual traffic flow, see [7, Sect. 2.3].

Nevertheless, here we introduce a limit model, because

- ✦ it yields a least a qualitatively correct representation of patterns observed in real traffic flow,
- ✦ it provides an important model problem for scalar non-linear conservation laws, see Section 8.1.3.

Ingredients of macroscopic (continuum) traffic model:

- spatial domain $\Omega = \mathbb{R} \mathrel{\hat{=}}$ infinitely long single highway lane ($\to$ Cauchy problem),
- traffic flow described by the macroscopic quantity

  normalized density of cars $u : \Omega \times [0,T] \mapsto [0,1]$ according to

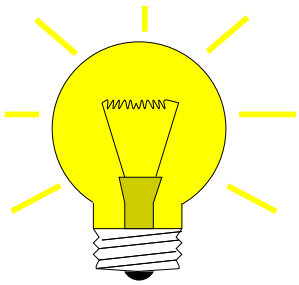
$$u_\delta(x,t) := \frac{\Delta_0}{\delta} \sharp\{i \in \{1,\dots,N\} : \; x-\delta \le x_i(t) < x+\delta\} \; , \tag{8.1.30}$$

- optimal velocity speed model (8.1.22) $\left(v_{\mathrm{opt}}(\Delta x) = v_{\max}(1 - \frac{\Delta_0}{\Delta x})\right)$.

(8.1.33) Macroscopic balance laws for traffic model

However, (8.1.22) and (8.1.21) do not fit the spirit of macroscopic modeling: neither Δx_i nor $\dot{x}_i(t)$ is a macroscopic quantity!

Required: concept of a macroscopic velocity



Idea: spatial averaging of velocities of cars

$$\boxed{v_\delta(x,t) = \frac{\sum_{i \in \mathcal{U}_\delta(x)} \dot{x}_i(t)}{\sharp \mathcal{U}_\delta}(x)} \; , \tag{8.1.34}$$

$$\mathcal{U}_\delta(x) := \{i \in \{1,\dots,N\} : \; x-\delta \le x_i(t) < x+\delta\} \; .$$

► From density and velocity we derive another macroscopic quantity:

(normalized) flux of cars: $q_\delta(x,t) = u_\delta(x,t) v_\delta(x,t)$. (8.1.35)

Interpretation: $q(x,t) \approx$ no. of cars passing site x in unit time around instance t in time.

approximate balance law
("conservation of cars" in a "space-time box")

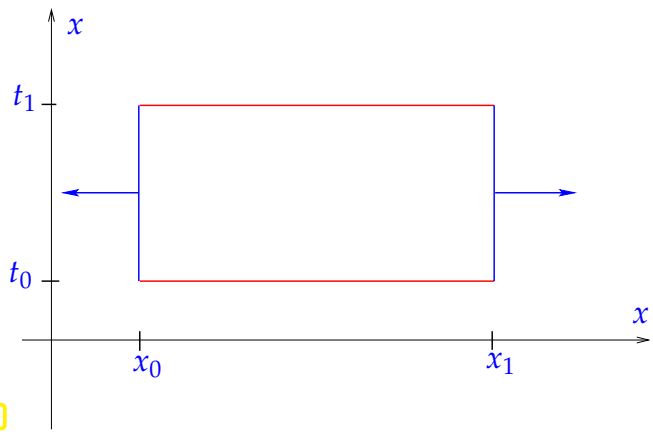


Fig. 338

$$\underbrace{\int_{x_0}^{x_1} u_\delta(x,t_1)\,\mathrm{d}t - \int_{x_1}^{x_1} u_\delta(x,t_0)\,\mathrm{d}t}_{\text{change of no. of cars on } [x_0,x_1] \text{ in } [t_0,t_1]} \approx \underbrace{\int_{t_0}^{t_1} q_\delta(x_0,t)\,\mathrm{d}x - \int_{t_0}^{t_1} q_\delta(x_1,t)\,\mathrm{d}x}_{\text{no. of cars entering/leaving } [x_0,x_1] \text{ in } [t_0,t_1]} \quad . \tag{8.1.36}$$

(8.1.37) Traffic flow: continuum limit of particle model

Now we consider $N \to \infty$ (many cars) and $\delta \sim N^{-1} \to 0$ and drop the subscript δ, which hints at the averaging.

The balance law (8.1.38) will remain valid in the limit and will even become exact **!**

$$\underbrace{\int_{x_0}^{x_1} u(x,t_1)\,\mathrm{d}t - \int_{x_1}^{x_1} u(x,t_0)\,\mathrm{d}t}_{\text{change of no. of cars on } [x_0,x_1] \text{ in } [t_0,t_1]} = \underbrace{\int_{t_0}^{t_1} q(x_0,t)\,\mathrm{d}x - \int_{t_0}^{t_1} q(x_1,t)\,\mathrm{d}x}_{\text{no. of cars entering/leaving } [x_0,x_1] \text{ in } [t_0,t_1]} \quad . \tag{8.1.38}$$

In the "infinitely many cars" limit $u(x,t)$, $v(x,t)$, and $q(x,t)$ can be expected to become (piecewise) smooth functions. This justifies the transition to a differential (PDE) macroscopic model:

Temporarily assume that $u = u(x,t)$ is smooth in both x and t and set $x_1 = x_0 + h$, $t_1 = t_0 + \tau$. First approximate the integrals in (8.1.38).

$$\int_{x_0}^{x_1} u(x,t_1) - u(x,t_0)\,\mathrm{d}x = h(u(x_0,t_1) - u(x_0,t_0)) + O(h^2) \quad \text{for} \quad h \to 0\,,$$

$$\int_{t_0}^{t_1} q(x_1,t) - q(x_0,t)\,\mathrm{d}t = \tau(q(x_1,t_0) - q(x_0,t_0)) + O(\tau^2) \quad \text{for} \quad \tau \to 0\,.$$

Then employ Taylor expansion for the differences:

$$u(x_0,t_1) - u(x_0,t_0) = \frac{\partial u}{\partial t}(x_0,t_0)\tau + O(\tau^2) \quad \text{for} \quad \tau \to 0\,,$$

$$q(x_1,t_0) - q(x_0,t_0) = \frac{\partial q}{\partial x}(x_0,t_0)h + O(h^2) \quad \text{for} \quad h \to 0\,.$$

Finally, divide by h and τ and take the limit $\tau \to 0$, $h \to 0$:

$$\frac{\partial u}{\partial t}(x,t) + \frac{\partial q}{\partial x}(x,t) = 0 \quad \text{in } \Omega \times]0,T[\,. \tag{8.1.39}$$

This is a first-order partial differential equation.

We still need to link u and q: From (8.1.22) (with $v_{\max} = 1$ after rescaling) we deduce the macroscopic constitutive relationship between the (averaged and normalized) density ($\to$ (8.1.30)) of cars and their averaged speed ($\to$ (8.1.34)):

$$v(x,t) = 1 - u(x,t) \overset{(8.1.35)}{\Rightarrow} q(x,t) = u(x,t)(1-u(x,t))\,. \tag{8.1.40}$$

(8.1.39) & (8.1.40) & (8.1.35)

$$\boxed{\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(u(1-u)) = 0 \quad \text{in } \Omega \times]0,T[}\,. \tag{8.1.41}$$

+ macroscopic counterpart of initial conditions (8.1.23):

$$u(x,0) = u_0(x)\,, \quad x \in \mathbb{R}\,. \tag{8.1.42}$$

8.1.3 Inviscid gas flow

Introduction. In this section we study modeling in fluid mechanics, a special field of continuum mechanics. In spirit this is close to the modeling of trafic flow in Sect. 8.1.2, because the macroscopic behavior of fluids also results from the interaction of many small particles (molecules). However, in fluid mechanics the limit model for infinitely many particles enjoys a much more solid foundation than that for traffic, because the number of particles involved is tremendous ($\approx 10^{20} - 10^{30}$).

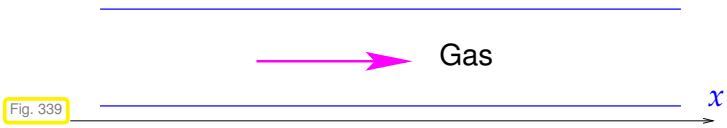


Fig. 339

Frictionless gas flow in (infinitely) long pipe

Terminology: frictionless $\hat{=}$ inviscid

Assumption:	variation of gas density negligible ("near incompressibility")
	motion of fluid driven by inertia $\leftrightarrow$ *conservation of linear momentum*

(8.1.43) Inviscid gas flow: balance law

We derive a continuum model for inviscid, nearly incompressible fluid in a straight infinitely long pipe $\leftrightarrow$ $\Omega = \mathbb{R}$ (Cauchy problem).

This simple model will be based on *conservation of linear momentum*, whereas conservation of mass and energy will be neglected (and violated). Hence, the crucial conserved quantity will be the momentum.

by near incompressibility

Unkown: $u = u(x,t)$ = momentum *density* $\sim$ local velocity $v = v(x,t)$ of fluid

Conserved quantity:	(linear) momentum of fluid $u = u(x,t)$

➢ flux of linear momentum $f \sim v \cdot u$ (after scaling: $f(u) = \frac{1}{2}u \cdot u$)
("momentum u advected by velocity u")

Conservation of linear momentum ($\sim u$): for all control volumes $V :=]x_0, x_1[\subset \Omega$:

$$\underbrace{\int_{x_0}^{x_1} u(x,t_1) - u(x,t_0)\,\mathrm{d}x}_{\text{change of momentum in } V} + \underbrace{\int_{t_0}^{t_1} \tfrac{1}{2}u^2(x_1,t) - \tfrac{1}{2}u^2(x_0,t)\,\mathrm{d}t}_{\text{outflow of momentum}} = 0 \quad \forall 0 < t_0 < t_1 < T\,. \tag{8.1.44}$$

(8.1.45) Burgers equation modelling inviscid gas flow

Temporarily assume that $u = u(x,t)$ is smooth in both x and t and set $x_1 = x_0 + h$, $t_1 = t_0 + \tau$. First approximate the integrals in (8.1.44).

$$\int_{x_0}^{x_1} u(x,t_1) - u(x,t_0)\,\mathrm{d}x = h(u(x_0,t_1) - u(x_0,t_0)) + O(h^2) \quad \text{for} \quad h \to 0\,,$$

$$\int_{t_0}^{t_1} \tfrac{1}{2}u^2(x_1,t) - \tfrac{1}{2}u^2(x_0,t)\,\mathrm{d}t = \tau(\tfrac{1}{2}u^2(x_1,t_0) - \tfrac{1}{2}u^2(x_0,t_0)) + O(\tau^2) \quad \text{for} \quad \tau \to 0\,.$$

Then employ Taylor expansion for the differences:

$$u(x_0,t_1) - u(x_0,t_0) = \frac{\partial u}{\partial t}(x_0,t_0)\tau + O(\tau^2) \quad \text{for} \quad \tau \to 0\,,$$

$$\tfrac{1}{2}u^2(x_1,t_0) - \tfrac{1}{2}u^2(x_0,t_0) = \frac{\partial}{\partial x}(\tfrac{1}{2}u^2)(x_0,t_0)h + O(h^2) \quad \text{for} \quad h \to 0\,.$$

Finally, divide by h and τ and take the limit $\tau \to 0$, $h \to 0$:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\tfrac{1}{2}u^2\right) = 0 \quad \text{in } \Omega \times]0,T[\,. \tag{8.1.46}$$

(8.1.46) = Burgers equation: a one-dimensional scalar conservation law (without sources)

Remark 8.1.47 (Euler equations)

The above gas model blatantly ignores the fundamental laws of conservation of mass and of energy. These are taken into account in a famous more elaborate model of inviscid fluid flow:

Euler equations [15], a more refined model for inviscid gas flow in an infinite pipe

$$\frac{\partial}{\partial t}\begin{pmatrix}\rho\\ \rho u\\ E\end{pmatrix} + \frac{\partial}{\partial x}\begin{pmatrix}\rho u\\ \rho u^2 + p\\ (E+p)u\end{pmatrix} = 0 \quad \text{in} \quad \mathbb{R} \times]0,T[\,, \tag{8.1.48}$$

$$u(x,0) = u_0(x) \quad , \quad \rho(x,0) = \rho_0(x) \quad , \quad E(x,0) = E_0(x) \quad \text{for } x \in \mathbb{R}\,,$$

where
- ✦ $\rho = \rho(x,t) \hat{=}$ fluid density, $[\rho] = \mathrm{kg\,m^{-1}}$,
- ✦ $u = u(x,t) \hat{=}$ fluid velocity, $[u] = \mathrm{m\,s^{-1}}$,
- ✦ $p = p(x,t) \hat{=}$ fluid pressure, $[p] = \mathrm{N}$,
- ✦ $E = E(x,t) \hat{=}$ total energy density, $[E] = \mathrm{J\,m^{-1}}$.

+ state equation (material specific constitutive equations), e.g., for ideal gas

$$p = (\gamma - 1)(E - \tfrac{1}{2}\rho u^2)\,, \quad \text{with adiabatic index} \quad 0 < \gamma < 1\,.$$

Conserved quantities (densities):

$$\rho \quad \leftrightarrow \quad \text{mass density} \quad , \quad \rho u \quad \leftrightarrow \quad \text{momentum density} \quad , \quad E \quad \leftrightarrow \quad \text{energy density.}$$

Underlying physical conservation principles for individual densities:

- First equation $\dfrac{\partial \rho}{\partial t} + \dfrac{\partial}{\partial x}(\rho u) = 0 \quad \leftrightarrow$ *conservation of mass*,
- Second equation $\dfrac{\partial (\rho u)}{\partial t} + \dfrac{\partial}{\partial x}(\rho u^2 + p) = 0 \quad \leftrightarrow$ *conservation of momentum*,
- Third equation $\dfrac{\partial E}{\partial t} + \dfrac{\partial}{\partial x}((E+p)u) = 0 \quad \leftrightarrow$ *conservation of energy*.

Euler equations (8.1.48) = non-linear system of conservation laws (in 1D)

As is typical of non-linear systems of conservations laws, the analysis of the Euler equations is intrinsically difficult: hitherto not even existence and uniqueness of solutions for general initial values could be established. Moreover, solutions display a wealth of complicated structures. Therefore, this course is confined to scalar conservation laws, for which there is only one unknown real-valued function of space and time.

?! *Review question(s) 8.1.49.* (Conservation based transport problems)

1. Consider the Cauchy problem (8.1.4) for linear advection for $d = 2$ and the velocity field $\mathbf{v}(\boldsymbol{x}) = \begin{bmatrix} -x_2 \\ x_1 \end{bmatrix}$. Write down the solution $u = u(\boldsymbol{x}, t)$ in terms of the initial data $u_0 = u_0(\boldsymbol{x})$.
2. Show that your solution $u = u(\boldsymbol{x}, t)$ satisfies $\frac{\partial u}{\partial t} + \mathrm{div}_{\boldsymbol{x}}(\mathbf{v}u) = 0$ in the sense of classical calculus, if u_0 is continuously differentiable.
3. In an $x - t$ diagram sketch the trajectory of a car starting at $t = 0$, $x = 0$ and moving with *constant acceleration* to right.
4. Which traffic flow conservation law arises, when (8.1.22) is replaced with $v_{\mathrm{opt}}(\Delta x) = v_{\max} \cos(\frac{\pi}{2}\frac{\Delta_0}{\Delta x})$.

8.2 Scalar conservation laws in 1D

8.2.1 Integral and differential form

What we have seen so far (except for Euler's equations in Rem. 8.1.47)

$$\text{Burgers equation:} \quad \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\tfrac{1}{2}u^2\right) = 0 \quad \text{in } \Omega \times]0, T[\,, \tag{8.1.46}$$

$$\text{traffic flow equation:} \quad \frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(u(1-u)) = 0 \quad \text{in } \Omega\times]0,T[\,, \tag{8.1.41}$$

$$\text{linear advection:} \quad \frac{\partial}{\partial t}(\rho u) + \operatorname{div}(\mathbf{v}(\boldsymbol{x},t)(\rho u)) = f(\boldsymbol{x},t) \quad \text{in} \quad \mathbb{R}^d\times]0,T[\,. \tag{8.1.2}$$

Now, we learn about a class of Cauchy problems to which these three belong. First some notations and terminology:

- $\Omega \subset \mathbb{R}^d \mathrel{\hat{=}}$ fixed (bounded/unbounded) spatial domain ($\Omega = \mathbb{R}^d$ = Cauchy problem)
- computational domain: space-time cylinder $\widetilde{\Omega} := \Omega\times]0,T[$, $T > 0$ final time
- $U \subset \mathbb{R}^m$ ($m \in \mathbb{N}$) $\mathrel{\hat{=}}$ phase space (state space) for conserved quantitities u_i (usually $U = \mathbb{R}^m$)
  ➤ A vector $\in U$ is often called a state.

Our focus below: *scalar* case $m = 1$

Conservation law for transient state distribution $u : \widetilde{\Omega} \mapsto U$: $u = u(\boldsymbol{x},t)$, for $0 \le t \le T$

$$\frac{d}{dt}\int_V u\,\mathrm{d}\boldsymbol{x} + \int_{\partial V} \mathbf{f}(u,\boldsymbol{x})\cdot\mathbf{n}\,\mathrm{d}S(\boldsymbol{x}) = \int_V s(u,\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x} \quad \forall \text{ "control volumes" } V \subset \Omega\,. \tag{8.2.1}$$

change of amount — inflow/outflow — production term

Terminology:
- ▷ flux function $\mathbf{f} : U \times \Omega \mapsto \mathbb{R}^d$
- ▷ source function $s : U \times \Omega\times]0,T[\mapsto \mathbb{R}$ (here usually $s = 0$)

- For Burgers equation (8.1.46): $f(u,x) = f(u) = \frac{1}{2}u^2$, $s = 0$,
- For traffic flow equation (8.1.41): $f(u,x) = f(u) = u(1-u)$, $s = 0$,
- For linear advection (8.1.2): $\mathbf{f}(u,\boldsymbol{x}) = \mathbf{v}(\boldsymbol{x},t)u$, $s = f(\boldsymbol{x},t)$
  (Note: in this case the conserved quantity is actually ρu, which was again denoted by u)

☞ (8.2.1) has the same structure as the "conservation of energy law" (6.1.3) for heat conduction.

Conservation of energy:

$$\frac{d}{dt}\int_V \rho u\,\mathrm{d}\boldsymbol{x} + \int_{\partial V} \mathbf{j}\cdot\boldsymbol{n}\,\mathrm{d}S = \int_V f\,\mathrm{d}\boldsymbol{x} \quad \text{for all "control volumes" } V \tag{6.1.3}$$

energy stored in V — power flux through ∂V — heat generation in V

In this case the heat flux was given by

$$\text{Fourier's law} \quad \mathbf{j}(\boldsymbol{x}) = -\kappa(\boldsymbol{x})\,\mathbf{grad}\,u(\boldsymbol{x})\,, \quad \boldsymbol{x} \in \Omega\,, \tag{2.6.5}$$

or its extended version (7.1.5). In Fourier's law the flux is a *linear* function of *derivatives* of u.

Conversely, for the flux function $\mathbf{f} : U \times \Omega \mapsto \mathbb{R}^d$ in (8.2.1) we assume

$\mathbf{f}$ depends only on local state u, not on derivatives of u: $f(u,\boldsymbol{x}) = f(u(\boldsymbol{x}),x)$.

On the other hand we go far beyond Fourier's law, since

$\mathbf{f}$ will, in general, be a non-linear function of u!

Remark 8.2.2 (Diffusive flux)

Taking into account the relationship with heat "diffusion", a flux function of the form of Fourier's law (2.6.5)

$$\mathbf{f}(u) = -\kappa(\boldsymbol{x})\,\mathbf{grad}\,u\ ,$$

is called a diffusive flux.

Now, integrate (8.2.1) over time period $[t_0, t_1] \subset [0, T]$ (space-time box, see Fig. 338) and use the fundamental theorem of calculus in the time direction:

- Space-time integral form of (8.2.1), *cf.* (8.1.44),

$$\int_V u(\boldsymbol{x}, t_1)\,\mathrm{d}\boldsymbol{x} - \int_V u(\boldsymbol{x}, t_0)\,\mathrm{d}\boldsymbol{x} + \int_{t_0}^{t_1}\int_{\partial V} \mathbf{f}(u,\boldsymbol{x})\cdot\boldsymbol{n}\,\mathrm{d}S(\boldsymbol{x})\mathrm{d}t = \int_{t_0}^{t_1}\int_V s(u,\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x}\mathrm{d}t \tag{8.2.3}$$

for all $V \subset \Omega$, $0 < t_0 < t_1 < T$, $\boldsymbol{n} \hat{=}$ exterior unit normal at ∂V

- [Gauss theorem Thm. 2.5.7] (local) differential form of (8.2.1):

$$\boxed{\frac{\partial}{\partial t}u + \mathrm{div}_{\boldsymbol{x}}\mathbf{f}(u,\boldsymbol{x}) = s(u,\boldsymbol{x},t) \quad \text{in } \widetilde{\Omega}} \ . \tag{8.2.4}$$

div acting on spatial variable $\boldsymbol{x}$ only

+ initial condition $\qquad u(\boldsymbol{x},0) = u_0(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega$

Special case $d = 1$ $\leftrightarrow$ (8.2.4) = one-dimensional scalar conservation law for "density" $u : \widetilde{\Omega} \mapsto \mathbb{R}$

$$\frac{\partial u}{\partial t}(x,t) + \frac{\partial}{\partial x}(f(u(x,t),x)) = s(u(x,t),x,t) \quad \text{in }]\alpha,\beta[\times]0,T[,\ \alpha,\beta \in \mathbb{R}\cup\{\pm\infty\}\ . \tag{8.2.5}$$

Remark 8.2.6 (Boundary values for conservation laws)

Suitable boundary values on $\partial\Omega\times]0,T[$? $\qquad\rightarrow$ usually tricky question (highly $\mathbf{f}$-dependent)

Reason: remember discussion in Rem. 8.1.13, meaningful boundary conditions hinge on knowledge of local (in space and time) transport direction, which, in a *non-linear* conservation law, will usually depend on the unknown solution $u = u(x,t)$.

This obviously compounds difficulties $\succ$ only Cauchy problems considered in this chapter.

8.2.2 Characteristics

In this section we will come across a surprising ostensible solution formula for non-linear scalar conservation laws in one spatial dimension. Yet, at second glance, we will see that this formula has problem. Its breakdown will teach us that *discontinuous solutions* are meaningful and very common in the case of conservation laws.

We consider Cauchy problem ($\Omega = \mathbb{R}$) for one-dimensional scalar conservation law (8.2.5):

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= 0 && \text{in } \mathbb{R}\times]0,T[\,, \\ u(x,0) &= u_0(x) && \text{in } \mathbb{R}\,. \end{aligned} \tag{8.2.7}$$

by chain rule: (8.2.7) $\Leftrightarrow$ $\dfrac{\partial u}{\partial t} + f'(u)\dfrac{\partial u}{\partial x} = 0\,,$

$\updownarrow$

relate with linear advection (8.1.10) $\dfrac{\partial u}{\partial t} + v\dfrac{\partial u}{\partial x} = 0\,.$

➢ The derivative $f'(u)$ plays the role of a u-dependent velocity of transport.

If this dependence was not there, the formula (8.1.11) would give us the solution. Now we will see how this formula can be generalized.

Assumption 8.2.8. Monotonicity of

The flux function $f : \mathbb{R} \mapsto \mathbb{R}$ is smooth ($f \in C^2$), and convex or concave [53, Def. 5.5.2].

Recall [53, Thm. 5.5.2]:

f convex $\Rightarrow$ derivative f' increasing
f concave $\Rightarrow$ derivative f' decreasing

flux function for Burgers' equation (8.1.46)

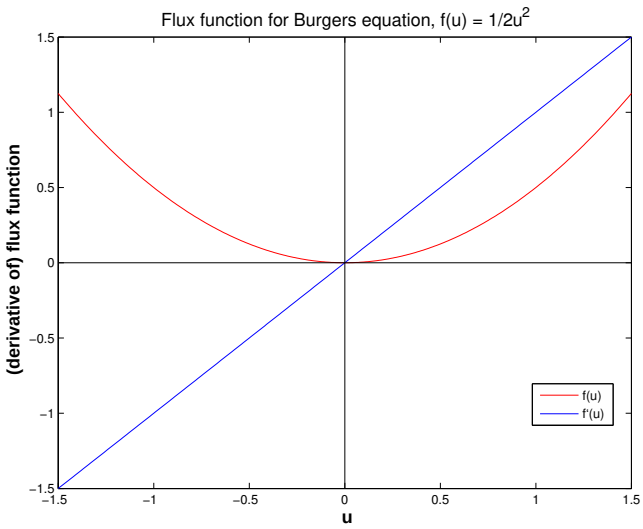


Fig. 340

f *convex*

flux function for traffic flow equation (8.1.41)

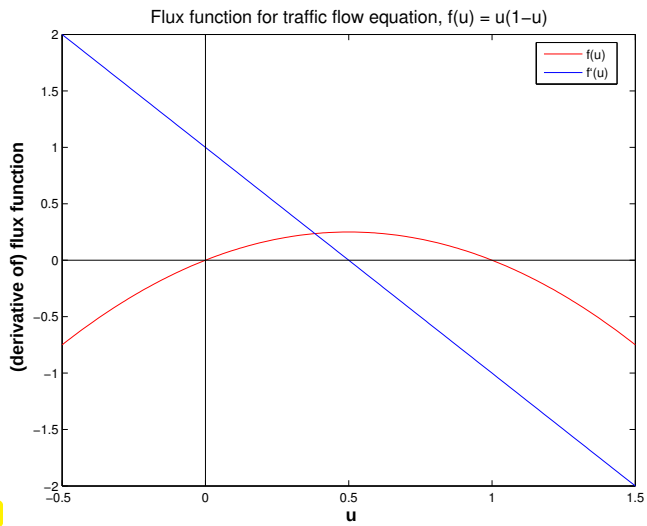


Fig. 341

f *concave*

Burgers' equation (8.1.46) and the traffic flow equation (8.1.41) will serve as main examples for scalar conservation laws in one spatial dimension. The opposite curvatures of their flux functions will be reflected by a "mirror symmetric" behavior of their solutions in many cases. Below most examples will be discussed for both model problems in order to elucidate these differences, but the reader may focus on only one model problem.

Definition 8.2.9. Characteristic curve for one-dimensional scalar conservation law

A curve $\Gamma := (\gamma(\tau), \tau) : [0, T] \mapsto \mathbb{R}\times]0, T[$ in the (x, t)-plane is a characteristic curve for the conservation law (8.2.7), if

$$\frac{d}{d\tau}\gamma(\tau) = f'(u(\gamma(\tau), \tau)) \ , \quad 0 \leq \tau \leq T \ , \tag{8.2.10}$$

where u is a continuously differentiable solution of (8.2.7).

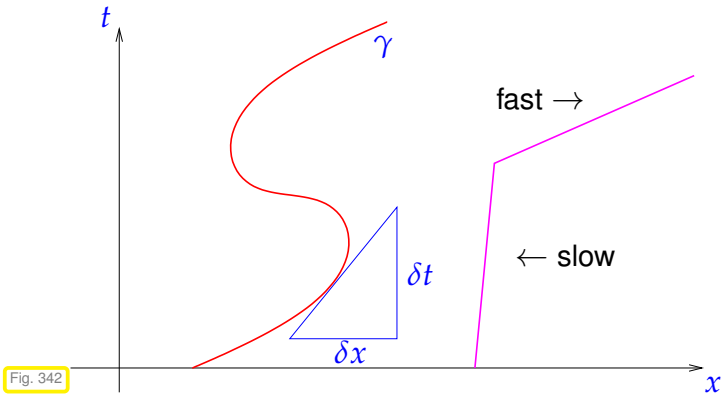


Fig. 342

◁ curves in an $x - t$-diagram, described by a function $x = \gamma(t)$
↔ movement of a point on the real axis.
◁ $x - t$-diagram

$$\frac{d}{d\tau}\gamma(\tau) = \text{speed of interface } \gamma.$$

Example 8.2.11 (Characteristics for advection)

Constant linear advection (8.1.10): $f(u) = vu$

➽ characteristics $\gamma(\tau) = v\tau + c$, $c \in \mathbb{R}$.

solution (8.1.11) $u(x, t) = u_0(x - vt)$

meaningful for *any* u_0 ! (*cf.* Sect. 7.3.2)

▷

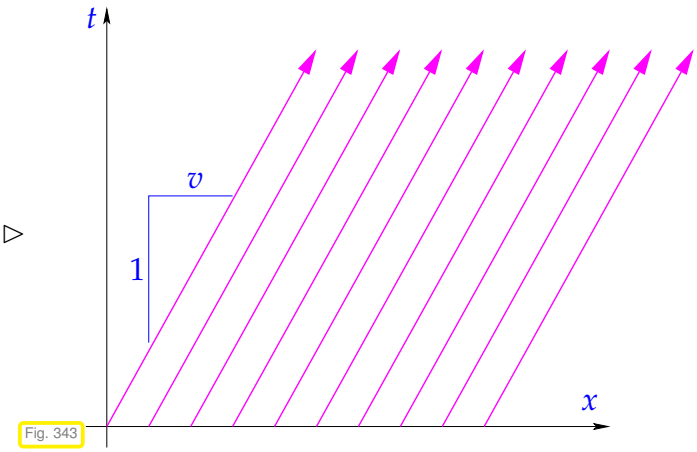


Fig. 343

Ex. 8.2.11 reveals a close relationship between streamlines (→ Section 7.1.1) and characteristic curves. That the latter are a true generalization of the former is also reflected by the following simple observation, which generalizes the considerations in Section 7.3.2, (7.3.9).

Lemma 8.2.12. Classical solutions and characteristic curves

Smooth solutions of (8.2.7) *are constant along characteristic curves.*

Proof. Apply chain rule twice, *cf.* (7.3.9), and use the defining equation (8.2.10) for a characteristic curve:

$$\begin{aligned}\frac{d}{d\tau}u(\gamma(\tau), \tau) &\overset{\text{chain rule}}{=} \frac{\partial u}{\partial x}(\gamma(\tau), \tau)\frac{d}{d\tau}\gamma(\tau) + \frac{\partial u}{\partial t}(\gamma(\tau), \tau) \\ &\overset{(8.2.10)}{=} \frac{\partial u}{\partial x}(\gamma(\tau), \tau) \cdot f'(u(\gamma(\tau), \tau)) + \frac{\partial u}{\partial t}(\gamma(\tau), \tau) \\ &\overset{\text{chain rule}}{=} \left(\frac{\partial}{\partial x}f(u)\right)(\gamma(\tau), \tau) + \frac{\partial u}{\partial t}(\gamma(\tau), \tau) = 0 \ .\end{aligned}$$

✎ notation: $f' \hat{=}$ derivative of flux function $f : U \subset \mathbb{R} \mapsto \mathbb{R}$

So, u is constant on a characteristic curve.

➢ $f'(u)$ is constant on a characteristic curve.

(8.2.10) $\Rightarrow$ slope of characteristic curve is constant!

▶ Characteristic curve through $(x_0, 0)$ is a straight line $(x_0 + f'(u_0(x_0))\tau, \tau)$, $0 \leq \tau \leq T$!

!? implicit solution formula for (8.2.7) (f' monotone !):

$$u(x,t) = u_0(x - f'(u(x,t))t) \ . \tag{8.2.13}$$

This is a non-linear equation for $u(x,t)$.

(8.2.14) Breakdown of characteristic solution formula

The key problem of formula (8.2.13) is that it may have multiple solutions:

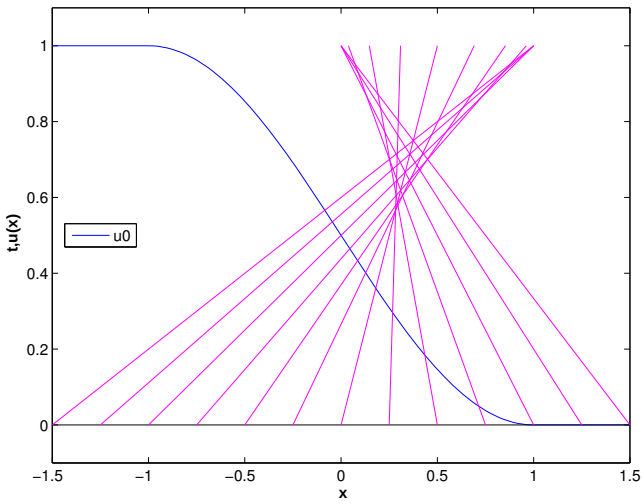


Fig. 344

For Burger's equation (8.1.46):
($f(u) = \frac{1}{2}u^2$ smooth and strictly convex)

▷ $f'(u) = u$ (increasing)

◁ if u_0 smooth and decreasing.

➤ characteristic curves intersect !

➤ solution formula (8.2.13) becomes invalid

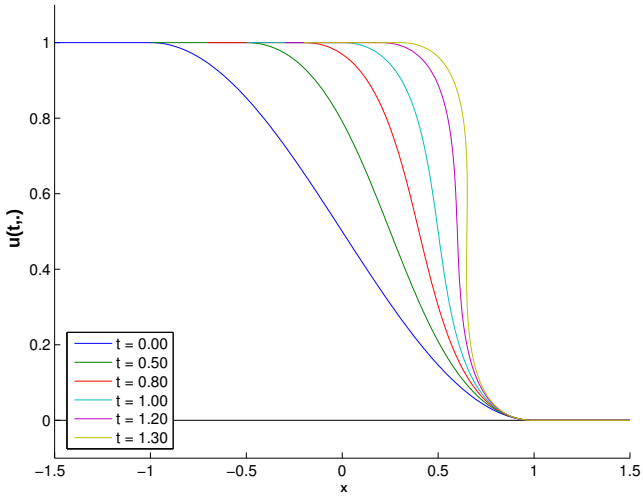


Fig. 345

$t < 1.3$: solution by (8.2.13)

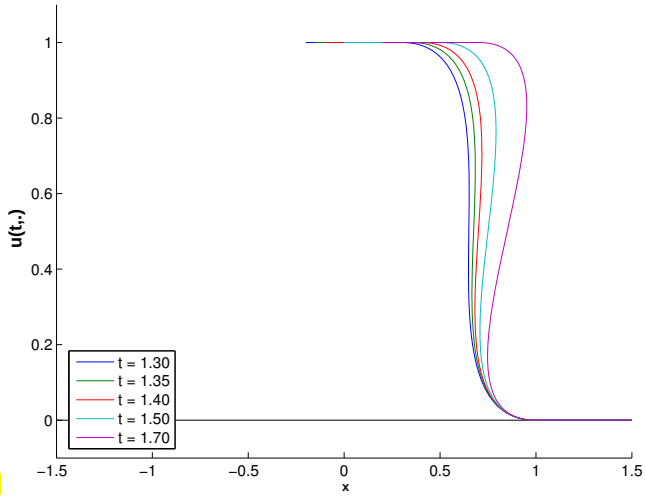


Fig. 346

The wave breaks: "multivalued solution"

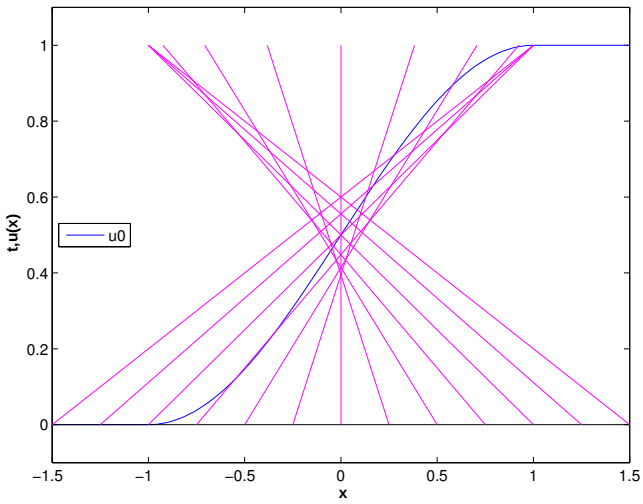


Fig. 347

For traffic flow equation (8.1.41):
($f(u) = u(1-u)$ smooth and strictly concave)

▷ $f'(u) = 1 - 2u$ (decreasing)

◁ if u_0 smooth and increasing.

➤ characteristic curves intersect **!**

➤ solution formula (8.2.13) becomes invalid

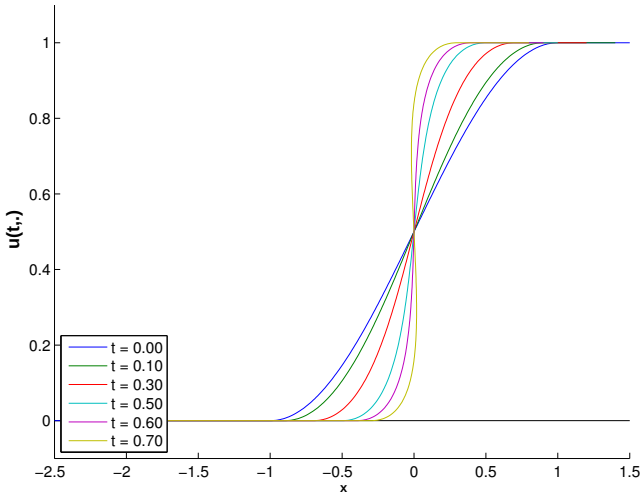


Fig. 348

$t < 0.7$: solution by (8.2.13)

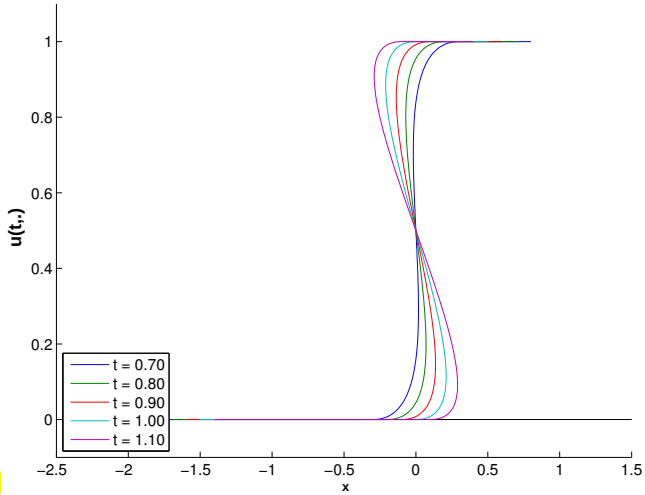


Fig. 349

the wave breaks: "multivalued solution"

---

breakdown of classical solutions & Ex. 8.2.11 ➥ new concept of solution of (8.2.7)

Remark 8.2.15 (Meaning of characteristics)

Concerning the interpretation of characteristics in the case of the traffic flow model (**??**) with $f(u) = u(1-u)$ we find

$$\begin{array}{rl} \text{Equation for characteristics} & \dot{\gamma}(t) = -2u(\gamma(t),t) + 1\,, \\ \text{Equation for car trajectories} & \dot{x}(t) = 1 - u(x(t),1). \end{array}$$

Hence, characteristics do not give the paths of cars; cars always drive to the right, while characteristics may be slanted to the left!

Yet, Lemma 8.2.12 tells us that for a smooth solution of a non-linear scalar conservation law, the characteristic running through $(x^*,t^*) \in \mathbb{R}\times]0,T[$ gives the locus of space-time points $(x,t) \in \mathbb{R}\times]0,T[$, on which the solution value $u(x^*,t^*)$ depends (for $t < t^*$) or on which it will have an influence (for $t > t^*$).

For a scalar conservation law information "flows" along characteristic curves.

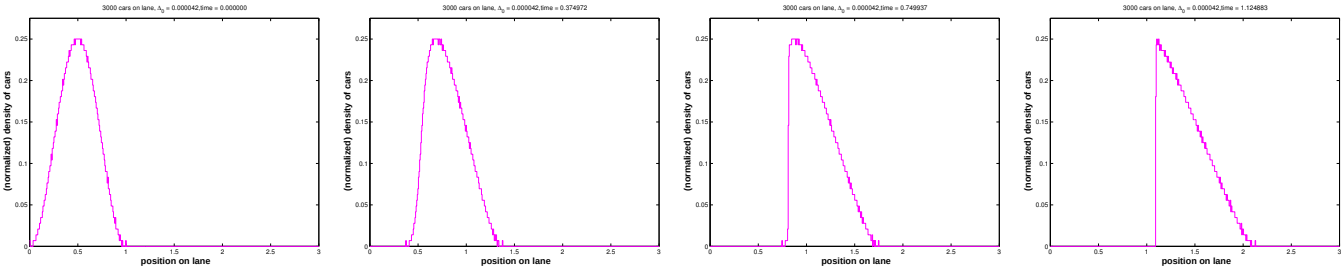
Example 8.2.16 (Traffic flow: Evolution of smooth initial density)

For the traffic flow model we should always expect a unique car density for all times. Thus, in order to see the consequences of the breakdown of the solution formula, we return to the particle model for single lane traffic flow from Section 8.1.2.1. For a large number of cars it should give us a hint how the density will be affected by the intersection of characteristics.

We solve the particle model, that is the evolution according to ODE (8.1.21), (8.1.22), implemented in Code 8.1.27, for $N = 3000$ cars.

The initial car positions derived from a smooth car density u_0

$$x_i(0) = \Phi^{-1}\left(\frac{i-1}{N-1}\right) , \quad i = 1,\dots,N , \quad \Phi(\xi) = \int_0^{\xi} u_0(x)\,\mathrm{d}x , \quad u_0(x) := 2\sin^2(\pi x) .$$



After some time a *discontinuity* in the density of cars crops up ("breaking wave", see Figure 349). This suggests that the emergence of discontinuities despite smooth initial data is an intrinsic feature of the traffic flow model, which reflects "physical reality".

8.2.3 Weak solutions

Of course, discontinuous solutions of (8.2.7) cannot be solutions in he sense of classical calculus. Yet, the fact that physically meaningful solutions fail to meet the smoothness requirements for classical solutions is familiar to us: we saw this already for the elastic string model, where we had to admit solutions with a kink in Ex. 1.3.37. This forced us to develop weak concepts of solutions. For the elastic string models these were solutions of the associated variational equation. In the case of conservation laws a similar concept of weak solutions will turn out to capture all physically meaningful solutions.

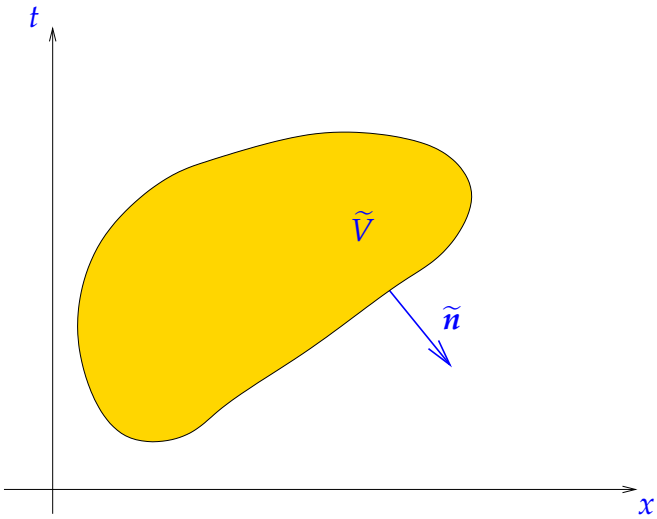
The integral form of a conservation law that we have already seen in (8.2.3) points the way.

"Space-time Gaussian theorem"

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \qquad (8.2.17)$$

$$\Updownarrow$$

$$\operatorname{div}_{(x,t)} \begin{bmatrix} f(u) \\ u \end{bmatrix} = 0 \quad \text{in } \widetilde{\Omega}\,. \qquad (8.2.18)$$



$\blacktriangleright$ $\forall$ "space-time control volumes" $\widetilde{V} \subset \widetilde{\Omega}$:

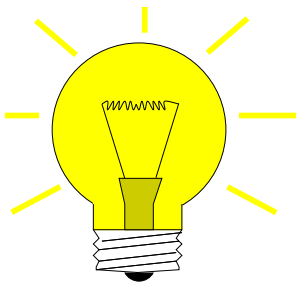
$$\int_{\partial \widetilde{V}} \begin{bmatrix} f(u(\widetilde{\boldsymbol{x}})) \\ u(\widetilde{\boldsymbol{x}}) \end{bmatrix} \cdot \begin{bmatrix} n_x(\widetilde{\boldsymbol{x}}) \\ n_t(\widetilde{\boldsymbol{x}}) \end{bmatrix} \mathrm{d}S(\widetilde{\boldsymbol{x}}) = 0\,,$$

$\widetilde{\boldsymbol{n}} := (n_x, n_t)^T \mathrel{\hat{=}}$ space-time unit normal

(8.2.18) for space-time rectangle $\widetilde{V} =]x_0, x_1[\times]t_0, t_1[$ $\blacktriangleright$ integral form of (8.2.17), *cf.* (8.2.3):

$$\int_{x_0}^{x_1} u(x,t_1)\,\mathrm{d}x - \int_{x_0}^{x_1} u(x,t_0)\,\mathrm{d}x = \int_{t_0}^{t_1} f(u(x_0,t))\,\mathrm{d}t - \int_{t_0}^{t_1} f(u(x_1,t))\,\mathrm{d}t\,. \qquad (8.2.19)$$

Still, (8.2.19) encounters problems, if a discontinuity of u coincides with an edge of the space-time rectangle.



The idea is similar to that behind the derivation of the weak form for 2nd-order elliptic BVPs in Section 2.9. For the Cauchy problem

I: test the conservation law PDE with a smooth function,
II: integrate by parts one in space & time,
III: take into account the initial conditions.

STEP I: Test (8.2.18) with compactly supported smooth function $\Phi : \widetilde{\Omega} \mapsto \mathbb{R}$, $\Phi(\cdot, T) = 0$, and integrate over space-time cylinder $\widetilde{\Omega} = \mathbb{R} \times [0,T]$:

$$(8.2.18) \quad \blacktriangleright \quad \int_{\widetilde{\Omega}} \operatorname{div}_{(x,t)} \begin{bmatrix} f(u) \\ u \end{bmatrix} \Phi(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x}\,\mathrm{d}t = 0\,.$$

STEP II: Perform integration by parts using Green's first formula Thm. 2.5.9 on $\widetilde{\Omega}$:

$$\int_{\widetilde{\Omega}} \operatorname{div}_{(x,t)} \begin{bmatrix} f(u) \\ u \end{bmatrix} \Phi(\boldsymbol{x},t)\,\mathrm{d}\boldsymbol{x}\,\mathrm{d}t = 0$$

$$\overset{\text{Thm. 2.5.9}}{\Rightarrow} \quad \int_{\widetilde{\Omega}} \begin{bmatrix} f(u) \\ u \end{bmatrix} \cdot \mathbf{grad}_{(x,t)}\,\Phi\,\mathrm{d}\boldsymbol{x}\,\mathrm{d}t + \int_{-\infty}^{\infty} u(x,0)\Phi(x,0)\,\mathrm{d}x = 0\,,$$

because $\partial\widetilde{\Omega} = \mathbb{R} \times \{0\} \cup \mathbb{R} \times \{T\}$ with "normals" $\boldsymbol{n} = \binom{0}{-1}$ ($t = 0$ boundary) and $\boldsymbol{n} = \binom{0}{1}$ ($t = T$ boundary), which has to be taken into account in the boundary term in Green's formula. The "$t = T$ boundary part" does not enter as $\Phi(\cdot, T) = 0$.

Note that $u(x,0)$ is fixed by the initial condition: $u(x,0) = u_0(x)$.

Definition 8.2.20. Weak solution of Cauchy problem for scalar conservation law

For $u_0 \in L^\infty(\mathbb{R})$, $u : \mathbb{R}\times]0,T[\mapsto \mathbb{R}$ is a weak solution of the Cauchy problem (8.2.7), if

$$u \in L^\infty(\mathbb{R}\times]0,T[) \quad \wedge \quad \int_{-\infty}^{\infty}\int_0^T \left\{ u\frac{\partial\Phi}{\partial t} + f(u)\frac{\partial\Phi}{\partial x}\right\} \mathrm{d}t\mathrm{d}x + \int_{-\infty}^{\infty} u_0(x)\Phi(x,0)\,\mathrm{d}x = 0\,,$$

for all $\Phi \in C_0^\infty(\mathbb{R}\times[0,T[)$, $\Phi(\cdot,T) = 0$.

Remark 8.2.21 (Properties of weak solutions)

By reversing integration by parts, it is easy to see that

$$u \text{ weak solution of (8.2.7) } \& \; u \in C^1 \iff u \text{ classical solution of (8.2.7).}$$

Arguments from mathematical integration theory confirm

$$u \in L^\infty_{\mathrm{loc}}(\mathbb{R}\times]0,T[) \text{ weak solution of (8.2.7)} \quad \Rightarrow \quad \begin{array}{c} u \text{ satisfies integral form (8.2.19)} \\ \text{for "almost all" } x_0 < x_1,\ 0 < t_0 < t_1 < T. \end{array}$$

8.2.4 Jump conditions

Now we want to explore the discontinuities compatible with our concept of a weak solution from Def. 8.2.20.

For piecewise smooth divergence-free vectorfield $\mathbf{j} : \Omega \subset \mathbb{R}^2$ we find by a "pillbox thought experiment":

$$\text{"}\operatorname{div}\mathbf{j} = 0\text{"}$$

$$\Updownarrow$$

$$\int_{\partial V} \mathbf{j}\cdot\boldsymbol{n}\,\mathrm{d}S = 0 \quad \forall \text{ control volumes } V \subset \Omega$$

Necessary condition:

Continuity of **normal components** across discontinuities

discontinuous divergence-free vectorfield ▷

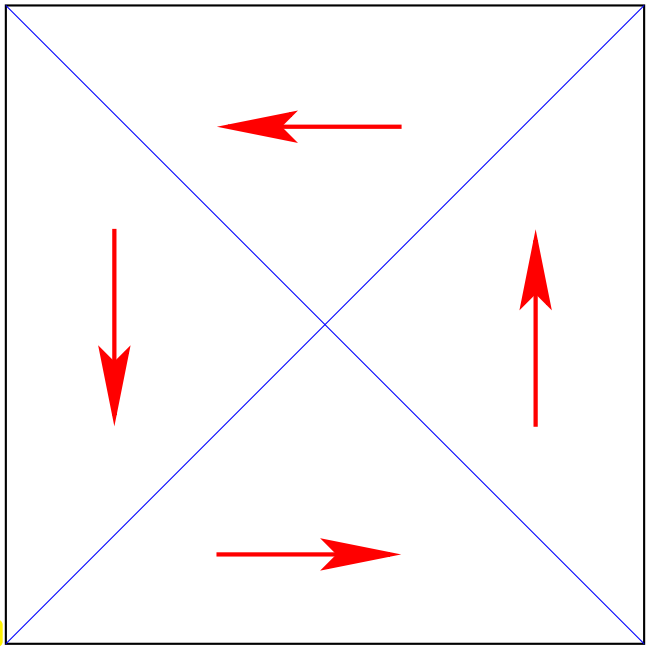


Fig. 350

To see this, consider a slender tiny rectangle aligned with a line of discontinuity of $\mathbf{j}$. In the absence of normal continuity a net flux through its boundary will result, provided that the rectangle is small enough ("pillbox argument").

Apply this insight to vectorfield on space-time domain $\widetilde{\Omega} = \mathbb{R}\times]0,T[$:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \Leftrightarrow \quad \operatorname{div}_{(x,t)} \underbrace{\begin{bmatrix} f(u) \\ u \end{bmatrix}}_{=:\mathbf{j}} = 0 \quad \text{in } \widetilde{\Omega}\,. \tag{8.2.18}$$

Normal at C^1-curve $\Gamma := \tau \mapsto (\gamma(\tau),\tau)$ in $(\gamma(\tau),\tau)$

$$\widetilde{\boldsymbol{n}} = \frac{1}{\sqrt{1+|\dot{s}|^2}} \begin{bmatrix} 1 \\ -\dot{s} \end{bmatrix}, \quad \dot{s} := \frac{d\gamma}{d\tau}(\tau) \quad \text{"speed of curve"}\,.$$

To see this, recall that the normal is orthogonal to the tangent vector $\binom{\dot{s}}{1}$ and that in 2D the direction orthogonal to $\binom{x_1}{x_2}$ is given by $\binom{-x_2}{x_1}$.

"normal continuity" of

piecewise smooth vectorfield $(f(u),u)^T$

$$\Updownarrow$$

$$\begin{bmatrix} 1 \\ -\frac{d\gamma}{d\tau} \end{bmatrix} \cdot \begin{bmatrix} [\![f(u)]\!] \\ [\![u]\!] \end{bmatrix} = 0\,, \tag{8.2.22}$$

where $[\![\cdot]\!] \hat{=}$ jump across Γ ("from left to right", e.g. $[\![u]\!] = u_l - u_r$, where subscripts 'l' and 'l' designates values in $\widetilde{\Omega}_l$, $\widetilde{\Omega}_r$).

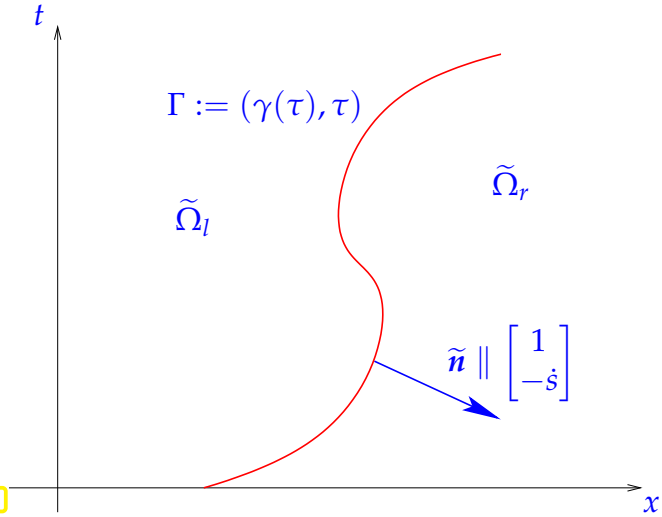


Fig. 351

Terminology: (8.2.22) = Rankine-Hugoniot (jump) condition, shorthand notation:

$$\boxed{\dot{s}(u_l - u_r) = f_l - f_r} \quad , \quad \dot{s} := \frac{d\gamma}{d\tau} \quad \text{"propagation speed of discontinuity"} \tag{8.2.23}$$

(8.2.24) Discontinuity connecting constant states

The simplest situtation compliant with Rankine-Hugoniot jump condition: *constant states* to the left and right of the curve of discontinuity (8.2.22):

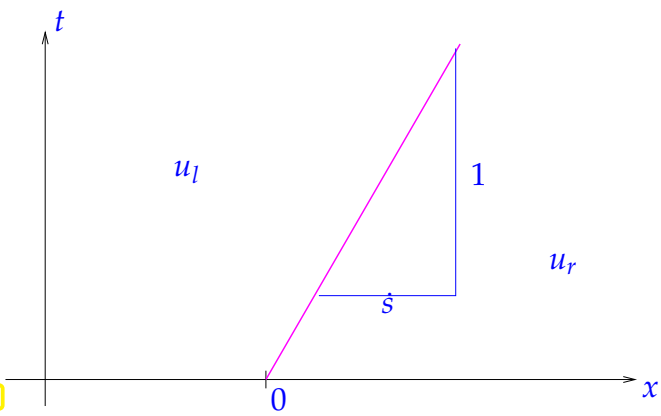


Fig. 352

$$u(x,t) = \begin{cases} u_l \in \mathbb{R} & \text{, for} \quad x < \dot{s}t\,, \\ u_r \in \mathbb{R} & \text{, for} \quad x < \dot{s}t\,, \end{cases} \tag{8.2.25}$$

with **constant** speed $\dot{s}$ of discontinuity, according to (8.2.23) given by (for $u_l \neq u_r$)

$$\dot{s} = \frac{f(u_l) - f(u_r)}{u_l - u_r}\,.$$

8.2.5 Riemann problem

The situation of locally constant states discussed in § 8.2.24 is particularly easy.

Consider: Cauchy-problem (8.2.7) for piecewise constant initial data u_0.

Definition 8.2.26. Riemann problem

$$u_0(x) = \begin{cases} u_l \in \mathbb{R} & \text{, if } x < 0\,, \\ u_r \in \mathbb{R} & \text{, if } x > 0\,. \end{cases} \quad \hat{=} \quad \text{Riemann problem for (8.2.7)}$$

Setting, *cf.* Section 8.2.2: flux function $f : \mathbb{R} \mapsto \mathbb{R}$ smooth & convex

f' non-decreasing ➤ pattern of characteristic curves for Riemann problem:

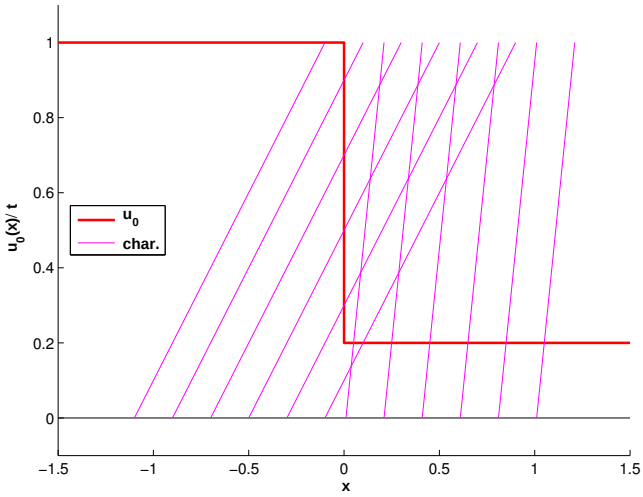


Fig. 353

intersecting characteristics

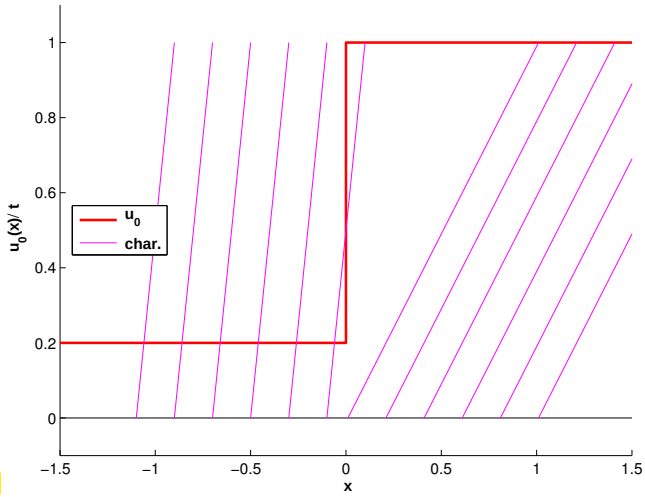


Fig. 354

diverging characteristics

Setting, *cf.* Section 8.2.2: flux function $f : \mathbb{R} \mapsto \mathbb{R}$ smooth & concave

f' non-increasing ➤ pattern of characteristic curves for Riemann problem:

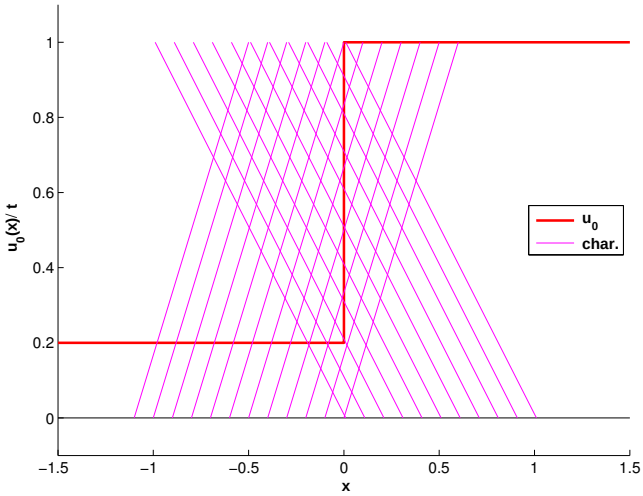


Fig. 355

intersecting characteristics

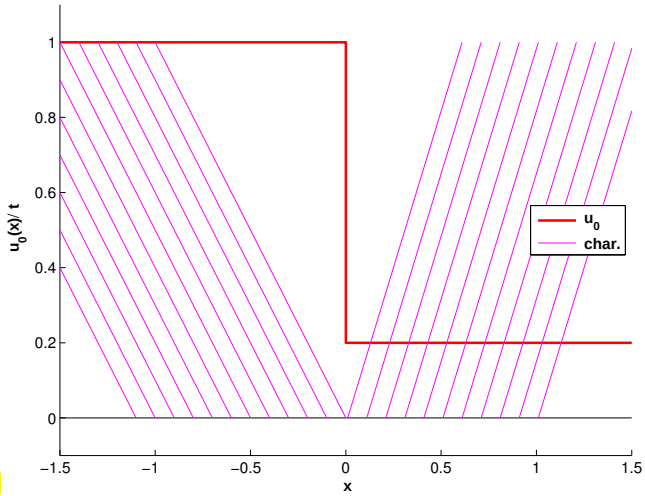


Fig. 356

diverging characteristics

Definition 8.2.27. Shock

If Γ is a smooth curve in the (x,t)-plane and u a weak solution of (8.2.7), a discontinuity of u across Γ is called a shock.

By § 8.2.24 ➤ the shock speed s is given by the Rankine-Hugoniot jump conditions:

$$(x_0,t_0)\in\Gamma:\quad \dot{s}=\frac{f(u_l)-f(u_r)}{u_l-u_r}\quad,\quad \begin{array}{l} u_l:=\lim_{\epsilon\to 0}u(x_0-\epsilon,t_0)\;,\\ u_r:=\lim_{\epsilon\to 0}u(x_0+\epsilon,t_0)\;.\end{array} \tag{8.2.28}$$

Lemma 8.2.29. Shock solution of Riemann problem

For any two states $u_l,u_l\in\mathbb{R}$ the piecewise constant function

$$u(x,t):=\begin{cases} u_l & \text{for } x<\dot{s}t\;,\\ u_r & \text{for } x>\dot{s}t\;,\end{cases}\quad \dot{s}:=\frac{f(u_l)-f(u_r)}{u_l-u_r}\;,\quad x\in\mathbb{R},\,0<t<T\;,$$

is weak solution ($\to$ Def. 8.2.20) of the related Riemann problem ($\to$ Section 8.2.5) for the 1D scalar conservation law (8.2.7).

Now we study the dependence of shock solutions on the initial states u_l and u_r. We take a close look at the connection between characteristics and shocks. In the following $x-t$ diagrams, shocks are marked with —, characteristics with — and u_0 is indicated by —.

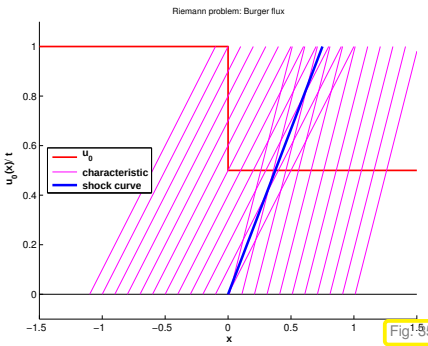


Fig. 357

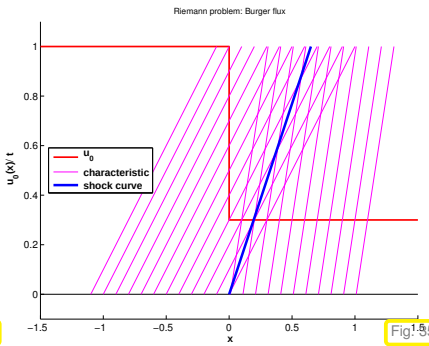


Fig. 358

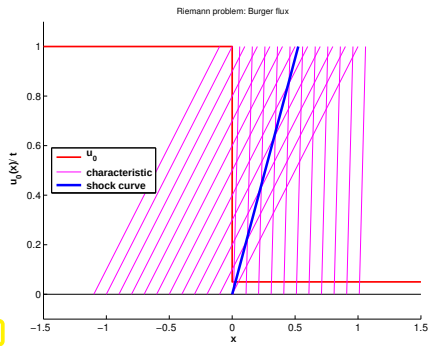


Fig. 359

Burgers flux $f(u)=\frac{1}{2}u^2$, $u_l>u_r$: characteristic curves impinge on shock

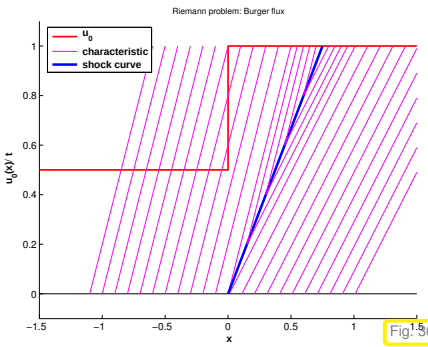


Fig. 360

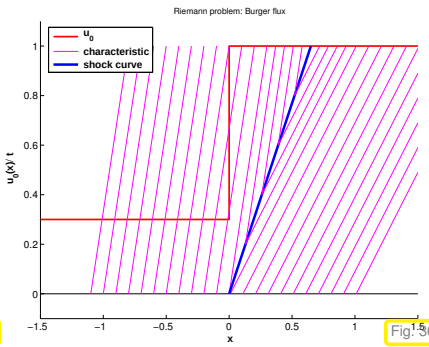


Fig. 361

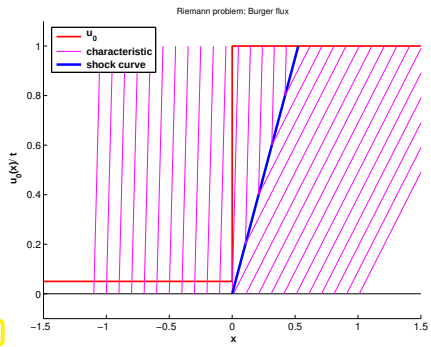


Fig. 362

Burgers flux $f(u)=\frac{1}{2}u^2$, $u_l<u_r$: characteristic curves emanate from shock (expansion shock)

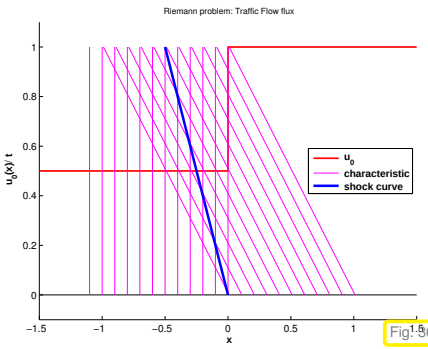


Fig. 363

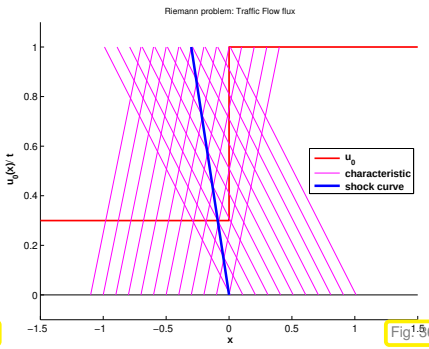


Fig. 364

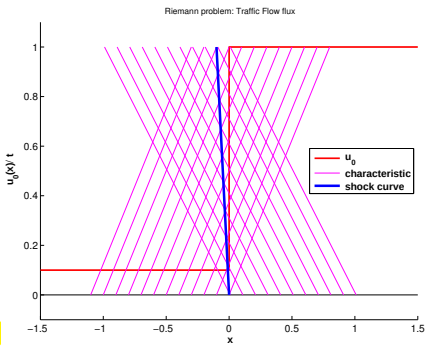


Fig. 365

Traffic Flow flux $f(u) = u(1-u)$, $u_l < u_r$: characteristic curves impinge on shock

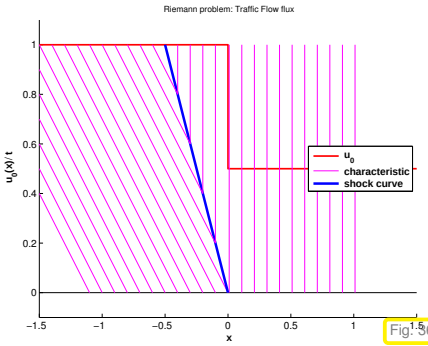


Fig. 366

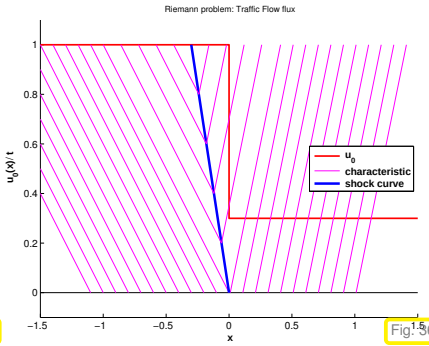


Fig. 367

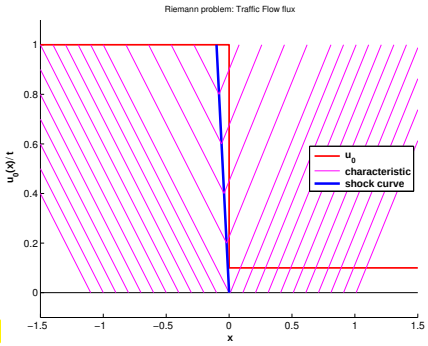


Fig. 368

Traffic flow flux $f(u) = u(1-u)$, $u_l > u_r$: characteristic curves emanate from shock (expansion shock)

Example 8.2.30 (Actual shock patterns in traffic flow)

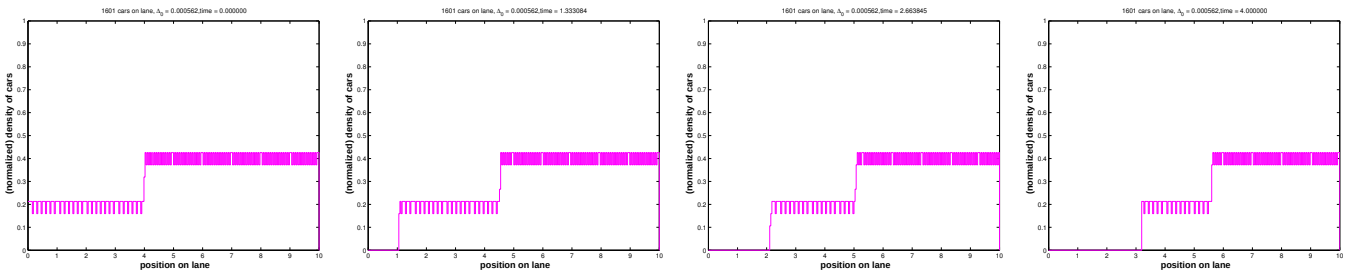
In order to tell the physical relevance of shock solutions for the car density we try to obtain them approximately from the particle model of traffic flow using many cars.

We conduct a simulation of microscopic particle model of traffic flow as in Exp. 8.1.31, with initial car distribution

`x0 = [(0:0.01:4),(4.005:0.005:10)]` (MATLAB syntax),

$\Delta_0 = 0.002$, normalized car density by averaging.

Situation: column of fast going cars approaches a zone of dense traffic.



Observation: abrupt changes of car density (= shocks) present in initial conditions persist throughout the evolution. Sites of discontinuity travel with constant speed close to the speed predicted by the jump conditions (8.2.23).

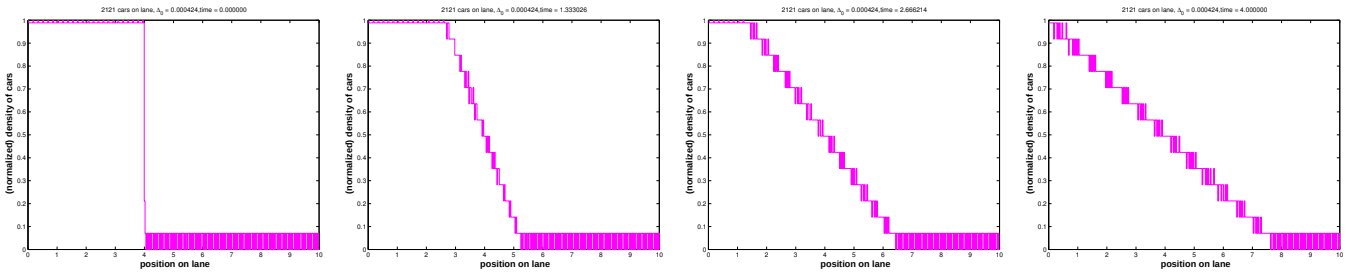
Example 8.2.31 (Fan patterns in traffic flow)

Simulation of microscopic particle model of traffic flow as in Ex. 8.1.31, initial car distribution

```
x0 = [(0:0.002:4),(4.05:0.05:10)]
```
(MATLAB syntax),

$\Delta_0 = 0.002$, normalized car density by averaging.

Situation: front end of a traffic jam



Observation: abrupt changes of car density present in initial conditions disappear and are replaced with a zone of *linearly decreasing* car density, whose edges move with constant speed in opposite direction.

No shock solution!

Example 8.2.32 (Vanishing viscosity for Burgers equation)

Recall the modeling approach explained in Sect. 8.1.3. There is no such material as an "invsicid" fluid in nature, because in any physical system there will be a tiny amount of friction. This leads us to the very general understanding that conservation laws can usually be regarded as limit problems $\epsilon = 0$ for singularly perturbed transport-diffusion problems with an "ϵ-amount" of diffusion.

In 1D, for any $\epsilon > 0$ these transport-diffusion problems will possess a unique smooth solution. Studying its behavior for $\epsilon \to 0$ will tell us, what are "physically meaningful" solutions for the conservation law. This consideration is called the vanishing viscosity method to define solutions for conservation laws.

Here we pursue this idea for Burgers equation, see Sect. 8.1.3.

Viscous Burgers equation:
$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}\left(\tfrac{1}{2}u^2\right) = \underbrace{\epsilon\frac{\partial^2 u}{\partial x^2}}_{\text{dissipative (viscous) term}} . \tag{8.2.33}$$

Travelling wave solution of Riemann problem for (8.2.33) via Cole-Hopf transform $\to$ [23, Sect. 4.4.1]

$$u_\epsilon(x,t) = w(x - \dot{s}t) \quad , \quad w(\xi) = u_r + \tfrac{1}{2}(u_l - u_r)\left(1 - \tanh\left(\frac{\xi(u_l - u_r)}{4\epsilon}\right)\right) , \quad \dot{s} = \tfrac{1}{2}(u_l + u_r) .$$

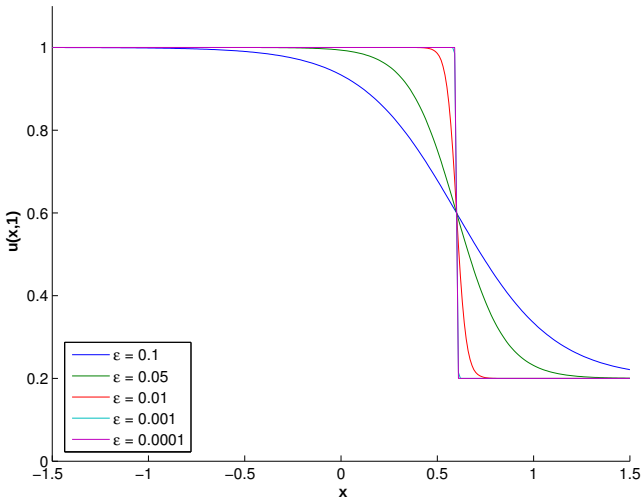


Fig. 369

$u_\epsilon(x,t)$ = classical solution of (8.2.33) for all $t > 0$, $x \in \mathbb{R}$ (only for $u_l > u_r$!).

$\triangleleft$ $u_l > u_r, \quad t = 0.5$

emerging shock for $\epsilon \to 0$

$u_\epsilon \to u$ from Lemma 8.2.29 in $L^\infty(\mathbb{R})$.

Highly accurate numerical solution of Riemann problem for (8.2.33)

$u_l < u_r$ $\qquad$ $u_\epsilon(x, 0.5) \triangleright$

no shock as $\epsilon \to 0$!

$u_\epsilon \to$ a piecewise linear function!

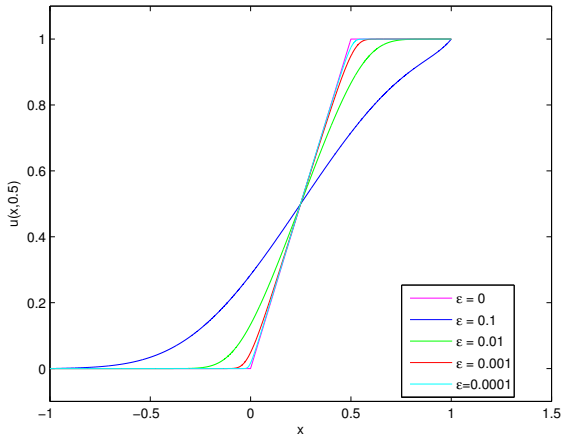


Fig. 370

(8.2.34) Similarity solution

Let us try to derive a (weak) solution of the homogeneous scalar conservation law (8.2.17) with the structure observed in Ex. 8.2.31 and Ex. 8.2.32.

Idea: conservation law (8.2.17) homogeneous in spatial/temporal derivatives:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R} \times \mathbb{R}^+ \quad \Rightarrow \quad \frac{\partial u_\lambda}{\partial t} + \frac{\partial}{\partial x} f(u_\lambda) = 0 \quad \text{in } \mathbb{R} \times \mathbb{R}^+ ,$$

where $u_\lambda(x,t) := u(\lambda x, \lambda t), \quad \lambda > 0.$

In addition, for the Riemann problem ($\to$ Def. 8.2.5) the initial condition also satisfies $u_0(\lambda x) = u_0(x)$.

This suggests that we look for solutions of the Riemann problem that are constant on all straight lines in the $x - t$-plane that cross $(0,0)^T$.

try similarity solution: $\qquad u(x,t) = \psi(x/t)$

$\leftarrow$ insert in $\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0$

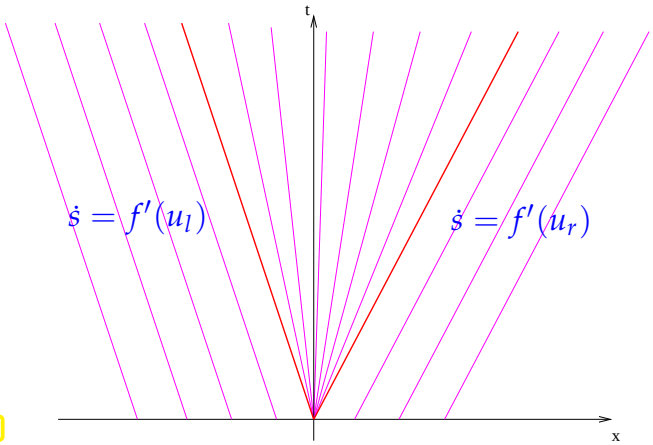
$$f'(\psi(x/t))\psi'(x/t) = (x/t)\psi'(x/t) \quad \forall x \in \mathbb{R}, 0 < t < T .$$

$$\psi' \equiv 0 \quad \vee \quad f'(\psi(w)) = w \quad \Leftrightarrow \quad \psi(w) = (f')^{-1}(w) .$$

f' strictly monotone !

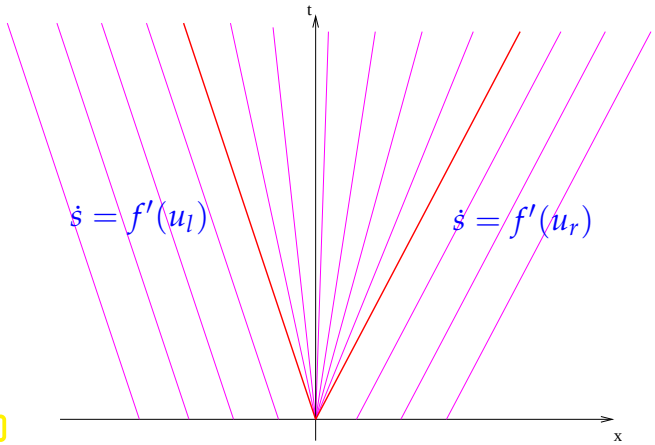
We can apply the formula for a similarity solution to the situation of a Riemann problem, because the initial data a compatible with it. Assuming monotonicity of the derivative of the (smooth) flux function f, we obtain he following similarity solutions:

Fig. 371



$f(u)$ strictly convex, $u_l < u_r$

Fig. 372



$f(u)$ strictly concave, $u_r < u_l$

Lemma 8.2.35. Rarefaction solution of Riemann problem

If $f \in C^2(\mathbb{R})$ *is strictly* $\begin{cases} \text{convex and} & u_l < u_r, \\ \text{concave and} & u_r < u_l, \end{cases}$ *then*

$$u(x,t) := \begin{cases} u_l & \text{for } x < \min\{f'(u_l), f'(u_r)\} \cdot t\,, \\ g(\frac{x}{t}) & \text{for } \min\{f'(u_l), f'(u_r)\} < \frac{x}{t} < \max\{f'(u_l), f'(u_r)\}\,, \\ u_r & \text{for } x > \max\{f'(u_l), f'(u_r)\} \cdot t\,, \end{cases}$$

$g := (f')^{-1}$, *is a weak solution of the Riemann problem (→ Def. 8.2.5).*

Proof. We show that the rarefaction solution is a weak solution according to Def. 8.2.20 ➢ for $\Phi \in C_0^\infty(\mathbb{R} \times]0,T[)$

$$\int_0^T \left\{ \int_{-\infty}^{f'(u_l)t} u_l \frac{\partial \Phi}{\partial t} + f(u_l)\frac{\partial \Phi}{\partial x}\,\mathrm{d}x + \int_{f'(u_l)t}^{f'(u_r t)} g(\tfrac{x}{t})\frac{\partial \Phi}{\partial t} + f(g(\tfrac{x}{t}))\frac{\partial \Phi}{\partial x}\,\mathrm{d}x + \int_{f(u_r)t}^{\infty} u_r \frac{\partial \Phi}{\partial t} + F(u_r)\frac{\partial \Phi}{\partial x}\,\mathrm{d}x \right\} \mathrm{d}t$$

$$= \int_0^T \int_{f'(u_l)t}^{f'(u_r)t} g'(\tfrac{x}{t})\tfrac{x}{t^2}\,\Phi - f'(g(\tfrac{x}{t}))\tfrac{1}{t}g'(\tfrac{x}{t})\,\Phi\,\mathrm{d}x\,\mathrm{d}t = 0\,,$$

because $(f' \circ g)(x/t) = x/t$ and by fundamental theorem of calculus. □

Terminology: solution of Lemma 8.2.35 = rarefaction wave: *continuous solution* !

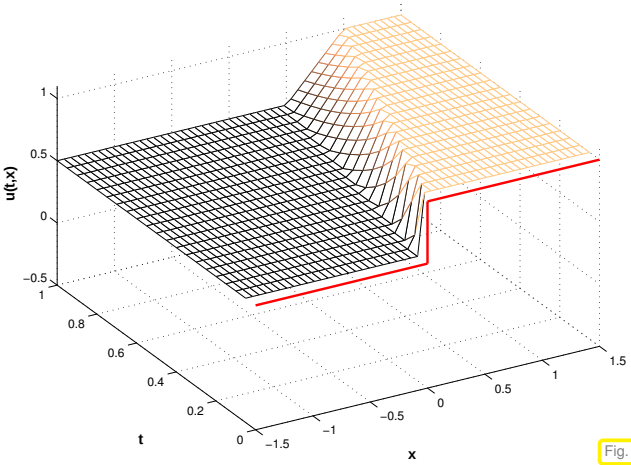


Fig. 373

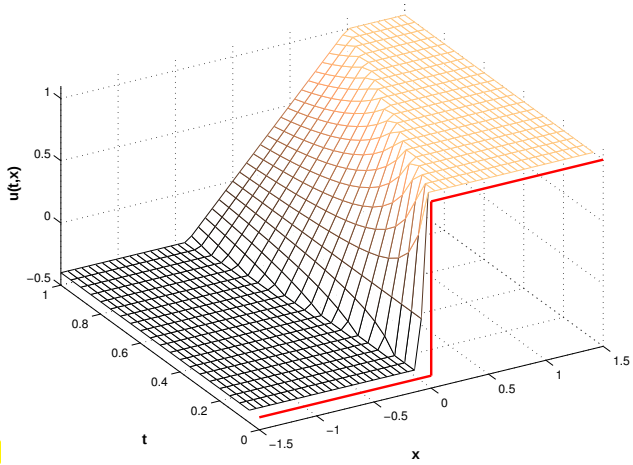


Fig. 374

Burger flux function $f(u) = \frac{1}{2}u^2$, $u_l < u_r$: rarefaction wave solutions

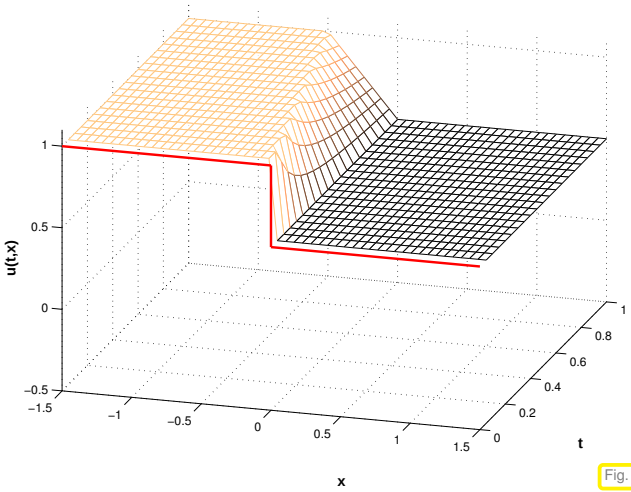


Fig. 375

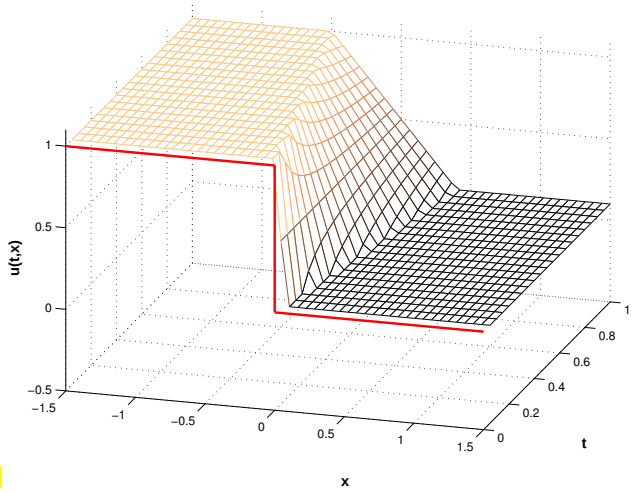


Fig. 376

Traffic flow flux function $f(u) = \frac{1}{2}u(1-u)$, $u_l > u_r$: rarefaction wave solutions

8.2.6 Entropy condition

In Section 8.2.5 we discovered that weak solutions of a scalar conservation law need not be unique. If f' is decreasing as in the traffic flow equation (8.1.41) and $u_l > u_r$ both a shock and a rarefaction wave provide valid weak solutions.

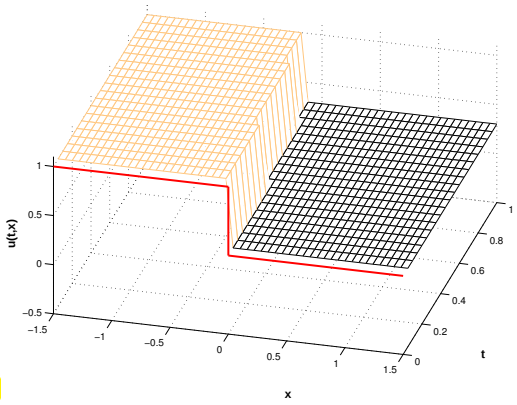


Fig. 377

Riemann solution: shock

$\overset{?}{\longleftrightarrow}$

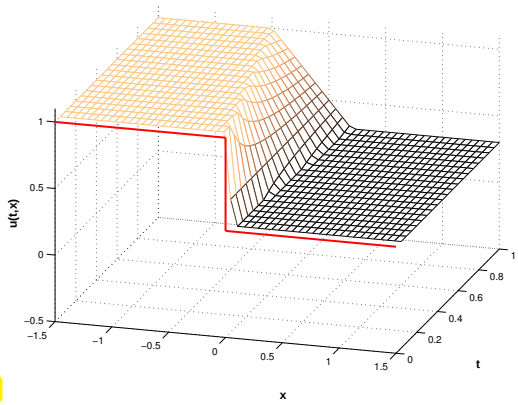


Fig. 378

Riemann solution: rarefaction wave

How to select "physically meaningful" = admissible solution ?

❶ Comparison with results from microscopic models, see Ex. 8.2.31 for the case of traffic flow.

❷ Vanishing viscosity technique (→ Ex. 8.2.32 for Burgers' equation): add an "ϵ-amount" of diffusion ("friction") and study solution for $\epsilon \to 0$.

However, desirable: simple selection criteria (entropy conditions)

Definition 8.2.36. Lax entropy condition

$u \mathrel{\hat{=}}$ weak solution of (8.2.7), piecewise classical solution in a neigborhood of C^2-curve $\Gamma := (\gamma(\tau), \tau)$, $0 \le \tau \le T$, discontinuous across Γ.

u satisfies the Lax entropy condition in $(x_0, t_0) \in \Gamma$ $:\Leftrightarrow$ $f'(u_l) > \dot{s} := \dfrac{f(u_l) - f(u_r)}{u_l - u_r} > f'(u_r)$.

$\Updownarrow$

Characteristic curves must not emanate from shock $\leftrightarrow$ no "generation of information"

➤ The expansion shocks from Fig. 360–**??**, Fig. 366-368 are not allowed.

Parlance: shock satisfying Lax entropy condition = physical shock

Note: f' increasing / decreasing ➤ by Def 8.2.36 necessary for physical shock $\boxed{u_l > u_r}$ / $\boxed{u_l < u_r}$

Physically meaningful weak solution of conservation law = entropy solution

For *scalar* conservation laws with locally Lipschitz-continuous flux function f [23, Sect. 11.4.3]:

Existence & uniqueness of entropy solutions

Remark 8.2.37 (General entropy solution for 1D scalar Riemann problem → [41])

In fact there is a general formula for the entropy solution of the Riemann problem (→ Section 8.2.5) for (8.2.7) with arbitrary $f \in C^1(\mathbb{R})$:

$$u(x,t) = \psi(x/t) \quad , \quad \psi(\xi) := \begin{cases} \operatorname*{argmin}_{u_l \le u \le u_r} (f(u) - \xi u) & \text{, if } u_l < u_r \text{ ,} \\ \operatorname*{argmax}_{u_r \le u \le u_l} (f(u) - \xi u) & \text{, if } u_l \ge u_r \text{ .} \end{cases} \tag{8.2.38}$$

Example 8.2.39 (Entropy solution of Burgers equation)

An analytic solution is available for Burgers eqution (8.1.46) with intial data, see [23, Sect. 3.4, Ex. 3]

$$u_0(x) = \begin{cases} 0 & \text{, if } x < 0 \text{ or } x > 1 \text{ ,} \\ 1 & \text{, if } 0 \le x \le 1 \text{ .} \end{cases}$$

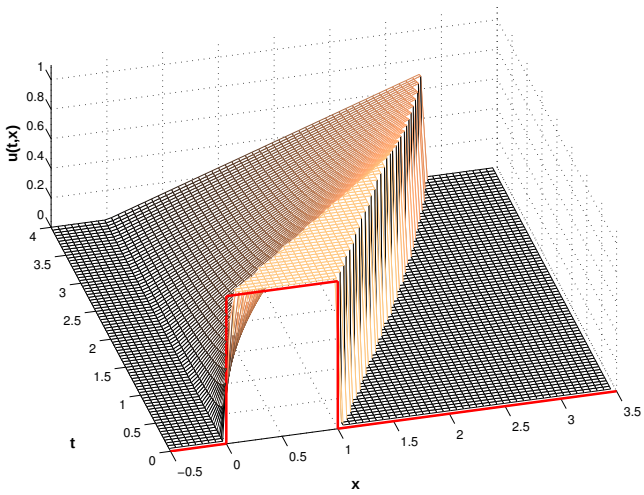


Fig. 379

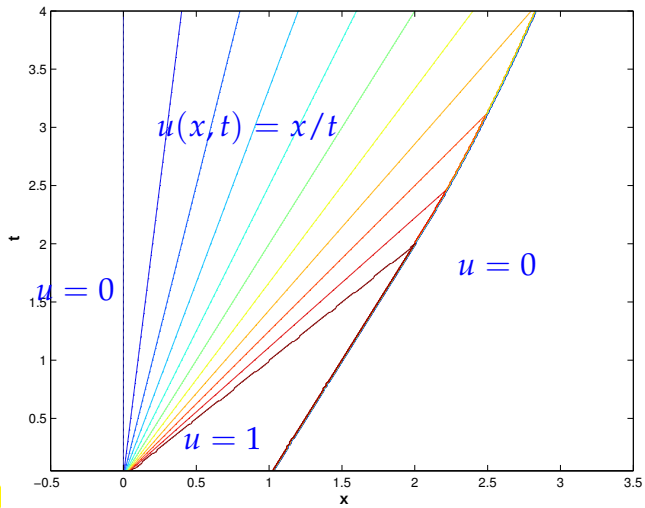


Fig. 380

Vector field in $x - t$-plane

$$\begin{bmatrix} f(u(x,t)) \\ u(x,t) \end{bmatrix}$$

for entropy solution $u = u(x,t)$ ▷
Observe the normal continuity across the shock: the vector field is tangential to the shock curve.

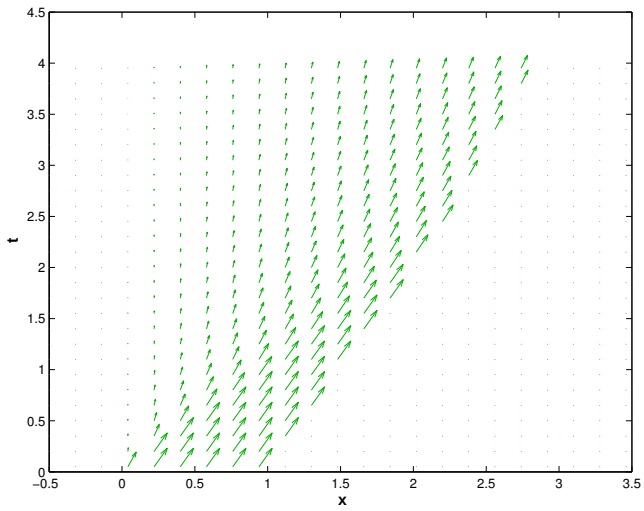


Fig. 381

Example 8.2.40 (Entropy solution of Traffic Flow equation)

An analytic solution is also available for the traffic flow eqution (8.1.41) with intial data, see [23, Sect. 3.4, Ex. 3]

$$u_0(x) = \begin{cases} 0.5 & \text{, if } x < 0 \text{ or } x > 1 \text{ ,} \\ 1 & \text{, if } 0 \leq x \leq 1 \text{ .} \end{cases}$$

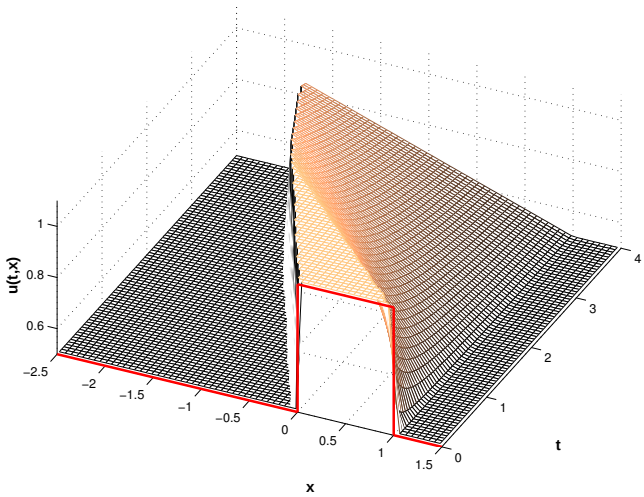


Fig. 382

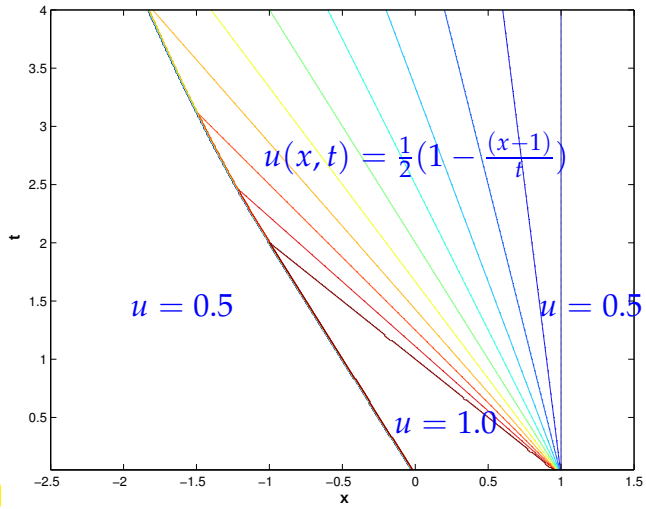


Fig. 383

Vector field in $x - t$-plane

$$\begin{bmatrix} f(u(x,t)) \\ u(x,t) \end{bmatrix}$$

for entropy solution $u = u(x,t)$ ▷.
Observe the normal continuity across the shock!

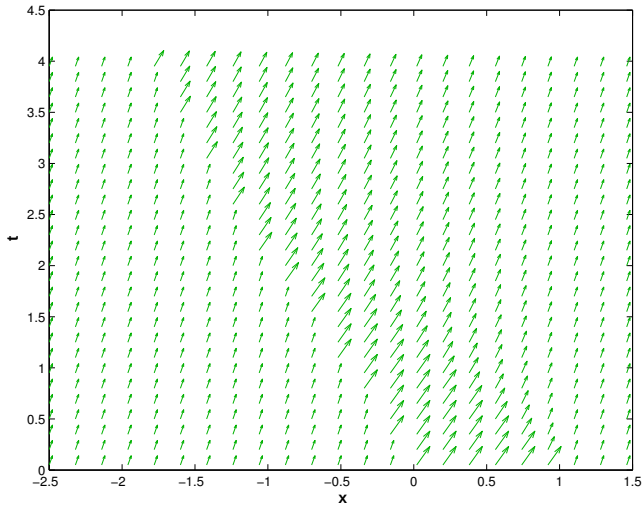


Fig. 384

8.2.7 Properties of entropy solutions

Existence and uniqueness of entropy solutions for 1D scalar conservation laws is guaranteed by theory.

Setting: $u \in L^\infty(\mathbb{R}\times]0,T[)$ weak (→ Def. 8.2.20) entropy solution of Cauchy problem

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R}\times]0,T[\quad , \quad u(x,0) = u_0(x) \ , \quad x \in \mathbb{R} \ . \tag{8.2.7}$$

with flux function $f \in C^1(\mathbb{R})$ (not necessarily convex/concave).

Notation: $\bar{u} \in L^\infty(\mathbb{R}\times]0,T[) \hat{=}$ entropy solution w.r.t. initial data $\bar{u}_0 \in L^\infty(\mathbb{R})$.

Theorem 8.2.41. Comparison principle for scalar conservation laws

If $u_0 \leq \bar{u}_0$ *a.e. on* $\mathbb{R}$ $\Rightarrow$ $u \leq \bar{u}$ *a.e. on* $\mathbb{R}\times]0,T[$

With obvious consequences, because we get constant solutions for constant initial values:

◤ $$u_0(x) \in [\alpha,\beta] \text{ on } \mathbb{R} \quad \Rightarrow \quad u(x,t) \in [\alpha,\beta] \text{ on } \mathbb{R}\times]0,T[$$

Note: this guarantees the normalization condition $0 \leq u(x,t) \leq 1$ for the traffic flow model, if it is satisfied for the initial data u_0.

L^∞-stability (➢ no blow-up can occur!)

$$\forall 0 \le t \le T: \quad \|u(\cdot,t)\|_{L^\infty(\mathbb{R})} \le \|u_0\|_{L^\infty(\mathbb{R})} \,. \tag{8.2.42}$$

Theorem 8.2.43. L^1-contractivity of evolution for scalar conservation law

$$\forall t \in]0,T[,\, R>0: \quad \int_{|x|<R} |u(x,t)|\,\mathrm{d}x \le \int_{|x|<R+\dot{s}t} |u_0(x)|\,\mathrm{d}x \,,$$

with *maximal speed of propagation*

$$\dot{s} := \max\{|f'(\xi)|: \inf_{x\in\mathbb{R}} u_0(x) \le \xi \le \sup_{x\in\mathbb{R}} u_0(x)\} \,. \tag{8.2.44}$$

Thm. 8.2.43 ➤ finite speed of propagation in conservation law, bounded by $\dot{s}$ from (8.2.44):

As in the case of the wave equation → Sect. 6.2.2:

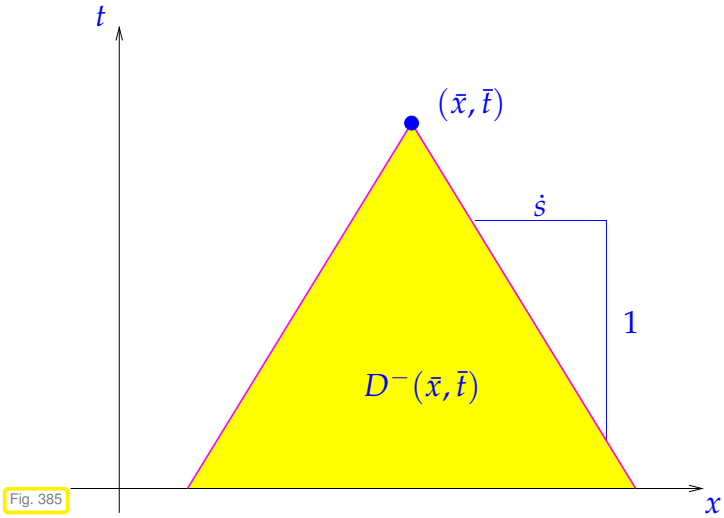


Fig. 385

◁ maximal domain of dependence of $(\bar{x},\bar{t}) \in \widetilde{\Omega}$

$$D^-(\bar{x},\bar{t}) := \left\{(x,t) \in \mathbb{R}\times\mathbb{R}^+ :\ \bar{x} - \dot{s}t \le x \le \bar{x} + \dot{s}t\right\} \,.$$

(Characteristics through a point outside $D^-(\bar{x},\bar{t})$ can never hit $(\bar{x},\bar{t}) \in \widetilde{\Omega}$.)

maximal domain of influence of $I_0 \subset \mathbb{R}$ ▷

For $I_0 = [a,b]$

$$D^+([a,b]) := \left\{(x,t) \in \mathbb{R}\times\mathbb{R}^+ :\ a - \dot{s}t \le x \le b + \dot{s}t\right\} \,.$$

(Characteristics starting in I_0 will always remain in $D^+(I_0)$.)

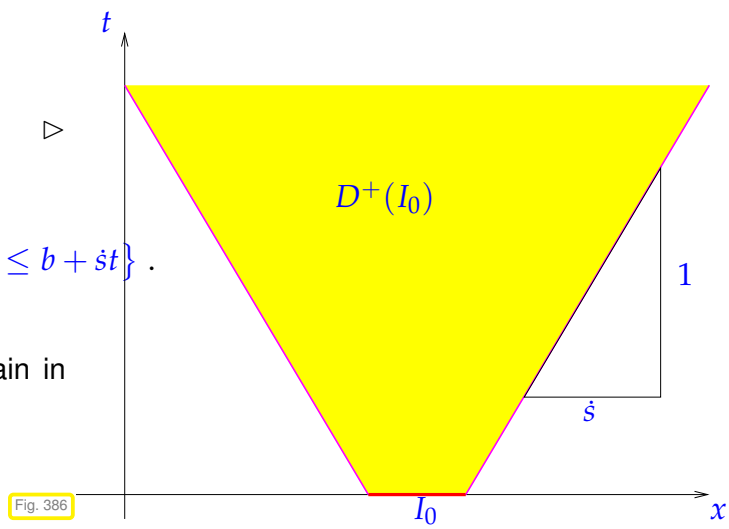


Fig. 386

Analoguous to Thm. 6.2.25:

Corollary 8.2.45. Domain of dependence for scalar conservation law $\rightarrow$ **[19, Cor. 6.2.2]**

The value of the entropy solution at $(\bar{x},\bar{t}) \in \widetilde{\Omega}$ *depends only on the restriction of the initial data to* $\{x \in \mathbb{R}: |x-\bar{x}| < \dot{s}\bar{t}\}$, *where* $\dot{s}$ *is defined in* (8.2.44).

Another strand of theoretical results asserts that the solution of a 1D scalar conservation law cannot develop oscillations:

u solves (8.2.7) ➤ No. of local extrema (in space) of $u(\cdot,t)$ decreasing with time

?! *Review question(s) 8.2.46.* (Scalar conservation laws in 1D)

1. Write down the general form of a Cauchy problem for a 1D scalar conservation law (without source terms).
2. For a scalar 1D conservation law with flux functions
   (a) $f(u) = u^2$,
   (b) $f(u) = \sin(\pi u)$,
   (c) $f(u) = \cos(\pi u)$

   and initial data $u_0(x) = 1$ for $-1 \leq x \leq 1$, $u_0(x) = 0$ elsewhere, sketch the family of characteristic curves ($\rightarrow$ Def. 8.2.9) in a $x-t$ diagram.
3. Show that $u(x,t) = u_0(x-vt)$, $u_0 \in L^\infty(\mathbb{R})$, is a weak solution of the linear advection equation $\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0$.
4. What is the Lax entropy condition and why is it important?
5. For $u_0(x) = 0$ for $x < 0$, $u_0(x) = 1$ for $x \geq 0$ give the formulas for the entropy solutions of the Riemann problems for the scalar 1D conservation laws with flux functions
   (a) $f(u) = u^4$,
   (b) $f(u) = \log(1+u)$,
   (c) $f(u) = 1 - e^u$,
   (d) $f(u) = \frac{1}{1+u}$.
6. Explain the notions of "domain of dependence" and "domain of influence" in connection with Cauchy problems for 1D scalar conservation laws.
7. Which formula yields the maximal speed of propagation for a Cauchy problem for a 1D scalar conservation law?

8.3 Conservative finite volume discretization

Example 8.3.1 (Naive finite difference scheme)

A popular way to discretize PDEs in a single space dimension is the finite difference approach, discussed for second-order two-point boundary value problems in Section 1.5.4. The simple idea is to replace derivatives by difference quotients anchored on a spatial grid, see § 1.5.142 and § 1.5.143.

Noe we present a warning example that pursuing this policy for conservation laws may yield a spurious scheme. We consider the Cauchy problem for Burgers equation (8.1.46) rewritten using product rule:

$$\frac{\partial u}{\partial t}(x,t) + u(x,t)\frac{\partial u}{\partial x}(x,t) = 0 \quad \text{in} \quad \mathbb{R}\times]0,T[\,.$$

$\leftrightarrow$ related to linear advection with velocity $v(x,t) = u(x,t)$:

$$\begin{array}{ccccccc} \frac{\partial u}{\partial t}(x,t) & + & u(x,t)\frac{\partial u}{\partial x}(x,t) & = & 0 & \text{in} & \mathbb{R}\times]0,T[\,. \\ \updownarrow & & \updownarrow & & & & \\ \frac{\partial u}{\partial t}(x,t) & + & v(x,t)\frac{\partial u}{\partial x}(x,t) & = & 0 & \text{in} & \mathbb{R}\times]0,T[\,. \end{array}$$

If $u_0(x) \geq 0$, then, by Thm. 8.2.41, $u(x,t) \geq 0$ for all $0 < t < T$, that is, positive direction of transport throughout.

Heeding the guideline from Section 7.3.1 we use an upwind discretization (backward differences) in space, which amounts to approximating $\frac{\partial u}{\partial x}$ by means of a one-sided difference quotient.

On an (infinite) equidistant spatial grid with meshwidth $h > 0$, that is, $x_j := hj$, $j \in \mathbb{Z}$, we obtain a semi-discrete problem for nodal values $\mu_j = \mu_j(t) \approx u(x_j,t)$

$$\begin{array}{ccccccc} \frac{\partial u}{\partial t}(x,t) & + & u(x,t)\frac{\partial u}{\partial x}(x,t) & = & 0 & \text{in} & \mathbb{R}\times]0,T[\,. \\ \updownarrow & & \updownarrow & & & & \\ \dot{\mu}_j(t) & + & \mu_j\dfrac{\mu_j - \mu_{j-1}}{h} & = & 0\,, & j\in\mathbb{Z}\,, & 0<t<T\,. \end{array} \tag{8.3.2}$$

Our numerical experiment tackles the Cauchy problem from Ex. 8.2.39, "box shaped" initial data u_0, $h = 0.08$, integration of (8.3.2) with adaptive explicit Runge-Kutta method `ode45`.

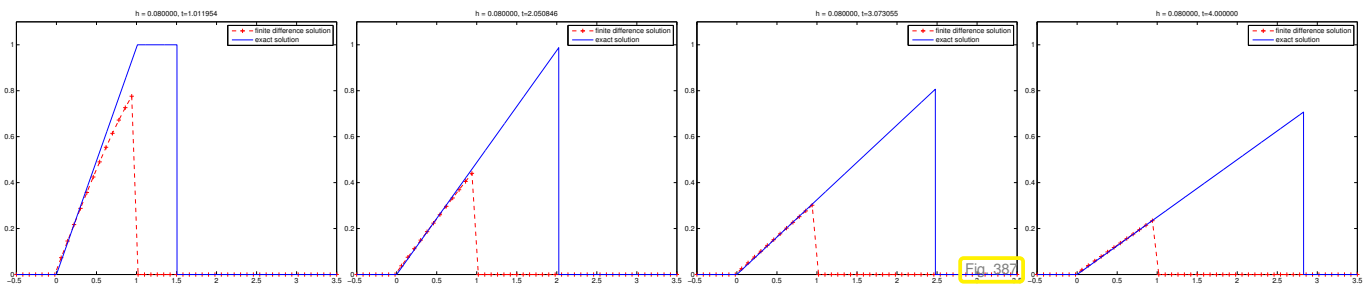


Fig. 387

Observation from numerical experiment: OK for rarefaction wave, but *scheme cannot capture speed of shock correctly*!

To understand the behavior of the scheme, we consider the Riemann problem with $u_0(x) = 1$ for $x < 0 - \epsilon$, and $u_0(x) = 0$ for $x > 0 - \epsilon$, $\epsilon \ll 1$. Accordingly, we choose as initial value for the semidiscrete evolution

$$\mu_j(0) = \begin{cases} 1 & \text{, if } j < 0\,, \\ 0 & \text{, if } j \geq 0\,, \end{cases}$$

Then, it is easy to see that $\dot{\mu}_j = 0$ for all $j \in \mathbb{Z}$.

Entropy solution (for this u_0) = travelling shock ($\rightarrow$ Lemma 8.2.29), speed $\dot{s} = \frac{1}{2} > 0$	$\lhd\rhd$	Numerical solution: $\vec{\mu}(t) = \vec{\mu}_0$ for all $t > 0$!

➤ 3-point FDM (8.3.2) "converges" to wrong solution !

In the next section we will learn an approach to the discretization of 1D conservation laws that has some built-in safeguards against failures as confronted in the above example.

8.3.1 Semi-discrete conservation form

Objective: spatial semi-discretization of a Cauchy problem for a general scalar conservation law in one spatial dimension:

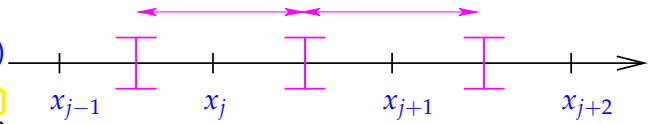
$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R}\times]0,T[\quad , \quad u(x,0) = u_0(x) \ , \quad x \in \mathbb{R} \ . \tag{8.2.7}$$

on an (infinite) equidistant spatial mesh with mesh width $h > 0$.

Remember: We have already seen spatial semi-discretization in the context of the method of lines, see Section 6.1.4. In a sense, our treatment of conservation laws follows a method of lines approach.

$$\mathcal{M} := \{]x_{j-1}, x_j[: x_j := jh, \ j \in \mathbb{Z}\} \ . \tag{8.3.3}$$

mesh cells and dual cells ▷ Fig. 388



The time-dependent unknowns of the semi-discrete scheme will be denoted by $\mu_j = \mu_j(t)$, $j \in \mathbb{Z}$. They play a similar role as the time-dependent basis expansion coefficients occurring as components of the vector $\vec{\mu} = \vec{\mu}(t)$ in the method of lines ODE Eq. (6.1.30).

We adopt a finite volume interpretation of the coefficients/unknowns $\mu_j(t)$, $j \in \mathbb{Z}$):

$\mu_j \leftrightarrow$ conserved quantities in dual cells $]x_{j-1/2}, x_{j+1/2}[$, midpoints $x_{j-1/2} := \frac{1}{2}(x_j + x_{j-1})$:

$$\boxed{\mu_j(t) \approx \frac{1}{h} \int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\,\mathrm{d}x} \ . \tag{8.3.4}$$

Relate $\vec{\mu}(t) := (\mu_j(t))_{j\in\mathbb{Z}} \in \mathbb{R}^{\mathbb{Z}} \quad \longleftrightarrow \quad u_N(x,t) = \sum_{j\in\mathbb{Z}} \mu_j(t)\, \chi_{]x_{j-1/2},x_{j+1/2}[}(x) \ .$ (8.3.5)

a function !

✎ notation: characteristic function $\chi_{]x_{j-1/2},x_{j+1/2}[}(x) = \begin{cases} 1 & \text{, if } x_{j-1/2} < x \le x_{j+1/2} \ , \\ 0 & \text{elsewhere.} \end{cases}$

➥ $(\mu_j(t))_{j\in\mathbb{Z}} \quad \longleftrightarrow \quad$ *piecewise constant* approximation $u_N(t) \approx u(\cdot,t)$

Note:
$u_N(t)$ is discontinuous at dual cell boundaries $x_{j+1/2}$!

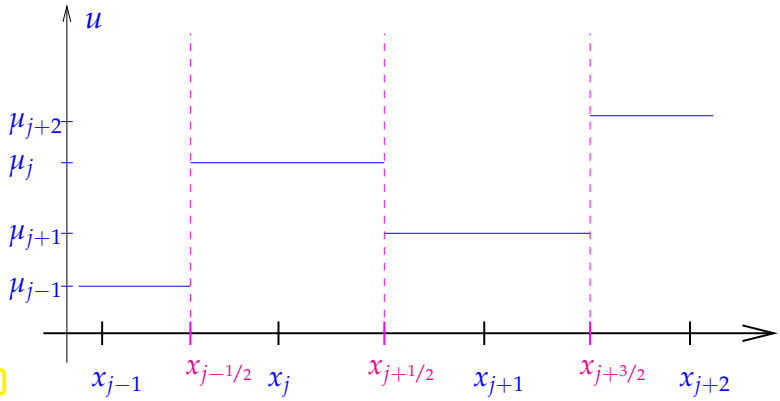


Fig. 389

By spatial integration over dual cells, which now play the role of the control volumes in (8.2.1), and applying the fundamental theorem of calculus, we obtain

$$\frac{d}{dt}\int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\,\mathrm{d}x + f(u(x_{j+1/2},t)) - f(u(x_{j-1/2},t)) = 0\,, \quad j \in \mathbb{Z}\,, \tag{8.3.6}$$

(8.3.4) ▶

$$\frac{d\mu_j}{dt}(t) + \frac{1}{h}\Big(\underbrace{f(u_N(x_{j+1/2},t))}_{?} - \underbrace{f(u_N(x_{j-1/2},t))}_{?}\Big) = 0\,, \quad j \in \mathbb{Z}\,. \tag{8.3.7}$$

Problem: owing to the jumps of $u_N(t)$ we face the ambiguity of the values $u_N(x_{j+1/2},t)$, $u_N(x_{j-1/2},t)$. (We encountered a similar situation it in the context of upwind quadrature in Section 7.2.2.1.)

Abstract "solution":

Approximation $\quad f(u_N(x_{j+1/2},t)) \approx f_{j+1/2}(t) := F(\mu_{j-m_l+1}(t),\dots,\mu_{j+m_r}(t))\,, \quad j \in \mathbb{Z}\,,$

with numerical flux function $F : \mathbb{R}^{m_l+m_r} \mapsto \mathbb{R}, \quad m_l, m_r \in \mathbb{N}_0$.

Note: If $f = f(u)$, then the **same** numerical flux function is usually used for all dual cells!

When we plug this approximation into (8.3.7) we end up with the following (formally infinite) system of ODEs:

Finite volume semi-discrete evolution for (8.2.7) **in conservation form**

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_{j-m_l+1}(t),\dots,\mu_{j+m_r}(t)) - F(\mu_{j-m_l}(t),\dots,\mu_{j+m_r-1}(t))\big)\,, \quad j \in \mathbb{Z}\,. \tag{8.3.9}$$

numerical flux (function) $F : \mathbb{R}^{m_l+m_r} \mapsto \mathbb{R}$

Special case: 2-point numerical flux ($m_l = m_r = 1$): $\quad F = F(v,w)$
($v \mathrel{\hat{=}}$ left state, $w \mathrel{\hat{=}}$ right state)

(8.3.9) ▶

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big)\,, \quad j \in \mathbb{Z}\,. \tag{8.3.10}$$

Assumption on numerical flux functions: F Lipschitz-continuous in each argument.

MATLAB Code 8.3.11: Wrapper code for finite volume evolution with 2-point flux

```
function ufinal = consformevl(a,b,N,u0,T,F)
% finite volume discrete evolution in conservation form with 2-point flux,
% see (8.3.10)
% Cauchy problem over time [0,T] restricted to finite interval [a,b],
% equidistant mesh with meshwidth N cells, meshwidth h := b-a/N.
% 2-point numerical flux function F = F(v,w) passed in handle F
h = (b-a)/N; x = a+0.5*h:h:b-0.5*h; % centers of dual cells
% vector μ_0 of initial cell averages (column vector)
% approximated by means of composite midpoint rule (1.5.86).
mu0 = u0(x)';
% right hand side function for MATLAB ode solvers
odefun = @(t,mu) (-1/h*fluxdiff(mu,F));
% Method of lines approach, c.f. Sect. 6.1.4: timestepping by
% MATLAB standard integrator (explicit Runge-Kutta method of order 5, Def. 6.1.40)
options = odeset('abstol',1E-8,'reltol',1E-6,'stats','on');
[t,MU] = ode45(odefun,[0 T],mu0,options);
% 3D graphical output of u(x,t) over space-time plane
[X,T] = meshgrid(x,t);
figure; surf(X,T,MU/h); colormap(copper);
xlabel('{\bf x}','fontsize',14);
ylabel('{\bf t}','fontsize',14);
zlabel('{\bf u}','fontsize',14);
ufinal = MU(:,end);
end

% difference of numerical fluxes on right hand side of (8.3.10)
function fd = fluxdiff(mu,F)
n = length(mu); fd = zeros(n,1);
% constant continuation of data outside [a,b]
fd(1) = F(mu(1),mu(2)) - F(mu(1),mu(1));
for j=2:n-1
  fd(j) = F(mu(j),mu(j+1)) - F(mu(j-1),mu(j)); % see (8.3.10)
end
fd(n) = F(mu(n),mu(n)) - F(mu(n-1),mu(n));
end
```

C++11 EIGEN code 8.3.12: Right hand side function for MOL-ODE (8.3.10) ➙ **GITLAB**

```
// arguments:
// (Finite) state vector μ of cell averages, see (8.3.4)
// Functor F : ℝ × ℝ ↦ ℝ, 2-point numerical flux
//
// return value:
// Vector with differences of numerical fluxes, which provides the
// right hand side of (8.3.10)
template<typename FunctionF>
VectorXd fluxdiff(const VectorXd& mu, FunctionF F) {
```

```
  unsigned n = mu.size(); // length of state vector
  VectorXd fd = VectorXd::Zero(n); // return vector

  // constant continuation of data for x ≤ a!
  fd[0] = F(mu[0],mu[1]) - F(mu[0],mu[0]);
  for (unsigned j=1; j < n-1; ++j) {
    fd[j] = F(mu[j],mu[j+1]) - F(mu[j-1],mu[j]); // see (8.3.10)
  }
  // constant continuation of data for x ≥ b!
  fd[n-1] = F(mu[n-1],mu[n-1]) - F(mu[n-2],mu[n-1]);
  // Efficient thanks to return value optimization (RVO)
  return fd;
}
```

C++11 EIGEN code 8.3.13: Wrapper code for finite volume evolution with 2-point flux
➙ GITLAB

```
// arguments:
// Real numbers a,b, the boundaries of the interval,
// unsigned int N, the number of cells,
// Functor u0 : ℝ ↦ ℝ, initial value,
// Final time T > 0,
// Functor F = F(v,w) for 2-point numerical flux function.
//
// return value:
// Vector with cell values at final time T
//
// Finite volume discrete evolution in conservation form with 2-point
// flux, see (8.3.10); Cauchy problem over time [0,T]
template<typename FunctionU0, typename FunctionF>
VectorXd consformevl(double a, double b, unsigned N,
                     FunctionU0 u0, double T, FunctionF F) {
  double h = (b-a)/N; // meshwidth
  // centers of dual cells
  VectorXd x =  VectorXd::LinSpaced(N, a+0.5*h, b-0.5*h);

  // vector μ⃗_0 of initial cell averages
  // obtained by point sampling of u_0 in grid points
  VectorXd mu0 = x.unaryExpr(u0);

  // right hand side function for ode solver
  auto odefun = [&] (const VectorXd& mu, VectorXd& dmdt, double t) {
    dmdt = -1./h*fluxdiff<FunctionF>(mu,F);  };

  // Method of lines approach, c.f.  Sect. 6.1.4:  timestepping by
  // Boost integrator (adaptive explicit Runge-Kutta method
  // of order 5, see also Def. 6.1.40)
  double abstol = 1E-8, reltol = 1E-6; // integration control
      parameters
  VectorXd t; // Returns temporal grid
```

```
  MatrixXd MU; // Returns state matrix
  std::tie(t, MU) = NPDE::ode45(odefun, 0, T, mu0, abstol, reltol); //
  // Final state vector is the rightmost column of MU.
  return MU.col(t.size()-1);
}
```

Note that in Code 8.3.13 we rely on high-order explicit Runge-Kutta timestepping in order to solve (8.3.9) approximately.

8.3.2 Discrete conservation property

We consider a Cauchy problem for a scalar conservation law

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R}\times]0,T[\quad , \quad u(x,0) = u_0(x) \ , \quad x \in \mathbb{R} \ . \tag{8.2.7}$$

and its conservative finite volume discretization on an (infinite) equidistant spatial mesh with mesh width $h > 0$:

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\left(F(\mu_{j-m_l+1}(t),\dots,\mu_{j+m_r}(t)) - F(\mu_{j-m_l}(t),\dots,\mu_{j+m_r-1}(t))\right) \ , \quad j \in \mathbb{Z} \ . \tag{8.3.9}$$

We abbreviate $$f_{j+1/2}(t) := F(\mu_{j-m_l+1}(t),\dots,\mu_{j+m_r}(t)).$$

(8.3.14) Preservation of constant data

An evident first property of finite volume methods in conservation form:

$$\mu_j(0) = \mu_0 \in \mathbb{R} \quad \forall j \in \mathbb{Z} \quad \Rightarrow \quad \mu_j(t) = \mu_0 \quad \forall j \in \mathbb{Z}, \quad \forall t > 0 \ . \tag{8.3.15}$$

that is, constant solutions are preserved by the method. Such methods are called well-balanced discretizations.

(8.3.16) Discrete flux balance

For conservation laws we found the fundamental local balance relation, see (6.1.3):

$$\frac{d}{dt}\int_a^b u(x,t)\,\mathrm{d}x = -(f(u(b,t)) - f(u(a,t))) \ . \tag{8.3.17}$$

A "telescopic sum argument" combined with the interpretation (8.3.5) shows that the conservation form (8.3.9) of the semi-discrete conservation law implies

$$\frac{d}{dt}\int_{x_{k-1/2}}^{x_{m+1/2}} u_N(x,t)\,\mathrm{d}x = h\sum_{l=k}^{m}\frac{d\mu_j}{dt}(t) = -\left(f_{m+1/2}(t) - f_{k-1/2}(t)\right) \quad \forall k,m \in \mathbb{Z} \ .$$

$$\Updownarrow$$

$$\frac{d}{dt}\int_{x_{k-1/2}}^{x_{m+1/2}} u(x,t)\,\mathrm{d}x = -\big(f(u(x_{m+1/2},t)) - f(u(x_{k-1/2},t))\big)\ ,$$

► With respect to unions of dual cells and numerical fluxes, the semidiscrete solution $u_N(t)$ satisfies a balance law of the same structure as a (weak) solution of (8.2.7).

Of course, the numerical flux function F has to fit the flux function f of the conservation law; the following is a minimal requirement for a viable numerical flux function.

Definition 8.3.18. 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

$$F(u,\dots,u) = f(u) \quad \forall u \in \mathbb{R}\ .$$

(8.3.19) Discrete shock speed

Focus: solution of Riemann problem (→ Def. 8.2.5) by finite volume method in conservation form (8.3.9):

Initial data "constant at $\pm\infty$": $\quad \mu_{-j}(0) = u_l \quad , \quad \mu_j(0) = u_r \quad$ for large j.

Consistency of the numerical flux function implies for large $m \gg 1$

$$\frac{d}{dt}\int_{-x_{-m-1/2}}^{x_{m+1/2}} u_N(x,t)\,\mathrm{d}x = -\big(F(u_r,\dots,u_r) - F(u_l,\dots,u_l)\big) = -\big(f(u_r) - f(u_l)\big)\ . \tag{8.3.20}$$

Exactly the same balance law holds for any weak solutions of the Riemann problem!

Situation : $u_r > u_l$ ➢ shock in traffic flow, discrete solution $u_N(t)$ increasing & supposed to *approximate* a shock; we cannot expect that u_N will also feature a sharp discontinuity, rather we may see a "smeared" transition from u_l to u_r.

Write $x_*(t) \in \mathbb{R}$ for the approximate location of the shock at time t, *defined* as

$$\int_{-\infty}^{x_*(t)} u_N(x,t) - u_l\,\mathrm{d}x = \int_{x_*(t)}^{\infty} u_r - u_N(x,t)\,\mathrm{d}x$$

equality of yellow areas ▷

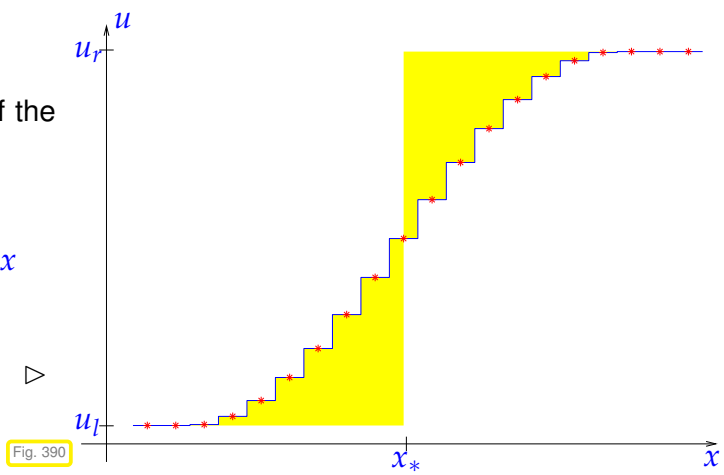


Fig. 390

► $$\int_{x_{-m-1/2}}^{x_{m+1/2}} u_N(x,t)\,\mathrm{d}x = (x_*(t) + x_{-m-1/2})u_l + (x_{m+1/2} - x_*(t))u_r\ .$$

$$\overset{(8.3.20)}{\Longrightarrow} \quad \frac{dx_*}{dt}(t) = \frac{1}{u_l - u_r} \sum_{j\in\mathbb{Z}} \frac{d\mu_j}{dt}(t) = \frac{f(u_l) - f(u_r)}{u_l - u_r} \overset{(8.2.23)}{=} \dot{s} \ .$$

Conservation form with consistent numerical flux yields correct "discrete shock speed"
(immune to spurious shock speeds as observed in Ex. 8.3.1)

8.3.3 Numerical flux functions

In this section concrete choices of consistent ($\rightarrow$ Def. 8.3.18) numerical flux functions will be presented and discussed. We restrict ourselves to 2-point numerical fluxes $F = F(v,w)$, $v \mathrel{\hat{=}}$ "left state", $w \mathrel{\hat{=}}$ "right state", see page 595.

It will turn out that finding appropriate numerical flux functions is by no means straightforward, because both instability and numerical solutions that violate the entropy condition (*to* Sect. 8.2.6) have to be avoided.

8.3.3.1 Central flux

A very simple choice for numerical flux functions relies on arithmetic averaging and yields the two central numerical fluxes

$$F_1(v,w) := \tfrac{1}{2}\big(f(v) + f(w)\big) \quad , \quad F_2(v,w) := f\big(\tfrac{1}{2}(v+w)\big) \ . \tag{8.3.21}$$

Obviously the 2-point numerical fluxes F_1 and F_2 are consistent according to Def. 8.3.18. The resulting spatially semi-discrete schemes are given by, see (8.3.10),

$$\begin{aligned} F_1\colon &\quad \frac{d\mu_j}{dt}(t) = -\frac{1}{2h}\big(f(\mu_{j+1}(t)) - f(\mu_{j-1}(t))\big) \ , \\ F_2\colon &\quad \frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(f(\tfrac{1}{2}(\mu_j(t) + \mu_{j+1}(t))) - f(\tfrac{1}{2}(\mu_j(t) + \mu_{j-1}(t)))\big) \ . \end{aligned}$$

Experiment 8.3.22 (Central flux for Burgers equation)

- Cauchy problem for Burgers equation (8.1.46) (flux function $f(u) = \frac{1}{2}u^2$) from Ex. 8.2.39 ("box" intial data)
- Spatial finite volume discretization in conservation form (8.3.9) with central numerical fluxes according tp (8.3.21).
- timestepping based on adaptive explicit Runge-Kutta method `ode45`.
  (in MATLAB: `opts = odeset('abstol',1E-7,'reltol',1E-6);`).

Fully discrete evolution for central numerical flux F_1: $h = 0.03$

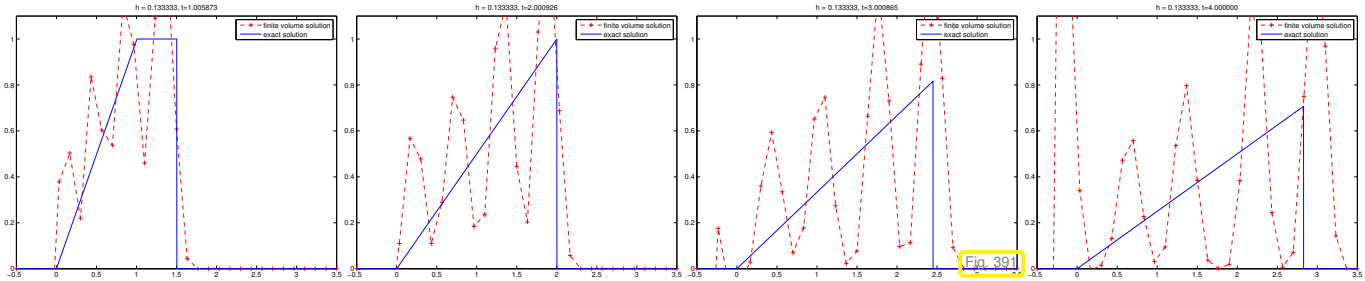
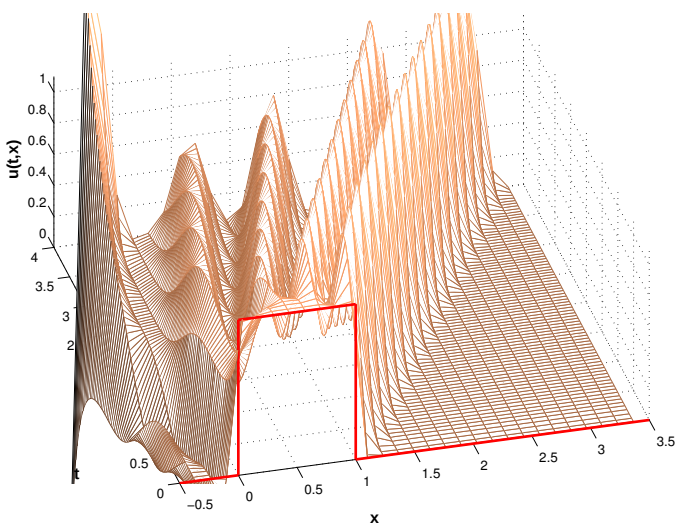
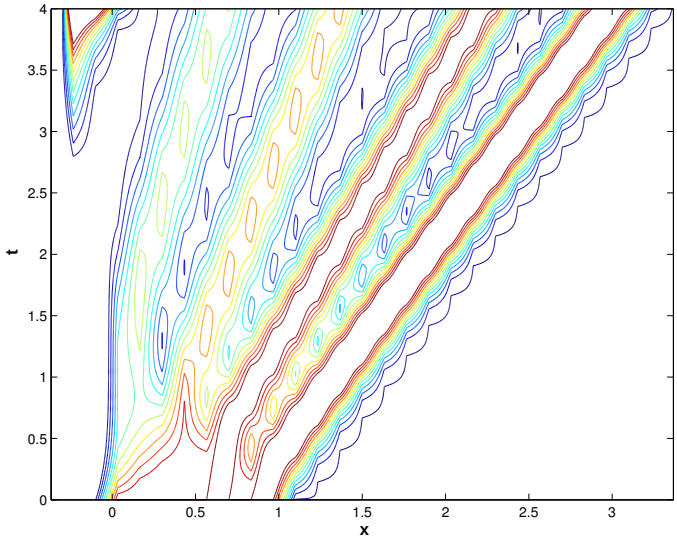


Fig. 391





Fully discrete evolution for central numerical flux F_2: $h = 0.017$

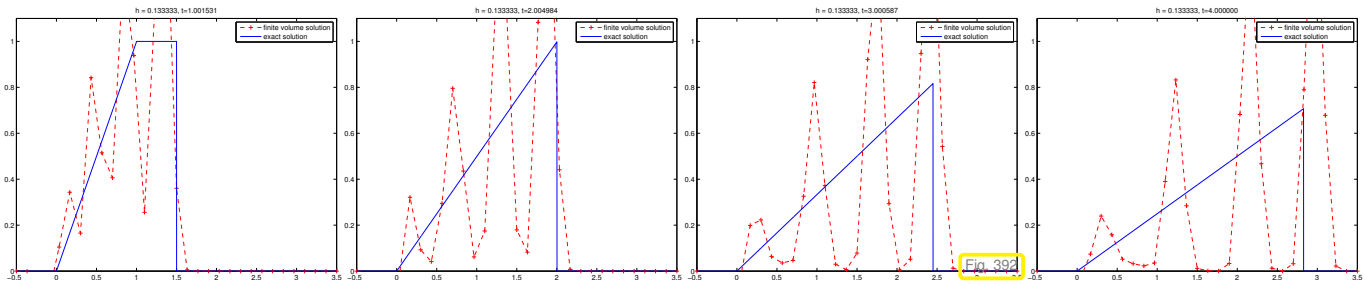
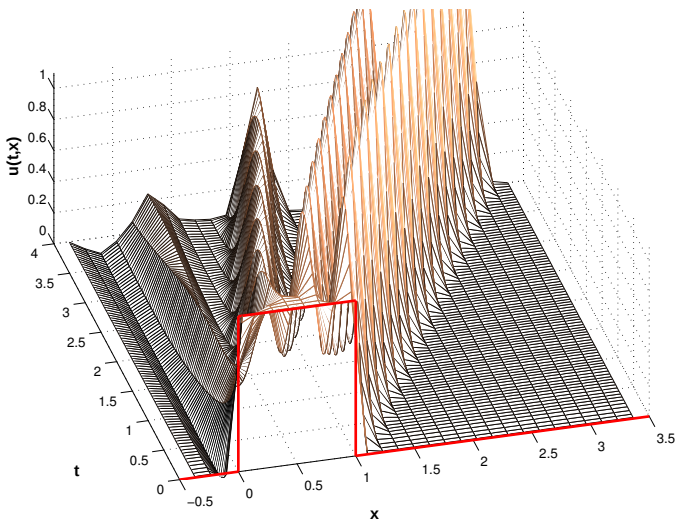
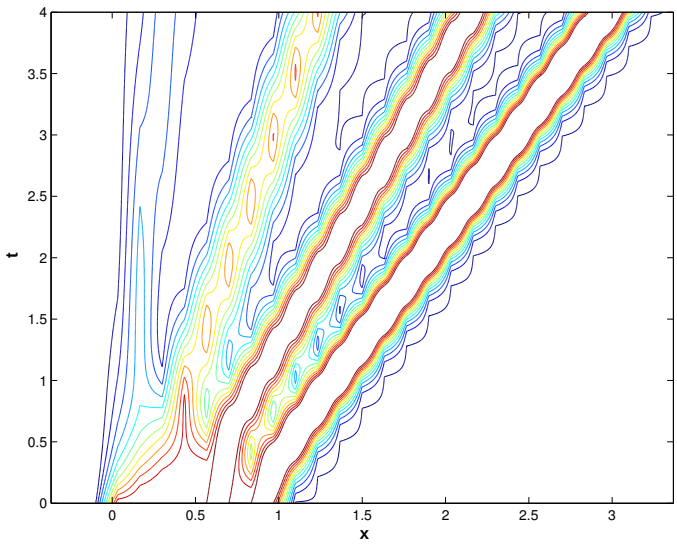


Fig. 392



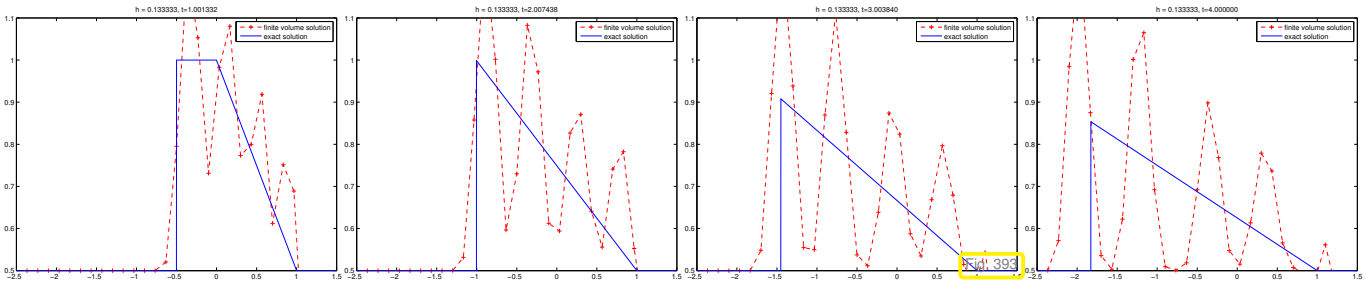


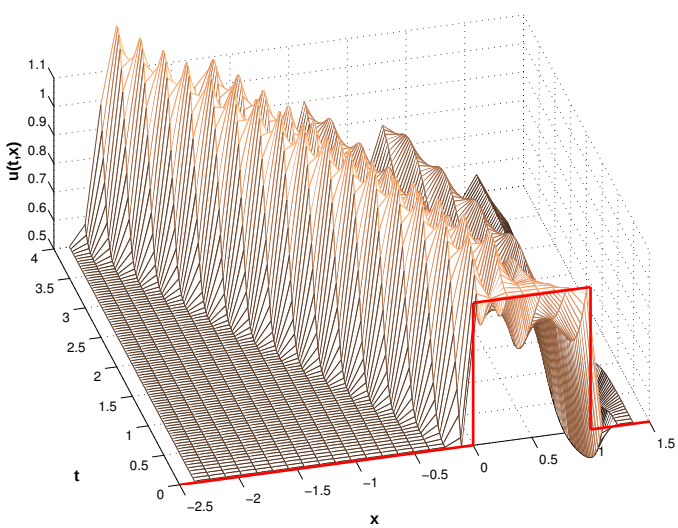
Observation: massive spurious oscillations utterly pollute numerical solution

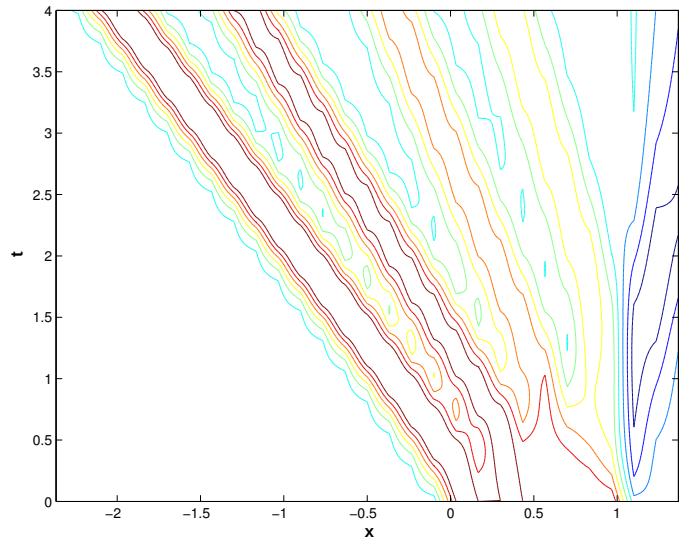
Experiment 8.3.23 (Central flux for Traffic Flow equation)

- Cauchy problem for Traffic Flow equation (8.1.41) (flux function $f(u) = u(1-u)$) from Ex. 8.2.40 ("box" intial data, $u_0 = \chi_{[0,1]}$)
- Spatial finite volume discretization in conservation form (8.3.9) with central numerical fluxes according tp (8.3.21).
- timestepping based on adaptive explicit Runge-Kutta method `ode45`.
  (in MATLAB: `opts = odeset('abstol',1E-7,'reltol',1E-6);`).

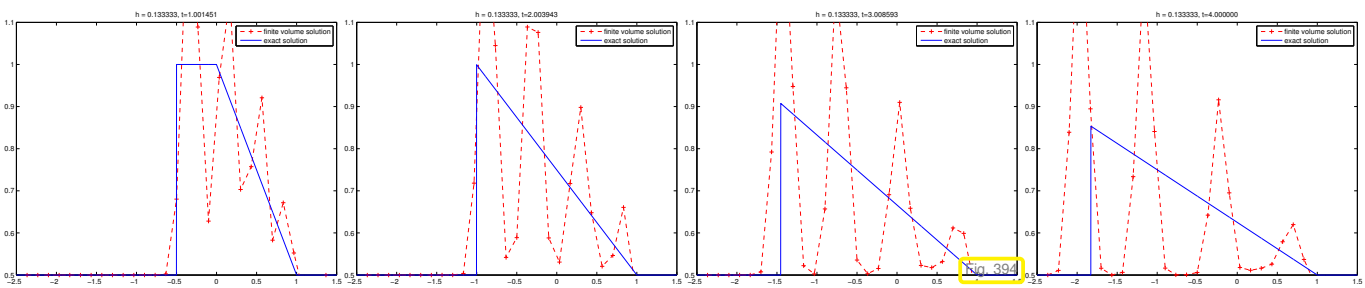
Fully discrete evolution for central numerical flux F_1: $h = 0.03$

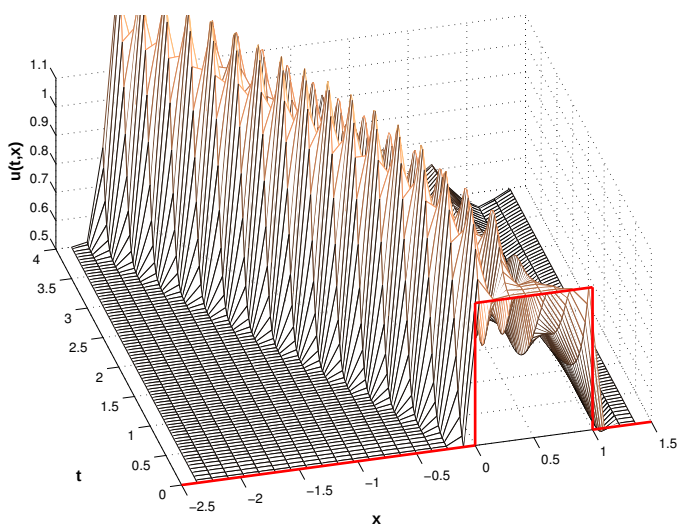


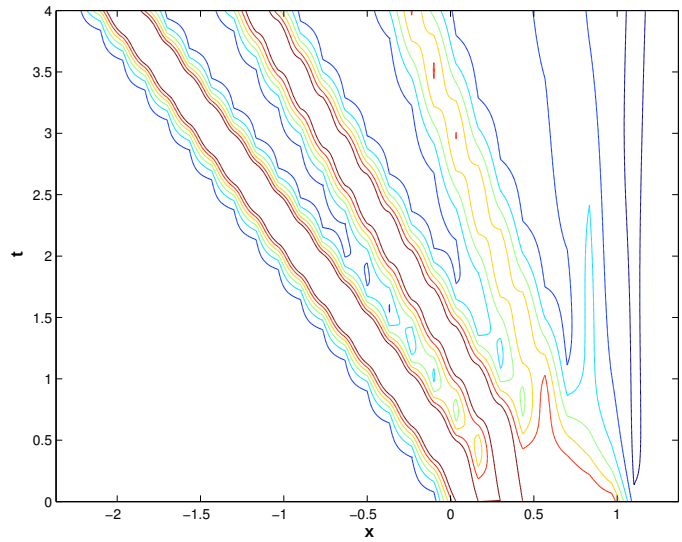




Fully discrete evolution for central numerical flux F_2: $h = 0.017$







Observation: massive spurious oscillations utterly pollute numerical solution

Experiment 8.3.24 (Central flux for linear advection)

In order to see whether the emergence of spurious oscillations is an inherent weakness of central fluxes we apply them to the simplest scalar conservation law, linear advection Section 8.1.1 with constant velocity.

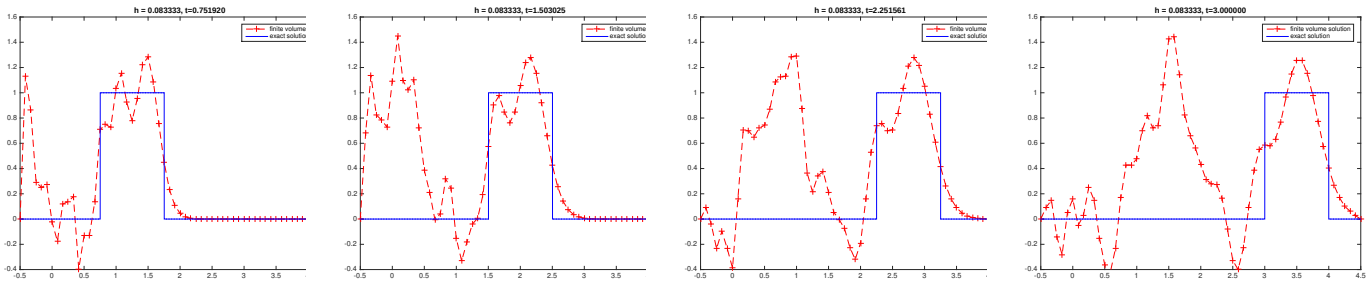
We consider the Cauchy problem (8.1.10): constant velocity scalar linear advection, $c = 1$, flux function $f(u) = cu$

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \quad \text{in} \quad \widetilde{\Omega} = \mathbb{R}\times]0,T[\,, \quad u(x,0) = u_0(x) \quad \forall x \in \mathbb{R}\,. \tag{8.1.10}$$

Finite volume spatial discretization in conservation form (8.3.9) with central numerical fluxes from (8.3.21):

$$\begin{array}{l} F_1(v,w) := \frac{1}{2}\big(f(v)+f(w)\big) \\ F_2(v,w) := f\big(\frac{1}{2}(v+w)\big) \end{array} \quad \Rightarrow \quad \frac{d\mu_j}{dt}(t) = -\frac{c}{2h}\big(\mu_{j+1}(t) - \mu_{j-1}(t)\big)\,, \quad j \in \mathbb{Z}\,. \tag{8.3.25}$$

For the numerical experiment we use "box shaped" initial data $u_0 = \chi_{[0,1]}$, an equidistant spatial mesh with meshwidth $h = 0.083$, `ode45` adaptive explicit Runge-Kutta timestepping.



Fig

Again, we observe tremendous spurious oscillations that render the computed solution completely useless.

Remark 8.3.26 (Connection with convection-diffusion IBVPs → Chapter 7)

Note that the Cauchy problem (8.1.10) is an initial value problem for the 1D transport equation (7.3.7)!

From Section 7.2.2, (7.2.18) we see that the semi-discrete evolution

$$\frac{d\mu_j}{dt}(t) = -\frac{c}{2h}(\mu_{j+1}(t) - \mu_{j-1}(t)) \,, \quad j \in \mathbb{Z} \,, \tag{8.3.25}$$

agrees with what we obtain from straightforward spatial *linear finite element Galerkin semi-discretization*.

In Section 7.3.1 we learned that this method is *prone to spurious oscillations*, see Ex. 7.3.4. This offers an explanation also for its failure for Burgers equation/traffic flow equation, see Exp. 8.3.22.

8.3.3.2 Lax-Friedrichs/Rusanov flux

(8.3.27) Fighting oscillations with diffusion

According to § 8.1.1 the simple linear advection Cauchy problem

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \quad \text{in} \quad \widetilde{\Omega} = \mathbb{R}\times]0,T[\,, \quad u(x,0) = u_0(x) \quad \forall x \in \mathbb{R} \,. \tag{8.1.10}$$

models heat transport in a fluid moving with constant velocity c.

If u_0 is oscillatory (many local extrema), then these will just be carried along. However, if there is a non-zero heat conductivity $\kappa > 0$, then local extrema of the temperature can be expected to decay exponentially, while they are moving with the flow. For instance, for $c = \kappa = 1$ (dimensionless equations), we get

$$\begin{aligned} \tfrac{\partial u}{\partial t} + \tfrac{\partial u}{\partial x} - \tfrac{\partial^2 u}{\partial x^2} &= 0 \,, \\ u_0(x) &= \sin(x) \end{aligned} \quad \blacktriangleright \quad u(x,t) = e^{-t}\sin(x-t) \,, \quad x \in \mathbb{R},\, t \geq 0 \,. \tag{8.3.28}$$

diffusive term

Hence, let us consider the advection equation with *extra added diffusion*, whose strength can be controlled by the diffusion coefficient $\kappa > 0$,

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} - \kappa\frac{\partial^2 u}{\partial x^2} = 0 \,, \tag{8.3.29}$$

which amounts to a 1D scalar conservation law with the flux function ($\rightarrow$ Rem. 8.2.2)

$$f(u) = cu - \kappa\frac{\partial u}{\partial x} \,. \tag{8.3.30}$$

A related numerical flux on an equidistant mesh with meshwidth $h > 0$ can rely on a central flux (8.3.21) for the advective part, and on a simple difference quotient approximation for the derivative

$$\begin{array}{ccccc} f(u) & = & cu & - & \kappa\frac{\partial u}{\partial x} \,, \\ \downarrow & & \downarrow & & \downarrow \\ F(v,w) & = & \frac{c}{2}(v+w) & - & \kappa\frac{w-v}{h} \,. \end{array}$$

central numerical flux diffusive numerical flux

With this choice of numerical flux the semi-discrete evolution (8.3.10) becomes:

$$\dot{\mu}_j(t) + c\frac{\mu_{j+1}(t) - \mu_{j-1}(t)}{2h} + \kappa\frac{-\mu_{j+1}(t) + 2\mu_j(t) - \mu_{j-1}(t)}{h^2} = 0 \,. \tag{8.3.31}$$

(7.2.18), Section 1.5.4 ➢ (8.3.31) agrees with the method-of-lines ODE obtained from the linear finite element Galerkin discretization of (8.3.29) on an equidistant mesh!

Caution: the extra diffusion amounts to a *perturbation* of the Cauchy problem that must be kept as small as possible and, in any case, vanish for $h \to 0$, which entails $\kappa = \kappa(h)$.

Guideline: prevent diffusive flux from dominating central flux ➢ $\boxed{\kappa = \frac{ch}{2}}$ (8.3.32)

Remark 8.3.33 (Connection with artificial viscosity → Section 7.2.2.2)

As already pointed out in Rem. 8.3.26, the developments in this section are closely connected with similar considerations in Section 7.2.2, Section 7.3.1 in the context of stable spatial discretization of convection-diffusion problems (8.3.29).

In Section 7.2.2.2 we saw that artificial diffusion cures instability of central difference quotients. In (7.2.22) we found a new interpretation of the upwind discretization based on one-sided difference quotients:

$$\begin{array}{ccccccccc}
\frac{\partial u}{\partial t} & & & & + & c\frac{\partial u}{\partial x} & = & 0 & \text{in } \mathbb{R}\times]0,T[\,, \\
\updownarrow & & & & & \updownarrow & & & \\
\frac{\partial u}{\partial t} & + & (ch/2) & \underbrace{\frac{-\mu_{j-1}+2\mu_j-\mu_{j+1}}{h^2}}_{\hat{=}\text{ difference quotient for } \frac{d^2u}{dx^2}} & + & \underbrace{c\,\frac{\mu_{j+1}-\mu_{j-1}}{2h}}_{\hat{=}\text{ difference quotient for } c\frac{du}{dx}} & = & 0\,, & j\in\mathbb{Z}\,.
\end{array}$$

Can this be rewritten in conservation form (8.3.9)? YES!

$$(ch/2)\frac{-\mu_{j-1}+2\mu_j-\mu_{j+1}}{h^2} + c\,\frac{\mu_{j+1}-\mu_{j-1}}{2h} = \frac{1}{h}\big(F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big)\,,$$

with $\boxed{F(v,w) := \frac{c}{2}(v+w) - \frac{c}{2}(w-v)}$. (8.3.34)

central numerical flux h-weighted diffusive/viscous numerical flux

Recall from Rem. 8.2.2: the flux function $f(u) = -\frac{\partial u}{\partial x}$ models diffusion. Hence, the diffusive numerical flux amounts to a central finite difference discretization of the partial derivative in space:

$$-\frac{\partial u}{\partial x}(x,t)_{|x=x_{j+1/2}} \approx -\frac{1}{h}\big(u(x_{j+1},t) - u(x_j,t)\big)\,.$$

Thus, starting from upwind discretization, we also arrive at the scheme heuristically derived in § 8.3.27.

How to adapt the idea of extra diffusion to general scalar conservation laws? A simple manipulation connects these with linear advection:

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = \frac{\partial u}{\partial t} + f'(u)\frac{\partial u}{\partial x} = 0 \tag{8.3.35}$$

local speed of transport $\leftrightarrow c$

However, the speed $f'(u)$ of transport will depend on x, which suggests that the strength of artificial diffusion should vary. We choose it according to (8.3.32), but large enough to fit the maximal local velocity: we set $k = \frac{h}{2}\max\{|f'(u)| : \min\{v,w\} \leq u \leq \max\{v,w\}\}$ in the diffusive part of the numerical flux.

(local) Lax-Friedrichs/Rusanov flux

$$\boxed{F_{\mathrm{LF}}(v,w) = \tfrac{1}{2}(f(v)+f(w)) - \tfrac{1}{2}(w-v)\cdot \max_{\min\{v,w\}\leq u\leq \max\{v,w\}} |f'(u)|} \,. \tag{8.3.36}$$

The next two experiments investigate the performance of the (local) Lax-Friedrichs/Rusanov numerical flux for our model non-linear scalar conservation laws.

Example 8.3.37 (Lax-Friedrichs flux for Burgers equation)

- same setting and conservative discretization as in Ex. 8.3.22
- Numerical flux function: Lax-Friedrichs flux (8.3.36)

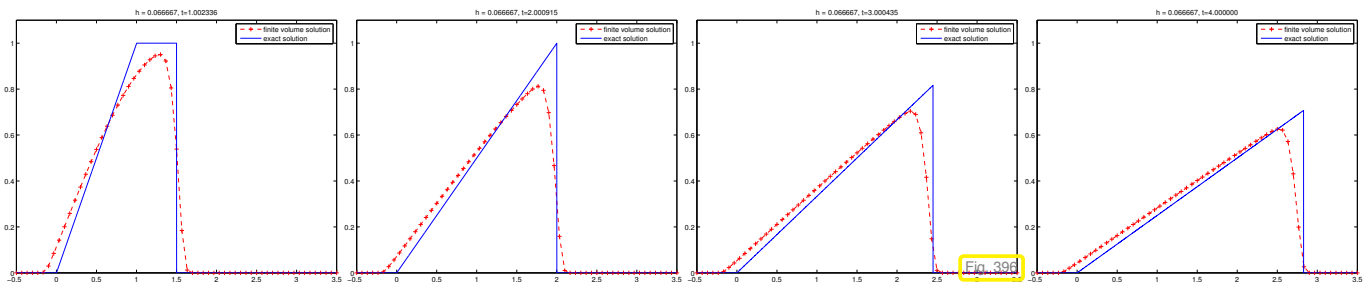
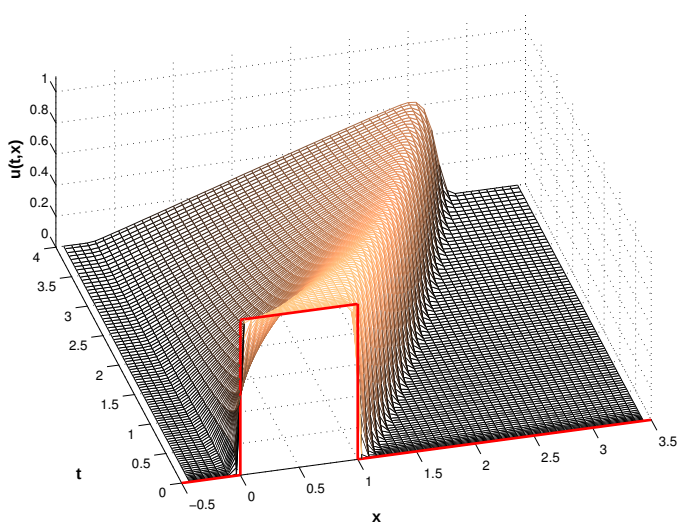
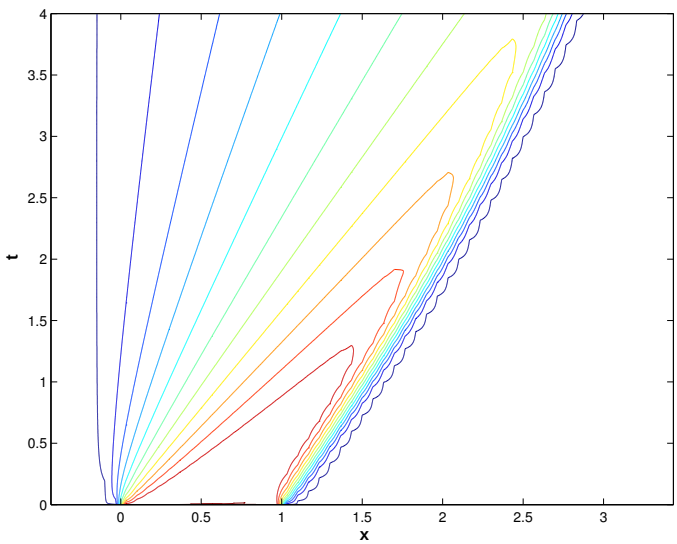


Fig. 396





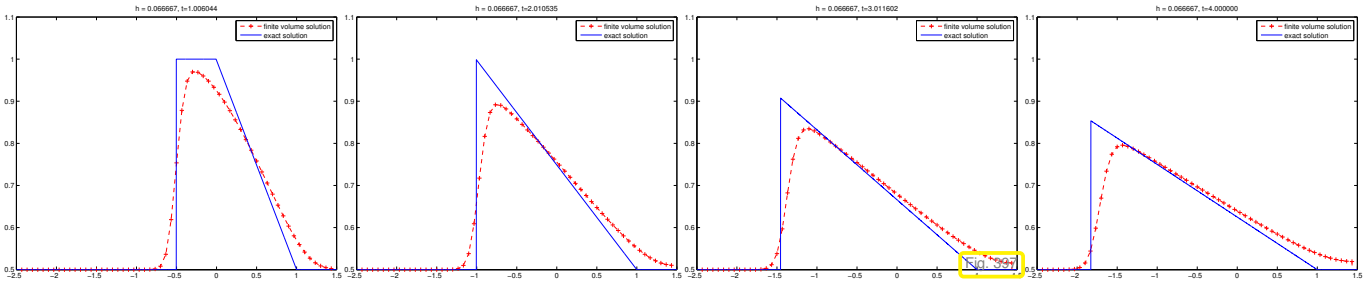
Observation: spurious oscillations are suppressed completely, qualitatively good resolution of both shock and rarefaction.

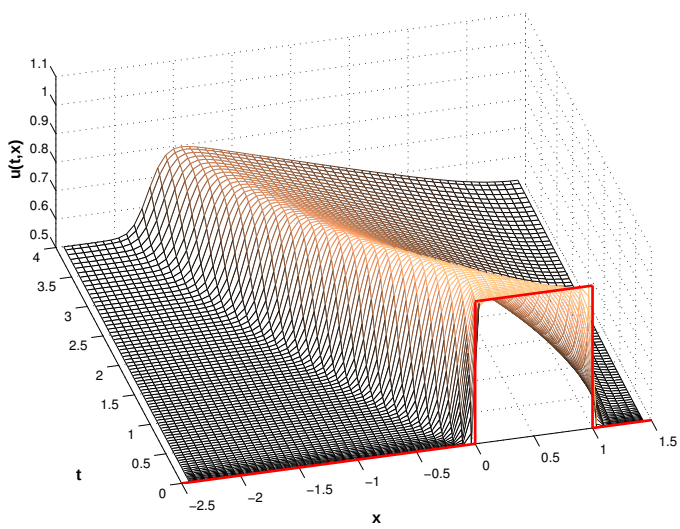
Effect of artificial diffusion: smearing of shock, *cf.* discussion in Ex. 7.2.31.

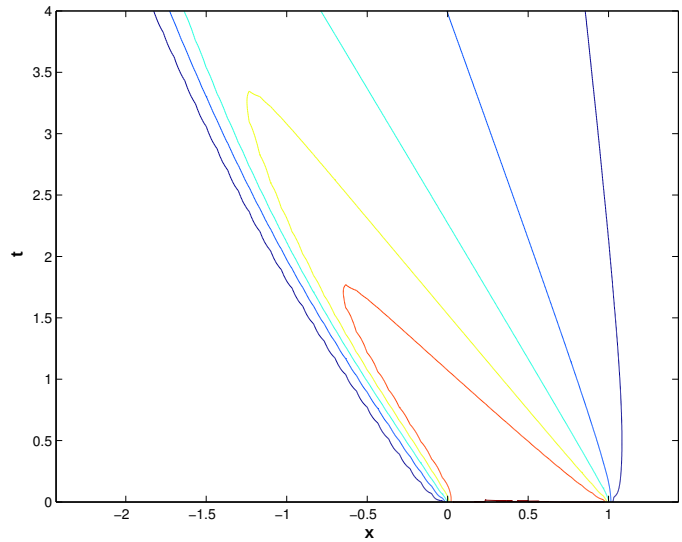
Example 8.3.38 (Lax-Friedrichs flux for traffic flow equation)

☞ same setting and conservative discretization as in Ex. 8.3.22

☞ Numerical flux function: Lax-Friedrichs flux (8.3.36)







Same observations as in Ex. 8.3.37: no spurious oscillations, qualitatively correct solution, but strong smearing of shock.

8.3.3.3 Upwind flux

Another idea for stable spatial discretization of stationary transport in Sect. 7.2.2.1 ("upwind quadrature"):

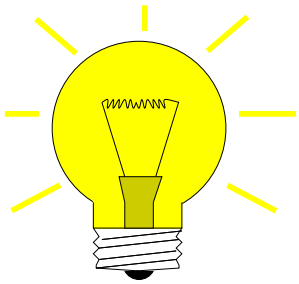
"upwinding" = obtain information from where transport brings it

☞ remedy for ambiguity of evaluation of discontinuous gradient in upwind quadrature

Owing to the discontinuity of u_N at $x_{k+1/2}$, ambiguity is also faced in the evaluation of the fluxes $f(u_N(x_{j+1/2}),t)$,$f(u$ see (8.3.7), which forced us to introduce numerical flux functions in (8.3.9). We may also seek to select the value of u_N from that side of $x_{k+1/2}$ where information comes from. In light of Rem. 8.2.15 we should examine the direction of the characteristic running through $(x_{k+1/2,t})$.

Def. 8.2.9, (8.3.35) ➢ The local slope of the characteristic curve (velocity of transport) at $(x,t) \in \widetilde{\Omega}$ is given by $f'(u(x,t))$.

local velocity of transport $f'(u_N(x_{k+1/2},t))$ is ambiguous too!



Idea: There is a "velocity of propagation" even at discontinuities of u!
Deduce it from Rankine-Hugoniot jump condition (8.2.23).

- local velocity of transport $= \begin{cases} f'(u) & \text{for unique state, } u = u_l = u_r \\ \frac{f(u_r)-f(u_l)}{u_r-u_l} & \text{at discontinuity.} \end{cases}$

(u_l, u_r $\hat{=}$ states to left and right of discontinuity)

- upwind numerical flux for scalar conservation law with flux function f:

$$\boxed{F_{\mathrm{uw}}(v,w) = \begin{cases} f(v) & \text{, if } \dot{s} \geq 0\,, \\ f(w) & \text{, if } \dot{s} < 0\,, \end{cases} \qquad \dot{s} := \begin{cases} \frac{f(w)-f(v)}{w-v} & \text{for } v \neq w\,, \\ f'(v) & \text{for } v = w\,. \end{cases}} \tag{8.3.39}$$

Now we investigate empirically the performance of the upwind numerical flux for our model non-linear scalar conservation laws.

Example 8.3.40 (Upwind flux for Burgers equation)

☞ same setting and conservative discretization as in Exp. 8.3.22

☞ Numerical flux function: upwind flux (8.3.39)

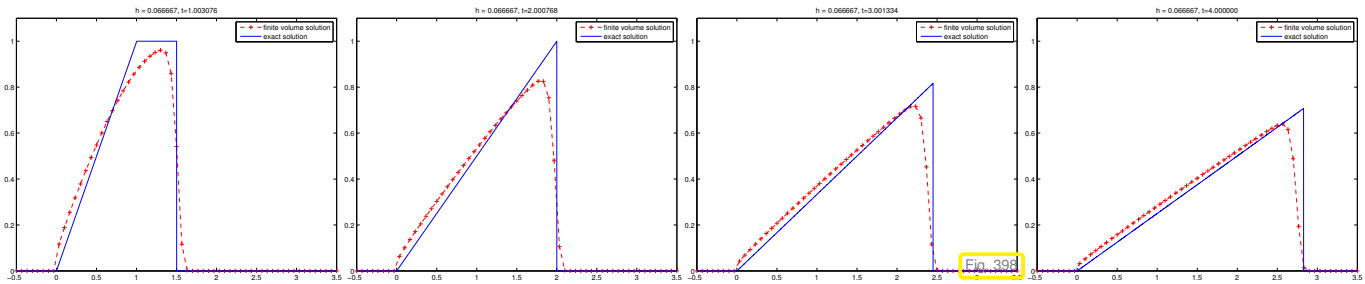
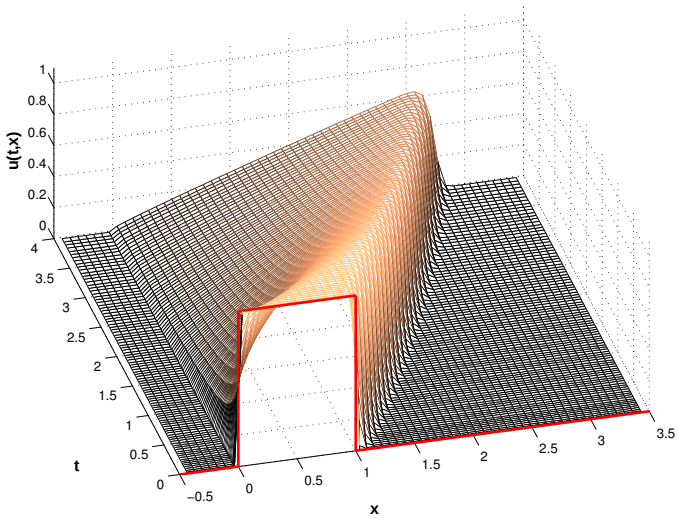
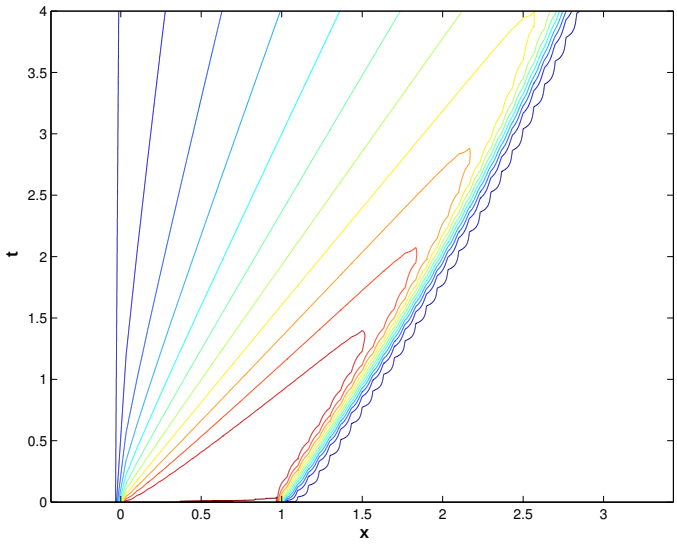


Fig. 398





Example 8.3.41 (Upwind flux for traffic flow simulation)

☞ Conservative finite volume discretization of Cauchy problem for traffic flow equation (8.1.41), flux function $f(u) = u(1-u)$

- ☞ Equidistant spatial mesh with meshwidth $h = 0.03$, adaptive explicit Runge-Kutta timestepping (MATLAB `ode45`)
- ☞ Numerical flux function: upwind flux (8.3.39)
- ☞ "Box shaped" initial data $u_0(x) = \begin{cases} 1 & \text{for } 0 \leq x \leq 1\,, \\ 0.5 & \text{elsewhere.} \end{cases}$

The solution will comprise a stationary shock and a rarefaction fan, which will merge eventually.

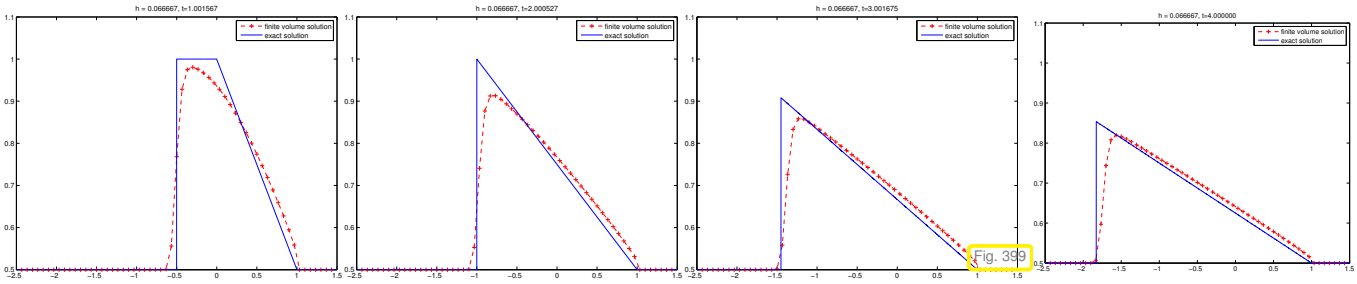
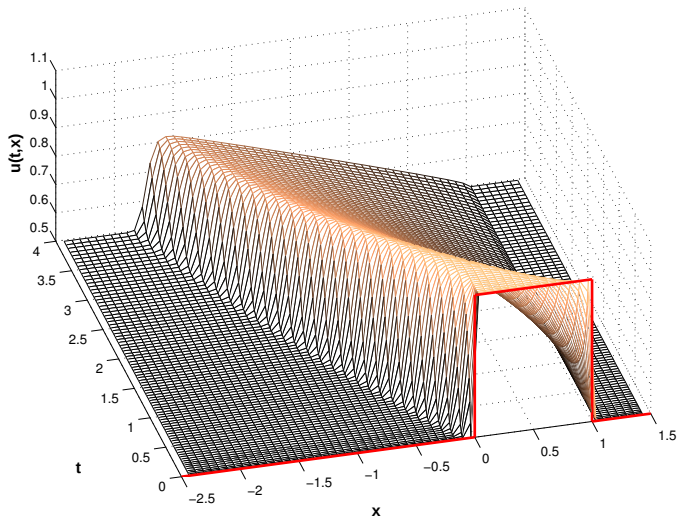
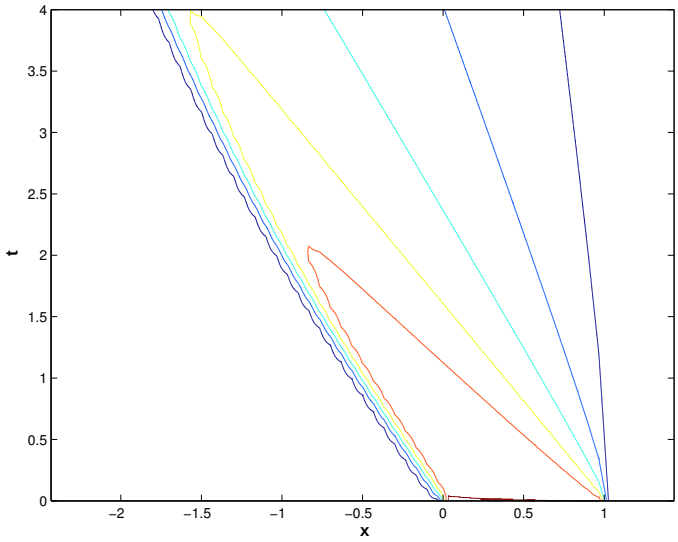


Fig. 389





We observe a satisfactory resolution of the shock and the rarefaction fan.

Example 8.3.42 (Upwind flux and transsonic rarefaction)

In this example we will witness a situation in which the use of the upwind numerical flux function produces a non-physical shock.

We consider the Cauchy problem (8.2.7) for Burgers equation (8.1.46), i.e., $f(u) = \frac{1}{2}u^2$ and initial data

$$u_0(x) = \begin{cases} -1 & \text{for} \quad x < 0 \text{ or } x > 1\,, \\ 1 & \text{for} \quad 0 < x < 1\,. \end{cases} \tag{8.3.43}$$

The analytic solution for this Cauchy problem is given in Ex. 8.2.39.

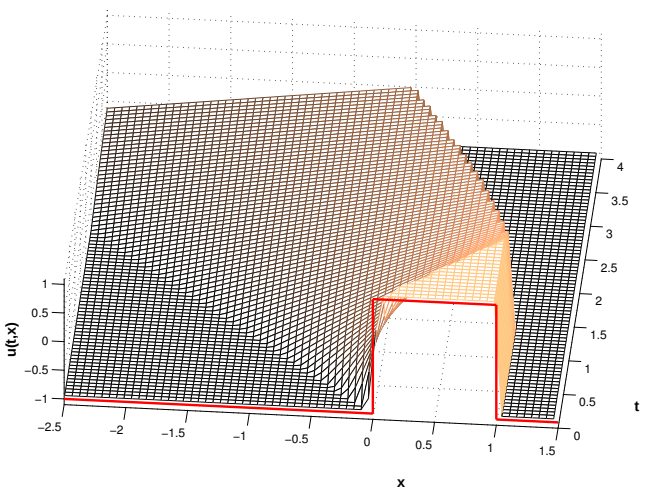


Fig. 400

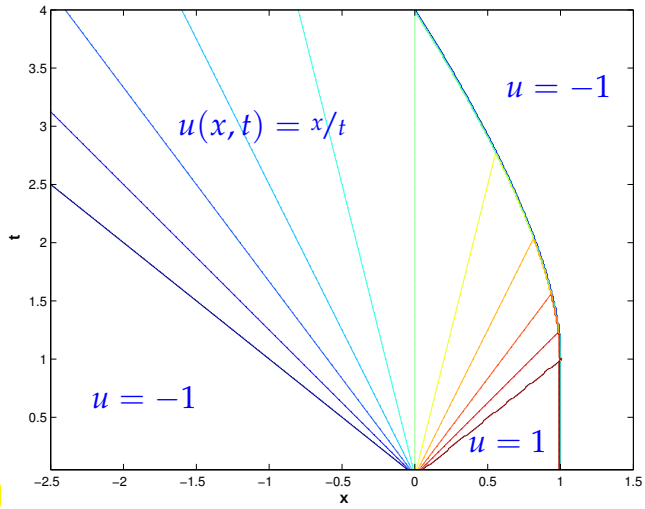


Fig. 401

There is a related Cauchy problem (8.2.7) for the traffic flow equation (8.1.41), i.e., $f(u) = u(1-u)$ and initial data

$$u_0(x) = \begin{cases} 0 & \text{for} \quad x < 0 \text{ or } x > 1\,, \\ 1 & \text{for} \quad 0 < x < 1\,. \end{cases} \tag{8.3.44}$$

Its analytic solution is plotted in Fig. 402 and given in Fig. 403.

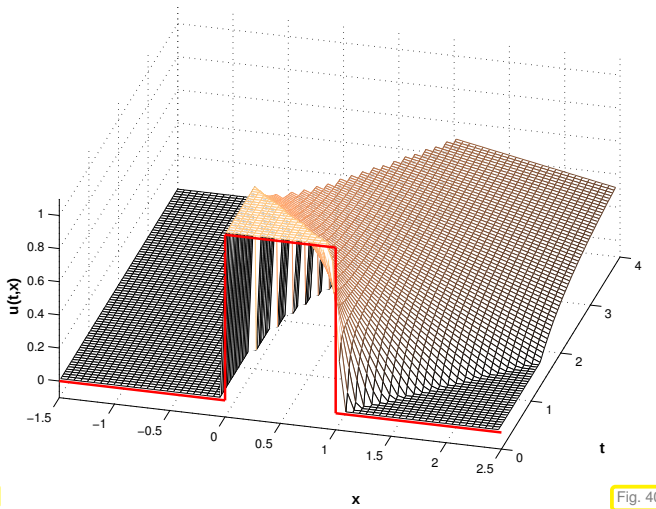


Fig. 402

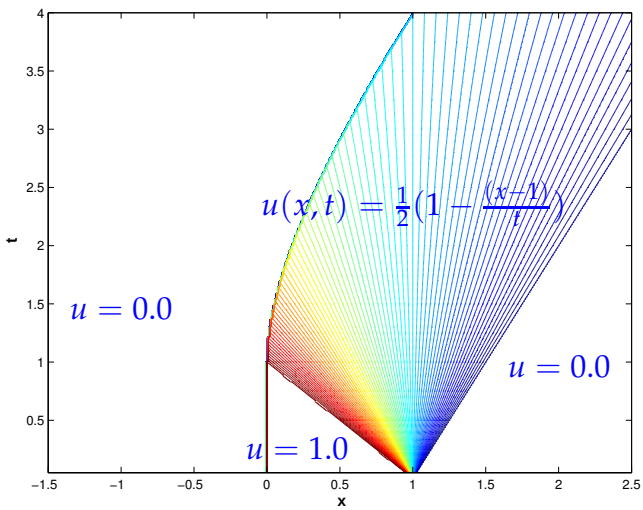


Fig. 403

The *entropy solution* ($\rightarrow$ Section 8.2.6) of these Cauchy problems features a transsonic rarefaction fan at $x = 1$: this is a rarefaction solution ($\rightarrow$ Lemma 8.2.35) whose "edges" move in opposite directions.

Burgers' equation, initial density (8.3.43): numerical solution with finite volume method with upwind flux (8.3.39).

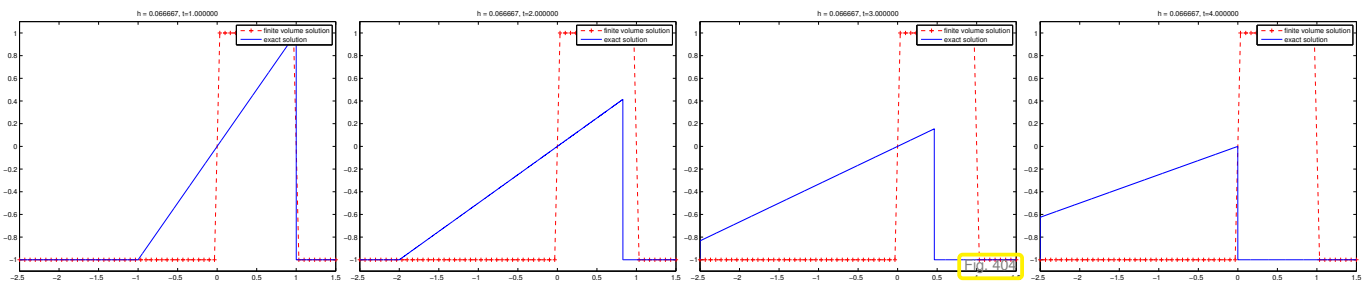
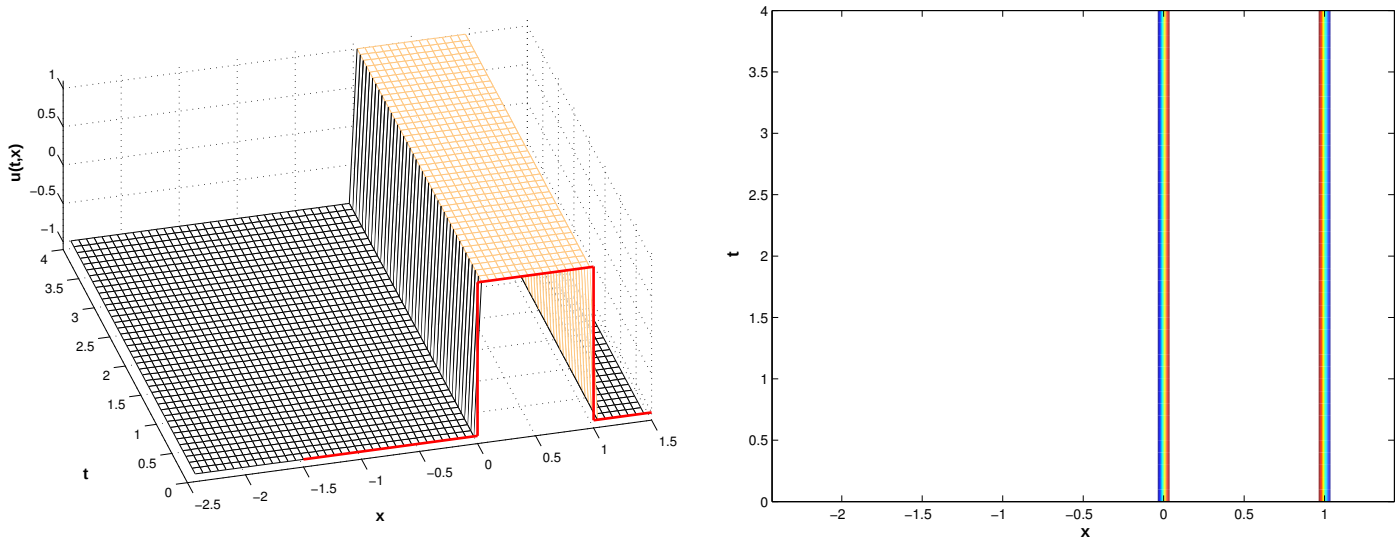
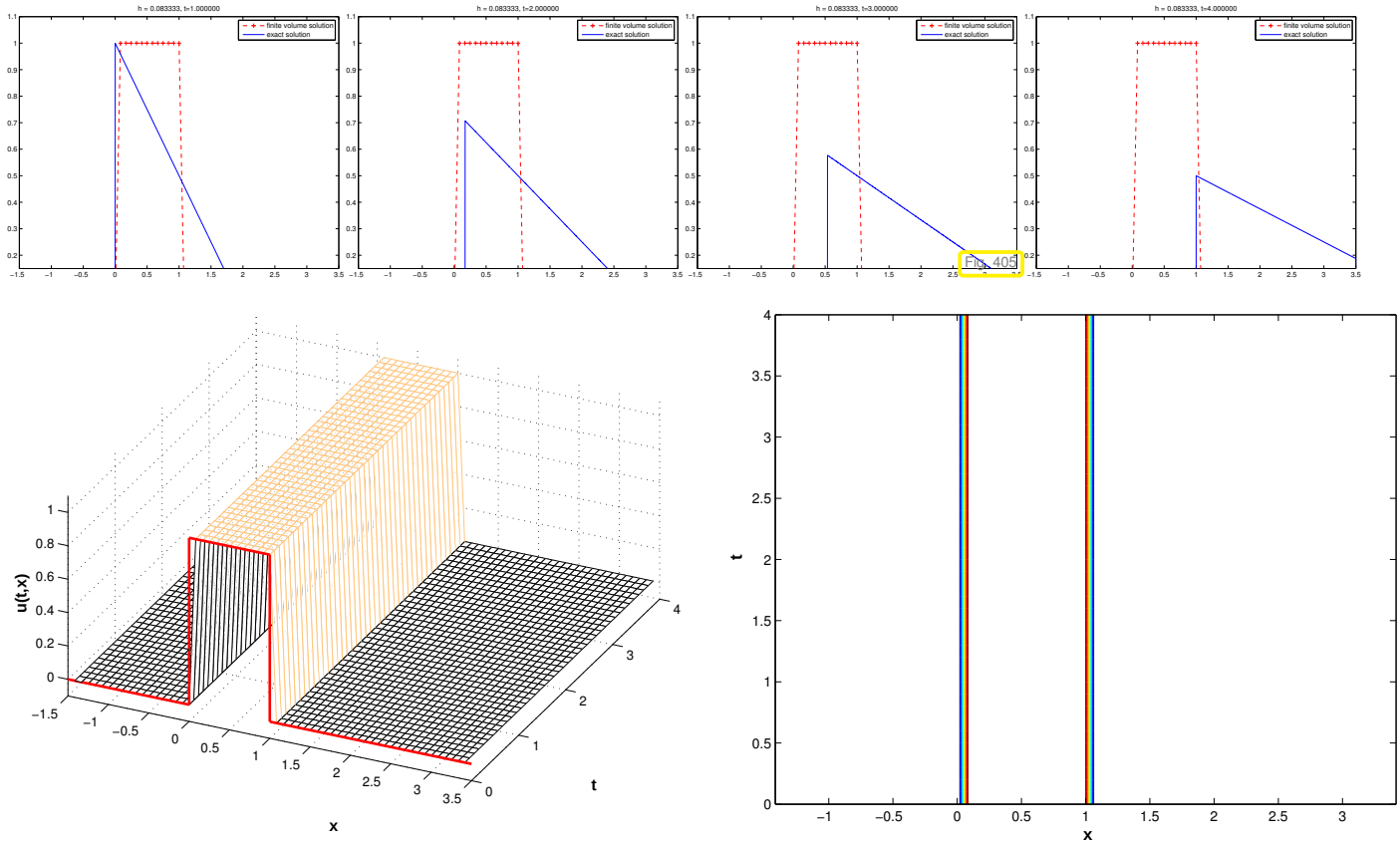


Fig. 404



Traffic flow equation, initial data (8.3.44): numerical solution with finite volume method with upwind flux (8.3.39).



Conservative finite volume discretization with upwind flux produces (stationary) *expansion shock* instead of transonic rarefaction!

Sect. 8.2.6: this is a weak solution, but it violates the entropy condition, "non-physical shock".

Example 8.3.45 (Upwind flux: Convergence to expansion shock)

In Ex. 8.3.42 we have seen that the use of the upwind flux can make a conservative finite volume discretization converge to non-physical expansion shocks. In this example simple computations will show

how this can happen. The setting is the following:

- Cauchy problem (8.2.7) for Burgers equation (8.1.46), i.e., $f(u) = \frac{1}{2}u^2$
- $u_0(x) = 1$ for $x > 0$, $u_0(x) = -1$ for $x < 0$
  - ➤ entropy solution = rarefaction wave (→ Lemma 8.2.35)
- FV in conservation form, upwind flux (8.3.39), on equidistant grid, $x_j = (j + \frac{1}{2})h$, meshwidth $h > 0$

➤ initial nodal values $\mu_j(0) = \begin{cases} -1 & \text{for } j < 0\,, \\ 1 & \text{for } j \geq 0\,. \end{cases}$

➤ Semi-discrete evolution equation:

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{2h} \cdot \begin{cases} \mu_{j+1}^2(t) - \mu_j^2(t) & \text{for } j \geq 0\,, \\ \mu_j^2(t) - \mu_{j-1}^2(t) & \text{for } j < 0\,. \end{cases}$$

➤ $\mu_j(t) = \mu_j(0)$ for all t ➤ for $h \to 0$, convergence to entropy violating expansion shock **!**

➤ conservative finite volume method may converge to non-physical weak solutions **!**

8.3.3.4 Godunov flux

Ex. 8.3.42 strikingly illustrated the failure of the a conservative finite volume discretization based on upwind flux to deal with transsonic rarefactions. In this section a different perspective on upwind fluxes will suggest a remedy.

(The following discussion is for *convex* flux functions only, that occur, for instance in Burgers equation (8.1.46). The reader is encouraged to figure out the modifications necessary if the flux function is concave, as in the traffic flow equation (8.1.41).)

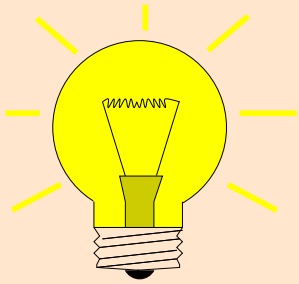
The upwind flux (8.3.39) is a numerical flux of the form

$$\boxed{F(v,w) = f(u^{\downarrow}(v,w))} \quad \text{with an intermediate state} \quad u^{\downarrow}(v,w) \in \mathbb{R}\,.$$

For the upwind flux the intermediate state is not really "intermediate", but coincides with one of the states v, w depending on the sign of the "local shock speed" $\dot{s} := \frac{f(w)-f(v)}{w-v}$.

(8.3.46) Local Riemann problems

We note that the intermediate state for the upwind numerical flux at the dual cell boundary $x_{j+1/2}$ agrees with the state produced for short times at $x_{j+1/2}$ by an *all-shock solution* of the conservation law with initial data $u_N(\cdot,t)$, with u_N the $\mathcal{M}$-piecewise constant function defined by the dual cell averages according to (8.3.5). This solution may feature non-physical (expansion) shocks, while rarefaction waves are missing. For this reason the simple upwind flux fails to capture rarefaction waves as we have witnessed in Ex. 8.3.42.



Idea: obtain suitable intermediate state as

$$u^{\downarrow}(v,w) = \psi(0)\,, \tag{8.3.47}$$

where $\overline{u}(x,t) = \psi(x/t)$ solves the Riemann problem ($\rightarrow$ Def. 8.2.5)

$$\frac{\partial \overline{u}}{\partial t} + \frac{\partial}{\partial x} f(\overline{u}) = 0 \quad , \quad \overline{u}(x,0) = \begin{cases} v & \text{, for } x < 0\,, \\ w & \text{, for } x \geq 0\,. \end{cases} \tag{8.3.48}$$

Remember Lemma 8.2.29, Lemma 8.2.35, and the reasons why we can count on the entropy solution of the Riemann problem to be a similarity solution of the form $\overline{u}(x,t) = \psi(x/t)$, see page 585.

We focus on $f : \mathbb{R} \mapsto \mathbb{R}$ strictly convex & smooth (e.g. Burgers equations (8.1.46))

➤ Riemann problem (8.3.48) ($\rightarrow$ Def. 8.2.5) has the *entropy solution* ($\rightarrow$ Sect. 8.2.6):

❶ If $v > w$ ➤ discontinuous solution, shock ($\rightarrow$ Lemma 8.2.29)

$$\overline{u}(t,x) = \begin{cases} v & \text{if } x < \dot{s}t\,, \\ w & \text{if } x > \dot{s}t\,, \end{cases} \qquad \dot{s} = \frac{f(v) - f(w)}{v - w}\,. \tag{8.3.49}$$

❷ If $v \leq w$ ➤ continuous solution, rarefaction wave ($\rightarrow$ Lemma 8.2.35)

$$\overline{u}(t,x) = \begin{cases} v & \text{if } x < f'(v)t\,, \\ g(x/t) & \text{if } f'(v) \leq x/t \leq f'(w)\,, \\ w & \text{if } x > f'(w)t\,, \end{cases} \qquad g := (f')^{-1}\,. \tag{8.3.50}$$

➢ Also from these formulas we see that all weak solutions of a Riemann problem are of the form $u(x,t) = \psi(x/t)$ (similarity solution) with a suitable function ψ, which is

- ✦ piecewise constant with a jump at $\dot{s} := \frac{f(w)-f(v)}{w-v}$ for a shock solution (8.3.49),
- ✦ the continuous function (in the case of strictly convex flux function f)

$$\psi(\xi) := \begin{cases} v & \text{, if } \xi < f'(v)\,, \\ (f')^{-1}(\xi) & \text{, if } f'(v) < \xi < f'(w)\,, \\ w & \text{, if } \xi > f'(v)\,, \end{cases}$$

provided that $w > v$ = situation of a rarefaction solution (8.3.50), see Lemma 8.2.35.

A graphical illustration of the various local Riemann solutions that can be found at dual cell boundaries is given next:

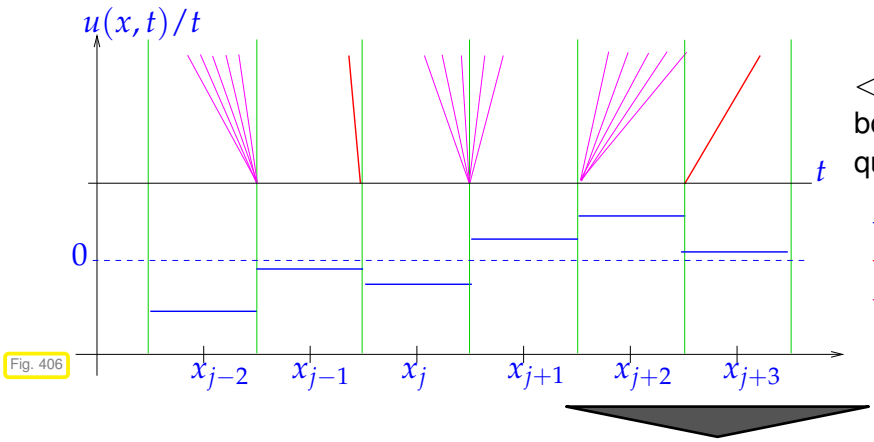


Fig. 406

◁ local Riemann problems at dual cell boundaries (for Burgers flux $f(u) = \frac{1}{2}u^2$, qualitative drawing)

— — ≙ piecewise constant function $u(0,t)$
— — ≙ shock in (t,x)-plane
— — ≙ rarefaction wave in (t,x)-plane

for *convex* flux function f

$$u^{\downarrow}(v,w) = \begin{cases} w & \text{, if } \begin{array}{l} v > w \wedge \dot{s} < 0 \text{ (shock ❶)}, \\ v < w \wedge f'(w) < 0 \text{ (rarefaction ❷)}, \end{array} \\ v & \text{, if } \begin{array}{l} v > w \wedge \dot{s} > 0 \text{ (shock ❸)}, \\ v < w \wedge f'(v) > 0 \text{ (rarefaction ❹)}, \end{array} \\ (f')^{-1}(0) & \text{, if } v < w \wedge f'(v) \leq 0 \leq f'(w) \text{ (rarefaction ❺)}. \end{cases} \tag{8.3.51}$$

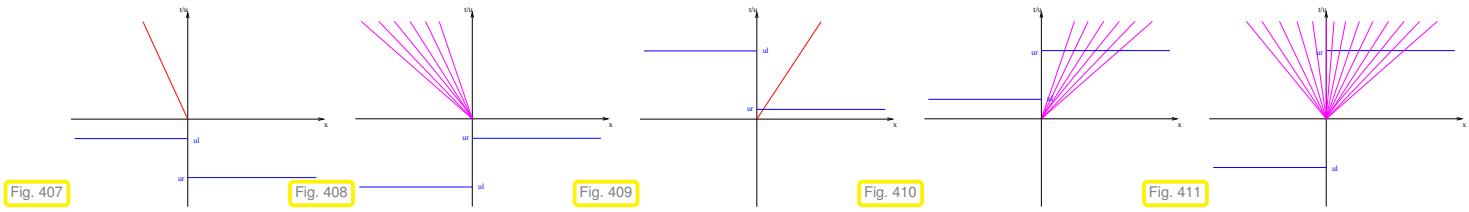


Fig. 407 Fig. 408 Fig. 409 Fig. 410 Fig. 411

❶: subsonic shock ❷: subsonic rarefaction ❸: supersonic shock ❹: supersonic rarefaction ❺: transonic rarefaction

(8.3.52) Formulas for Godunov numerical flux function

A detailed analysis of (8.3.51) yields fairly explicit formulas:

$$v > w \quad \text{(shock case):} \quad f(u^{\downarrow}(v,w)) = \begin{cases} f(v) & \text{, if } \frac{f(w)-f(v)}{w-v} > 0 \quad \Leftrightarrow \quad f(w) < f(v)\,, \\ f(w) & \text{, if } \frac{f(w)-f(v)}{w-v} \leq 0 \quad \Leftrightarrow \quad f(w) \geq f(v)\,. \end{cases}$$

$$\blacktriangleright \quad f(u^{\downarrow}(v,w)) = \max\{f(v), f(w)\}\,.$$

For a convex flux function f:

$$v < w \quad \Rightarrow \quad f'(v) \leq \frac{f(w)-f(v)}{w-v} \leq f'(w)\,.$$

$\blacktriangleright$ For $v < w$ (rarefaction case)

$$f(u^{\downarrow}(v,w)) = \begin{cases} f(v) & \text{, if } f'(v) > 0\,, \\ f(z) & \text{, if } f'(v) < 0 < f'(w)\,, \\ f(w) & \text{, if } f'(w) < 0\,, \end{cases}$$

where $f'(z) = 0 \Leftrightarrow f$ has a global minimum in z.

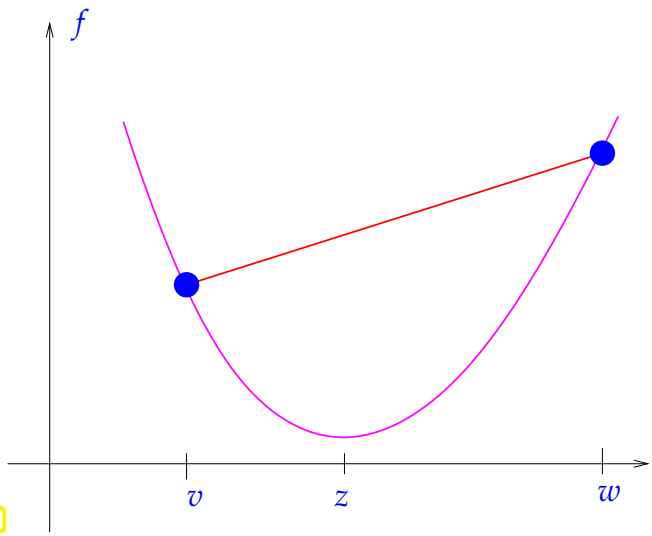


Fig. 412

2-point numerical flux function according to (8.3.47) and (8.3.48): Godunov numerical flux

Using general Riemann solution (8.2.38) we get for **any** flux function:

Godunov numerical flux function

$$F_{\mathrm{GD}}(v,w) = \begin{cases} \min\limits_{v\le u\le w} f(u) & \text{, if} \quad v < w\,, \\ \max\limits_{w\le u\le v} f(u) & \text{, if} \quad w \le v\,. \end{cases} \tag{8.3.53}$$

Obviously the Godunov numerical flux is consistent according to Def. 8.3.18.

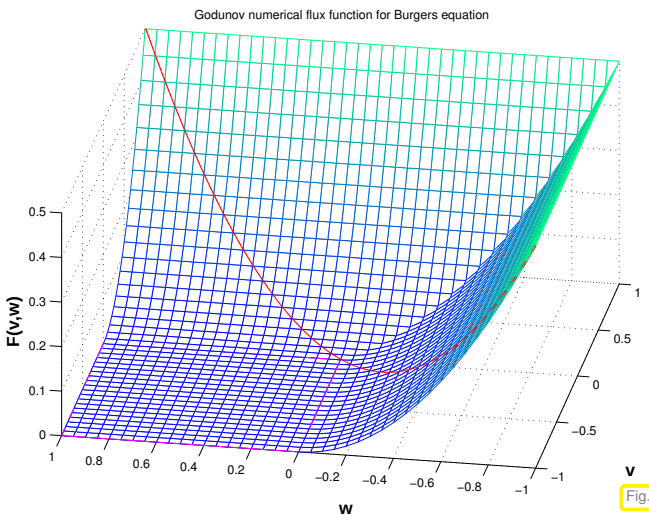


Fig. 413

for Burgers' equation (8.1.46)

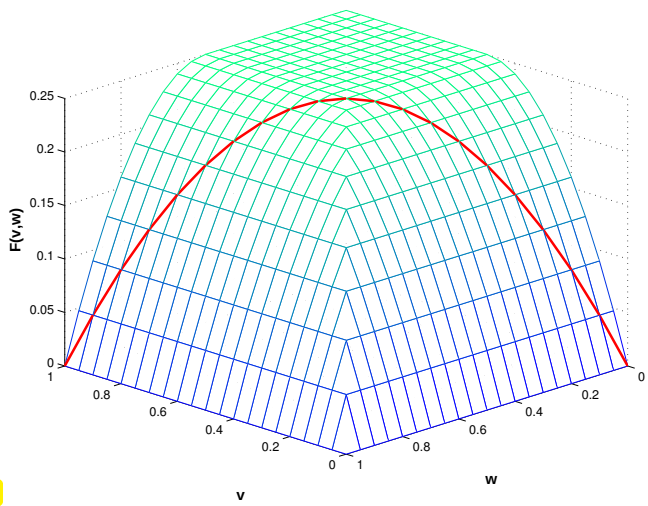


Fig. 414

for traffic flow equation (8.1.41)

Remark 8.3.54 (Upwind flux and expansion shocks)

For traffic flow equation (8.1.41) $(f(u) = u(1-u))$

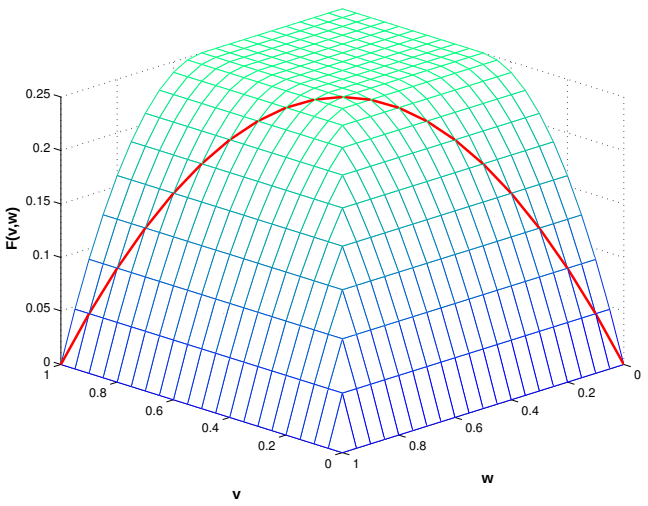


Fig. 415

Godunov flux $F_{\mathrm{DG}}(v,w)$

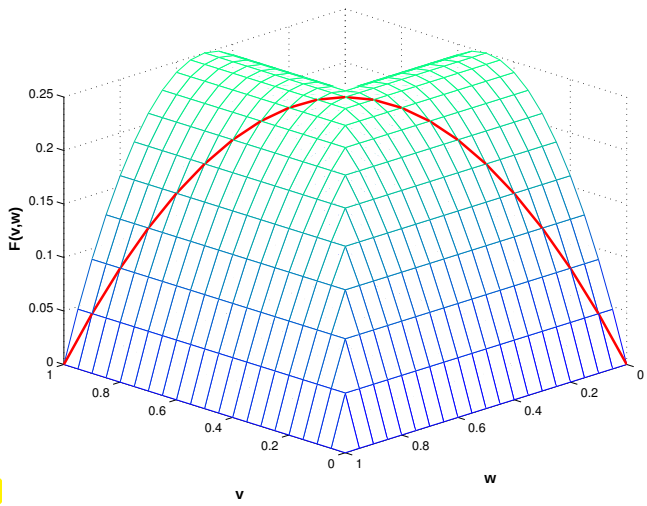


Fig. 416

upwind flux $F_{\mathrm{UW}}(v,w)$

$F_{\mathrm{uw}}(v,w) = F_{\mathrm{GD}}(v,w)$, **except** for the case of *transsonic rarefaction*!

(transsonic rarefaction = rarefaction fan with edges moving in opposite direction, see Ex. 8.3.42)

What does the upwind flux $F_{\mathrm{uw}}(v,w)$ from (8.3.39) yield in the case of transsonic rarefaction?

If f convex, $v < w$, $f'(v) < 0 < f'(w)$,

$$\blacktriangleright \quad F_{\mathrm{uw}}(v,w) = f(\psi(0)) \ ,$$

where $u(x,t) = \psi(x/t)$ is a non-physical *entropy-condition violating* ($\rightarrow$ Def. 8.2.36) expansion shock weak solution of (8.3.48).

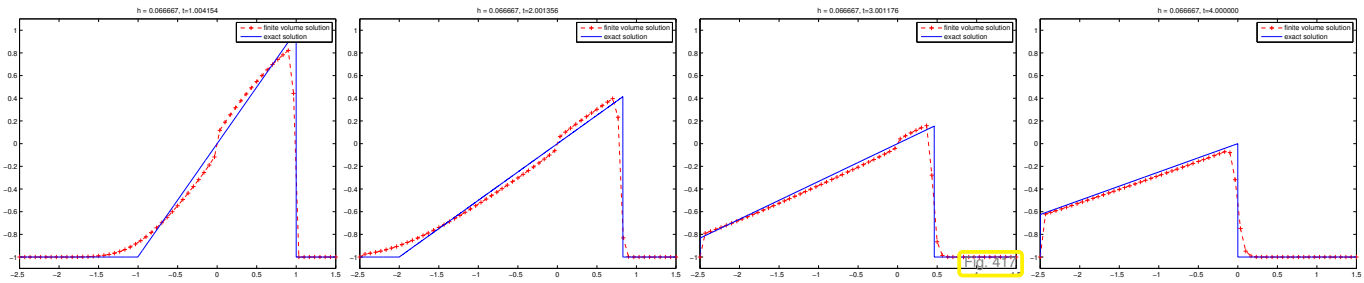
Upwind flux treats transsonic rarefaction as expansion shock!

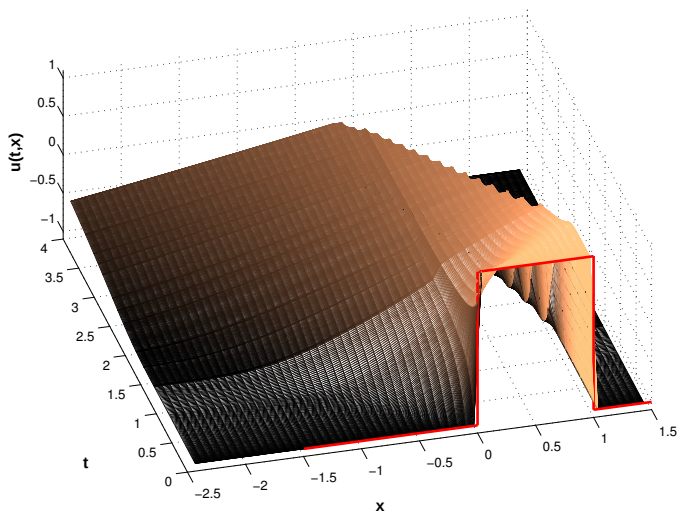
➢ Explanation for observation made in Ex. 8.3.42.

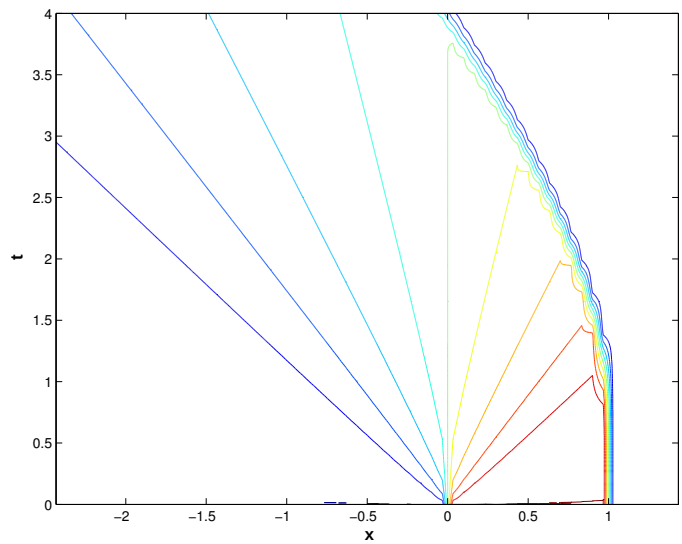
Example 8.3.55 (Godunov flux for Burgers equation)

☞ same setting and conservative discretization as in Ex. 8.3.42

☞ Numerical flux function: Godunov numerical flux (8.3.53)







Observation: Transonic rarefaction captured by discretization, but small remnants of an expansion shock still observed.

Example 8.3.56 (Godunov flux for traffic flow equation)

☞ same setting and conservative discretization as in Ex. 8.3.42

☞ Numerical flux function: Godunov numerical flux (8.3.53)

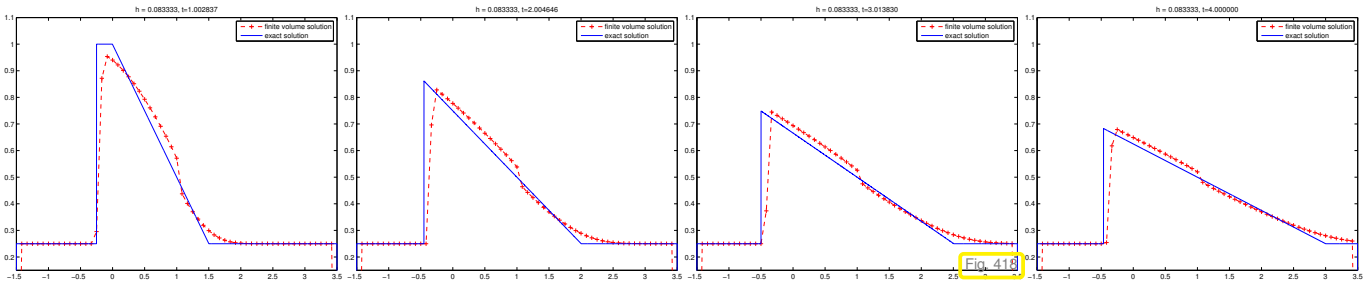
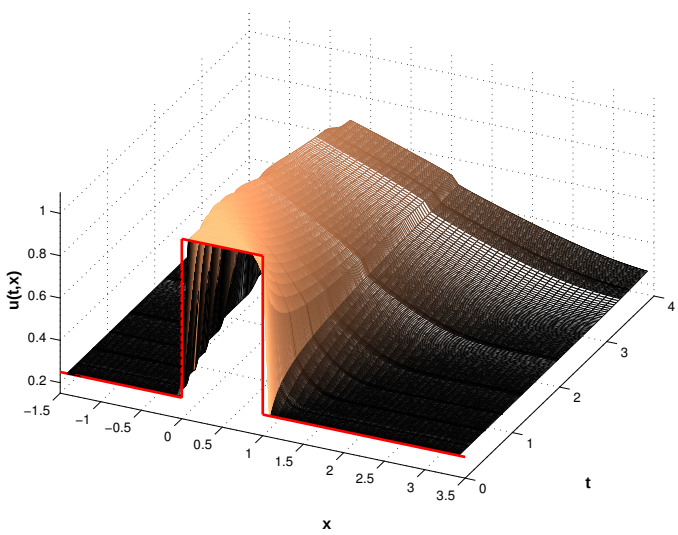
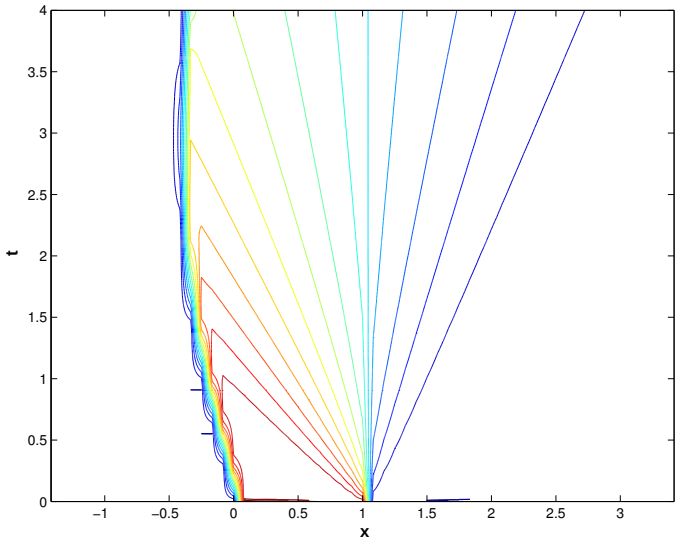


Fig. 418





Observation: Transonic rarefaction captured by discretization, but small remnants of an expansion shock still observed.

8.3.4 Monotone schemes

Observations made for some piecewise constant solutions $u_N(t)$ of semi-discrete evolutions arising from spatial finite volume discretization in conservation form (8.3.10):

Ex. 8.3.37 (Lax-Friedrichs numerical flux (8.3.36))
Ex. 8.3.55 (Godunov numerical flux (8.3.53)) :

✦ $\min_{x\in\mathbb{R}} u_0(x) \leq u_N(x,t) \leq \max_{x\in\mathbb{R}} u_0(x)$
✦ *no new* local extrema in numerical solution

In these respects the conservative finite volume discretizations based on either the Lax-Friedrichs numerical flux (→ Section 8.3.3.2) or the Godunov numerical flux (→ Section 8.3.3.4) inherit crucial structural properties of the exact solution, see Sect. 8.2.7, in particular, Thm. 8.2.41 and the final remark: they display structure preservation, *cf.* (5.7).

Is this coincidence for the special settings examined in Ex. 8.3.37 and Ex. 8.3.55**?**

(8.3.57) Discrete comparison principle

Focus: semi-discrete evolution (8.3.10) resulting from finite volume discretization in conservation form with 2-point numerical flux on an equidistant infinite mesh

(8.3.9) ► $$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big)\,, \quad j\in\mathbb{Z}\,, \tag{8.3.10}$$

for Cauchy problem

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R}\times]0,T[\quad , \quad u(x,0) = u_0(x) \; , \quad x \in \mathbb{R} \; , \tag{8.2.7}$$

induced by Lax-Friedrichs numerical flux (8.3.36)

$$F_{\mathrm{LF}}(v,w) = \tfrac{1}{2}(f(v)+f(w)) - \tfrac{1}{2} \max_{\min\{v,w\}\le u\le \max\{v,w\}} |f'(u)|(w-v) \; . \tag{8.3.36}$$

$$\frac{d\mu_j}{dt} = -\frac{1}{2h}\Big(f(\mu_{j+1}) - f(\mu_{j-1}) - \max_{u\in[\mu_j,\mu_{j+1}]} |f'(u)|(\mu_{j+1}-\mu_j) + \max_{u\in[\mu_{j-1},\mu_j]} |f'(u)|(\mu_j-\mu_{j-1})\Big) \; . \tag{8.3.58}$$

Goal: show that $u_N(t)$ linked to $\vec{\mu}(t)$ from (8.3.58) through piecewise constant reconstruction (8.3.5) satisfies

$$\min_{x\in\mathbb{R}} u_N(x,0) \le u_N(x,t) \le \max_{x\in\mathbb{R}} u_N(x,0) \quad \forall x \in \mathbb{R} \; , \quad \forall t \in [0,T] \; . \tag{8.3.59}$$

Recall from Sect. 8.2.7: estimate (8.3.59) for the exact solution $u(x,t)$ of (8.2.7) is a consequence of the comparison principle of Thm. 8.2.41 and the fact that constant initial data are preserved during the evolution. The latter property is straightforward for conservative finite volume spatial semi-discretization, see (8.3.15).

➢ Goal: Establish comparison principle for finite volume semi-discrete solutions based on Lax-Friedrichs numerical flux:

$$\left\{ \begin{array}{l} \vec{\mu}(t), \vec{\eta}(t) \text{ solve (8.3.58)} \; , \\ \eta_j(0) \le \mu_j(0) \quad \forall j \in \mathbb{Z} \end{array} \right\} \quad \Rightarrow \quad \eta_j(t) \le \mu_j(t) \quad \forall j \in \mathbb{Z} \; , \quad \forall 0 \le t \le T \; .$$

Assumption: $\vec{\mu} = \vec{\mu}(t)$ and $\vec{\eta} = \vec{\eta}(t)$ solve (8.3.58) and satisfy for some $t \in [0,T]$

$$\eta_k(t) \le \mu_k(t) \quad \forall k \in \mathbb{Z} \quad , \quad \xi := \eta_j(t) = \mu_j(t) \quad \text{for some } j \in \mathbb{Z} \; .$$

Can η_j raise above μ_j**?**

$$\frac{d}{dt}(\mu_j - \eta_j) = -\frac{1}{h}\Big(F_{\mathrm{LF}}(\xi,\mu_{j+1}) - F_{\mathrm{LF}}(\xi,\eta_{j+1}) + F_{\mathrm{LF}}(\eta_{j-1},\xi) - F_{\mathrm{LF}}(\mu_{j-1},\xi)\Big) \; .$$

To show: $\frac{d}{dt}(\mu_j - \eta_j) \ge 0$ ➢ $\mu_j(t)$ will stay above $\eta_j(t)$.

This can be concluded, if

$$F_{\mathrm{LF}}(\xi,\mu_{j+1}) - F_{\mathrm{LF}}(\xi,\eta_{j+1}) \le 0 \quad \text{and} \quad F_{\mathrm{LF}}(\eta_{j-1},\xi) - F_{\mathrm{LF}}(\mu_{j-1},\xi) \le 0 \; . \tag{8.3.60}$$

The only piece of information we are allowed to use is

$$\mu_{j+1} \ge \eta_{j+1} \quad \text{and} \quad \mu_{j-1} \ge \eta_{j-1} \; .$$

This would imply (8.3.60), if F_{LF} was increasing in the first argument and decreasing in the second argument. Such a trait of a two-point numerical flux is considered in the next definition.

Definition 8.3.61. Monotone numerical flux function

A 2-point numerical flux function $F = F(v,w)$ is called monotone, if

$$F \text{ is an increasing function of its first argument } v \quad (\forall w)$$

and

$$F \text{ is a decreasing function of its second argument } w \ (\forall v).$$

Corollary 8.3.62. Simple criterion for monotone flux function

A continuously differentiable 2-point numerical flux function $F = F(v,w)$ is montone, if and only if

$$\frac{\partial F}{\partial v}(v,w) \geq 0 \quad \text{and} \quad \frac{\partial F}{\partial w}(v,w) \leq 0 \quad \forall (v,w) \ . \tag{8.3.63}$$

The important 2-point numerical fluxes that we have studied in Section 8.3.3.2 and Section 8.3.3.4 enjoy the monotonicity property.

Lemma 8.3.64. Monotonicity of Lax-Friedrichs/Rusanov numerical flux and Godunov flux

For any continuously differentiable flux function f the associated Lax-Friedrichs/Rusanov flux (8.3.36) *and Godunov flux* (8.3.53) *are monotone.*

Proof.

❶ (Local) Lax-Friedrichs/Rusanov numerical flux:

$$F_{\mathrm{LF}}(v,w) = \tfrac{1}{2}(f(v)+f(w)) - \tfrac{1}{2}(w-v) \cdot \max_{\min\{v,w\} \leq u \leq \max\{v,w\}} |f'(u)| \ .$$

Application of the criterion (8.3.63) is straightforward:

$$\frac{\partial F_{\mathrm{LF}}}{\partial v}(v,w) = \tfrac{1}{2} f'(v) + \tfrac{1}{2} \max_{\min\{v,w\} \leq u \leq \max\{v,w\}} |f'(u)| \geq 0 \ ,$$
$$\frac{\partial F_{\mathrm{LF}}}{\partial w}(v,w) = \tfrac{1}{2} f'(w) - \tfrac{1}{2} \max_{\min\{v,w\} \leq u \leq \max\{v,w\}} |f'(u)| \leq 0 \ .$$

For the genuine Lax-Friedrichs numerical flux (8.3.36) the proof of monotonicity entails treating numerous cases separately, because the factor in front of the diffusive flux will also depend on v and w.

❷ Godunov numerical flux

$$F_{\mathrm{GD}}(v,w) = \begin{cases} \min\limits_{v \leq u \leq w} f(u) & \text{, if } \quad v < w \ , \\ \max\limits_{w \leq u \leq v} f(u) & \text{, if } \quad w \leq v \ . \end{cases} \tag{8.3.53}$$

$v < w$: If v increases, then the range of values over which the minimum is taken will shrink, which makes $F_{\mathrm{GD}}(v,w)$ increase.
If w is raised, then the minimum is taken over a larger interval, which causes $F_{\mathrm{GD}}(v,w)$ to become smaller.

$v \geq w$: If v increases, then the range of values over which the maximum is taken will grow, which makes $F_{\mathrm{GD}}(v,w)$ increase.
If w is raised, then the maximum is taken over a smaller interval, which causes $F_{\mathrm{GD}}(v,w)$ to decrease. □

Lemma 8.3.65. Comparison principle for monotone semi-discrete conservative evolutions

Let the 2-point numerical flux function $F = F(v,w)$ *be monotone (→ Def. 8.3.61) and* $\vec{\mu} = \vec{\mu}(t)$, $\vec{\eta} = \vec{\eta}(t)$ *solve* (8.3.10). *Then*

$$\eta_k(0) \leq \mu_k(0) \quad \forall k \in \mathbb{Z} \quad \Rightarrow \quad \eta_k(t) \leq \mu_k(t) \quad \forall k \in \mathbb{Z}\,, \quad \forall\, 0 \leq t \leq T\,.$$

The assertion of Lemma 8.3.65 means that for monotone numerical flux, the semi-discrete evolution satisfies the comparison principle of Thm. 8.2.41.

Proof (of Lemma 8.3.65, following the above considerations for the Lax-Friedrichs flux).

The two sequences of nodal values satisfy (8.3.10)

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big)\,, \quad j \in \mathbb{Z}\,, \tag{8.3.66}$$

$$\frac{d\eta_j}{dt}(t) = -\frac{1}{h}\big(F(\eta_j(t),\eta_{j+1}(t)) - F(\eta_{j-1}(t),\eta_j(t))\big)\,, \quad j \in \mathbb{Z}\,. \tag{8.3.67}$$

Let t_0 be the *earliest* time, at which $\vec{\eta}$ "catches up" with $\vec{\mu}$ in at least one node x_j, $j \in \mathbb{Z}$, of the mesh, that is

$$\eta_k(t_0) \leq \mu_k(t_0) \quad \forall k \in \mathbb{Z} \quad , \quad \xi := \eta_j(t_0) = \mu_j(t_0)\,.$$

By subtracting (8.3.66) and (8.3.67) we get

$$\frac{d}{dt}(\mu_j - \eta_j)(t_0) = -\frac{1}{h}\Big(F(\xi,\mu_{j+1}(t_0)) - F(\xi,\eta_{j+1}(t_0)) + F(\eta_{j-1}(t_0),\xi) - F(\mu_{j-1}(t_0),\xi)\Big) \geq 0\,,$$

because for a *monotone* numerical flux function (→ Def. 8.3.61)

$$\begin{aligned} \eta_{j-1}(t_0) \leq \mu_{j-1}(t_0) &\quad \overset{\text{increasing in first argument}}{\Rightarrow} \quad F(\eta_{j-1}(t_0),\xi) - F(\mu_{j-1}(t_0),\xi) \leq 0\,, \\ \eta_{j+1}(t_0) \leq \mu_{j+1}(t_0) &\quad \overset{\text{decreasing in second argument}}{\Rightarrow} \quad F(\xi,\mu_{j+1}(t_0)) - F(\xi,\eta_{j+1}(t_0)) \leq 0\,. \end{aligned}$$

This means that "η_j cannot overtake μ_j": no value η_j can ever raise above μ_j. □

(8.3.68) No creation of discrete local extrema

Now we want to study the "decrease of the number of local extrema" during a semi-discrete evolution, another *structural property* of exact solutions of conservations laws, see Sect. 8.2.7.

Intuitive terminology: $\vec{\mu}$ has a local maximum $u_m \in \mathbb{R}$, if

$$\exists j \in \mathbb{Z}\colon \quad \mu_j = u_m \quad \text{and} \quad \exists k_l < j < k_r \in \mathbb{N}\colon \quad \max_{k_l < l < k_r} \mu_l = u_m \quad \text{and} \quad \mu_{k_l} < u_m\,,\, \mu_{k_r} < u_m\,.$$

In analogous fashion, we define a local minimum. If $\vec{\mu}$ is constant for large indices, these values are also regarded as local extrema.

Counting local extrema of $\vec{\mu}$ and the associated piecewise constant reconstruction.

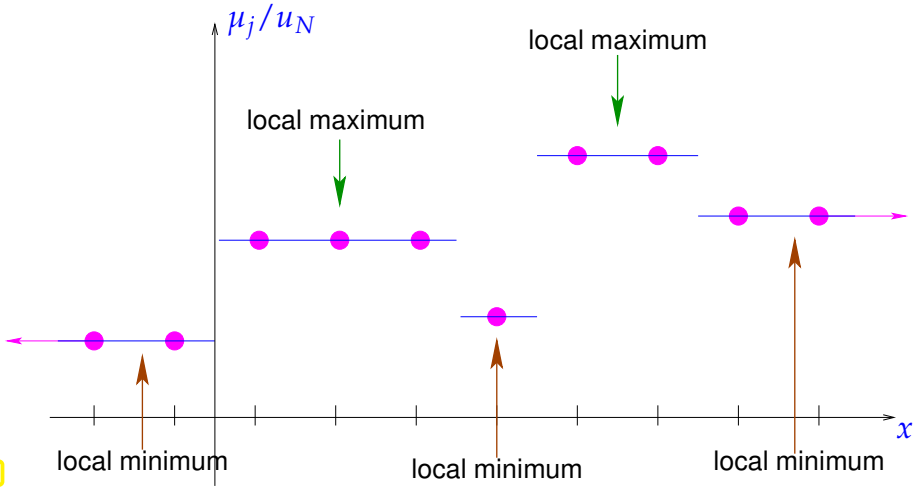


Fig. 419

Lemma 8.3.69. Non-oscillatory monotone semi-discrete evolutions

If $\vec{\mu} = \vec{\mu}(t)$ solves (8.3.10) *with a* monotone *numerical flux function $F = F(v,w)$ and $\vec{\mu}(0)$ has finitely many local extrema, then the number of local extrema of $\vec{\mu}(t)$ cannot be larger than that of $\vec{\mu}(0)$.*

Proof. $i \hat{=}$ index of local maximum of $\vec{\mu}(t)$, t fixed

$$\begin{array}{l} \mu_{i-1}(t) \leq \mu_i(t) \\ \mu_{i+1}(t) \leq \mu_i(t) \end{array} \overset{\text{monotone flux}}{\Longrightarrow} F(\mu_i,\mu_{i+1}) \geq F(\mu_i,\mu_i) \geq F(\mu_{i-1},\mu_i) \,,$$

$$\Rightarrow \quad \frac{d}{dt}\mu_i(t) = -\frac{1}{h}\big(F(\mu_i,\mu_{i+1}) - F(\mu_{i-1},\mu_i)\big) \leq 0 \,.$$

➢ maxima of $\vec{\mu}$ subside, (minima of $\vec{\mu}$ rise !)

Idea of proof:

No new (local) extrema can arise !

Adjacent values cannot "overtake":
local maximum: cannot move up
local minmum: cannot move down

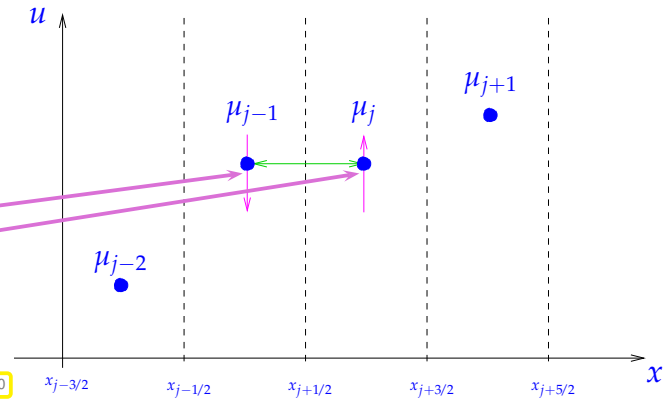


Fig. 420

8.4 Timestepping

In the spirit of the method of lines approach, we next we pursue the temporal discretization of the ordinary differential equation (ODE)

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_{j-m_l+1}(t),\dots,\mu_{j+m_r}(t)) - F(\mu_{j-m_l}(t),\dots,\mu_{j+m_r-1}(t))\big) \,, \quad j \in \mathbb{Z} \,, \tag{8.3.9}$$

which arises from the conservative finite volume spatial semi-discretization of the Cauchy problem for a generic 1D scalar conservation law (without sources)

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R}\times]0,T[\,, \quad u(x,0) = u_0(x) \text{in } \mathbb{R}\,. \tag{8.2.7}$$

Note that (8.3.9) is an ODE on the infinite-dimensional state space $\mathbb{R}^{\mathbb{Z}}$, but formally we can treat it like a regular ODE in $\mathbb{R}^N$. In particular, single step timestepping methods can be applied to (8.3.9).

(8.4.1) Runge-Kutta single step timestepping

Our focus:	*Explicit* Runge-Kutta timestepping methods (→ Def. 6.1.40)

Recall [33, Def. 11.4.9]: for explicit s-stage Runge-Kutta single step methods the coefficients a_{ij} vanish for $j \geq i$, $1 \leq i,j \leq s$ ➢ The increments $\mathbf{k}_i$ can be computed in turns (without solving a non-linear system of equations): For the abstract autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ an explicit s-stage Runge-Kutta single step method reads (for uniform timestep size $\tau > 0$)

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}^{(k)})\,,\\
\mathbf{k}_2 &= \mathbf{f}(\mathbf{y}^{(k)} + \tau a_{21}\mathbf{k}_1)\,,\\
&\vdots\\
\mathbf{k}_s &= \mathbf{f}(\mathbf{y}^{(k)} + \tau a_{s1}\mathbf{k}_1 + \cdots + \tau a_{s,s-1}\mathbf{k}_{s-1})\,,
\end{aligned} \quad , \quad \mathbf{y}^{(k+1)} := \mathbf{y}^{(k)} + \tau \sum_{l=1}^{s} b_l \mathbf{k}_l\,. \tag{8.4.2}$$

Here, $a_{ij} \in \mathbb{R}$ and $b_l \in \mathbb{R}$ are the coefficients from the Butcher scheme (6.1.42) describing the Runge-Kutta method. The vectors $\mathbf{k}_i$, $i = 1,\dots,s$, are called the increments. For explicit RK-methods the coefficient matrix $\mathfrak{A}$ is strictly lower triangular.

We consider an initial value problem for an abstract semi-discrete evolution in $\mathbb{R}^{\mathbb{Z}}$:

$$\frac{d\vec{\mu}}{dt}(t) = \mathcal{L}_h(\vec{\mu}(t))\,, \quad 0 \leq t \leq T \quad , \quad \vec{\mu}(0) = \vec{\mu}_0 \in \mathbb{R}^{\mathbb{Z}}\,. \tag{8.4.3}$$

Here: $\mathcal{L}_h : \mathbb{R}^{\mathbb{Z}} \mapsto \mathbb{R}^{\mathbb{Z}} \mathrel{\hat{=}}$ (non-linear) finite difference operator, e.g. for finite volume semi-discretization in conservation form with 2-point numerical flux:

$$\text{(8.3.10)} \quad ➢ \quad (\mathcal{L}_h\vec{\mu})_j := -\frac{1}{h}\big(F(\mu_j,\mu_{j+1}) - F(\mu_{j-1},\mu_j)\big)\,. \tag{8.4.4}$$

Note that for conservative finite volume discretization $\mathcal{L}_h$ is *local*: $(\mathcal{L}_h(\vec{\mu}))_j$ depends only on "neighboring values" $\mu_{j-n_l},\dots,\mu_{j+n_r}$:

$$(\mathcal{L}_h\vec{\mu})_j = \mathcal{L}_j(\mu_{j-n_l},\dots,\mu_{j+n_r})\,, \tag{8.4.5}$$

with suitable functions $\mathcal{L}_j : \mathbb{R}^{1+n_l+n_r} \to \mathbb{R}$.

From (8.4.2) we get the formulas for an explicit s-stage Runge-Kutta single step method for (8.4.3), timestep $\tau > 0$:

$$\begin{aligned}
\vec{\kappa}_1 &= \mathcal{L}_h(\vec{\mu}^{(k)})\,,\\
\vec{\kappa}_2 &= \mathcal{L}_h(\vec{\mu}^{(k)} + \tau a_{21}\vec{\kappa}_1)\,,\\
\vec{\kappa}_3 &= \mathcal{L}_h(\vec{\mu}^{(k)} + \tau a_{31}\vec{\kappa}_1 + \tau a_{32}\vec{\kappa}_2)\,,\\
&\vdots\\
\vec{\kappa}_s &= \mathcal{L}_h(\vec{\mu}^{(k)} + \tau \sum_{j=1}^{s-1} a_{sj}\vec{\kappa}_j)\,,
\end{aligned} \qquad \vec{\mu}^{(k+1)} = \vec{\mu}^{(k)} + \tau \sum_{l=1}^{s} b_l \vec{\kappa}_j\,. \tag{8.4.6}$$

All increments $\vec{\kappa}_i$, $i = 1, \dots, s$, belong to $\mathbb{R}^{\mathbb{Z}}$.

The formulas (8.4.6) are "explicit" in the sense that timestepping just relies on more evaluations of the operator $\mathcal{L}_h$. This greatly facilitates implementation, because $\mathcal{L}_h$ will, in general, a non-linear and even non-smooth mapping. Thus it might be very difficult and expensive to solve a system of non-linear equations involving $\mathcal{L}_h$.

(8.4.7) Fully discrete evolution

Timestepping converts (8.4.3) into a family of equations for functions on an infinite space-time grid.

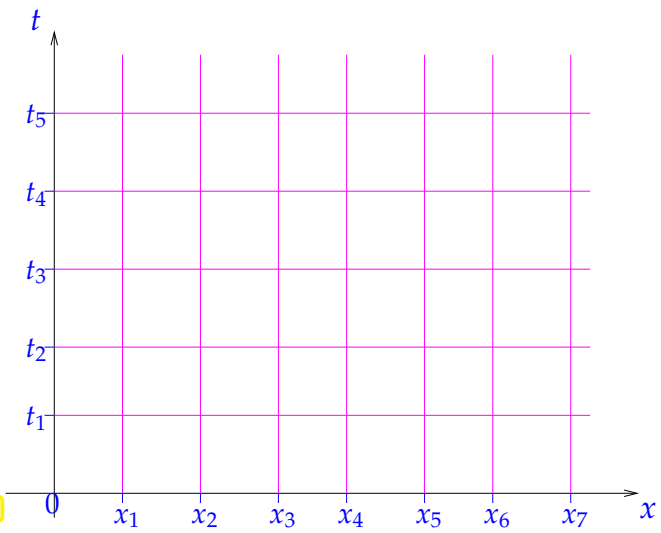


Fig. 421

Setting: equidistant spatial mesh $\mathcal{M}$, meshwidth $h > 0$, nodes $x_j := hj$, $j \in \mathbb{Z}$,

uniform timestep $\tau > 0$, $t_k := \tau k$, $k \in \mathbb{N}_0$.

Single step timestepping for (8.4.3) produces a sequence $\left(\vec{\mu}^{(k)}\right)_{k\in\mathbb{N}_0}$

$$\mu_j^{(k)} \approx u(x_j, t_k) \ , \quad j \in \mathbb{Z}, k \in \mathbb{N}_0 \ .$$

Fully discrete evolution

$$\vec{\mu}^{(k+1)} = \mathcal{H}_h(\vec{\mu}^{(k-1)}) \ , \quad k \in \mathbb{N}_0 \ .$$

$\mathcal{H}_h : \mathbb{R}^{\mathbb{Z}} \mapsto \mathbb{R}^{\mathbb{Z}}$: fully discrete evolution operator, arising from applying single step timestepping (8.4.6) to (8.4.3).

Example 8.4.8 (Fully discrete evolutions arising from conservative discretizations)

Fully discrete evolution arising from finite volume semi-discretization in conservation form with 2-point numerical flux $F = F(v, w)$

$$\text{(8.3.10)} \quad \succ \quad (\mathcal{L}_h\vec{\mu})_j := -\frac{1}{h}\big(F(\mu_j, \mu_{j+1}) - F(\mu_{j-1}, \mu_j)\big) \ . \tag{8.4.4}$$

in combination with *explicit Euler* timestepping ($\hat{=}$ 1-stage explicit RK-method)

$$\vec{\mu}^{(k+1)} = \vec{\mu}^{(k)} + \tau\mathcal{L}_h(\vec{\mu}^{(k)}) \ .$$

$$\boxed{(\mathcal{H}_h(\vec{\mu}))_j = \mu_j^{(k)} - \frac{\tau}{h}\big(F(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)}, \mu_j^{(k)})\big)} \ . \tag{8.4.9}$$

In the case of *explicit trapezoidal rule* timestepping [33, Eq. (11.4.6)] (2-stage RK-SSM, method of Heun)

$$\vec{\kappa} = \mu^{(k)} + \tau\mathcal{L}_h(\vec{\mu}^{(k)}) \quad , \quad \vec{\mu}^{(k+1)} = \mu^{(k)} + \frac{\tau}{2}\Big(\mathcal{L}_h(\mu^{(k)}) + \mathcal{L}_h(\vec{\kappa})\Big) \ .$$

$$\begin{aligned} \kappa_j := (\vec{\kappa})_j &= \mu_j^{(k)} - \frac{\tau}{h}\big(F(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)}, \mu_j^{(k)})\big) \ , \\ (\mathcal{H}_h(\vec{\mu}))_j &= \mu_j^{(k)} - \frac{\tau}{2h}\big(F(\kappa_j, \kappa_{j+1}) - F(\kappa_{j-1}, \kappa_j) + F(\mu_j^{(k)}, \mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)}, \mu_j^{(k)})\big) \ . \end{aligned} \tag{8.4.10}$$

For the *explicit midpoint rule*, another 2-stage RK-SSM, we get the recursion

$$\begin{aligned} \vec{\kappa} &= \boldsymbol{\mu}^{(k)} + \frac{\tau}{2}\boldsymbol{\mathcal{L}}_h(\vec{\boldsymbol{\mu}}^{(k)}) \quad , \quad \vec{\boldsymbol{\mu}}^{(k+1)} = \boldsymbol{\mu}^{(k)} + \tau\boldsymbol{\mathcal{L}}_h(\vec{\kappa}) \; . \\ \kappa_j &:= (\vec{\kappa})_j = \mu_j^{(k)} - \frac{\tau}{2h}\left(F(\mu_j^{(k)},\mu_{j+1}^{(k)}) - F(\mu_{j-1}^{(k)},\mu_j^{(k)})\right) \; , \\ (\boldsymbol{\mathcal{H}}_h(\vec{\boldsymbol{\mu}}))_j &= \mu_j^{(k)} - \frac{\tau}{h}\left(F(\kappa_j,\kappa_{j+1}) - F(\kappa_{j-1},\kappa_j)\right) \; . \end{aligned} \tag{8.4.11}$$

8.4.1 CFL-condition

As we have seen in Section 6.1.3 and Section 6.2.5, the use of explicit timestepping in the context of a method-of-lines approach to an initial boundary value problem for a PDE often faces a mesh-dependent timestep constraint in order to avoid catastrophic blow-up. This will also be the case for the conservative finite volume discretization of conservation laws.

(8.4.12) Difference stencils

We have already observed in (8.4.5) that the operators $\boldsymbol{\mathcal{L}}_h$ process cell averages μ_j locally. This allows a catchy representation of the structure of fully discrete evolutions.

Stencil notation: Visualization of flow of information in fully discrete *explicit* evolution (action of $\boldsymbol{\mathcal{H}}_h$), *cf.* Fig. 290.

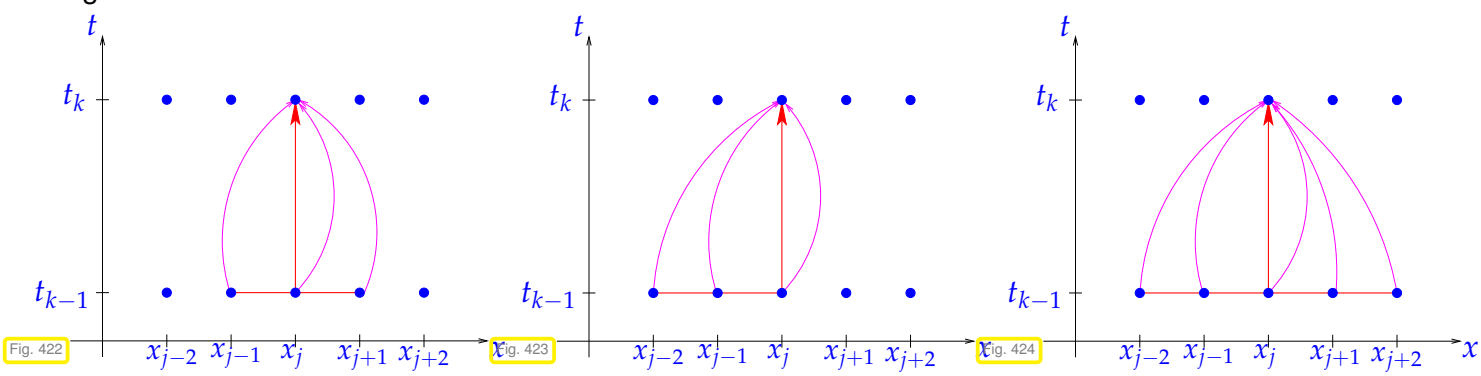


Fig. 422 2-point numerical flux & explicit Euler timestepping

Fig. 423 upwinding & explicit trapezoidal rule

Fig. 424 2-point numerical flux & explicit trapezoidal rule

(8.4.13) Common properties of conservative fully discrete evolutions

A consequence of the locality of $\boldsymbol{\mathcal{L}}_h$ combined with *explicit* timestepping: *locality* of fully discrete evolution operator:

$$\exists m_l, m_r \in \mathbb{N}_0: \quad (\boldsymbol{\mathcal{H}}(\vec{\boldsymbol{\mu}}))_j = \mathcal{H}_j(\mu_{j-m_l},\dots,\mu_{j+m_r}) \; . \tag{8.4.14}$$

If flux function f does not depend on x, $f = f(u)$ as in (8.2.7), we can expect

$$\boldsymbol{\mathcal{H}}_h \text{ is translation-invariant:} \qquad \mathcal{H}_j = \mathcal{H} \quad \forall j \in \mathbb{Z} \; .$$

This is the case for (8.4.9) and (8.4.10).

By inspection of (8.4.6): if $\boldsymbol{\mathcal{L}}_h$ is translation-invariant

$$(\boldsymbol{\mathcal{L}}_h(\vec{\boldsymbol{\mu}}))_j = \mathcal{L}(\mu_{j-n_l}, \dots \mu_{j+n_r}) \ , \quad j \in \mathbb{Z} \ ,$$

and timestepping relies on an s-stage explicit Runge-Kutta method, then we conclude for m_l, m_r in (8.4.14)

$$m_l \leq s \cdot n_l \quad , \quad m_r \leq s \cdot n_r \ .$$

(8.4.15) Domains of dependence

Now we revisit a concept from Sect. 6.2.5, see, in particular, Rem. 6.2.56:

Definition 8.4.16. Numerical domain of dependence

Consider explicit translation-invariant fully discrete evolution $\vec{\boldsymbol{\mu}}^{(k+1)} := \boldsymbol{\mathcal{H}}(\vec{\boldsymbol{\mu}}^{(k)})$ on uniform spatio-temporal mesh ($x_j = hj$, $j \in \mathbb{Z}$, $t_k = k\tau$, $k \in \mathbb{N}_0$) with

$$\exists m \in \mathbb{N}_0 \colon \quad (\boldsymbol{\mathcal{H}}(\vec{\boldsymbol{\mu}}))_j = \mathcal{H}(\mu_{j-m}, \dots, \mu_{j+m}) \ , \quad j \in \mathbb{Z} \ . \tag{8.4.17}$$

Then the numerical domain of dependence is given by

$$D_h^-(x_j, t_k) := \{(x_m, t_l) \in \mathbb{R} \times [0, t_k] \colon j - m(k-l) \leq m \leq j + m(k-l)\} \ .$$

From Thm. 8.2.43 recall the maximal analytical domain of dependence for a solution of (8.2.7)

$$D^-(\overline{x}, \overline{t}) := \{(x,t) \in \mathbb{R} \times [0, \overline{t}] \colon \dot{s}_{\min}(\overline{t} - t) \leq x - \overline{x} \leq \dot{s}_{\max}(\overline{t} - t)\} \ .$$

with maximals speeds of propagation

$$\dot{s}_{\min} := \min\{f'(\xi) : \inf_{x \in \mathbb{R}} u_0(x) \leq \xi \leq \sup_{x \in \mathbb{R}} u_0(x)\} \ , \tag{8.4.18}$$

$$\dot{s}_{\max} := \max\{f'(\xi) : \inf_{x \in \mathbb{R}} u_0(x) \leq \xi \leq \sup_{x \in \mathbb{R}} u_0(x)\} \ . \tag{8.4.19}$$

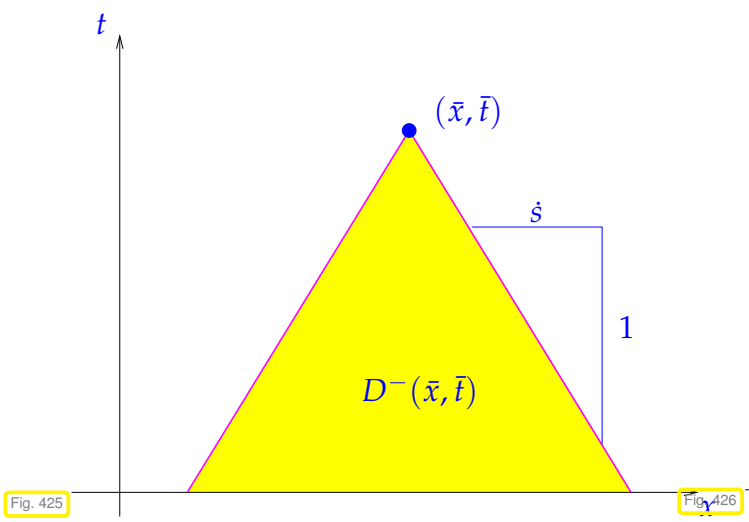


Fig. 425

$D^-(\bar{x}, \bar{t}) \subset \mathbb{R} \times [0, T]$

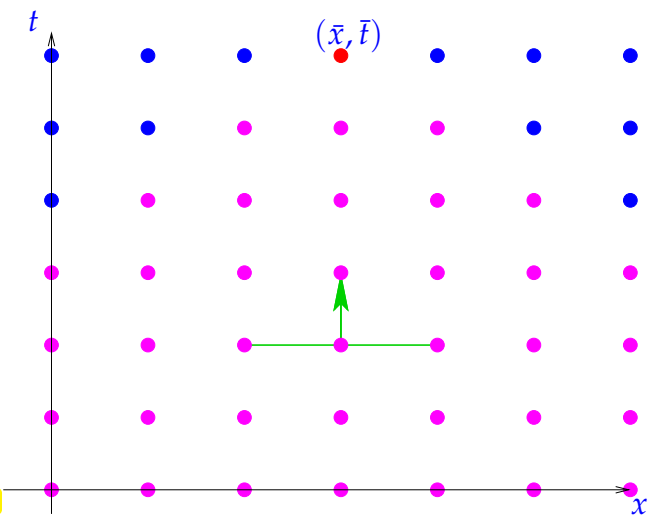


Fig. 426

$D_h^-(\bar{x}, \bar{t})$ for 3-point stencil

(8.4.20) CFL-condition

Definition 8.4.21. Courant-Friedrichs-Lewy (CFL-)condition → **Rem. 6.2.56**

An explicit translation-invariant local fully discrete evolution $\vec{\mu}^{(k+1)} := \mathcal{H}(\vec{\mu}^{(k)})$ on uniform spatio-temporal mesh ($x_j = hj$, $j \in \mathbb{Z}$, $t_k = k\tau$, $k \in \mathbb{N}_0$) as in Def. 8.4.16 satisfies the Courant-Friedrichs-Lewy (CFL-)condition, if the convex hull of its numerical domain of dependence contains the maximal analytical domain of dependence:

$$D^-(x_j, t_k) \subset \operatorname{convex}(D_h^-(x_j, t_k))$$

By definition of $D^-(\overline{x}, \overline{t})$ and $D_h^-(x_j, t_k)$ sufficient for the CFL-condition is

$$\boxed{\frac{\tau}{h} \leq \frac{m}{\dot{s}}} \quad \longleftrightarrow \quad \textit{timestep constraint}! \,. \tag{8.4.22}$$

This is a timestep constraint similar to the one encountered in Sect. 6.2.5 in the context of leapfrog timestepping for the semi-discrete wave equation.

Remember Rem. 6.1.101, page 481: stability induced timestep constraint can lead to an inefficient discretization. Also in the case of the "ODE" (8.3.9) implicit timestepping can circumvent the CFL-condition. Yet, at the price of having to solve *non-linear systems* of equations, which may be prohibitive and makes people put up with the moderate timestep constraint (8.4.22) gladly.

As discussed in Rem. 6.2.56,

We cannot expect convergence for *fixed ratio* $\tau : h$, for $h \to 0$ in case the CFL-condition is violated.

Refer to Fig. 427 for a "graphical argument":

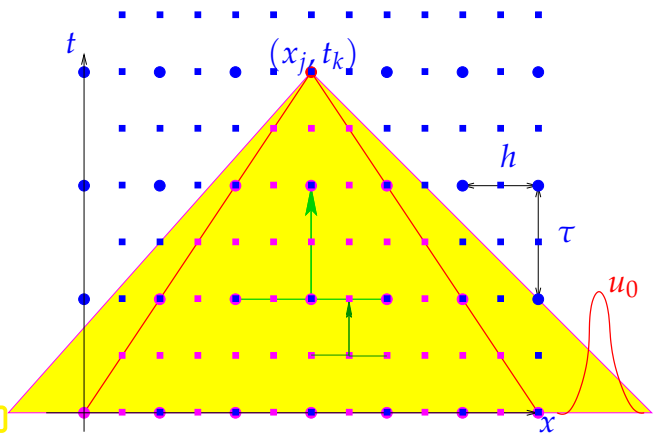


Fig. 427

(• ≙ coarse grid, ■ ≙ fine grid, □ ≙ d.o.d)

◁ Sequence of equidistant space-time grids of $\mathbb{R} \times [0, T]$ with $\tau = \gamma h$ (τ/h = meshwidth in time/space)

If $\gamma >$ CFL-constraint (8.4.22) then

analytical domain of dependence $\not\subset$ numerical domain of dependence

Heuristic reasoning: Initial data u_0 supported outside the numerical domain of dependence can influence the exact solution in the point (x_j, t_k), which is contained in all spatio-temporal grids of the sequence. However, $\mu_j^{(k)}$ will never be influenced by initial data inside the support of u_0. Hence, there can be cases, when $\mu_j^{(k)} \not\to u(x_j, t_k)$ though $h, \tau \to 0$.

8.4.2 Linear stability

In Section 6.1.5.2 (parabolic evolutions) and Section 6.2.5 (linear wave equations) we found that for the method of lines combined with *explicit* timestepping

timestep constraints $\tau \leq O(h^r)$, $r \in \{1,2\}$, *necessary* to avoid exponential blow-up (*instability*)

Is the timestep constraint (8.4.22) suggested by the CFL-condition also stipulated by stability requirements**?**

(8.4.23) Focus on linear advection

We are going to investigate the question only for the Cauchy problem for scalar *linear* advection in 1D with constant velocity $v > 0$:

$$\frac{\partial u}{\partial t} + v\frac{\partial u}{\partial x} = 0 \quad \text{in} \quad \mathbb{R}\times]0,T[\,, \quad u(x,0) = u_0(x) \quad \forall x \in \mathbb{R}\,. \tag{8.1.10}$$

Method of lines approach: Semi-discretization in space on equidistant mesh with meshwidth $h > 0$ leads to a

➢ *linear, local, and translation-invariant* semi-discrete evolution

$$\frac{d\vec{\mu}}{dt}(t) = \mathcal{L}_h(\vec{\mu}(t))\,, \qquad \text{with} \qquad \boxed{(\mathcal{L}_h(\vec{\mu}))_j = \sum_{l=-m}^{m} c_l \mu_{j+l}\,, \quad j \in \mathbb{Z}}\,, \tag{8.4.24}$$

for suitable weights $c_l \in \mathbb{R}$. This is also called a stencil formula, *cf.* § 8.4.12, m is the width of the stencil.

Explanation of terminology:

- *linear*: The finite difference operator $\mathcal{L}_h : \mathbb{R}^{\mathbb{Z}} \mapsto \mathbb{R}^{\mathbb{Z}}$ is linear.
- *local*: $(\mathcal{L}_h(\vec{\mu}))_j$ depends only on a few coefficients μ_{j+l} for small $|l|$, *cf.* page 624.
- *translation-invariant*: if $\eta_j := \mu_{j+1}$, then $(\mathcal{L}_h(\vec{\eta}))_j = (\mathcal{L}_h(\vec{\mu}))_{j+1}$ (the finite difference operator commutes with shifts of the coefficient vector, *cf.* page 624).

Example 8.4.25 (Upwind difference operator for linear advection)

Finite volume semi-discretization of (8.1.10) in conservation form with Godunov numerical flux (8.3.53) (, which agrees with the upwind flux (8.3.39) in this case)

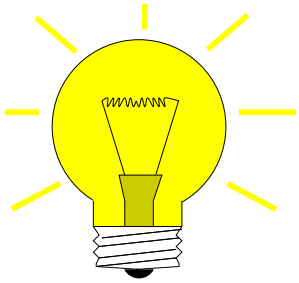
$$(\mathcal{L}_h(\vec{\mu}))_j = -\frac{v}{h}(\mu_j - \mu_{j-1})\,. \tag{8.4.26}$$

► Coefficients in (8.4.24): $c_0 = -\frac{v}{h}$, $c_{-1} = \frac{v}{h}$.

Note: In this case the (loca) Lax-Friedrichs/Rusanov numerical flux (8.3.36) yields the same $\mathcal{L}_h$.

As in Section 6.1.5.2 and Section 6.2.5 we employ a **diagonalization technique** (with a new twist). The policy was to expand the vector of unknowns of the semi-discrete evolution into eigenvectors of the "right-hand-side operator" of the method-of-lines ODE.

Now the new twist is that $\mathcal{L}_h$ acts on the sequence space $\mathbb{R}^{\mathbb{Z}}$!



Idea: trial expression for "eigenvectors" (in $\mathbb{C}^{\mathbb{Z}}$)

"complex waves" $\left(\vec{\psi}_\xi\right)_j := \exp(\imath\xi j)\,, \quad j \in \mathbb{Z}\,, \quad -\pi < \xi \leq \pi\,.$ (8.4.27)

An infinite set of eigenvectors, matching $\dim \mathbb{C}^{\mathbb{Z}} = \infty$!

Remark 8.4.28 (Diagonalization in $\mathbb{C}$)

Why do we have to consider complex-valued eigenvectors? Well, remember from linear algebra that purely real matrices may have complex eigenvalues. Here, purely real finite difference operators have a complex spectrum!

By straightforward computations, using $\exp(x+y) = \exp(x)\exp(y)$, we verify the eigenvector property and compute the corresponding eigenvalues:

$$(\mathcal{L}_h(\vec{\mu}))_j = \sum_{l=-m}^{m} c_l \mu_{j+l} \quad \Rightarrow \quad \mathcal{L}_h \psi^\xi = \underbrace{\left(\sum_{l=-m}^{m} c_l \exp(i\xi l)\right)}_{\text{"eigenvalue" } \hat{c}_h(\xi)} \psi^\xi\,.$$

spectrum of $\mathcal{L}_h$: $\sigma(\mathcal{L}_h) = \{\hat{c}_h(\xi) := \sum_{l=-m}^{m} c_l \exp(\imath\xi l) \colon -\pi < \xi \leq \pi\}\,.$ (8.4.29)

Terminology: The function $\hat{c}_h(\xi)$ is known as the symbol of the difference operator $\mathcal{L}_h$, *cf.* the concept of symbol of a differential operator.

Remark 8.4.30 (Eigenvectors of translation invariant linear operators)

In [33, Chapter 9] periodic linear time-invariant filters are introduced as linear operators on the space of n-periodic sequences that commute with translation, see [33, Def. 9.1.13]. They are described by *circulant matrices*, see [33, Def. 9.1.17]. The linear difference operator $\mathcal{L}_h$ from (8.4.24) generalizes this concept, because it still features translation invariance, but acts on infinite sequencies. In fact, $\mathcal{L}_h$ can be represented by means of an infinite banded circulant matrix with respect to the "unit vector basis" of the space of sequences $\mathbb{R}^{\mathbb{Z}}$

$$\mathcal{L}_h \sim \begin{bmatrix} \ddots & & \ddots & & & & & \\ & \ddots & & \ddots & 0 & & & \\ \ddots & & c_0 & \dots & c_m & 0 & & \\ 0 & c_{-m} & \dots & c_0 & \dots & c_m & 0 & \\ & 0 & c_{-m} & \dots & c_0 & \dots & c_m & 0 \\ & & & \ddots & & \ddots & & \ddots \end{bmatrix}$$

.

According to [33, Lemma 9.2.11] the columns of the Fourier matrix [33, Eq. (9.2.8)], the vectors $(\exp(\frac{2\pi jk}{n}))_{j=0}^{n-1} \in \mathbb{C}^n$, $k = 0, \dots, n-1$, provide the eigenvectors of any circulant matrix $\in \mathbb{C}^n$. The generalization of these "Fourier harmonics" to $\mathbb{R}^{\mathbb{Z}}$ are the complex waves defined in (8.4.27). Therefore we can expect them to furnish eigenvectors for $\boldsymbol{\mathcal{L}}_h$.

Example 8.4.31 (Spectrum of upwind difference operator)

Apply formula (8.4.29) with $c_0 = -\frac{v}{h}$, $c_{-1} = \frac{v}{h}$ (from (8.4.26)):

For $\boldsymbol{\mathcal{L}}_h$ from (8.4.26): $\sigma(\boldsymbol{\mathcal{L}}_h) = \left\{ \frac{v}{h}(\exp(-\imath\xi) - 1) \colon -\pi < \xi \leq \pi \right\}$

Spectrum of upwind finite difference operator for linear advection with velocity $v > 0$ (meshwidth $h > 0$) as a subset of $\mathbb{C}$ ▷

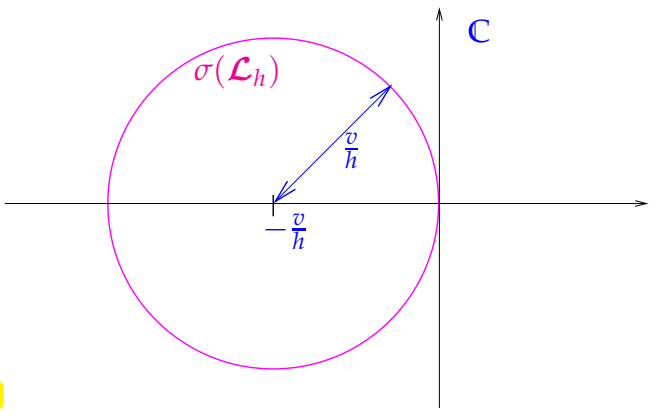


Fig. 428

(8.4.32) Diagonalization of semi-discrete evolution

The eigenvalue $\hat{c}_h(\xi)$ will govern the evolution when we choose $\vec{\boldsymbol{\psi}}_\xi$ as initial value:

$$\boldsymbol{\mathcal{L}}\vec{\boldsymbol{\psi}}_\xi = \hat{c}(\xi)\vec{\boldsymbol{\psi}}_\xi \,,$$

$$\frac{d\vec{\boldsymbol{\mu}}}{dt}(t) = \boldsymbol{\mathcal{L}}_h(\vec{\boldsymbol{\mu}}(t)) \; bcom \qquad \Rightarrow \quad \vec{\boldsymbol{\mu}}(t) = \exp(\hat{x}_\xi)\vec{\boldsymbol{\xi}}_\xi \,, \tag{8.4.33}$$

$$\vec{\boldsymbol{\mu}}(0) = \vec{\boldsymbol{\psi}}_\xi$$

as can be seen by simply differentiation.

In § 6.1.58 the principal idea was an expansion of the time-dependent vector of unknown coefficients as a finite linear combination of eigenvector of the spatially discrete evolution operator. However, now we have to deal with uncountably many "eigenvectors" $\vec{\boldsymbol{\psi}}_\xi$, $-\pi < \xi \leq \pi$, so that linear combination becomes integration over $[-\pi, \pi]$:

$$\vec{\boldsymbol{\mu}}(t) = \int_{-\pi}^{\pi} \hat{\mu}(t,\xi)\vec{\boldsymbol{\psi}}_\xi \,\mathrm{d}\xi \quad \Leftrightarrow \quad \mu_j(t) = \int_{-\pi}^{\pi} \hat{\mu}(t,\xi)\, \exp(\imath\xi j)\,\mathrm{d}\xi \,. \tag{8.4.34}$$

$$\blacktriangleright \quad \frac{d\vec{\boldsymbol{\mu}}}{dt}(t) = \boldsymbol{\mathcal{L}}_h(\vec{\boldsymbol{\mu}}(t)) \quad \Rightarrow \quad \boxed{\frac{\partial\hat{\mu}}{\partial t}(t,\xi) = \hat{c}_h(\xi)\hat{\mu}(t,\xi)} \,. \tag{8.4.35}$$

This is a family of *decoupled* scalar, linear ODEs parameterized by $\xi \in]-\pi,\pi]$.

Remark 8.4.36 (Fourier series $\rightarrow$ [33, Section 9.2.5])

Up to normalization the relationship

$$\vec{\mu}^{(0)} \in \mathbb{R}^{\mathbb{Z}} \quad \leftrightarrow \quad \hat{\mu}^{(0)} :]-\pi,\pi] \mapsto \mathbb{C}$$

from (8.4.34) is the Fourier series transform, which maps a sequence to a 2π-periodic function. It has the important isometry property

$$\sum_{j=-\infty}^{\infty} |\mu_j|^2 = 2\pi \int_{-\pi}^{\pi} |\hat{\mu}(\xi)|^2 \,\mathrm{d}\xi \,.$$

➢ The symbol $\hat{c}_h$ can be viewed as the *representation of a difference operator in Fourier domain*.

The decoupling manifest in (8.4.35) carries over to Runge-Kutta timestepping in the sense of the commuting diagram (6.1.85). If $\mathbf{\Psi}^\tau$ is the discrete evolution operator ($\rightarrow$ § 6.1.34) induced by an s-stage Runge-Kutta single step method according to Def. 6.1.40 with timestep $\tau > 0$ for the ODE $\dot{\vec{\mu}} = \mathcal{L}_h(\vec{\mu})$, $\mathcal{L}_h$ from (8.4.24), then straightforward computations yield

$$\mathbf{\Psi}^\tau \vec{\boldsymbol{\psi}}_\xi = \Psi^\tau_\xi \hat{c}(\xi)\, \vec{\boldsymbol{\xi}}_\xi \,, \tag{8.4.37}$$

where $\Psi^\tau_\xi \in \mathbb{C}$ is the (multiplication) discrete evolution operator describing the application of the same RK-SSM to the scalar ODE $\dot{\mu} = \hat{c}(\xi)\mu$.

To put these considerations into the diagonalization framework, we introduce the Fourier transforms of the members of the sequence $\left(\vec{\mu}^{(k)}\right)_k$ created by Runge-Kutta timestepping

$$\vec{\mu}^{(k)} = \int_{-\pi}^{\pi} \hat{\mu}^{(k)}(\xi)\vec{\boldsymbol{\psi}}_\xi \,\mathrm{d}\xi \quad \Leftrightarrow \quad \mu_j^{(k)} = \int_{-\pi}^{\pi} \hat{\mu}^{(k)}(\xi)\, \exp(\imath\xi j)\,\mathrm{d}\xi \,. \tag{8.4.38}$$

Then from (8.4.37), formally appealing to the linearity of $\mathcal{L}_h$, we conclude that

$$\vec{\mu}^{(k+1)} = \mathbf{\Psi}^\tau \vec{\mu}^{(k)} = \int_{-\pi}^{\pi} \hat{\mu}^{(k)}(\xi)\mathbf{\Psi}^\tau \vec{\boldsymbol{\psi}}_\xi \,\mathrm{d}\xi = \int_{-\pi}^{\pi} \hat{\mu}^{(k)}\Psi^\tau_\xi \vec{\boldsymbol{\psi}}_\xi \,\mathrm{d}\xi \,. \tag{8.4.39}$$

Hence, $\xi \mapsto \hat{\mu}^{(k)}(\xi)\Psi^\tau_\xi$ has been identified as the Fourier transform of $\vec{\mu}^{(k+1)}$ and we find

$$\hat{\mu}^{(k)} = \left(\Psi^\tau_\xi\right)^k \hat{\mu}^{(0)} \,, \quad k \in \mathbb{N} \,. \tag{8.4.40}$$

Example 8.4.41 (Explicit Euler in Fourier domain)

Let us apply the above formulas to explicit Euler timestepping [33, Eq. (11.2.7)] for semi-discrete evolution (8.4.24), see also (8.4.9),

$$\vec{\mu}^{(k+1)} = \vec{\mu}^{(k)} + \tau\mathcal{L}_h\vec{\mu}^{(k)} \,.$$

$$\int_{-\pi}^{\pi} \hat{\mu}^{(k+1)}(\xi)\,\vec{\boldsymbol{\psi}}_{\xi}\,\mathrm{d}\xi = (\mathsf{Id} + \tau\boldsymbol{\mathcal{L}}_h)\int_{-\pi}^{\pi} \hat{\mu}^{(k)}(\xi)\vec{\boldsymbol{\psi}}_{\xi}\,\mathrm{d}\xi = \int_{-\pi}^{\pi} \hat{\mu}^{(k)}(\xi)(1+\tau\hat{c}_h(\xi))\,\mathrm{d}\xi\,.$$

$$\hat{\mu}^{(k+1)}(\xi) = \hat{\mu}^{(k)}(\xi)(1+\tau\hat{c}_h(\xi))\,.$$

In Fourier domain a single explicit Euler timestep corresponds to a multiplication of $\hat{\mu} :]-\pi,\pi] \mapsto \mathbb{C}$ with the function $(1+\tau\hat{c}_h) :]-\pi,\pi] \mapsto \mathbb{C}$.

We get the same result when applying an explicit Euler step to the ODE $\frac{\partial\hat{\mu}}{\partial t}(t,\xi) = \hat{c}_h(\xi)\hat{\mu}(t,\xi)$ from (8.4.35) with paramter ξ:

$$\hat{\mu}^{(k+1)}(\xi) = (1+\tau\hat{c}_h(\xi))\hat{\mu}^{(k)}(\xi)\,.$$

We summarize the observation made in the previous example: For the sequence $\left(\vec{\boldsymbol{\mu}}^{(k)}\right)_{k\in\mathbb{N}_0}$ generated by an RK-SSM for the linear MOL-ODE (8.4.24) holds

$$\vec{\boldsymbol{\mu}}^{(k)} = \int_{-\pi}^{\pi} \hat{\mu}^{(k)}(\xi)\vec{\boldsymbol{\psi}}_{\xi}\,\mathrm{d}\xi\,,$$

where $\left(\hat{\mu}^{(k)}(\xi)\right)_{k\in\mathbb{N}_0}$ is the sequence of approximations created by the Runge-Kutta method when applied to the scalar linear initial value problem

$$\dot{y} = \hat{c}(\xi)\,y \quad , \quad y(0) = \hat{\mu}^{(0)}(\xi)\,.$$

Clearly, timestepping can only be stable, if blowup $|\hat{\mu}^{(k)}(\xi)| \to \infty$ for $k \to \infty$ can be avoided **for all** $-\pi < \xi \leq \pi$.

From [33, Thm. 12.1.15] we know a rather explicit formula for the (complex) numbers Ψ_{ξ}^{τ}:

Theorem 8.4.42. Stability function of explicit Runge-Kutta methods

The execution of one step of size $\tau > 0$ of an explicit *s-stage Runge-Kutta single step method (→ Def. 6.1.40) with Butcher scheme* $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$ *(see* (6.1.42)*) for the scalar linear ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{C}$, amounts to a multiplication with the number*

$$\Psi_{\lambda}^{\tau} = \underbrace{1 + z\mathbf{b}^T(\mathbf{I} - z\mathfrak{A})^{-1}\mathbf{1}}_{\text{stability function } S(z)} = \det(\mathbf{I} - z\mathfrak{A} + z\mathbf{1}\mathbf{b}^T)\,, \quad z := \lambda\tau\,, \quad \mathbf{1} = (1,\dots,1)^T \in \mathbb{R}^s\,.$$

Example 8.4.43 (Stability functions of explicit RK-methods)

- Explicit Euler method (8.4.9) : $\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array}$ ➢ $S(z) = 1 + z\,.$

- Explicit trapezoidal rule (8.4.10) : $\begin{array}{c|cc} 0 & 0 & 0 \\ 1 & 1 & 0 \\ \hline & \frac{1}{2} & \frac{1}{2} \end{array}$ ➢ $S(z) = 1 + z + \frac{1}{2}z^2\,.$

- Classical RK4-method [33, Ex. 11.4.13]: $\begin{array}{c|cccc} 0 & 0 & 0 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ \hline & \frac{1}{6} & \frac{2}{6} & \frac{2}{6} & \frac{1}{6} \end{array}$ ➢ $S(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + \frac{1}{24}z^4\,.$

Thm 8.4.2 together with the combinatorial formula for the determinant means that $\Psi_\lambda^\tau(z)$ is a polynomial of degree $\leq s$ in $z \in \mathbb{C}$.

So we conclude for the evolution of the "Fourier transforms" $\hat{\mu}^{(k)}(\xi)$:

$$\boxed{\hat{\mu}^{(k+1)}(\xi) = S(\tau\hat{c}(\xi)) \cdot \hat{\mu}^{(k)}(\xi)\,, \quad k \in \mathbb{N}_0\,, \quad -\pi < \xi \leq \pi}\,,$$

where $z \mapsto S(z)$ is the stability function of the Runge-Kutta timestepping method, see Thm. 8.4.2. For the explicit Euler method we recover the formula of Ex. 8.4.41.

$$\boxed{\text{Stability of RK-timestepping of linear semi-discrete evolution} \iff \max_{-\pi<\xi\leq\pi} |S(\tau\hat{c}(\xi))| \leq 1}$$

The linear stability analysis based on Fourier symbols of difference operators for Cauchy problems is often referred to as von Neumann stability analysis.

Remark 8.4.44 (Stability domains)

Terminology in the theory of Runge-Kutta single step methods [33, Def. 12.1.49]:

Stability domain: $\{z \in \mathbb{C}\colon |S(z)| \leq 1\}\,.$

Stability domains:

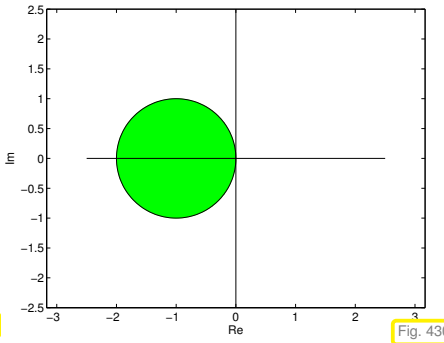


Fig. 429 Fig. 430

explicit Euler method

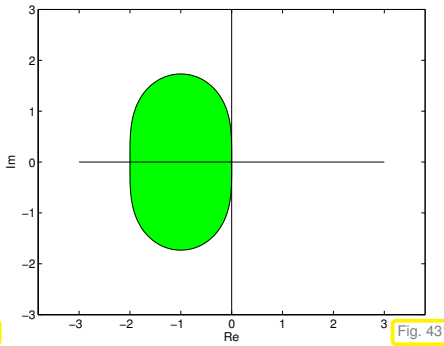
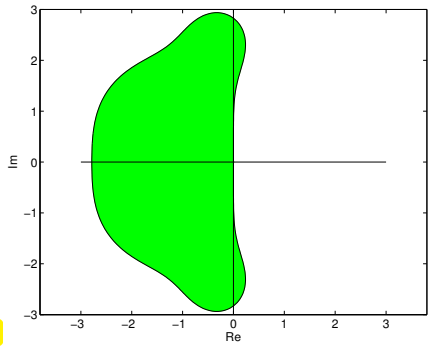


Fig. 431

explicit trapezoidal rule



Classical RK4-method

For explicit RK-SSM the stability function $S(z)$ is a polynomial, see [33, § 12.1.47]. Therefore, their stability domains will invariably be *bounded* sets in $\mathbb{C}$.

► Necessary stability condition for RK-SSM for linear evolutions in $\mathbb{R}^{\mathbb{Z}}$:

$$\{\tau\hat{c}(\xi)\,,\ -\pi < \xi \leq \pi\} \subset \text{stability domain of RK-SSM}$$

Example 8.4.45 (Stability and CFL condition)

Consider: upwind spatial discretization (8.4.26) & explicit Euler timestepping

➢ symbol of difference operator (→ Ex. 8.4.31): $\hat{c}_h(\xi) = \frac{v}{h}(\exp(-\imath\xi) - 1)$,

stability function: $S(z) = 1 + z$.

Locus of

$$\Sigma := S(\tau\hat{c}(\xi))\,, \quad -\pi < \xi \leq \pi\,,$$

in the complex plane

(Unit circle in green)

▷

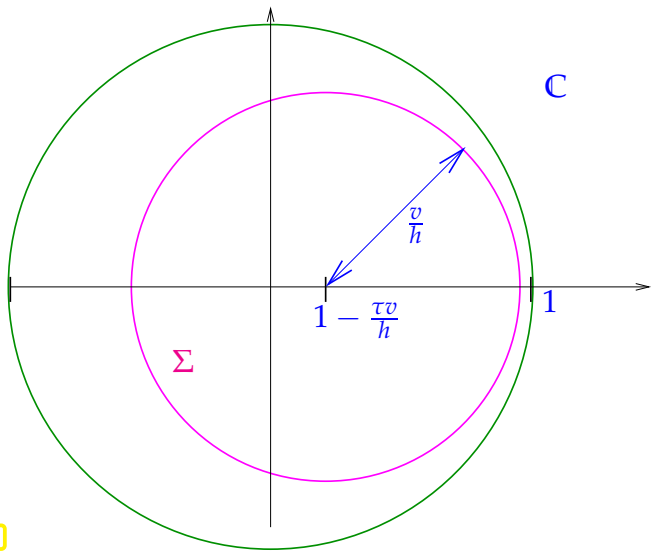


Fig. 432

► $$|S(\tau\hat{c}(\xi))| \leq 1 \quad \forall -\pi < \xi \leq \pi \iff v\frac{\tau}{h} \leq 1\,.$$

= CFL-condition of Def. 8.4.21! Note that the maximal analytic region of dependence for constant velocity v linear advection is merely a line with slope v in the $x - t$-plane, see Ex. 8.2.11.

Consider: upwind spatial discretization (8.4.26) & explicit trapezoidal rule: stability function $S(z) = 1 + z + \frac{1}{2}z^2$

Plots for $v = 1$, $\tau = 1$

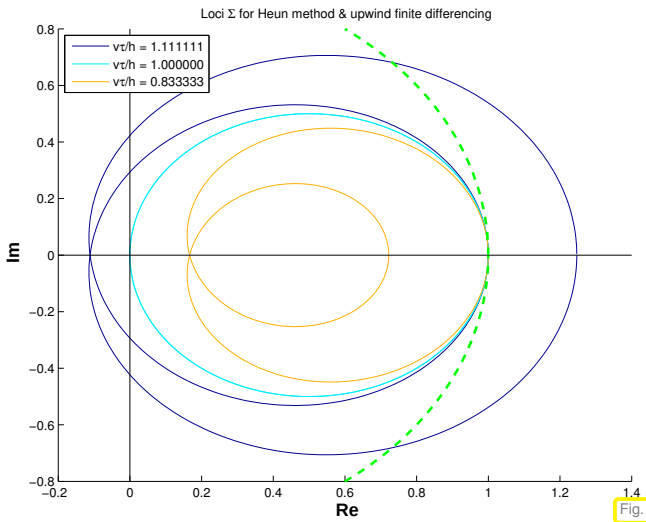


Fig. 433

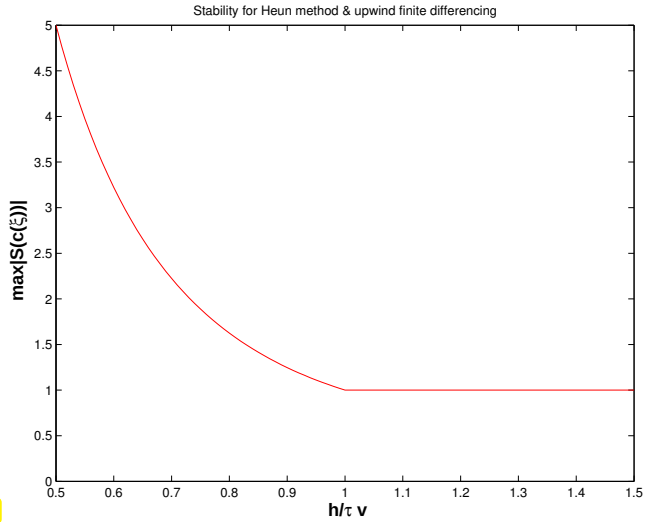


Fig. 434

$$\blacktriangleright \quad |S(\tau\hat{c}(\xi))| \leq 1 \quad \forall -\pi < \xi \leq \pi \quad \Longleftrightarrow \quad v\frac{\tau}{h} \leq 1 \,.$$

= *tighter timestep constraint* than stipulated by mere CFL-condition (8.4.22). To see this note that the explicit trapezoidal rule is a 2-stage Runge-Kutta method. Hence, the spatial stencil has width 2 in upwind direction, see Fig. 423.

Stability induced timestep constraint

For an *explicit* Runge-Kutta single-step method applied to a linear semi-discrete evolution (8.4.24) the necessary stability condition $\max\limits_{-\pi\leq\xi\leq\pi} |S(\tau\hat{c}(\xi))| \leq 1$ implies a *timestep constraint*.

8.4.3 Convergence

Experiment 8.4.47 (Convergence of fully discrete finite volume methods for Burgers equation)

This example presents a comprehensive *empirical* investigation of the convergence of simple finite volume methods.

- Cauchy problem for Burgers equation (8.1.46)

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x}(\tfrac{1}{2}u^2) = 0 \quad \text{in } \mathbb{R}\times]0,T[\,, \quad u(x,0) = u_0(x)\,, \quad x \in \mathbb{R}\,.$$

- smooth, non-smooth and discontinuous initial data, supported in $[0,1]$:

$$u_0(x) = 1 - \cos^2(\pi x)\,, \quad 0 \leq x \leq 1\,, \quad 0 \text{ elsewhere}\,, \qquad \text{(BUMP)}$$
$$u_0(x) = 1 - 2 * |x - \tfrac{1}{2}|\,, \quad 0 \leq x \leq 1\,, \quad 0 \text{ elsewhere}\,, \qquad \text{(WEDGE)}$$
$$u_0(x) = 1\,, \quad 0 \leq x \leq 1\,, \quad 0 \text{ elsewhere}\,. \qquad \text{(BOX)}$$

➢ maximum speed of propagation $\dot{s} = 1$.

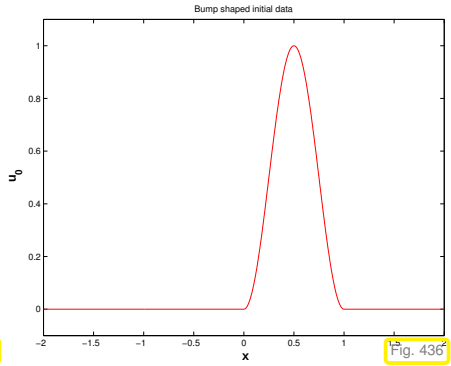


Fig. 435 Fig. 436

(BUMP)

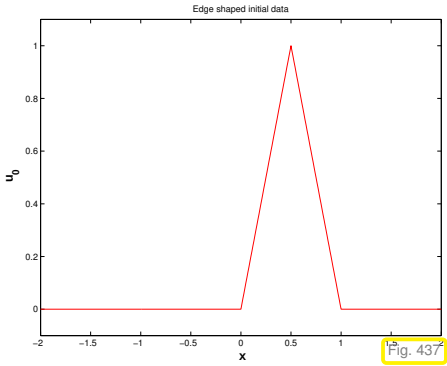
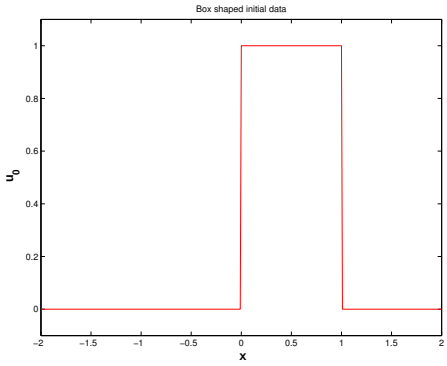


Fig. 437

(WEDGE)



(BOX)

- Spatial discretization on equidistant mesh with meshwidth $h > 0$ based on finite volume method in conservation form with

  ❶ (local) Lax-Friedrichs numerical flux (8.3.36),
  ❷ Godunov numerical flux (8.3.53).

- Initial values $\vec{\mu}^{(0)}$ obtained from dual cell averages.
- Explicit Runge-Kutta (order 4) timestepping with uniform timestep $\tau > 0$.
- Fixed ratio: $\tau : h = 1$ (➢ CFL-condition satisfied)
- Monitored: error norms (log-log plots)

$$\mathrm{err}_1(h) := \max_{k>0} h \sum_j |\mu_j^{(k)} - u(x_j, t_k)| \approx \max_{k>0} \left\| u_N^{(k)} - u(\cdot, t_k) \right\|_{L^1(\mathbb{R})} , \tag{8.4.48}$$

$$\mathrm{err}_\infty(h) := \max_{k>0} \max_{j\in\mathbb{Z}} |\mu_j^{(k)} - u(x_j, t_k)| \approx \max_{k>0} \left\| u_N^{(k)} - u(\cdot, t_k) \right\|_{L^\infty(\mathbb{R})} . \tag{8.4.49}$$

for different final times $T = 0.3, 4$, $h \in \{\frac{1}{20}, \frac{1}{40}, \frac{1}{80}, \frac{1}{160}, \frac{1}{320}, \frac{1}{640}, \frac{1}{1280}\}$.

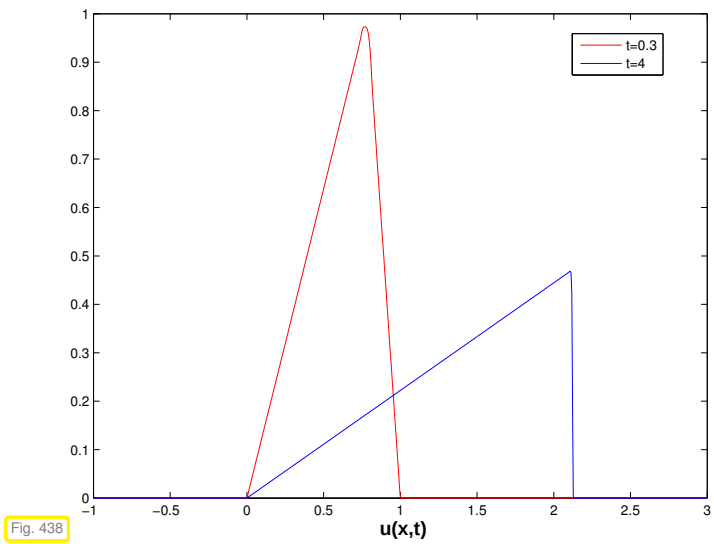


Fig. 438

"Exact solution": Initial data (WEDGE)

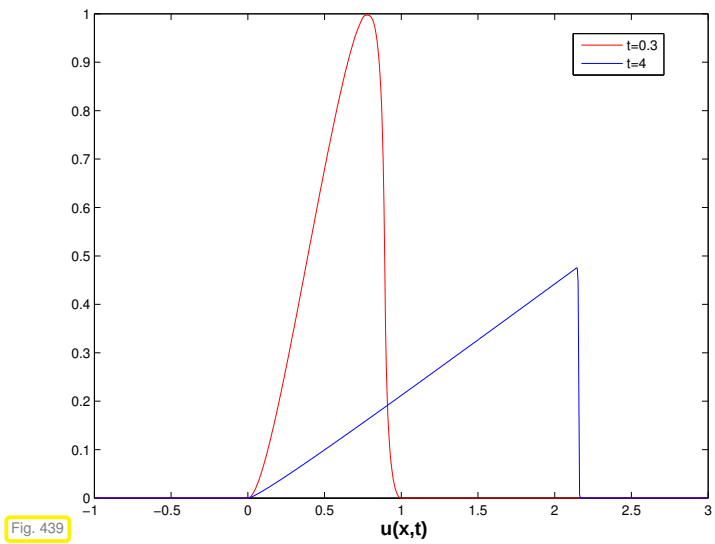


Fig. 439

"Exact solution": Initial data (BUMP)

These "exact solutions' were computed with a MUSCL scheme (→ Sect. 8.5.3) on an equidistant mesh with $h = 10^{-4}$

Note: for bump initial data (BUMP) we can still expect $u(\cdot, 0.3)$ to be smoot, because characteristics will not intersect before that time, *cf.* (8.2.13) and Ex. 8.2.14.

Why do we study the particular error norms (8.4.48) and (8.4.49)?

From Thm. 8.2.41 and Thm. 8.2.43 we know that the evolution for a scalar conservation law in 1D enjoys stability on the norms $\|\cdot\|_{L^1(\mathbb{R})}$ and $\|\cdot\|_{L^\infty(\mathbb{R})}$. Hence, these norms are the natural norms for measuring discretization errors, *cf.* the use of the energy norm for measuring the finite element discretization error for 2nd order elliptic BVP.

$T = 0.3$, error err_1

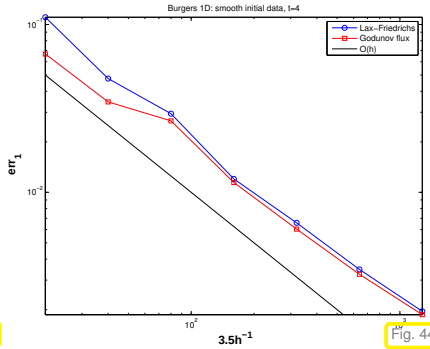


Fig. 440

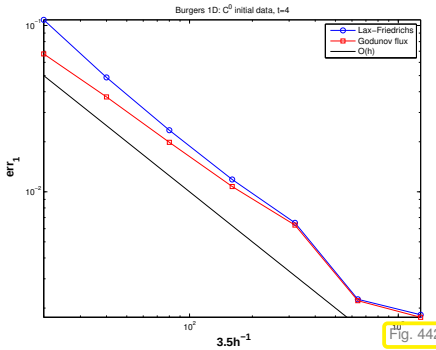


Fig. 441

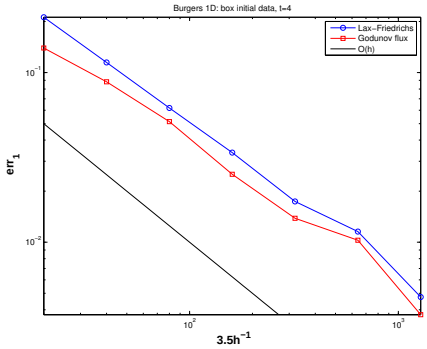


Fig. 442

$T = 0.3$: error err_∞

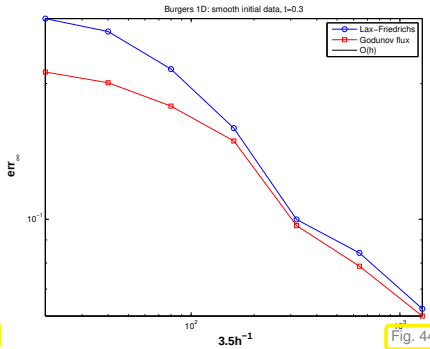


Fig. 443

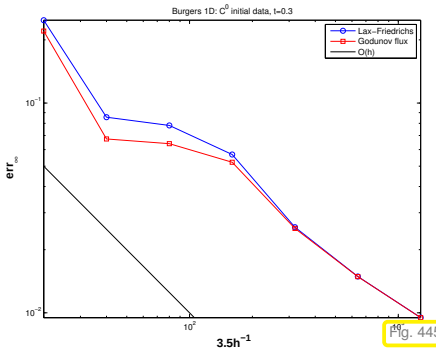


Fig. 444

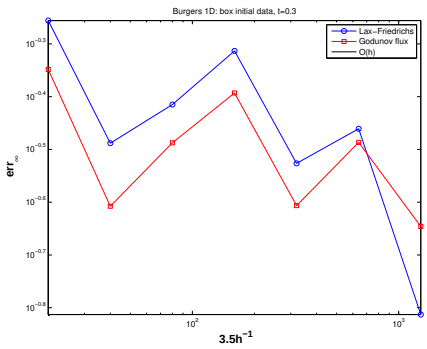


Fig. 445

$T = 4$: error err_∞

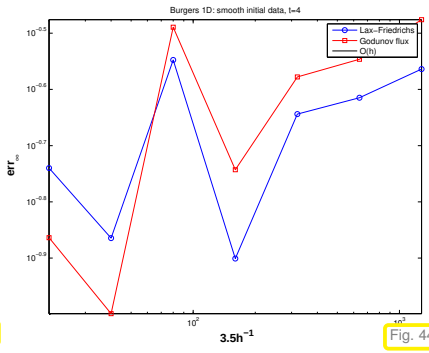


Fig. 446

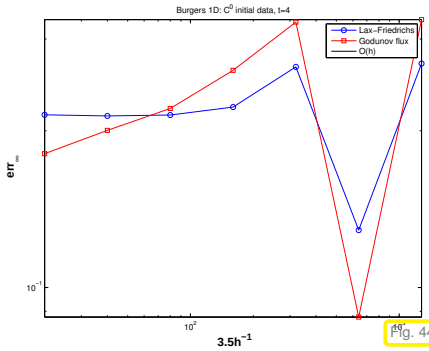


Fig. 447

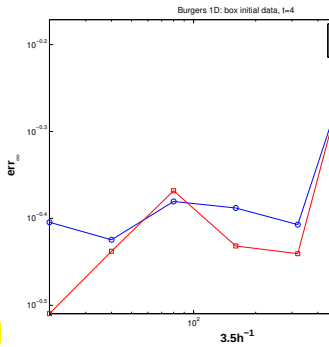


Fig. 448

Error obtained by comparison with numerical "reference solution" obtained on a very fine spatio-temporal grid.

Oberservations: for either numerical flux function

- (near) first order algebraic convergence ($\rightarrow$ Def. 1.6.24) w.r.t. mesh width h in err_1,
- algebraic convergence w.r.t. mesh width h in err_∞ *before* the solution develops discontinuities (shocks),
- no covergence in norm err_∞ after shock formation.

Best we get: merely first order algebraic convergence $O(h)$

Heuristic explanation for limited order:

$u = u(x,t) \hat{=}$ *smooth* entropy solution of Cauchy problem

$$\frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) = 0 \quad \text{in } \mathbb{R}\times]0,T[\quad , \quad u(x,0) = u_0(x) \ , \quad x \in \mathbb{R} \ . \tag{8.2.7}$$

We study the so-called consistency error of the numerical flux $F = F(v,w)$

$$(\vec{\tau}_F(t))_j = F(u(x_j,t), u(x_{j+1},t)) - f(u(x_{j+1/2},t)) \ , j \in \mathbb{Z} \quad ,$$

which measures the deviation of the approximate flux and the true flux, when the approximate solution agreed with the exact solution at the nodes of the mesh.

What we are interested in

behavior of $(\vec{\tau}_F(t))_j$ as mesh width $h \to 0$,

where an equidistant spatial mesh is assumed. Terminology:

$$\max_{j\in\mathbb{Z}} (\vec{\tau}_F(t))_j = O(h^q) \quad \text{for } h \to 0 \quad \leftrightarrow \quad \text{numerical flux consistent of order } q \in \mathbb{N} \ . \tag{8.4.50}$$

Rule of thumb: Order of consistency of numerical flux function limits (algebraic) order of convergence of (semi-discrete and fully discrete) finite volume schemes.

Example 8.4.51 (Consistency error of upwind numerical flux)

Assumption: f continuously differentiable $u_0 \geq 0$ and $f'(u) \geq 0$ for $u \geq 0$ ➢ no transsonic rarefactions!

In this case the upwind numerical flux (8.3.39) agrees with the Godunov flux (8.3.53), see Rem. 8.3.54 and

$$F_{\mathrm{uw}}(u(x_j,t), u(x_{j+1},t)) = f(u(x_j),t) \ , \quad j \in \mathbb{Z} \ .$$

$$\begin{aligned}
(\vec{\tau}_{F_{\mathrm{uw}}}(t))_j &= f(u(x_j,t)) - f(u(x_{j+1/2},t)) \\
&= f'(u(x_{j+1/2},t))(u(x_j,t) - u(x_{j+1/2},t)) + O(|u(x_j,t) - u(x_{j+1/2},t)|^2) \\
&= -f'(u(x_{j+1/2},t)) \frac{\partial u}{\partial x}(x_{j+1/2},t) \tfrac{1}{2} h + O(h^2) \quad \text{for } h \to 0 \ ,
\end{aligned}$$

by *Taylor expansion* of f and u.

This means that the upwind/Godunov numerical flux is (only) *first order consistent*.

Example 8.4.52 (Consistency error of Lax-Friedrichs/Rusanov numerical flux)

Assumption: smooth flux function

Recall: The (local) Lax-Friedrichs numerical flux

$$F_{\mathrm{LF}}(v,w) = \tfrac{1}{2}(f(v)+f(w)) - \tfrac{1}{2}\max_{\min\{v,w\}\leq u\leq \max\{v,w\}} |f'(u)|(w-v)\,, \tag{8.3.36}$$

is composed of the central flux and a diffusive flux.

We examine the consistency error for both parts separately, using Taylor expansion:

❶ central flux:

$$\begin{aligned}
&\tfrac{1}{2}(f(u(x_j,t)) + f(u(x_{j+1},t))) - f(u(x_{j+1/2},t)) \\
&\qquad = \tfrac{1}{2}f'(u(x_{j+1/2},t))(u(x_j,t) - u(x_{j+1/2},t) + u(x_{j+1},t) - u(x_{j+1/2},t)) + O(h^2) \qquad (8.4.53)\\
&\qquad = \tfrac{1}{2}f'(u(x_{j+1/2},t))\big(\frac{\partial u}{\partial x}(x_{j+1/2},t)(-\tfrac{1}{2}h + \tfrac{1}{2}h) + O(h^2)\big) + O(h^2)\\
&\qquad = O(h^2) \quad \text{for } h \to 0\,.
\end{aligned}$$

➢ The central flux is *second order consistent*.

However, due to instability the central flux on its own is useless, see Section 8.3.3.1.

❷ diffusive flux part:

$$u(x_{j+1},t) - u(x_j,t) = \frac{\partial u}{\partial x}(x_{j+1/2},t)h + O(h^2) \quad \text{for } h \to 0\,.$$

$$\Downarrow$$

$$F_{\mathrm{LF}}(u(x_j,t),u(x_{j+1},t)) - f(u(x_{j+1/2},t)) = O(h) \quad \text{for } h \to 0\,,$$

that is the Lax-Friedrichs/Rusanov numerical flux is only first order consistent, because the consistency error is dominated by the diffusive flux, which is necessary for the sake of stability.

The observations made in the above examples are linked to a general fact:

Order barrier for monotone numerical fluxes

Monotone numerical fluxes (→ Def. 8.3.61) are at most first order consistent.

?! *Review question(s) 8.4.55.* (Timestepping for semi-discrete conservation laws)

1. Conduct a vonNeumann stability analysis for the linear evolution

$$\dot{\mu}_j = \frac{\mu_{j+1} - 2\mu_j + \mu_{j-1}}{h^2}\,, \quad h > 0\,, \tag{8.4.56}$$

when explicit Euler timestepping/the explicit trapezoidal rule is used for discretization in time.

8.5 Higher-order conservative schemes

Formally, high-order conservative finite volume methods are distinguished by numerical flux functions that are consistent of order ≥ 2, see (8.4.50).

However, solutions of (systems of) conservation laws will usually not even be continuous (because of shocks emerging even in the case of smooth u_0, see (8.2.14)), let alone smooth, so that the formal order of consistency may not have any bearing for the (rate of) convergence observed for the method for a concrete Cauchy problem.

Therefore in the field of numerics of conservation laws "high-order" is desired not so much for the promise of higher rates of convergence, but for the following advantages:

- for the same spatial resolution. high-order methods frequently provide more accurate solutions in the sense of global error norms as first-order methods,
- high-order methods often provide *better resolution of local features* of the solution (shocks, etc.).

In standard semi-discrete finite volume schemes in conservation form for 2-point numerical flux function,

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(\mu_j(t),\mu_{j+1}(t)) - F(\mu_{j-1}(t),\mu_j(t))\big)\ , \quad j \in \mathbb{Z}\ , \tag{8.3.10}$$

the numerical flux function is evaluated for the cell averages μ_j, which can be read as approximate values of a projection of the exact solution onto piecewise constant functions (on dual cells)

$$\mu_j(t) \approx \frac{1}{h}\int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\,\mathrm{d}x\ . \tag{8.3.4}$$

By Taylor expansion we find for $u \in C^1$

$$u(x_{j+1/2},t) - \frac{1}{h}\int_{x_{j-1/2}}^{x_{j+1/2}} u(x,t)\,\mathrm{d}x = O(h) \quad \text{for } h \to 0\ ,$$

and, unless some lucky cancellation occurs as in the case of the central flux, see Ex. 8.4.52, this does not allow more than first order consistency.

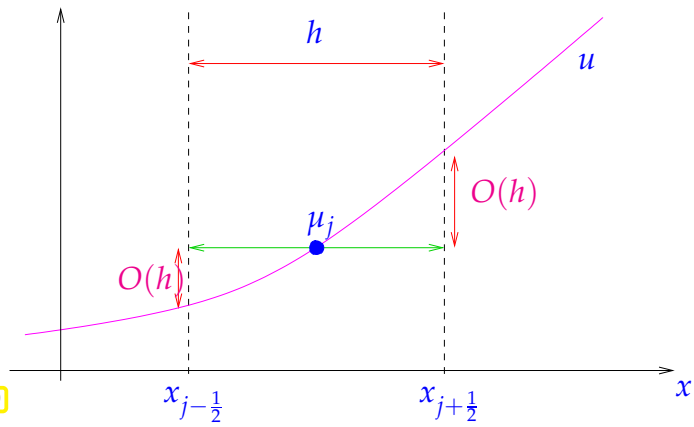
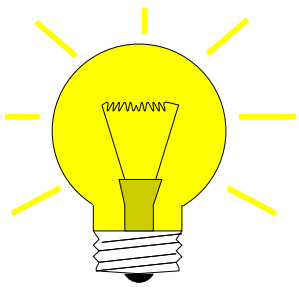


Fig. 449

8.5.1 Piecewise linear reconstruction



Idea: Plug "better" approximations of $u(x_{j\pm 1/2},t)$ into numerical flux function in (8.3.10)

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\big(F(v_j^+(t),v_{j+1}^-(t)) - F(v_{j-1}^+(t),v_j^-(t))\big)\ , \quad j\in\mathbb{Z}\ , \tag{8.5.1}$$

where $v_j^\pm$ are obtained by piecewise linear reconstruction from the (dual) cell values μ_j.

$$\begin{aligned} v_j^-(t) &:= \mu_j(t) - \tfrac{1}{2}h\sigma_j(t)\ , \\ v_j^+(t) &:= \mu_j(t) + \tfrac{1}{2}h\sigma_j(t)\ , \end{aligned} \quad j\in\mathbb{Z}\ , \tag{8.5.2}$$

with suitable slopes $\sigma_j(t) = \sigma(\vec{\mu}(t))$.

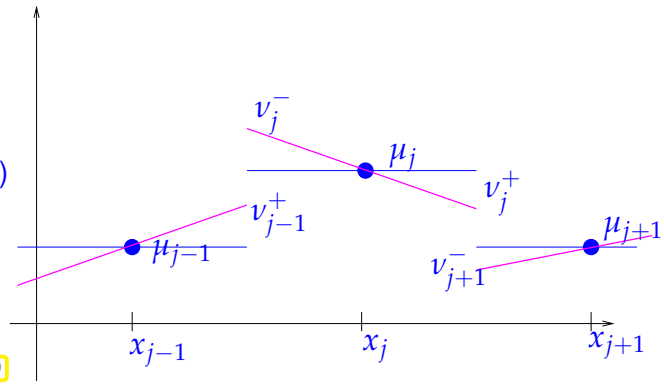


Fig. 450

Analogy: piecewise cubic Hermite interpolation with reconstructed slopes, discussed in the context of shape preserving interpolation in [33, Section 3.4.2]. However, we do not aim for smooth functions now.

Definition 8.5.3. Linear reconstruction

Given an (infinite) mesh $\mathcal{M} := \{]x_{j-1},x_j[\}_{j\in\mathbb{Z}}$ ($x_{j-1} < x_j$), a linear reconstruction operator $\mathsf{R}_\mathcal{M}$ is a mapping

$$\mathsf{R}_\mathcal{M} : \mathbb{R}^\mathbb{Z} \mapsto \{v\in L^\infty(\mathbb{R}) \colon v \text{ linear on }]x_{j-1/2},x_{j+1/2}[\ \forall j\in\mathbb{Z}\}\ ,$$

taking a sequence $\vec{\mu}\in\mathbb{R}^\mathbb{Z}$ of cell averages to a possibly *discontinuous* function $\mathsf{R}_\mathcal{M}\vec{\mu}$ that is piecewise linear on dual cells.

Linear reconstruction & (8.5.1) ➢ semi-discrete evolution in conservation form, *cf.* (8.3.9)

For 2-point numerical flux $F = F(u,w)$

$$\frac{d\mu_j}{dt}(t) = -\frac{1}{h}\Big(F(v_j^+(t),v_{j+1}^-(t)) - F(v_{j-1}^+(t),v_j^-(t))\Big)\ , \quad j\in\mathbb{Z}\ . \tag{8.5.4}$$

MATLAB Code 8.5.5: Conservative FV with linear reconstruction: `ode45` timestepping

```
function ufinal = highresevl(a,b,N,u0,T,F,slopes)
% finite volume discrete evolution in conservation form with linear
% reconstruction, see (8.5.4)
% Cauchy problem over time [0,T] restricted to finite interval [a,b],
% equidistant mesh with meshwidth N cells, meshwidth h := (b-a)/N.
% 2-point numerical flux function F = F(v,w) passed in handle F
% 3-point slope recostruction rule passed as handle slopes = @(v,u,w)
% ...
% (Note: no division by h must be done in slope computation)
% returns cell averages for approximate solution at final time in a row
```

```
    vector
h = (b-a)/N; x = a+0.5*h:h:b-0.5*h; % cell centers
mu0 = h*u0(x)';  % vector of initial cell averages (column vector)
% right hand side function for MATLAB ode solvers
odefun = @(t,mu) (-1/h*fluxdiff(h,mu,F,slopes));
% timestepping by explicity Runge-Kutta method of order 5
options = odeset('abstol',1E-8,'reltol',1E-6,'stats','on');
[t,MU] = ode45(odefun,[0 T],mu0,options);
% Graphical output
[X,T] = meshgrid(x,t);
figure; surf(X,T,MU/h); colormap(copper);
xlabel('{\bf x}','fontsize',14);
ylabel('{\bf t}','fontsize',14);
zlabel('{\bf u}','fontsize',14);
ufinal = MU(end,:)';
end
```

C++11 EIGEN code 8.5.6: Conservative FV with linear reconstruction: ode45 timestepping
➙ GITLAB

```
// Arguments:
// real numbers a,b.  a < b, the boundaries of the domain
// unsigned int N the number of grid cells
// Functor u0 : ℝ ↦ ℝ:  initial data
// real number T > 0:  final time
// Functor F = F(v,w):  2-point numerical flux function
// Functor slopes = σ(v,u,w):  3-point slope recostruction rule
// (Note:  no division by h needs to be done in slopt computation
//
// returns:
// Vector of cell averages at final time
//
// finite volume discrete evolution in conservation form with linear
// reconstruction, see (8.5.4).
// Cauchy problem over time [0,T], equidistant mesh
template<typename FunctionU0, typename FunctionF,
      typename FnSlopes>
Eigen::VectorXd highresevl(double a, double b, unsigned N,
                  FunctionU0 u0, double T, FunctionF F,
                  FnSlopes slopes) {
  double h = (b-a)/N; // mesh width
  // positions of grid points
  Eigen::VectorXd x = Eigen::VectorXd::LinSpaced(N,a+0.5*h,b-0.5*h);
  // vector of initial cell averages (column vector) from sampling u_0
  Eigen::VectorXd mu0 = h*x.unaryExpr(u0);

  // right hand side lambda function for ODE solver
  auto odefun = [&] (const Eigen::VectorXd& mu,
                Eigen::VectorXd& dmdt, double t) {
    dmdt = -1./h*fluxdiff<FunctionF,FnSlopes>(h, mu, F, slopes); };
```

```

  // timestepping by explicit adaptive Runge-Kutta single-step
  // method of order 5.  Adaptivity control according to [33, Section 11.5]
  double abstol = 1E-8, reltol = 1E-6;
  Eigen::VectorXd t;  // Temporal adaptive integration mesh
  Eigen::MatrixXd MU; // State vectors $\vec{\mu}^{(k)}$
  std::tie(t, MU) = NPDE::ode45(odefun, 0, T, mu0, abstol, reltol);
  // Extract state vector for final time
  return MU.col(t.size()-1);
}
```

MATLAB Code 8.5.7: Operator $\mathcal{L}_h$ for spatial semidiscretization with conservative FV with linear reconstruction and 2-point numerical flux

```
function fd = fluxdiff(h,mu,F,slopes)
% MATLAB function that realizes the right hand side operator $\mathcal{L}_h$ for the ODE
% (8.4.3) arising from conservative finite volume semidiscretization of the
% Cauchy problem for a 1D scalar conservation law (8.2.7).
% h:  meshwidth of equidistant spatial grid
% mu:  (finite) vector $\vec{\mu}$ of cell averages
% F: handle to 2-point numerical flux function $F = F(v,w)$
% slope:  handle to slope function $\sigma_j = \mathrm{slopes}(\mu_{j-1},\mu_j,\mu_{j+1})$
n = length(mu); sigma = zeros(n,1); fd = zeros(n,1);
% Computation of slopes $\sigma_j$, uses $\mu_0 = \mu_1$,
% $m_{N+1} = \mu_N$, which amounts to constant extension of state beyond domain of
% influence $[a,b]$ of non-constant intial data.
sigma(1) = slopes(mu(1),mu(1),mu(2));
for j=2:n-1, sigma(j) = slopes(mu(j-1),mu(j),mu(j+1)); end
sigma(n) = slopes(mu(n-1),mu(n),mu(n));
% Compute linear reconstruction at endpoints of dual cells (8.5.2)
nup = mu+0.5*sigma; % $\nu_j^+$ at right endpoint
num = mu-0.5*sigma; % $\nu_j^-$ at left endpoint
% As in Code 8.3.11:  constant continuation of data outside $[a,b]$ !
fd(1) = F(nup(1),num(2)) - F(mu(1),num(1));
for j=2:n-1
  fd(j) = F(nup(j),num(j+1)) - F(nup(j-1),num(j)); % see (8.5.4)
end
fd(n) = F(nup(n),mu(n)) - F(nup(n-1),num(n));
end
```

C++11 EIGEN 8.5.8: Operator $\mathcal{L}_h$ for spatial semidiscretization with conservative FV with linear reconstruction and 2-point numerical flux ➙ GITLAB

```
// arguments:
// double texttth:  meshwidth of equidistant spatial grid
// Vector textttmu:  (finite) vector $\vec{\mu}$ of cell averages
```

```
// Functor textttF: handle to 2-point numerical flux function F = F(v,w)
// Functor textttslope:  handle to slope function σ_j = slopes(μ_{j-1}, μ_j, μ_{j+1})
//
// returns:
// Vector with differences of numerical fluxes
//
// Function that realizes the right hand side operator L_h for the ODE
// (8.4.3) arising from conservative finite volume semidiscretization of
// the Cauchy problem for a 1D scalar conservation law (8.2.7).
template<typename FunctionF, typename FunctionSlopes>
Eigen::VectorXd fluxdiff(double h, const Eigen::VectorXd& mu,
                         FunctionF F, FunctionSlopes slopes) {
  unsigned n = mu.size(); // Number of grid cells
  Eigen::VectorXd sigma = Eigen::VectorXd::Zero(n); // Vector of slopes
  Eigen::VectorXd fd = Eigen::VectorXd::Zero(n);

  // 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 Code 8.3.12
  sigma[0] = slopes(mu[0], mu[0], mu[1]);
  for (unsigned 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 (8.5.2)
  Eigen::VectorXd nup = mu + 0.5*sigma;
  Eigen::VectorXd num = mu - 0.5*sigma;

  // As in Code 8.3.11:  constant continuation of data outside [a,b] !
  fd[0] = F(nup[0], num[1]) - F(mu[0], num[0]);
  for (unsigned j=1; j < n-1; ++j)
    fd[j] = F(nup[j], num[j+1]) - F(nup[j-1], num[j]); // see (8.3.10)
  fd[n-1] = F(nup[n-1], mu[n-1]) - F(nup[n-2], num[n-1]);

  return fd;
}
```

"Natural" choice: central slope (averaged slope)

$$\sigma_j(t) = \frac{1}{2}\left(\frac{\mu_{j+1}(t)-\mu_j(t)}{h} + \frac{\mu_j(t)-\mu_{j-1}(t)}{h}\right) = \frac{1}{2}\frac{\mu_{j+1}(t)-\mu_{j-1}(t)}{h} . \tag{8.5.9}$$

By Taylor expansion: for $u \in C^2$ (that is, u *sufficiently smooth*), central slope (8.5.9), $v_j^{\pm}$ according to (8.5.2)

$$|v_j^-(t) - u(x_{j-1/2},t)|, |v_j^+(t) - u(x_{j+1/2},t)| = O(h^2) .$$

Example 8.5.10 (Convergence of FV with linear reconstruction)

- Cauchy problem for Burgers equation (8.1.46) (flux function $f(u) = \frac{1}{2}u^2$) from Ex. 8.2.39 with C^1 bump initial data (BUMP)
- Equidistant spatial mesh with meshwidth $h =$
- Linear reconstruction with central slope (8.5.9)
- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- 2n-order Runge-Kutta timestepping (method of Heun), timestep $\tau = 0.5h$ ("$\mathrm{CFL} = 0.5$")

Monitored: Approximate L^1- and L^∞-norms of error at final time $T = 0.3$ (exact solution still *smooth* at this time, see Ex. 8.4.47)

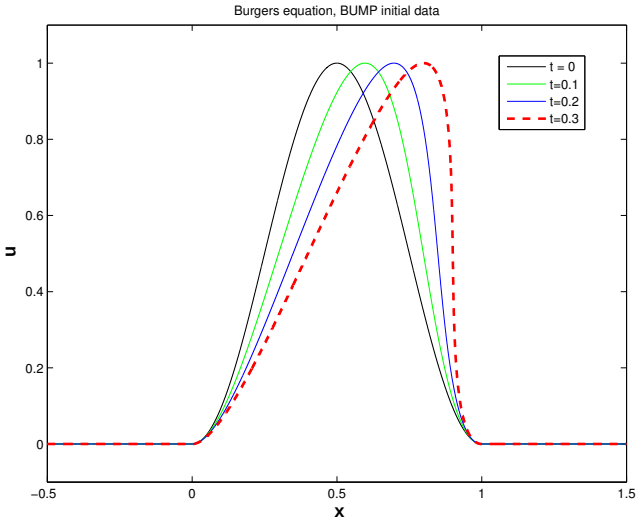


Fig. 451

◁ "exact solution"

computed by means of a high-order finite volume method (WENO) on a equidistant mesh with 2^{14} points., U. Fjordholm (SAM)

Observation: 2nd-order convergence in both norms

Example 8.5.11 (Linear reconstruction with central slope (Burgers' equation))

Cauchy problem of Ex. 8.3.22:

- Cauchy problem for Burgers equation (8.1.46) (flux function $f(u) = \frac{1}{2}u^2$) from Ex. 8.2.39 ("box" intial data)
- Equidistant spatial mesh with meshwidth $h =$
- Linear reconstruction with central slope (8.5.9)
- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- timestepping based on adaptive Runge-Kutta method `ode45` of MATLAB (`opts = odeset('abstol',1E-7,'reltol',1E-6);`).

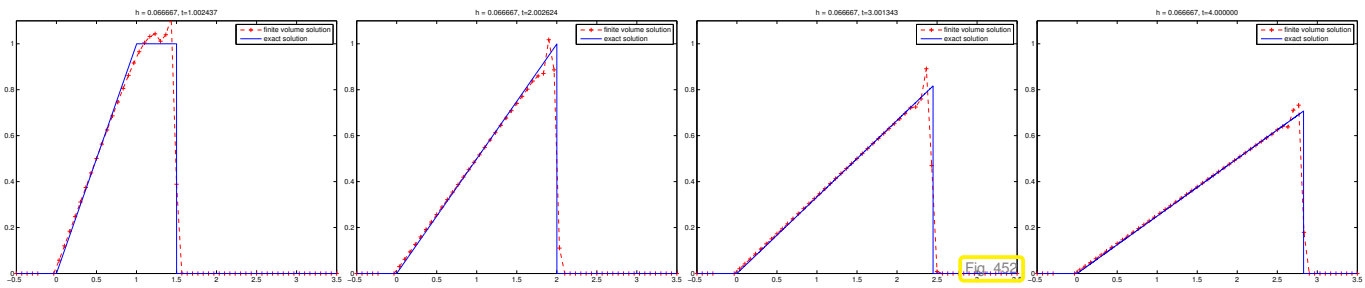
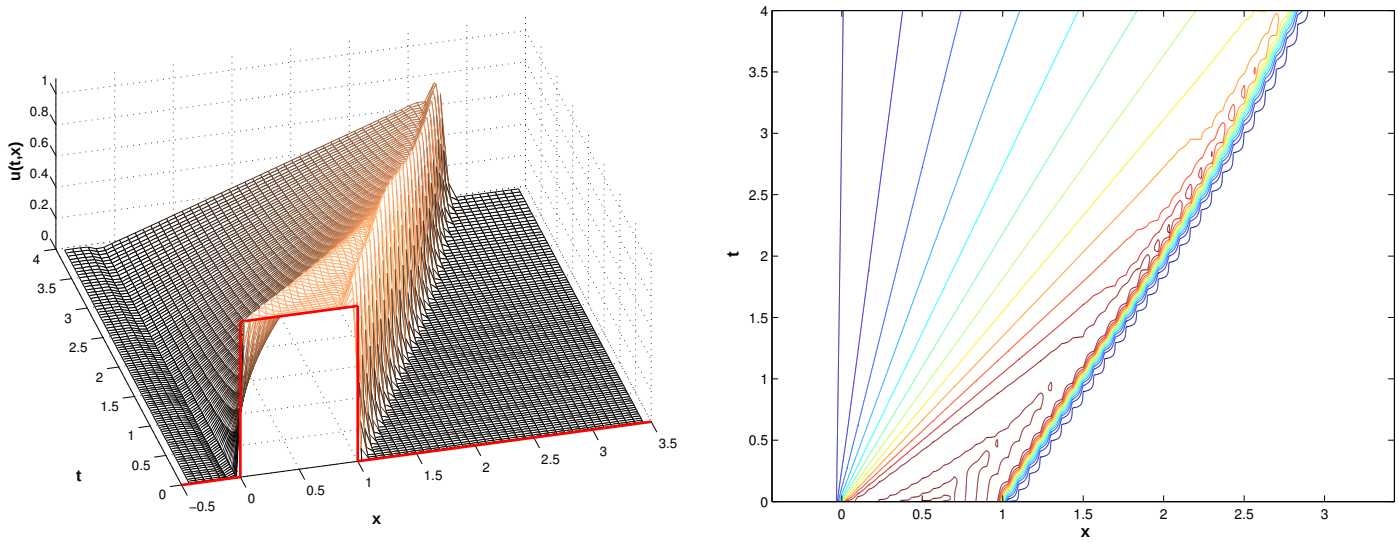


Fig. 452



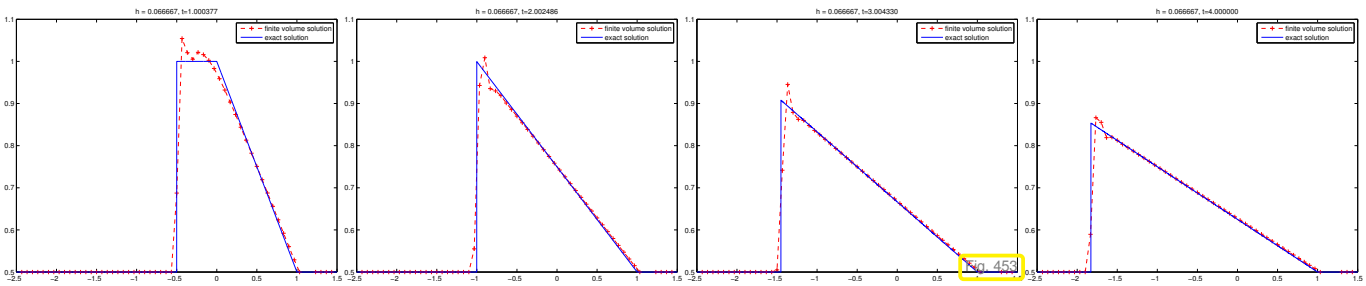
Emergence of spurious oscillations in the vicinity of shock (in violation of structural properties of the exact solution, see (8.2.42).)

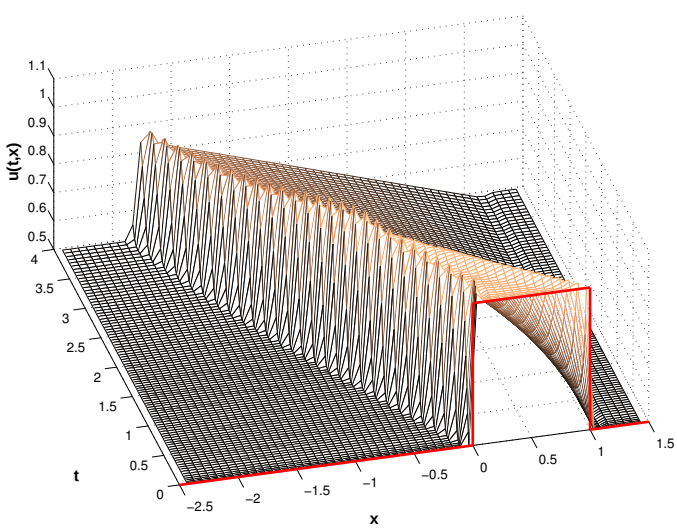
Compare: Oscillations occurring in FV schemes relying on central flux, see Ex. 8.3.22.

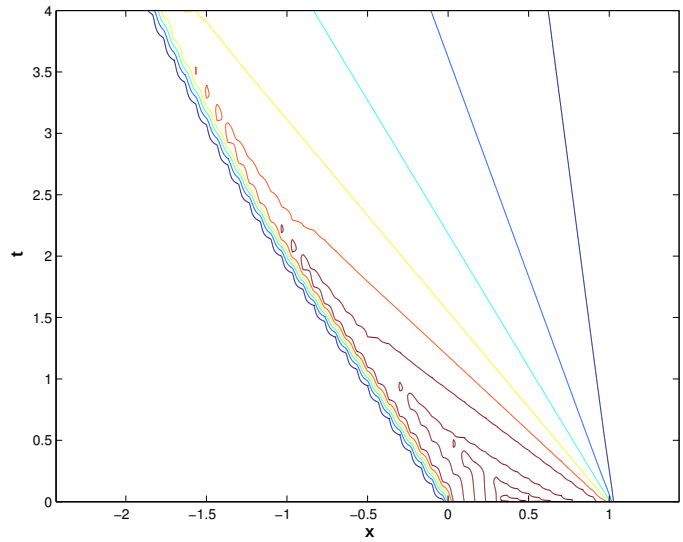
Example 8.5.12 (Linear reconstruction with central slope (traffic flow))

Cauchy problem of Ex. 8.3.23:

- Cauchy problem for Traffic Flow equation (8.1.41) (flux function $f(u) = u(1-u)$) from Ex. 8.2.40 ("box" intial data)
- Equidistant spatial mesh with meshwidth $h =$
- Linear reconstruction with central slope (8.5.9)
- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- timestepping based on adaptive Runge-Kutta method `ode45` of MATLAB (`opts = odeset('abstol',1E-7,'reltol',1E-6);`).







Emergence of spurious oscillations in the vicinity of shock (in violation of structural properties of the exact solution, see (8.2.42).)

Compare: Oscillations occurring in FV schemes relying on central flux, see Ex. 8.3.23.

In Ex. 8.3.22, 7.2.20, the spurious oscillations can be blamed on the unstable central flux/central finite differences. Maybe, this time the central slope formula is the culprit. Thus, we investigate slope reconstruction connected with backward and forward difference quotients.

Example 8.5.13 (Linear reconstruction with one-sided slopes (Burgers' equation))

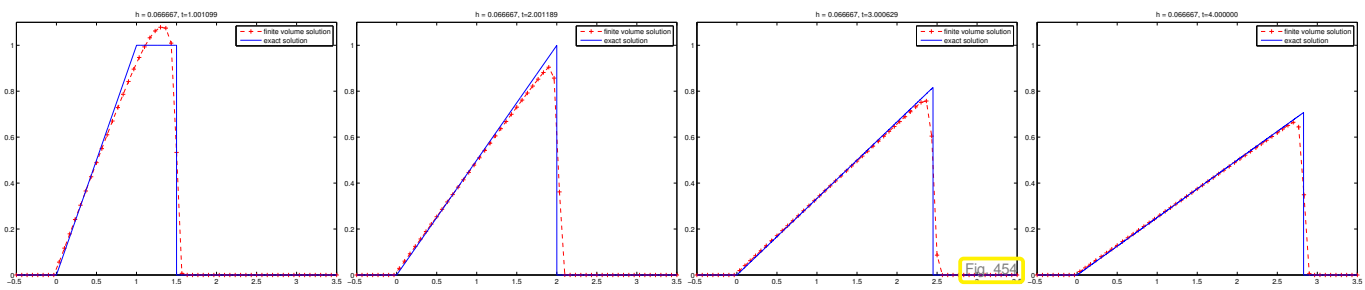
One-sided slopes for use in (8.5.2)

$$\text{Right slope:} \quad \sigma_j(t) = \frac{\mu_{j+1}(t) - \mu_j(t)}{h}\,, \tag{8.5.14}$$

$$\text{Left slope:} \quad \sigma_j(t) = \frac{\mu_j(t) - \mu_{j-1}(t)}{h}\,. \tag{8.5.15}$$

Same setting as in Ex. 8.5.11, with central slope replaced with one-sided slopes.

Left slope:



Right slope:

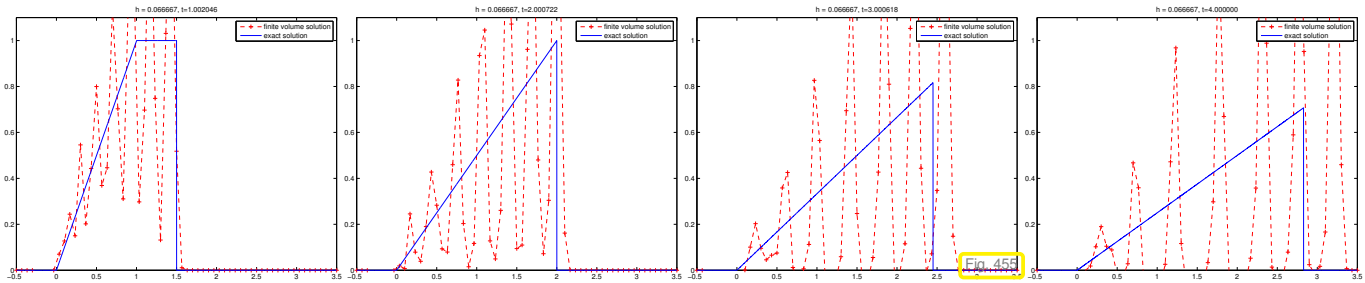


Fig. 455

Observation: spurious oscillations/overshoots, massive and global for (8.5.14), moderate close to shock for (8.5.15).

Example 8.5.16 (Linear reconstruction with one-sided slopes (traffic flow))

Left slope:

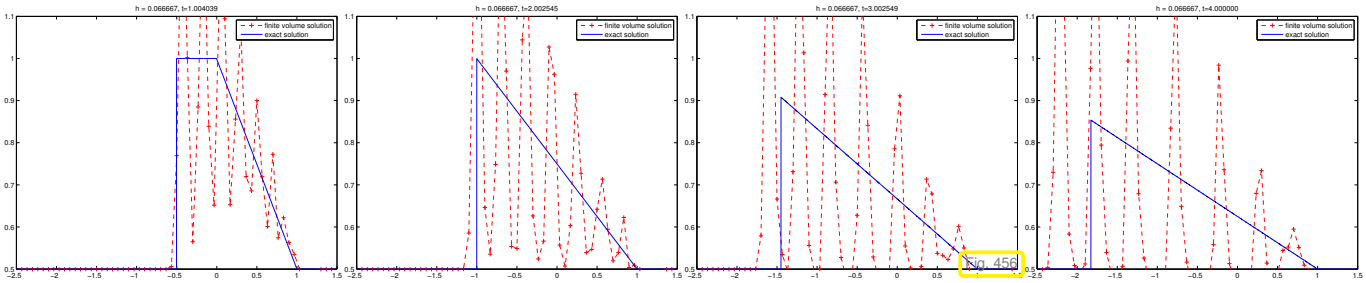


Fig. 456

Right slope:

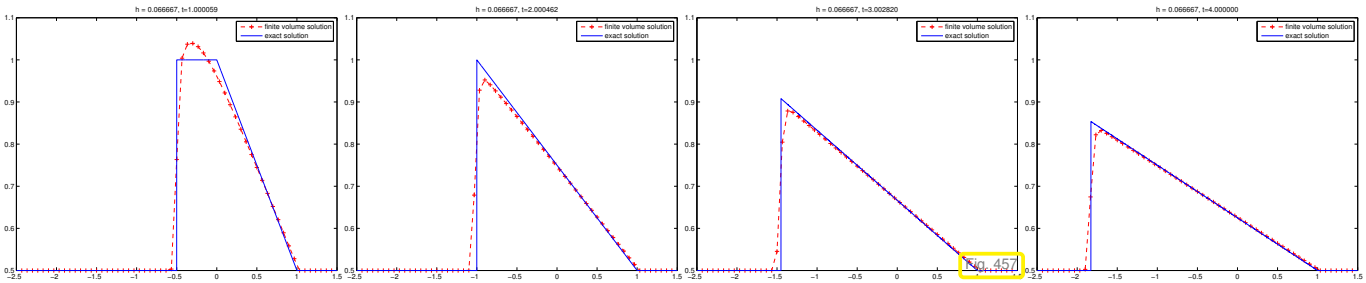


Fig. 457

Observation: spurious oscillations/overshoots, massive and global for (8.5.14), moderate close to shock for (8.5.15).

It seems to be the very process of linear reconstruction that triggers oscillations near shocks. These oscillations can be traced back to "overshooting" of linear reconstruction at jumps.

Example 8.5.17 (Over-/Undershoots in linear reconstruction)

In this example we apply the slope formulas proposed above to particular "synthetic cell averages" derived from a function (blue graph in plots) featuring a jump, a kink, and a local maximum.

Slope from central differencing:

$$\sigma_j = \frac{1}{2h}(\mu_{j+1} - \mu_{j-1}) \; . \qquad (8.5.9)$$

(— $\hat{=}$ cell averages, — $\hat{=}$ piecewise linear reconstruction)

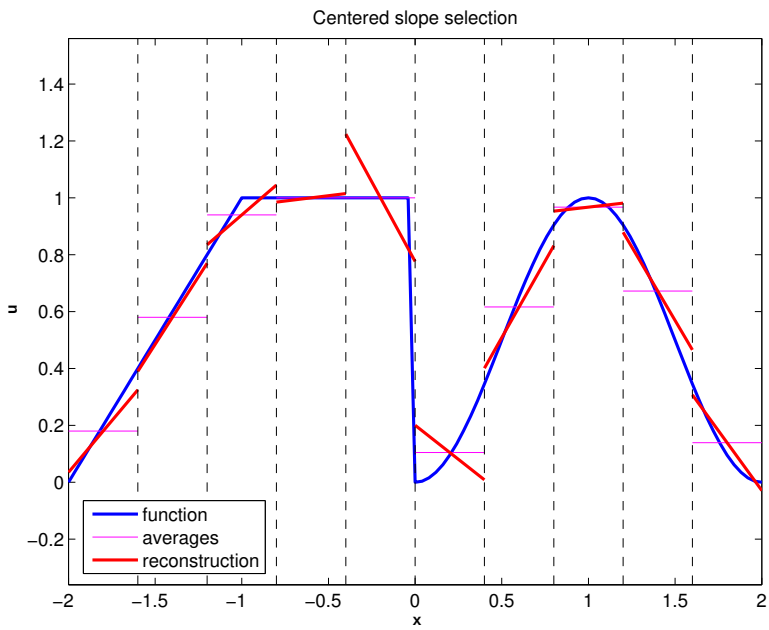


Fig. 458

Slope from forward differencing:

$$\sigma_j = \frac{1}{h}(\mu_{j+1} - \mu_j) \; . \qquad (8.5.14)$$

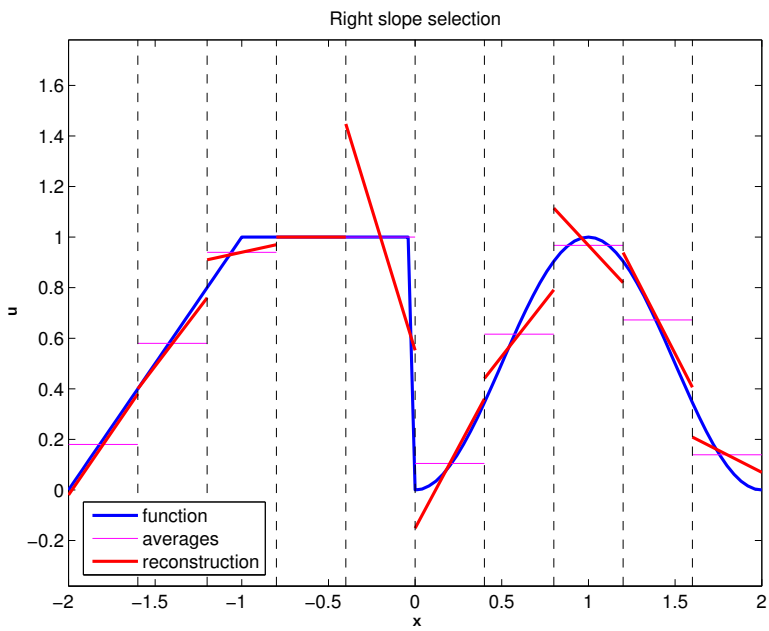


Fig. 459

Slope from backward differencing:

$$\sigma_j = \frac{1}{h}(\mu_j - \mu_{j-1}) \; . \qquad (8.5.15)$$

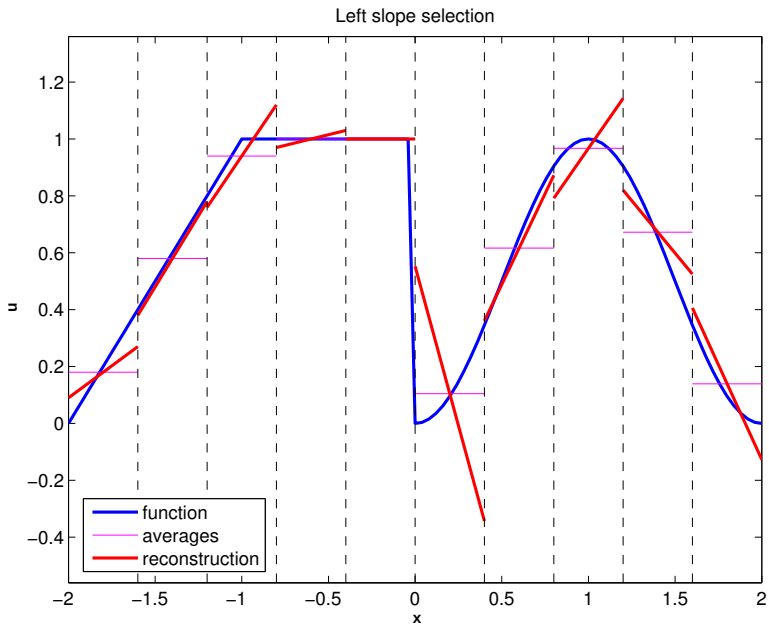


Fig. 460

We observe that the piecewise linear reconstruction develops over- and undershoots regardless of the slope formula used.

8.5.2 Slope limiting

We want to find a piecewise linear reconstruction method with a guarantee for the bsuppression of "over-/undershoots" (→ Fig. 458, Fig. 459, Fig. 460).

Use local monotonicity preservation of linear reconstruction

Definition 8.5.18. Monotonicity preserving linear interpolation

An linear reconstruction operator $\mathsf{R}_{\mathcal{M}}$ (→ Def. 8.5.3) is monotonicity preserving, if

$$(\mathsf{R}_{\mathcal{M}}\vec{\mu})(x_j) = \mu_j \quad \wedge \quad \begin{array}{lll} \mu_j \leq \mu_{j+1} & \Rightarrow & \mathsf{R}_{\mathcal{M}}\vec{\mu} \text{ non-decreasing in }]x_j, x_{j+1}[\,, \\ \mu_j \geq \mu_{j+1} & \Rightarrow & \mathsf{R}_{\mathcal{M}}\vec{\mu} \text{ non-increasing in }]x_j, x_{j+1}[\,. \end{array}$$

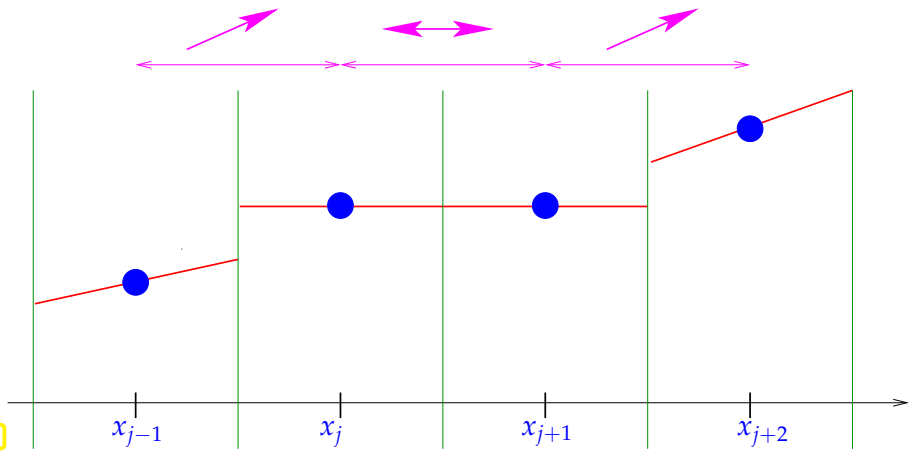


Fig. 461

Monotonicity preserving linear reconstruction:

- constant at plateaus
- constant at (local) extrema

(Magenta arrows ≙ admissible slope of p.w. linear reconstruction)

Related: shape preserving Hermite interpolation, see [33, Section 3.4.2], achieved by using

- zero slope, in case of local slopes with opposite sign, see [33, Eq. (3.4.12)],
- harmonic averaging of local slopes, see [33, Eq. (3.4.14)].

Remark 8.5.19 (Consequence of monotonicity preservation)

A monotonicity preserving linear reconstruction operator $\mathsf{R}_{\mathcal{M}}$ (→ Def. 8.5.18)

- respects the range of cell averages

$$\min\{\mu_k \mu_{k+1}, \dots, \mu_m\} \leq (\mathsf{R}\vec{\mu})(x) \leq \max\{\mu_k, \mu_{k+1}, \dots, \mu_m\}\,, \quad x_k < x < x_m\,. \tag{8.5.20}$$

  ↔ "range preservation" by entropy solutions, see Thm. 8.2.41.

- does not allow the creation of new extrema

$$\sharp\{\text{extrema of } \mathsf{R}_{\mathcal{M}}\vec{\mu}\} \leq \sharp\{\text{extrema of } \vec{\mu}\}\,. \tag{8.5.21}$$

  ↔ preservation of number of extrema in entropy solution, Sect. 8.2.7.

Remark 8.5.22 (Linearity and monotonicity preservation)

The linear reconstruction operators (→ Def. 8.5.3) based on the slope formulas (8.5.9) (central slope), (8.5.14) (forward slope), (8.5.15) (backward slope) are *linear* in the sense that

$$\mathsf{R}_{\mathcal{M}}(\alpha\vec{\mu}+\beta\vec{v}) = \alpha\mathsf{R}_{\mathcal{M}}(\vec{\mu})+\beta\mathsf{R}_{\mathcal{M}}\vec{v}) \quad \forall\vec{\mu},\vec{v}\in\mathbb{R}^{\mathbb{Z}},\ \alpha,\beta\in\mathbb{R}\ . \tag{8.5.23}$$

Lemma 8.5.24. Linear monotonicity preserving reconstruction trivial

Every linear*, monotonicity preserving (→ Def. 8.5.18) linear reconstruction yields piecewise constant functions.*

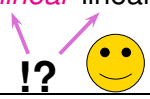
Proof. Define $\vec{\epsilon}^k \in \mathbb{R}^{\mathbb{Z}}$, $k\in\mathbb{Z}$, by

$$\epsilon_j^k = \begin{cases} 1 & \text{for } k=j\ , \\ 0 & \text{else.} \end{cases}$$

The $\vec{\epsilon}^k$ form a basis of $\mathbb{R}^{\mathbb{Z}}$. Thus, due to linearity, $\mathsf{R}_{\mathcal{M}}$ is fixed by its action on the basis vectors $\vec{\epsilon}^k$ and its image is spanned by $\left\{\mathsf{R}_{\mathcal{M}}\vec{\epsilon}^k\right\}_{k\in\mathbb{Z}}$.

However, monotonicity preservation entails that $\mathsf{R}_{\mathcal{M}}\vec{\epsilon}^k$ is piecewise constant, see Fig. 461. □

Necessary (for monotonicity preservation): *Non-linear* linear reconstruction



A simple consideration, see Fig. 461

$$\mu_{j-1}\le\mu_j \quad\text{and}\quad \mu_j\ge\mu_{j+1} \quad\Rightarrow\quad \mathsf{R}_{\mathcal{M}}\vec{\mu}\equiv\text{const} \quad\text{on}\quad \left]x_{j-1/2},x_{j+1/2}\right[\ , \tag{8.5.25}$$

for any monotonicity preserving (→ Def. 8.5.18) linear reconstruction operator $\mathsf{R}_{\mathcal{M}}$ (→ Def. 8.5.3).

➢ monotonicity preserving linear reconstruction $\mathsf{R}_{\mathcal{M}}\vec{\mu}$ must be constant at local extrema of $\vec{\mu}$!

Definition 8.5.26. Minmod reconstruction

The minmod reconstruction R_{mm} is a piecewise linear reconstruction (→ Def. 8.5.3) defined by

$$(\mathsf{R}_{\mathrm{mm}}\vec{\mu})(x) = \mu_j+\sigma_j(x-x_j)$$
$$\text{for}\quad x_{j-1/2}<x<x_{j+1/2}\ , j\in\mathbb{Z}\ ,$$
$$\sigma_j := \mathrm{minmod}\left(\frac{\mu_{j+1}-\mu_j}{x_{j+1}-x_j},\frac{\mu_j-\mu_{j-1}}{x_j-x_{j-1}}\right)\ ,$$
$$\mathrm{minmod}(v,w) := \begin{cases} v & ,\ vw>0,\ |v|<|w|\ , \\ w & ,\ vw>0,\ |w|<|v|\ , \\ 0 & ,\ vw\le 0\ . \end{cases}$$

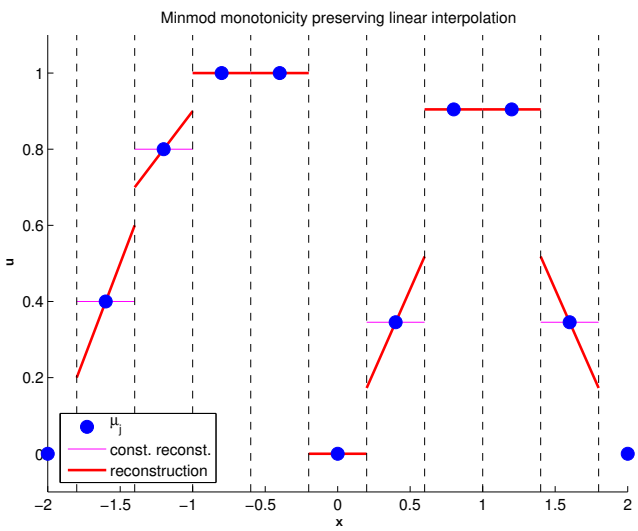


Fig. 462

Lemma 8.5.27. Monotonicity preservation of minmod reconstruction

Minmod reconstruction (→ Def. 8.5.26) is monotonicity preserving (→ Def. 8.5.18)

Proof. w.l.o.g. assume $\mu_{j+1} \geq \mu_j \Rightarrow \quad \sigma_j \geq 0 \quad \wedge \sigma_{j+1} \geq 0$
$\Rightarrow \quad \mu_j + \frac{1}{2}h\sigma_j \leq \frac{1}{2}(\mu_j + \mu_{j+1}) \leq \mu_{j+1} - \frac{1}{2}h\sigma_{j+1}$ □

Terminology: effect of minmod-function in R_{mm}: slope limiting: minmod = slope limiter

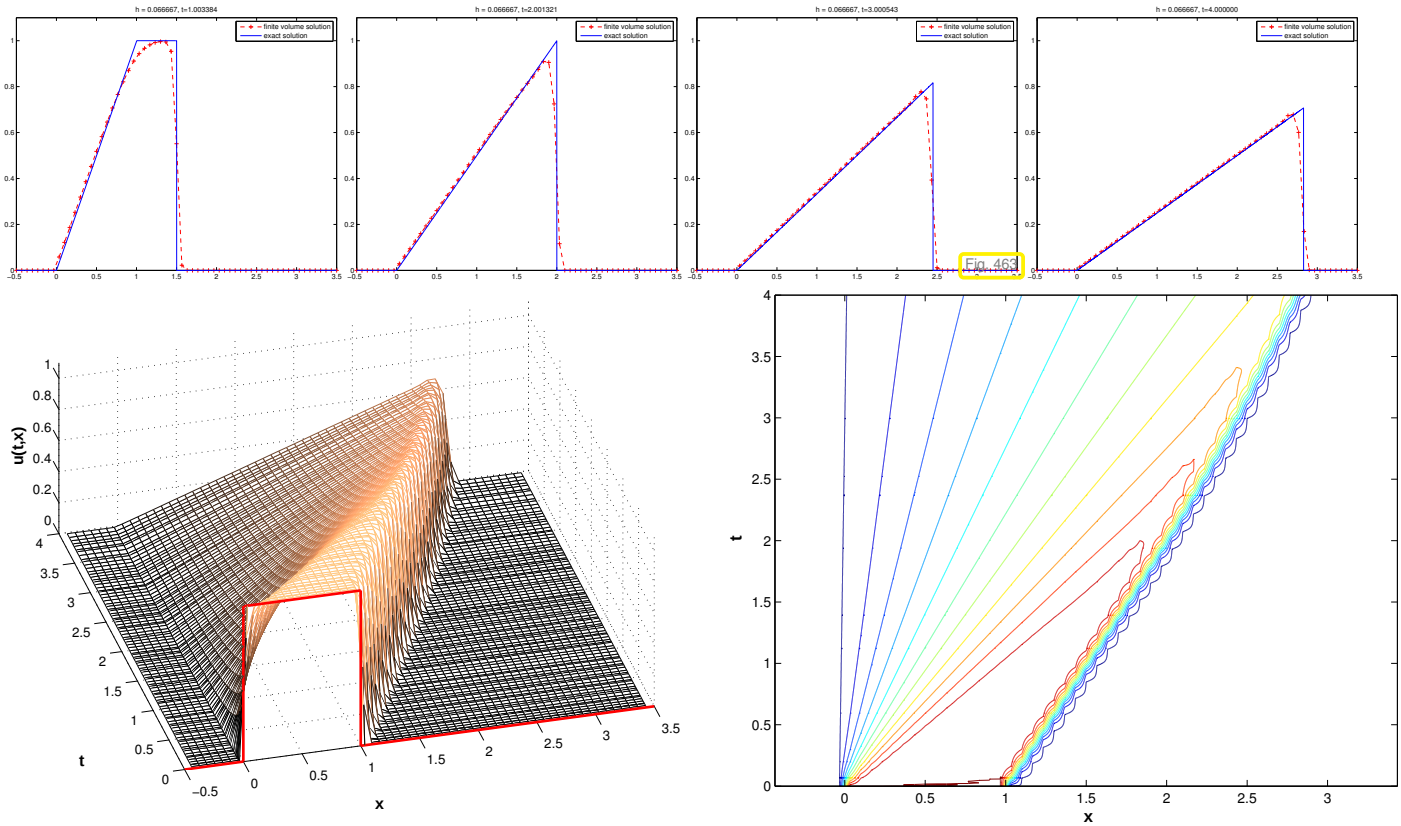
Example 8.5.28 (Linear reconstruction with minmod limiter (Burgers' equation))

Same setting as in Ex. 8.5.11, Cauchy problem as in Ex. 8.3.22:

- Cauchy problem for Burgers equation (8.1.46) (flux function $f(u) = \frac{1}{2}u^2$) from Ex. 8.2.39 ("box" intial data)
- Equidistant spatial mesh with meshwidth $h = \frac{1}{15}$
- Linear reconstruction with minmod limited slope (→ Def. 8.5.26)

$$\sigma_j := \mathrm{minmod}\left(\frac{\mu_j - \mu_{j-1}}{h}, \frac{\mu_{j+1} - \mu_j}{h}\right) .$$

- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- timestepping based on adaptive Runge-Kutta method `ode45` of MATLAB (`opts = odeset('abstol',1E-7,'reltol',1E-6);`).



Observation: spurious oscillations successfully suppressed!

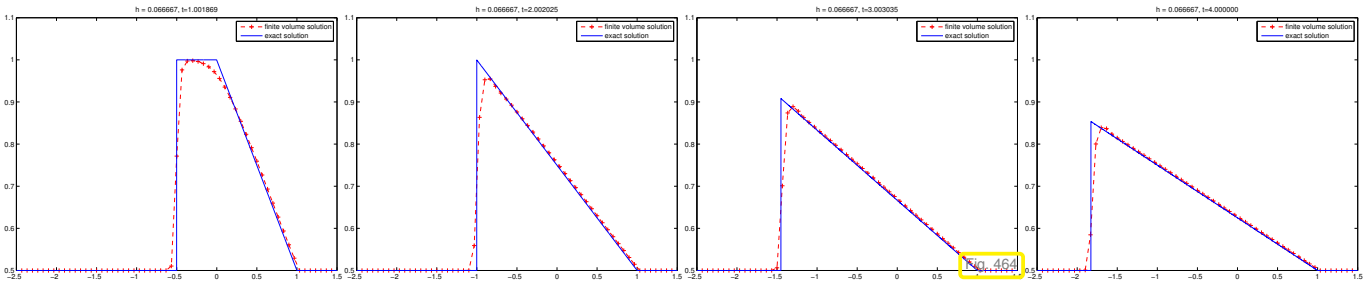
Example 8.5.29 (Linear reconstruction with minmod limiter)

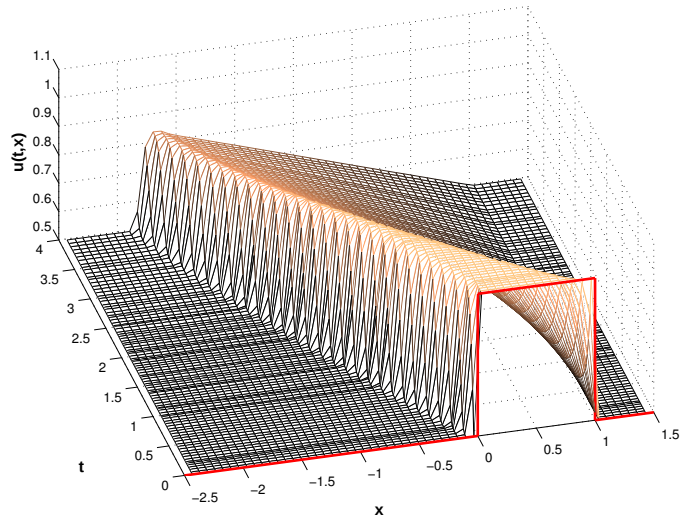
Same setting as in Ex. 8.5.11, Cauchy problem as in Ex. 8.3.23:

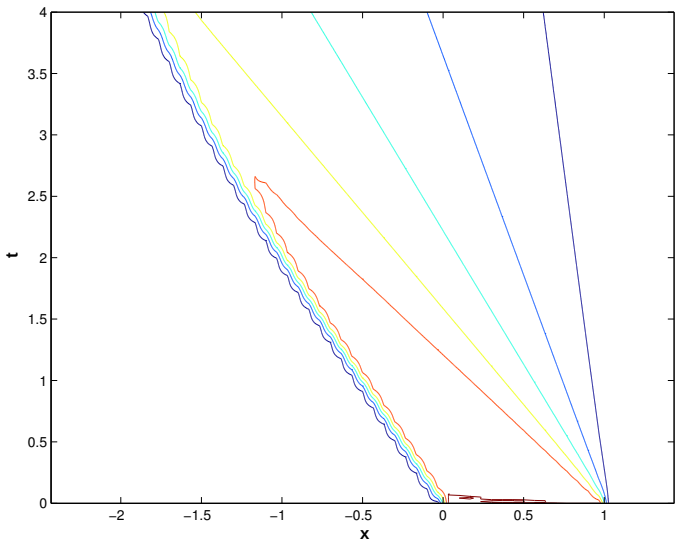
- Cauchy problem for Traffic Flow equation (8.1.41) (flux function $f(u) = u(1-u)$) from Ex. 8.2.40 ("box" intial data)
- Equidistant spatial mesh with meshwidth $h = \frac{1}{15}$
- Linear reconstruction with minmod limited slope ($\rightarrow$ Def. 8.5.26)

$$\sigma_j := \operatorname{minmod}\left(\frac{\mu_j - \mu_{j-1}}{h}, \frac{\mu_{j+1} - \mu_j}{h}\right) .$$

- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- timestepping based on adaptive Runge-Kutta method `ode45` of MATLAB (`opts = odeset('abstol',1E-7,'reltol',1E-6);`).







Observation: spurious oscillations successfully suppressed!

Example 8.5.30 (Improved resolution by limited linear reconstruction)

- Same setting as in Ex. 8.3.42: Cauchy problem for Burgers equation (8.1.46) (flux function $f(u) = \frac{1}{2}u^2$) from Ex. 8.2.39 (shifted "box" intial data, $u_0(x) = -1$ for $x \notin [0,1]$, $u_0(x) = 1$ for $x \in [0,1]$)
- Equidistant spatial mesh with meshwidth $h = \frac{1}{15}$

- "High-order" method based on linear reconstruction with minmod limited slope (→ Def. 8.5.26)

$$\sigma_j := \operatorname{minmod}\left(\frac{\mu_j - \mu_{j-1}}{h}, \frac{\mu_{j+1} - \mu_j}{h}\right) .$$

- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- timestepping based on adaptive Runge-Kutta method `ode45` of MATLAB (`opts = odeset('abstol',1E-10,'reltol',1E-8);`).

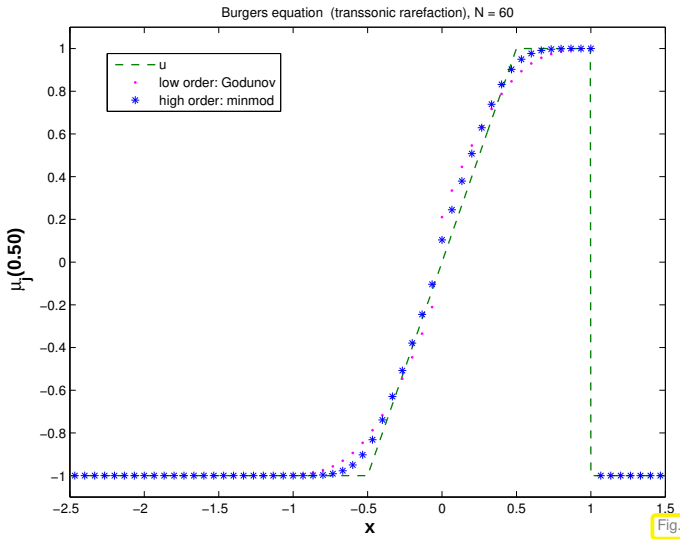


Fig. 465

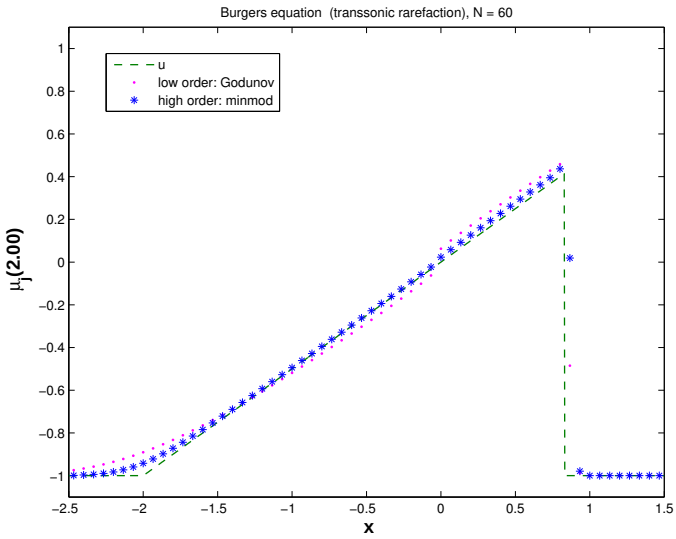


Fig. 466

Observation: *Better resolution* of rarefaction fan compared with the conservative finite volume method based on of Godunov numerical flux without linear reconstruction. Good resolution of shock.

This improved resolution is the main rationale for the use of piecewise linear reconstruction.

8.5.3 MUSCL scheme

= Monotone Upwind Scheme for Conservation Laws

Case of equidistant spatial mesh with meshwidth $h > 0$:

- Conservative finite volume spatial discretization (8.5.1) with monotone consistent 2-point flux, e.g., Godunov numerical flux (8.3.53)
- Piecewise linear reconstruction (→ Def. 8.5.3) with minmod slope limiting (→ Def. 8.5.26):

$$v_j^{\pm} := \mu_j \pm \tfrac{1}{2} \operatorname{minmod}\left(\mu_{j+1} - \mu_j, \mu_j - \mu_{j-1}\right) . \tag{8.5.31}$$

- 2nd-order Runge-Kutta timestepping for (8.5.1): method of Heun, *cf.* (8.4.10):

If the right hand side of (8.5.1) is abbreviated by

$$\mathcal{L}_h(\vec{\mu}) := -\frac{1}{h}\left(F(v_j^+(t), v_{j+1}^-(t)) - F(v_{j-1}^+(t), v_j^-(t))\right) ,$$

then the fully discrete scheme (uniform timestep $\tau > 0$) reads (8.5.1)

$$\begin{aligned} \vec{\kappa} &:= \vec{\mu}^{(k)} + \tfrac{1}{2}\tau\mathcal{L}_h(\vec{\mu}^{(k)}) \;, \\ \vec{\mu}^{(k+1)} &:= \vec{\mu}^{(k)} + \tau h\mathcal{L}_h(\vec{\kappa}) \;. \end{aligned} \tag{8.5.32}$$

Example 8.5.33 (Adequacy of 2nd-order timestepping)

- Same setting as in Ex. 8.3.42: Cauchy problem for Burgers equation (8.1.46) (flux function $f(u) = \frac{1}{2}u^2$) from Ex. 8.2.39 (shifted "box" intial data, $u_0(x) = -1$ for $x \notin [0,1]$, $u_0(x) = 1$ for $x \in [0,1]$)
- Equidistant spatial mesh with meshwidth $h = \frac{1}{15}$
- Linear reconstruction with minmod limited slope ($\rightarrow$ Def. 8.5.26)

$$\sigma_j := \operatorname{minmod}\left(\frac{\mu_j - \mu_{j-1}}{h}, \frac{\mu_{j+1} - \mu_j}{h}\right) .$$

- Godunov numerical flux (8.3.53): $F = F_{\mathrm{GD}}$
- Two options for timestepping
  1. timestepping based on adaptive Runge-Kutta method `ode45` of MATLAB (`opts = odeset('abstol',1E-10,'reltol',1E-8);`).
  2. Heun timestepping (8.5.32) with uniform timestep $\tau = h$

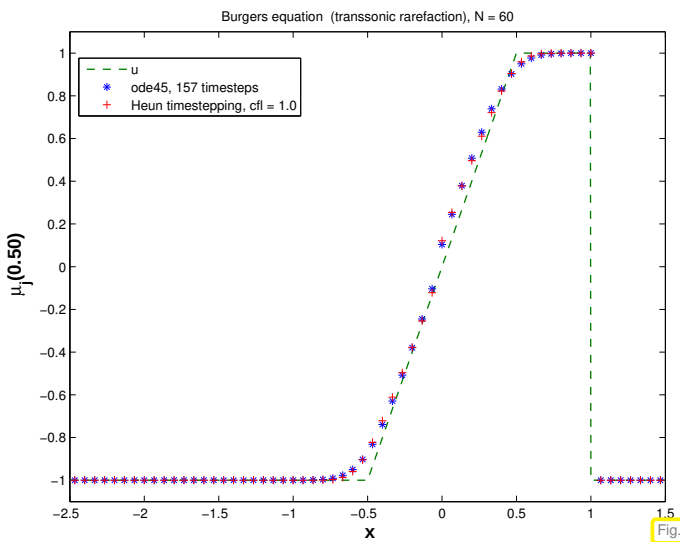


Fig. 467

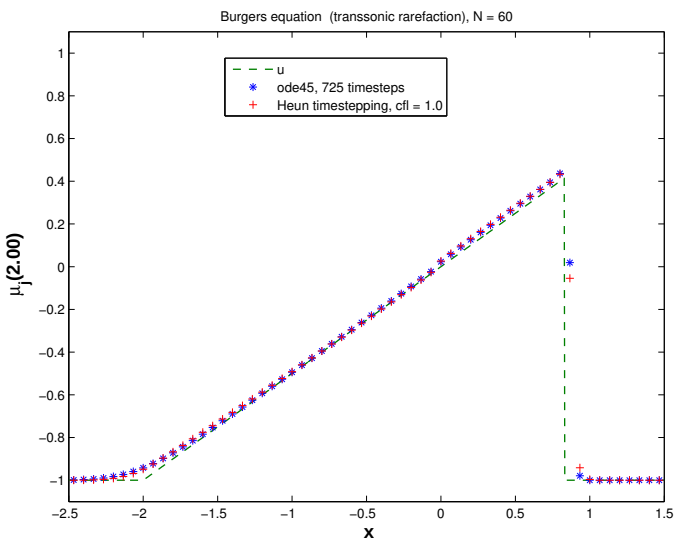


Fig. 468

Observation: 2nd-order Runge-Kutta method (8.5.32) provides same accuracy as "overkill integration" by means of `ode45` with tigth tolerances.

➢ For the sake of efficiency balance order of spatial and temporal discretizations and use Heun timestepping.

Example 8.5.34 (Convergence of MUSCL scheme)

Numerical experiments of Ex. 8.4.47 repeated for

- conservative finite volume discretization with Godunov numerical flux and minmod-limited linear reconstruction, see Ex. 8.5.28 (`ode45` timestepping),
- MUSCL scheme as introduced above with fixed timestep $\tau = 0.5h$.

Monitored: "discrete" error norms (8.4.48), (8.4.49)

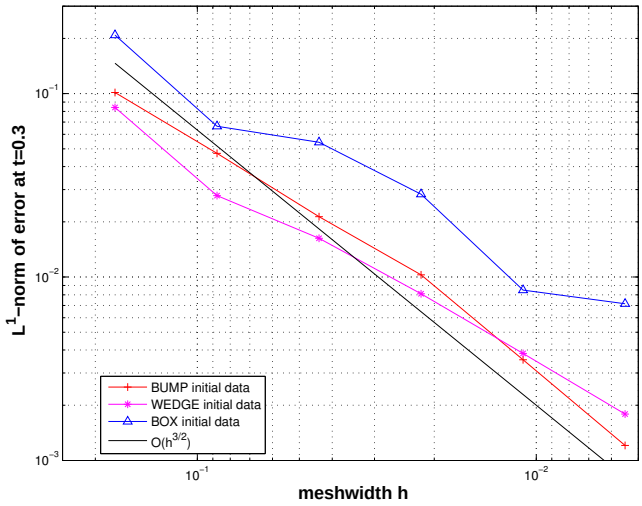


Fig. 469

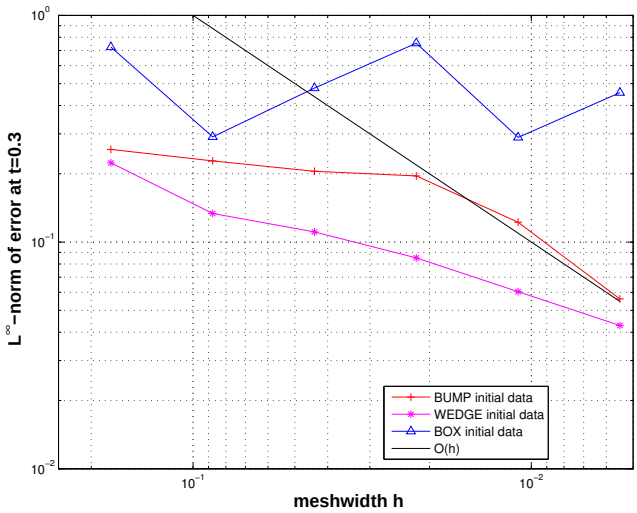


Fig. 470

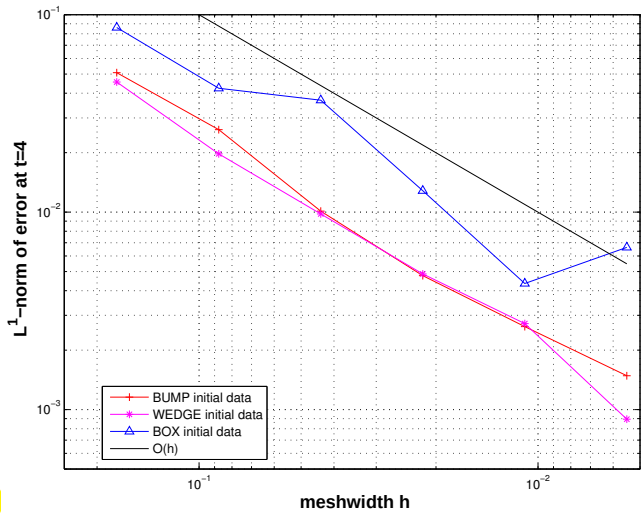


Fig. 471

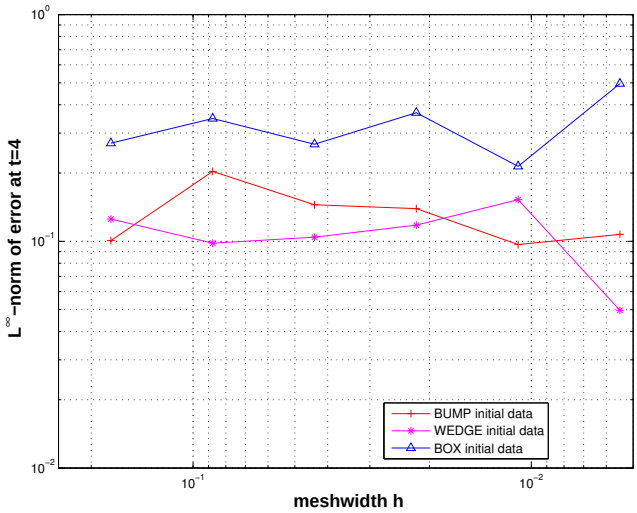


Fig. 472

Observation: 2nd-order Heun method produces solutions whose convergence and accuracy matches those of solutions obtained by highly accurate high-order Runge-Kutta timestepping.

➢

?! *Review question(s) 8.5.35.* (Higher order conservative finite volume methods)

1. Argue why a linear monotonicity preserving piecewise linear reconstruction must impose vanishing slopes throughout.
2. Explain the meanings of "linear" in the phrase "non-*linear linear* reconstruction" (in the context of high-order finite volume methods for 1D conservation laws).
3. What will be the width of the stencil for the fully discrete evolution operator, when using two-point numerical fluxes, piecewise linear reconstruction based on min-mod limiting, and an s-stable explicit RK-SSM for timestepping for the discretization of a scalar conservation law in 1D.

8.6 Outlook: systems of conservation laws

Learning outcomes

In this chapter about the numerical treatment of conservation laws you should have learned

- the general form of a scalar conservation law in one spatial dimension, and the balance law expressed by it.
- the notions of weak solutions, shock solutions, entropy conditions and entropy solutions.
- the general policy of constructing a conservative finite volume spatial semi-discretization.
- important consistent numerical flux functions, in particular the Godunov flux.
- the structure preservation inherent in conservative finite volume methods based on monotone numerical fluxes.
- the concept and significance of the CFL-condition for fully discrete conservative finite volume schemes.
- the construction of "high-order" spatial discretizations based on slope limited piecewise linear reconstruction.

Chapter 9

Finite Elements for the Stokes Equations

Contents

9.1 Viscous fluid flow

Task: simulation of *stationary fluid flow*

computation of the velocity $\mathbf{v} = \mathbf{v}(\boldsymbol{x})$ of a fluid moving in a container $\Omega \subset \mathbb{R}^d$, $d = 2, 3$, under the influence of an external force field $\mathbf{f} : \Omega \mapsto \mathbb{R}^d$.

($d = 2$? $\leftrightarrow$ translational symmetry ➡ dimensionally reduced model)

✎ notation: as before, bold typeface for vector valued functions

Recall: description of fluid motion through a velocity field $\rightarrow$ Sect. 7.1.1

We restrict ourselves to incompressible fluids $\rightarrow$ Def. 7.1.7

Thm. 7.1.12 ► Constraint $\operatorname{div} \mathbf{v} = 0$. (9.1.1)

► configuration space for incompressible fluid

$$V := \left\{ \mathbf{v} : \overline{\Omega} \mapsto \mathbb{R}^d \text{ continuous} , \ \operatorname{div} \mathbf{v} = 0 \right\} . \tag{9.1.2}$$

Flow regimes of an incompressible Newtonian fluid (a fluid, for which stress is linearly proportional to strain) are distinguished by the size of a fundamental *non-dimensional* quantity, the

$$\text{Reynolds number} \qquad \mathrm{Re} := \frac{\rho V L}{\mu} ,$$

where (for $d = 3$)

- $\rho \mathrel{\hat{=}}$ density ($[\rho] = \mathrm{kg\,m^{-3}}$)
- $V \mathrel{\hat{=}}$ mean velocity ($[V] = \mathrm{m\,s^{-1}}$)
- $L \mathrel{\hat{=}}$ characteristic length of region of interest ($[L] = \mathrm{m}$)
- $\mu \mathrel{\hat{=}}$ dynamic viscosity ($[\mu] = \mathrm{kg\,m^{-1}s^{-1}}$)

Reynolds number = ratio of inertia forces : viscous (friction) forces

The Reynolds number becomes small, if

- the speed of the flow is very small (slowly flowing fluids), or
- the flow is studied at tiny length scales (micro flows), or
- the fluid is highly viscous ("sticky").

In this case acceptably accurate modelling can neglect inertia forces ➢ creeping flow

Viscous fluids "stick to the walls of the container"

$$\text{no-slip boundary conditions:} \qquad \mathbf{v} = 0 \quad \text{on} \quad \partial\Omega . \tag{9.1.3}$$

▶ configuration space for viscous incompressible fluid

$$V := \left\{ \begin{array}{c} \mathbf{v} : \overline{\Omega} \mapsto \mathbb{R}^d \text{ continuous,} \\ \operatorname{div} \mathbf{v} = 0 , \quad \mathbf{v}_{|\partial\Omega} = 0 \end{array} \right\} . \tag{9.1.4}$$

We appeal to an extremal principle to derive governing equations for incompressible creeping flow: the state of the system renders a physical quantity minimal.

For the elastic string (→ Sect. 1.2), taut membrane (→ Sect. 2.2.1), electrostatic field (→ Sect. 2.2.2) this quantity was the total potential energy. For stationary viscous fluid flow, this role is played by the energy dissipation:

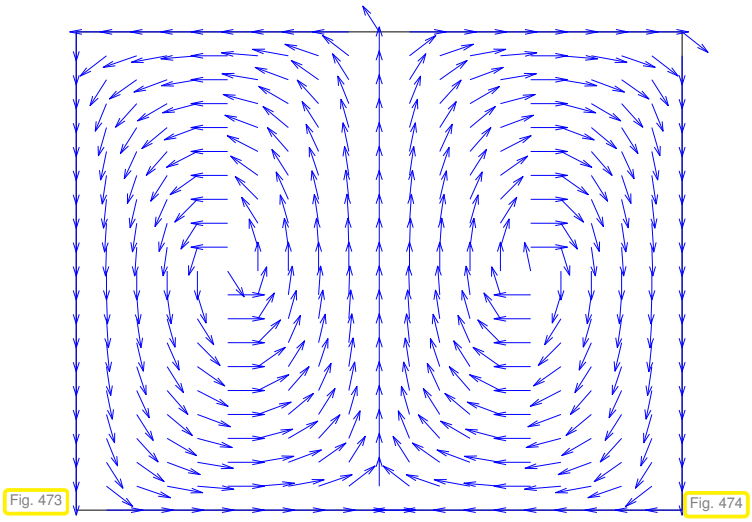
energy dissipation = conversion of kinetic energy into internal energy (heat)
(↔ entropy production)

AXIOM: energy dissipation functional for viscous fluid ($[P_{\mathrm{diss}}] = \mathrm{W}$)

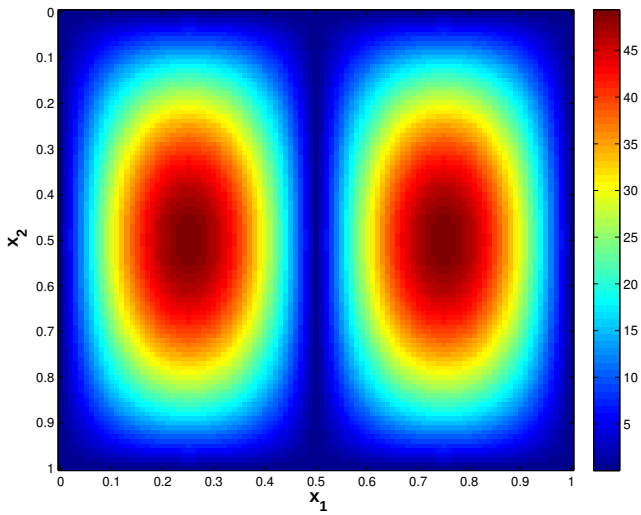
$$P_{\mathrm{diss}}(\mathbf{v}) = \int_\Omega \mu \|\mathbf{curl}\, \mathbf{v}(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \tag{9.1.5}$$

rotation/curl $\mathrel{\hat{=}}$ first-order differential operator

$$\mathbf{curl}\, \mathbf{v} := \begin{pmatrix} \dfrac{\partial v_2}{\partial x_3} - \dfrac{\partial v_3}{\partial x_2} \\ \dfrac{\partial v_3}{\partial x_1} - \dfrac{\partial v_1}{\partial x_3} \\ \dfrac{\partial v_1}{\partial x_2} - \dfrac{\partial v_2}{\partial x_1} \end{pmatrix} \quad \text{for} \quad d = 3 , \quad \mathbf{curl}\, \mathbf{v} := \frac{\partial v_1}{\partial x_2} - \frac{\partial v_2}{\partial x_1} \quad \text{for} \quad d = 2 . \tag{9.1.6}$$



"eddy field", $\operatorname{div} \mathbf{v} = 0$



Plot of $\|\mathbf{curl}\,\mathbf{v}\|$ for eddy field

$\mathbf{curl}\,\mathbf{v}$ = "density of eddies/vortices in flow field"

Thus, in viscous fluid flow the conversion of kinetic energy into heat due to friction presumably happens in vortical flow patterns (eddies).

Second law of thermodynamics for creeping flow:

$$\text{Maximization of energy dissipation in flow} \tag{9.1.7}$$

entropy production

First law of thermodynamics: conservation of energy/power balance

$$\int_\Omega \mu \|\mathbf{curl}\,\mathbf{v}(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{f} \cdot \mathbf{v} \,\mathrm{d}\boldsymbol{x} \,. \tag{9.1.8}$$

dissipated energy energy injected through forces

First equilibrium condition for viscous stationary flow:

$$\mathbf{v}^* = \operatorname{argmax}\left\{ \int_\Omega \mu \|\mathbf{curl}\,\mathbf{v}(\boldsymbol{x})\|^2 \,\mathrm{d}\boldsymbol{x} \colon \mathbf{v} \in V \,,\ \mathbf{v} \text{ satisfies (9.1.8)} \right\} \tag{9.1.9}$$

$\hat{=}$ *constrained optimization problem* with constraint (9.1.8).

Goal: Convert (9.1.9) into a "more standard" optimization problem.

To that end we study a related problem in finite dimensional context $\mathbb{R}^n$:

$$\mathbf{x}^* = \operatorname*{argmax}_{\mathbf{x}^T\mathbf{A}\mathbf{x} = \mathbf{b}^T\mathbf{x}} \mathbf{x}^T\mathbf{A}\mathbf{x} \,, \tag{9.1.10}$$

with s.p.d. $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$. With the transformation $\mathbf{y} = \mathbf{A}^{-1/2}\mathbf{x}$ ($\rightarrow$ [8, Rem. 8.3.2]) we arrive at the equivalent maximization problem

$$\mathbf{y}^* = \operatorname*{argmax}_{\|\mathbf{y}\|^2 = (\mathbf{A}^{-1/2}\mathbf{b})^T\mathbf{y}} \|\mathbf{y}\|^2 \,.$$

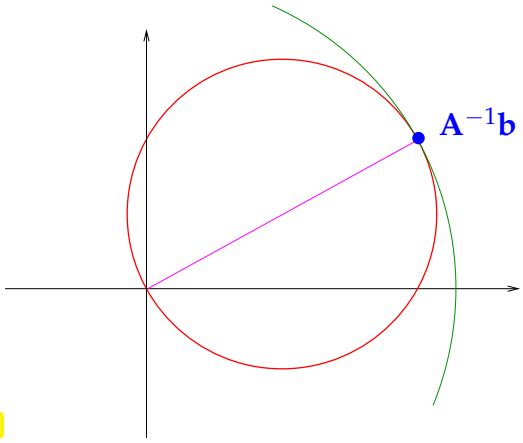


Fig. 475

The set $\{\mathbf{y} : \|\mathbf{y}\|^2 = (\mathbf{A}^{-1/2}\mathbf{b})^T\mathbf{y}\}$ is a sphere through 0 around $\frac{1}{2}\mathbf{A}^{-1/2}\mathbf{b}$ and we are looking for its point farthest away from 0. By "geometric considerations" this will be the point $\mathbf{y}^* = \mathbf{A}^{-1/2}\mathbf{b}$ ➢ $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$.

Recall: relationship between linear systems of equations and quadratic minimization problems, see [8, Section 8.1.1] and Sect. 2.2.3.

▶ $\mathbf{x}^* = \mathbf{A}^{-1}\mathbf{b}$ can be obtained as solution of

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^n} \tfrac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{b}^T\mathbf{x} \,. \tag{9.1.11}$$

To have faith that this reasoning applies to (9.1.9) as well, the bilinear form $(\mathbf{u},\mathbf{v}) \mapsto \int_\Omega \mathbf{curl}\,\mathbf{u}\cdot\mathbf{curl}\,\mathbf{v}\,\mathrm{d}\boldsymbol{x}$ should be positive definite (→ Def. 2.2.40) ➢ see Lemma 9.2.1 below.

Another issue, of course, is, whether the above arguments remain true for (infinite dimensional) function spaces ➢ theory of variational calculus [11, Ch. 49], not elaborated here.

▶ Second equilibrium condition for viscous stationary flow, *cf.* (2.2.12), (2.2.24):

$$\mathbf{v}^* = \operatorname*{argmin}_{\mathbf{v}\in V} \tfrac{1}{2}\int_\Omega \mu\|\mathbf{curl}\,\mathbf{v}(\boldsymbol{x})\|^2\,\mathrm{d}\boldsymbol{x} - \int_\Omega \mathbf{f}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} \,. \tag{9.1.12}$$

9.2 The Stokes equations

9.2.1 Constrained variational formulation

Lemma 9.2.1. $-\Delta = \mathbf{curl}\,\mathbf{curl} - \mathbf{grad}\,\mathrm{div}$

For $\mathbf{v} \in C^2(\overline{\Omega})$, $\mathbf{v}_{|\partial\Omega} = 0$, *holds*

$$\int_\Omega \|\mathbf{curl}\,\mathbf{v}\|^2\,\mathrm{d}\boldsymbol{x} + \int_\Omega |\,\mathrm{div}\,\mathbf{v}|^2\,\mathrm{d}\boldsymbol{x} = \int_\Omega \|D\mathbf{v}\|_F^2\,\mathrm{d}\boldsymbol{x} \,.$$

✎ notations: $D\mathbf{v} := \left(\dfrac{\partial v_i}{\partial x_j}\right)_{i,j=1}^d : \Omega \mapsto \mathbb{R}^{d,d}$ Jacobian,

$\|\mathbf{M}\|_F \mathrel{\hat{=}}$ Frobenius matrix norm (→ [8, Def. 7.5.37])

Proof (of Lemma 9.2.1)

Use the variant of Green's first formula Thm. 2.5.9

$$\int_\Omega \frac{\partial u}{\partial x_j} v \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \frac{\partial v}{\partial x_j} u \,\mathrm{d}\boldsymbol{x} \quad \forall u,v \in C^1(\overline{\Omega}) \,, \quad u,v = 0 \quad \text{on } \partial\Omega \,, \tag{9.2.2}$$

and the fact that different partial derivatives can be interchanged, which implies

$$\int_\Omega \frac{\partial u}{\partial x_j}\frac{\partial v}{\partial x_k} \,\mathrm{d}\boldsymbol{x} = \int_\Omega \frac{\partial u}{\partial x_k}\frac{\partial v}{\partial x_j} \,\mathrm{d}\boldsymbol{x} \,, \quad k,j = 1,\dots,d \,.$$

Then use the definitions of $\mathbf{curl}$ and div. □

In light of the properties $\mathrm{div}\,\mathbf{v} = 0$, $\mathbf{v} = 0$ on $\partial\Omega$, for eligible fluid velocity fields, see (9.1.4), we have the equivalence:

$$(9.1.12) \quad \overset{\text{Lemma 9.2.1}}{\Longleftrightarrow} \quad \mathbf{v}^* = \operatorname*{argmin}_{\mathbf{v}\in V} \tfrac{1}{2} \underbrace{\int_\Omega \mu \|D\mathbf{v}\|_F^2 \,\mathrm{d}\boldsymbol{x}}_{=:\mathsf{a}(\mathbf{v},\mathbf{v})} - \underbrace{\int_\Omega \mathbf{f}\cdot\mathbf{v} \,\mathrm{d}\boldsymbol{x}}_{=:\ell(\mathbf{v})} \,. \tag{9.2.3}$$

$\hat{=}$ quadratic minimization problem ($\rightarrow$ Def. 2.2.32) on function space V.

Rewrite quadratic form ($\mu \equiv \mathrm{const}$)

$$\mathbf{v} = (v_1,\dots,v_d)^T: \quad \int_\Omega \mu \|D\mathbf{v}\|_F^2 \,\mathrm{d}\boldsymbol{x} = \mu \sum_{i=1}^d \|\mathbf{grad}\, v_i\|^2 \,\mathrm{d}\boldsymbol{x} \,.$$

By the first Poincaré-Friedrichs inequality of Thm. 2.3.31

$$\|\mathbf{v}\|_{L^2(\Omega)}^2 \le \mathrm{diam}(\Omega)^2 \int_\Omega \|D\mathbf{v}\|_F^2 \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{v} \in V \subset (H_0^1(\Omega))^3 \,.$$

► Bilinear form a from (9.2.3) is positive definite ($\rightarrow$ Def. 2.2.40).

Remark 9.2.4 (Decoupling of velocity components ?)

Rewrite (9.2.3) in terms of components v_i of velocity (with force field $\mathbf{f} = (f_1,f_2,f_3)^T$):

$$(9.2.3) \quad \Leftrightarrow \quad \operatorname*{argmin}_{\mathbf{v}\in V} \sum_{i=1}^3 \Big(\tfrac{1}{2} \int_\Omega \mu \|\mathbf{grad}\, v_i\|^2 \,\mathrm{d}\boldsymbol{x} - \int_\Omega f_i v_i \,\mathrm{d}\boldsymbol{x} \Big) \,. \tag{9.2.5}$$

Well, three copies of (2.2.24) **?!**

NO! $\mathrm{div}\,\mathbf{v} = 0$ constraint (9.1.1) links components of velocity field $\mathbf{v}$.

This constraint in the space V represents the crucial difference compared to minimization problems (2.2.12), (2.2.24) underlying scalar 2nd-order elliptic variational equations.

As in Sect. 2.3: put (9.2.3) into Hilbert space (more precisely, Sobolev space) framework, where we have existence and uniqueness of solutions.

(9.2.5) offers hint on how to choose suitable Sobolev spaces.

Remember: function spaces for a (linear) variational problem are chosen as the largest (Hilbert) spaces on which the involved bilinear forms and linear forms are still *continuous*, *cf.* (2.2.55), (3.2.4).

(9.2.5), (9.1.3) ► appropriate Sobolev space for (9.2.3):

$$\boldsymbol{H}_0^1(\operatorname{div} 0,\Omega) := \left\{ \mathbf{v} \in (H_0^1(\Omega))^3 \colon \operatorname{div} \mathbf{v} = 0 \right\}$$

($(H_0^1(\Omega))^3 \hat{=}$ space of vector fields with components in $H_0^1(\Omega)$, alternative notation $\boldsymbol{H}_0^1(\Omega)$).

► As in Sect. 2.4.1 derive the linear variational problem

$$\mathbf{v} \in \boldsymbol{H}_0^1(\operatorname{div} 0,\Omega) \colon \quad \mathsf{a}(\mathbf{v},\mathbf{w}) = \ell(\mathbf{w}) \quad \forall \mathbf{w} \in \boldsymbol{H}_0^1(\operatorname{div} 0,\Omega) \,,$$

from (9.2.5), which reads in concrete terms:

Seek $\mathbf{v} \in \boldsymbol{H}_0^1(\operatorname{div} 0,\Omega) := \left\{ \mathbf{v} \in (H_0^1(\Omega))^3 \colon \operatorname{div} \mathbf{v} = 0 \right\}$ such that

$$\begin{gathered} \int_\Omega \mathbf{grad}\, v_i \cdot \mathbf{grad}\, w_i \,\mathrm{d}\boldsymbol{x} = \int_\Omega f_i w_i \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{w} \in \boldsymbol{H}_0^1(\operatorname{div} 0,\Omega) \,, \quad i = 1,2,3 \,, \\ \Updownarrow \\ \int_\Omega D\mathbf{v} : D\mathbf{w} \,\mathrm{d}\boldsymbol{x} = \int_\Omega \mathbf{f} \cdot \mathbf{w} \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{w} \in \boldsymbol{H}_0^1(\operatorname{div} 0,\Omega) \,. \end{gathered} \tag{9.2.6}$$

✎ notation: $\mathbf{A} : \mathbf{B} := \sum_{i,j} a_{ij} b_{ij}$ for matrices $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{m,n}$ ("componentwise dot product").

For this linear variational problem we verify

- Assumption 5.1.2 from Poincaré-Friedrichs inequality, see above,
- Assumption 5.1.3 for $\mathbf{f} \in (L^2(\Omega))^d$ by Cauchy-Schwarz inequality, see (2.3.30), (**??**),
- Assumption 5.1.4, since $\boldsymbol{H}_0^1(\operatorname{div} 0,\Omega)$ is a closed subspace of $\boldsymbol{H}^1(\Omega)$.

Thm. 5.1.5 ➡ existence & uniqueness of solutions of (9.2.6)

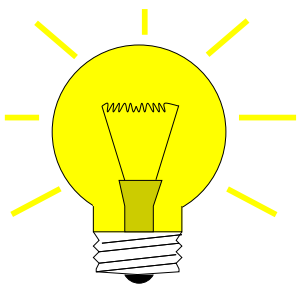
Remark 9.2.7 ($\boldsymbol{H}_0^1(\operatorname{div} 0,\Omega)$-conforming finite elements)

In principle, the linear variational problem could be tackled by means of a finite element Galerkin discretization.

However, finding finite element spaces $\subset \boldsymbol{H}_0^1(\operatorname{div} 0,\Omega)$ is complicated [9]: Continuous, piecewise polynomial, locally supported, and divergence free basis fields exist only for polynomial degree ≥ 4!

This remark motivates an approach that removes the constraint from trial and test space (and incorporates it into the variational formulation).

9.2.2 Saddle point problem



Idea: weak enforcement of divergence constraint (9.1.1) through Lagrange multiplier

Remark 9.2.8 (Heuristics behind Lagrangian multipliers)

Setting:

- $U, Q \hat{=}$ real Hilbert spaces with inner products $(\cdot,\cdot)_U$, $(\cdot,\cdot)_Q$,
- $J : U \mapsto \mathbb{R}$ convex and differentiable functional,
- $\mathsf{B} : U \mapsto Q$ linear operator (defining constraint)

Linearly constrained minimization problem

$$v^* = \operatorname*{argmin}_{v\in U, \mathsf{B}v=0} J(v) \; . \tag{9.2.9}$$

Introduce Lagrangian functional:

$$L(v,p) := J(v) + (p, \mathsf{B}v)_Q \quad \blacktriangleright \quad v^* = \operatorname*{argmin}_{v\in U} \sup_{p\in Q} L(v,p) \; , \tag{9.2.10}$$

because, if $\mathsf{B}v \neq 0$, the value of the inner supremum will be $+\infty$, and, thus, such a v can never be a candidate for a minimizer.

Terminology: p is called a Lagrange multiplier, Q the multiplier space.

Terminology: a min-max problem like (9.2.10) = saddle point problem

Lemma 9.2.11. Necessary conditions for existence of solution of saddle point problem → [11, Ch. 50]

Any solution v^* of (9.2.10) *will be the first component of a zero (v^*, p^*) of the derivative ("gradient") of the Lagrangian functional L.*

$\blacktriangleright$ (v^*, p^*) will satisfy

$$\begin{aligned} \lim_{t\to 0} \frac{L(v^*+tw,p^*) - L(v^*,p^*)}{t} &= 0 \quad \forall w \in U \; , \\ \lim_{t\to 0} \frac{L(v^*,p^*+tq) - L(v^*,p^*)}{t} &= 0 \quad \forall q \in Q \; . \end{aligned} \tag{9.2.12}$$

because by the very structure of the saddle point problem, see Fig. 476 for illustration,

$$L(v^*,p) \le L(v^*,p^*) \le L(v,p^*) \quad \forall v \in U,\; p \in Q \; . \tag{9.2.13}$$

Computing these "directional derivatives" as in Sect. 1.3.1 (for the elastic string energy functional there), we obtain

$$\begin{aligned} \langle DJ(v^*), w\rangle \;+\; (p^*, \mathsf{B}w)_Q &= 0 \quad \forall w \in U \; , \\ (q, \mathsf{B}v^*)_Q &= 0 \quad \forall q \in Q \; . \end{aligned} \tag{9.2.14}$$

This is a variational saddle point problem .

Special case: quadratic functional $J : U \mapsto \mathbb{R}$ → Def. 2.2.27

$$J(v) := \tfrac{1}{2}\mathsf{a}(v,v) - \ell(v) \; ,$$

with a positive definite, symmetric bilinear form $\mathsf{a} : U \times U \mapsto \mathbb{R}$ (→ Defs. 1.3.22, 2.2.40), continuous linear form $\ell : U \mapsto \mathbb{R}$.

(2.4.9)

$$\langle DJ(v^*), w\rangle = \mathsf{a}(v^*, w) - \ell(w) \;, \quad w \in U \;.$$

In this special case (9.2.14) becomes a linear variational saddle point problem:

Seek $v^* \in U$, $p^* \in Q$

$$\begin{aligned} \mathsf{a}(v^*, w) \quad + \quad (p^*, \mathsf{B}w)_Q &= \ell(w) \quad \forall w \in U \;, \\ (q, \mathsf{B}v^*)_Q \qquad\qquad\quad &= 0 \qquad \forall q \in Q \;. \end{aligned} \tag{9.2.15}$$

For rigorous mathematical treatment of constrained optimization in Banach spaces refer to [11, Ch. 49 & Ch. 50]. A discussion in finite-dimensional setting is given in [8, Section 6.4.1].

Solution of min-max problem:

saddle point

(non-extremal critical point)

The saddle point is a minimum when approached from the "U-direction", and a maximum, when approached from the "Q-direction".

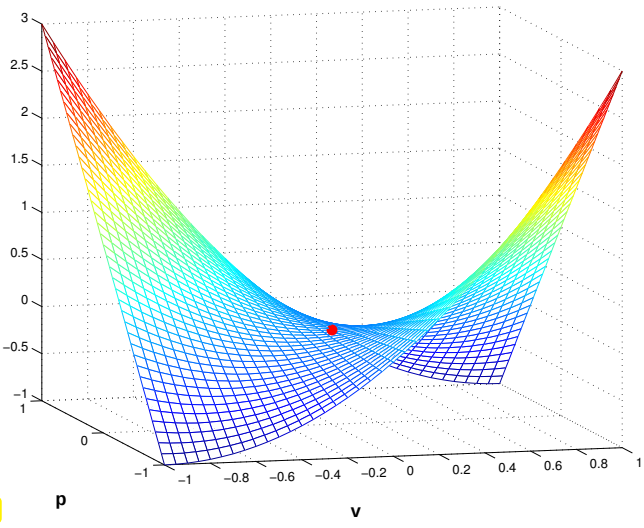


Fig. 476

Adapt abstract approach outline in Rem. 9.2.8 to (9.2.6):

- Hilbert spaces: $U = \boldsymbol{H}_0^1(\Omega)$, $Q = L^2(\Omega)$,
- Constraint $\operatorname{div} \mathbf{v} = 0$ ➢ $\mathsf{B} := \operatorname{div} : U \mapsto Q$ continuous,
- $J \leftrightarrow \mathbf{v} \mapsto \frac{1}{2}\int_\Omega \mu \|D\mathbf{v}\|^2 \,\mathrm{d}\boldsymbol{x} - \int_\Omega \mathbf{f}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x}$, a strictly convex quadratic functional (→ Def. 2.2.27)

Lagrangian functional for (9.2.6)

$$L(\mathbf{v}, p) = \frac{1}{2}\int_\Omega \mu \|D\mathbf{v}\|_F^2 \,\mathrm{d}\boldsymbol{x} - \int_\Omega \mathbf{f}\cdot\mathbf{v}\,\mathrm{d}\boldsymbol{x} + \int_\Omega \operatorname{div}\mathbf{v}\, p\,\mathrm{d}\boldsymbol{x} \;, \quad \mathbf{v} \in \boldsymbol{H}_0^1(\Omega),\; p \in L^2(\Omega) \;. \tag{9.2.16}$$

Next use formula for derivative of quadratic functionals, see Sect. 2.4.1, (2.4.9), which yields a concrete specimen of (9.2.15).

Stokes problem: Linear variational saddle point problem for viscous flow (preliminary version)

seek velocity $\mathbf{v} \in \boldsymbol{H}_0^1(\Omega)$, Lagrange multiplier $p \in L^2(\Omega)$

$$\begin{aligned} \int_\Omega \mu D\mathbf{v} : D\mathbf{w}\,\mathrm{d}\boldsymbol{x} \quad + \quad \int_\Omega \operatorname{div}\mathbf{w}\, p\,\mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{f}\cdot\mathbf{w}\,\mathrm{d}\boldsymbol{x} \qquad \forall \mathbf{w} \in \boldsymbol{H}_0^1(\Omega) \;, \\ \int_\Omega \operatorname{div}\mathbf{v}\, q\,\mathrm{d}\boldsymbol{x} \qquad\qquad\qquad &= 0 \qquad\qquad \forall q \in L^2(\Omega) \;. \end{aligned}$$

Lagrange multiplier p = pressure $([p] = \mathrm{N\,m^{-2}})$

No *differential constraints* in test/trial spaces for (9.2.19)!

Remark 9.2.17 (Ensuring uniqueness of pressure)

Notice: $$\int_\Omega \operatorname{div} \mathbf{v}\,\mathrm{d}\boldsymbol{x} = \int_{\partial\Omega} \mathbf{v}\cdot\boldsymbol{n}\,\mathrm{d}S = 0, \text{ since } \mathbf{v}_{|\partial\Omega} = 0.$$

Pressure solution p in (9.2.19) can be unique only up to an constant!

Compare: Non-uniqueness of solution of 2nd-order elliptic Neumann problem, Rem. 2.9.14.

Remedy, *cf.* (2.9.15)

$$\text{Choose} \quad p \in L^2_*(\Omega) := \{q \in L^2(\Omega) \colon \int_\Omega q\,\mathrm{d}\boldsymbol{x} = 0\}\,. \tag{9.2.18}$$

$\longleftrightarrow$ constraint on trial/test space $L^2(\Omega)$

Stokes problem: Variational saddle point problem for viscous flow

seek velocity $\mathbf{v} \in \boldsymbol{H}^1_0(\Omega)$, Lagrange multiplier $p \in L^2_*(\Omega)$

$$\begin{aligned} \int_\Omega \mu D\mathbf{v} : D\mathbf{w}\,\mathrm{d}\boldsymbol{x} \;+\; \int_\Omega \operatorname{div}\mathbf{w}\, p\,\mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{f}\cdot\mathbf{w}\,\mathrm{d}\boldsymbol{x} && \forall \mathbf{w} \in \boldsymbol{H}^1_0(\Omega)\,, \\ \int_\Omega \operatorname{div}\mathbf{v}\, q\,\mathrm{d}\boldsymbol{x} &= 0 && \forall q \in L^2_*(\Omega)\,. \end{aligned} \tag{9.2.19}$$

Theorem 9.2.20. Existence and uniqueness of weak solutions of Stokes problem

The linear variational saddle point problem (9.2.19) *("Stokes problem") has a unique solution.*

Proof. (crude outline; this sketch of the proof is included, because its ideas carry over to the discrete setting.)

Preparatory considerations: $\mathsf{a}(\mathbf{v},\mathbf{w}) := \int_\Omega \mu D\mathbf{v} : D\mathbf{w}\,\mathrm{d}\boldsymbol{x}$ is an inner product on $\boldsymbol{H}^1_0(\Omega)$.

$$\mathsf{a}\text{-orthogonal decomposition} \qquad \boldsymbol{H}^1_0(\Omega) = \boldsymbol{H}^1_0(\operatorname{div} 0,\Omega) \oplus V^\perp$$

❶ Unique solution $\mathbf{v} \in \boldsymbol{H}^1_0(\operatorname{div} 0,\Omega)$ of (9.2.6) ➢ unique $\mathbf{v}$-solution for (9.2.19)
(first test with $\mathbf{w} \in \boldsymbol{H}^1_0(\operatorname{div} 0,\Omega)$, then with $\mathbf{w} \in V^\perp$.)

❷ Use the following profound result from functional analysis [2, Thm. 5.3]:

Theorem 9.2.21. Existence of stable velocity potentials

$$\exists C = C(\Omega) > 0\colon \quad \forall q \in L^2_*(\Omega)\colon \quad \exists \mathbf{v} \in \boldsymbol{H}^1_0(\Omega)\colon \quad q = \operatorname{div}\mathbf{v} \quad \wedge \quad \|\mathbf{v}\|_{H^1(\Omega)} \le C\|q\|_{L^2(\Omega)}\,.$$

Idea: Assume $\mathbf{f} = 0$, test first equation with $\mathbf{w} \in V^{\perp}$ satisfying $\operatorname{div} \mathbf{w} = p$ ➤ $\|p\|_{L^2(\Omega)} = 0 \Leftrightarrow p = 0$, for any pressure solution $p \in L^2_*(\Omega)$.

uniqueness of pressure solution

❸ Existence of pressure solution from Riesz representation theorem (→ functional analysis) and Thm. 9.2.21, not elaborated here. □

Remaining issue: (9.2.18) introduces another *constraint* into (9.2.19)!

Relax, Lagrangian multipliers can deal with this, too. Now we study their use to enforce a zero mean constraint in the simpler setting of 2nd-order elliptic Neumann BVPs.

Remark 9.2.22 (Enforcing zero mean)

→ [1] As in Sect. 2.5, Rem. 2.9.14, we consider a 2nd-order linear Neumann BVP (with zero Neumann boundary c $h = 0$), *cf.* (2.9.16),

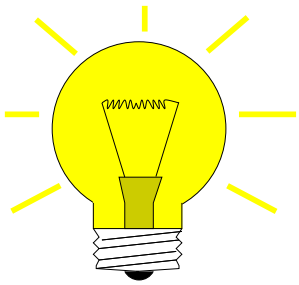
$$u \in H^1_*(\Omega): \quad \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} = \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1_*(\Omega)\,.$$

with the *constrained* trial/test space

$$H^1_*(\Omega) := \{v \in H^1(\Omega): \int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0\}\,. \tag{2.9.15}$$

The related quadratic minimization problem reads (→ Sect. 2.2.3)

$$u = \operatorname*{argmin}_{v \in H^1_*(\Omega)} J(v) \quad , \quad J(v) := \tfrac{1}{2}\int_\Omega \kappa(\boldsymbol{x}) \|\mathbf{grad}\, v\|^2 \,\mathrm{d}\boldsymbol{x} - \int_\Omega f v \,\mathrm{d}\boldsymbol{x}\,.$$



Idea: enforce linear constraint $\int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} = 0$ by means of Lagrangian multiplier, see Rem. 9.2.8

Here: scalar constraint ($Q = \mathbb{R}$) ➤ scalar multiplier $p \in \mathbb{R}$

► Lagrangian functional:

$$L(v,p) = J(v) + p\int_\Omega v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\,, \quad v \in H^1(\Omega)\,, \quad p \in \mathbb{R}\,.$$

► related (augmented) linear variational saddle point problem, specialization of (9.2.15):

seek $u \in H^1(\Omega)$, $p \in \mathbb{R}$

$$\begin{aligned} \int_\Omega \kappa(\boldsymbol{x})\, \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x} \;+\; p\int_\Omega v\,\mathrm{d}\boldsymbol{x} &= \int_\Omega f v \,\mathrm{d}\boldsymbol{x} \quad \forall v \in H^1(\Omega)\,, \\ \int_\Omega v\,\mathrm{d}\boldsymbol{x} &= 0\,. \end{aligned} \tag{9.2.23}$$

The same technique can be applied to (9.2.19).

Stokes variational saddle point problem with pressure normalization:

seek velocity $\mathbf{v} \in \boldsymbol{H}_0^1(\Omega)$, pressure $p \in L^2(\Omega)$, multiplier $\lambda \in \mathbb{R}$

$$\begin{aligned}
\int_\Omega \mu \nabla \mathbf{v} : \nabla \mathbf{w} \,\mathrm{d}\boldsymbol{x} + \int_\Omega \operatorname{div} \mathbf{w}\, p \,\mathrm{d}\boldsymbol{x} &= \int_\Omega \mathbf{f} \cdot \mathbf{w} \,\mathrm{d}\boldsymbol{x} \quad \forall \mathbf{w} \in \boldsymbol{H}_0^1(\Omega)\,, \\
\int_\Omega \operatorname{div} \mathbf{v}\, q \,\mathrm{d}\boldsymbol{x} + \lambda \int_\Omega q \,\mathrm{d}\boldsymbol{x} &= 0 \quad \forall q \in L^2(\Omega)\,, \\
\int_\Omega p \,\mathrm{d}\boldsymbol{x} &= 0 \quad .
\end{aligned} \tag{9.2.24}$$

9.2.3 Stokes system

As in Sect. 2.5: derivation of the BVP in PDE form corresponding to (9.2.24).

Approach: Remove spatial derivatives from test functions by integration by parts (1.3.40)

Assuming sufficient smoothness of solution $(\mathbf{v}, p)$, constant μ and (9.2.24) and taking into account boundary conditions, apply Green's formula of Thm 2.5.9:

$$\int_\Omega \mu \nabla \mathbf{v} : \nabla \mathbf{w} \,\mathrm{d}\boldsymbol{x} = \mu \sum_{i=1}^d \int_\Omega \mathbf{grad}\, v_i \cdot \mathbf{grad}\, w_i \,\mathrm{d}\boldsymbol{x} = -\mu \sum_{i=1}^d \int_\Omega \Delta v_i \, w_i \,\mathrm{d}\boldsymbol{x}\,,$$

$$\int_\Omega \operatorname{div} \mathbf{w}\, p \,\mathrm{d}\boldsymbol{x} = -\int_\Omega \mathbf{grad}\, p \cdot \mathbf{w} \,\mathrm{d}\boldsymbol{x}\,.$$

$$\Downarrow$$

$$(9.2.24) \quad \Rightarrow \quad \boxed{\begin{aligned}
-\mu \boldsymbol{\Delta} \mathbf{v} - \mathbf{grad}\, p &= \mathbf{f} \\
\operatorname{div} \mathbf{v} &= 0 \quad \text{in} \quad \Omega\,, \\
\int_\Omega p \,\mathrm{d}\boldsymbol{x} &= 0 \\
\mathbf{v} &= 0 \quad \text{on} \quad \partial\Omega\,.
\end{aligned}} \tag{9.2.25}$$

✎ notation: $\boldsymbol{\Delta} \mathrel{\hat{=}}$ componentwise Laplacian, see (2.5.15) ("vector Laplacian")

Remark 9.2.26 (Pressure Poisson equation)

Manipulating the PDEs in (9.2.25):

$$\begin{aligned}
\operatorname{div} \cdot (9.2.25) \quad &\Rightarrow \quad -\mu \operatorname{div} \boldsymbol{\Delta} \mathbf{v} + \operatorname{div} \mathbf{grad}\, p = \operatorname{div} \mathbf{f} \quad \text{in } \Omega\,, \\
&\Rightarrow \quad -\mu \Delta(\operatorname{div} \mathbf{v}) + \Delta p = \operatorname{div} \mathbf{f} \quad \text{in } \Omega\,, \\
&\overset{\operatorname{div} \mathbf{v} = 0}{\Rightarrow} \quad \Delta p = \operatorname{div} \mathbf{f}\,.
\end{aligned}$$

Appearance: (9.2.25) can be solved by solving $d+1$ Poisson equations,

- first solve pressure Poisson equation $\Delta p = \operatorname{div} \mathbf{f}$
- then solve Dirichlet boundary value problems for velocity components

$$-\Delta v_i = f_i + \frac{\partial p}{\partial x_i} \quad \text{in} \quad \Omega\,, \quad v_i = 0 \quad \text{on} \quad \partial\Omega\,.$$

☛ above manipulations only valid for *sufficiently smooth* $\mathbf{u}$ (not guaranteed).

Problems ☛ we cannot solve a "Poisson equation", we also need boundary conditions for p: not available!

9.3 Saddle point problems: Galerkin discretization

Example 9.3.1 (Naive finite difference discretization of Stokes system)

- BVP (9.2.25) on $\Omega =]0,1[^2$, $\mu \equiv 1$, $\mathbf{f} = \cos(\pi x_1)\binom{0}{1}$,
- Finite difference discretization on (→ Sect. 4.1) equidistant tensor product grid ▷
  Unknowns: $v_{1,ij}$, $v_{2,ij}$, p_{ij} $\hat{=}$ approximations of $v_1(ih,jh)$, $v_2(ih,jh)$, $p(ih,jh)$, $0 < i,j < N$.
- Zero boundary values for v_1, v_2, **and** p
- 5-point stencil discretization of $-\Delta$, see (4.1.3)
- Central finite difference approximation of $\mathbf{grad}\, p$, e.g.,

$$\frac{\partial p}{\partial x_1}_{|(ih,jh)} \approx \frac{1}{2h}\left(p_{i+1,j} - p_{i-1,j}\right)\,, 1 \le i,j < N\,.$$

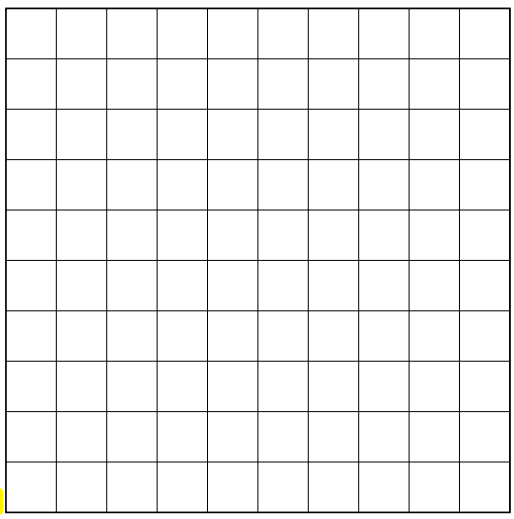


Fig. 477 finite difference grid

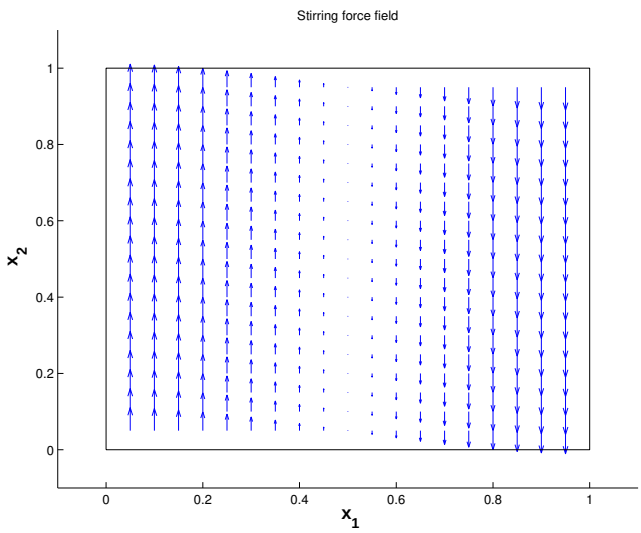


Fig. 478 force field $\mathbf{f}$

MATLAB Code 9.3.2: Central finite difference discretization of Stokes system

```
function [u1,u2,p] = StokesFD(N,f)
% Naive finite difference discretization of the Stokes system (9.2.25)
```

```
% N: number of grid cells in each direction.
% f:  handle to a (vector valued!)  function implementing the force field f
% Return values u1, u2 give the velocity components v = (v_1,v_2)^T
% in a matrix whose entries correspond to the vertices of the mesh,
% p returns the preassure.
h = 1/N;       % mesh width
x = h:h:1-h;  % coordinates of interior grid points
unk = N-1;    % number of interior points in each direction
n = 3 * unk^2; % total number of unknowns for v and p
% A line-by-line numbering (lexikographic numbering) of the grid points is assumed,
% see Sect. 4.1, Fig. 481.
A = gallery('poisson', unk); % Matrix for 5-point stencil discretization of −Δ

% Build matrix representation of grad p.  Note the efficient assembly based on the
% special structure of the matrices.
% Auxiliary 1D central finite difference matrix
e = ones(unk, 1); CD = spdiags([-h/2*e h/2*e], [-1 1], unk,unk);
% Central difference matrix for ∂/∂x_1:  This matrix is a block
% diagonal matrix with N−1 diagonal blocks corresponding to the grid rows.  Its diagonal
% blocks are skew-symmetric and bidiagonal with non-zero first off-diagonals only.
P1 = kron(speye(unk),CD);
% Central difference matrix for ∂/∂x_2:  This matrix is
% a block matrix with non-zero first off-diagonal blocks only.  Each non-zero block is
% a multiple of the identity.
P2 = kron(CD,speye(unk));
% Build the complete n×n system matrix and make sure that it is a sparse matrix.
Z = sparse(unk^2, unk^2); % Major mistake would be z = zeros(unk^2,unk^2);
H = [A Z P1; Z A P2; P1' P2' Z];

% Assemble the right hand side (sampling of f at interior grid points)
F = zeros(n,1);
pidx1 = 1; pidx2 = n/3+1;
for j = 1:size(x), for i = 1:size(x)
    frc = h^2*f(x(i),x(j));
    F(pidx1) = frc(1); F(pidx2) = frc(2);
    pidx1 = pidx1+1; pidx2 = pidx2 + 1;
end, end
% Direct solution of sparse indefinite symmetric system
X = H\F;

% Convert vectors of nodal values into matrix representations of grid functions
u1 = rot90(reshape(X(1:unk^2),unk,unk));
u2 = rot90(reshape(X(unk^2+1:2*unk^2),unk,unk));
p = rot90(reshape(X(2*unk^2+1:end),unk,unk));
```

```
end
```

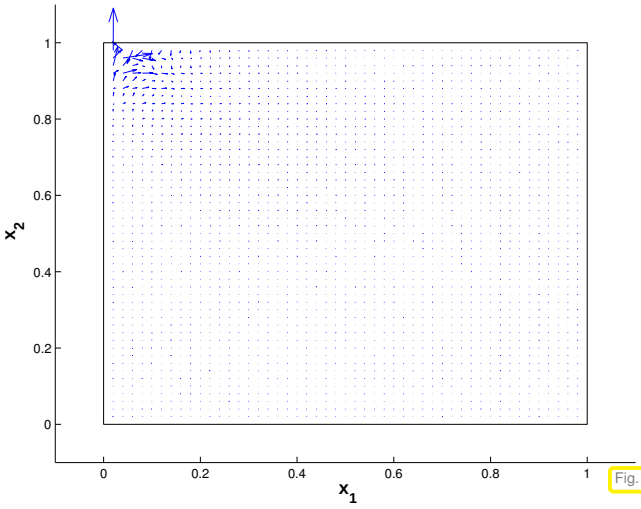


Fig. 479

velocity solution, $N = 50$

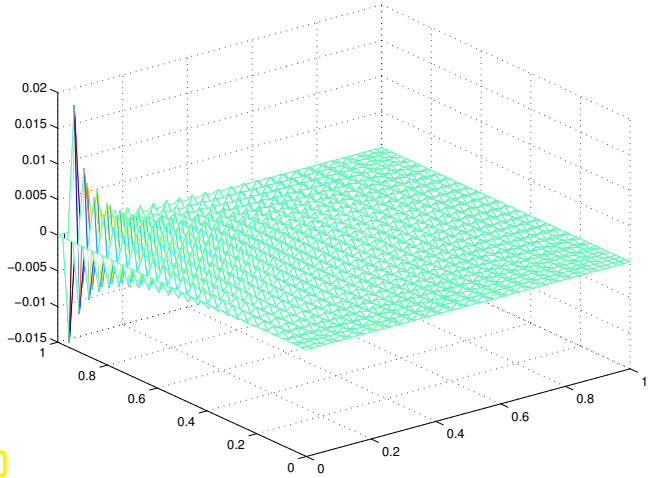


Fig. 480

pressure solution, $N = 50$

Physically meaningless solution marred by massive spurious oscillations of the pressure.

9.3.1 Pressure instability

Lesson learned: discretizing saddle point problems can be tricky!

Now, we examine the *Galerkin discretization* ($\rightarrow$ Sect. 3.2) of the linear variational problem (9.2.19) (Practical schemes will rely on (9.2.24), but here, for the sake of simplicity, we skirt the treatment of zero mean constraint.)

Shorthand notation for (9.2.19) ($\leftrightarrow$ abstract linear variational saddle point problem, see (9.2.15))

$$\begin{array}{l} \mathbf{v} \in U := \boldsymbol{H}_0^1(\Omega)\,, \\ p \in Q := L_*^2(\Omega) \end{array} \quad : \quad \begin{array}{lllll} \mathsf{a}(\mathbf{v},\mathbf{w}) & + & \mathsf{b}(\mathbf{w},p) & = & \ell(\mathbf{w}) \quad \forall \mathbf{w} \in U\,, \\ \mathsf{b}(\mathbf{v},q) & & & = & 0 \quad \forall q \in Q\,. \end{array} \tag{9.3.3}$$

with concrete *bilinear forms*

$$\mathsf{a}(\mathbf{v},\mathbf{w}) := \int_\Omega \mu\, D\mathbf{v} : D\mathbf{w}\,\mathrm{d}\boldsymbol{x} \quad , \quad \mathsf{b}(\mathbf{v},q) := \int_\Omega \operatorname{div}\mathbf{v}\, q\,\mathrm{d}\boldsymbol{x}\,. \tag{9.3.4}$$

First step of Galerkin discretization:

Replace $\begin{array}{c} \boldsymbol{H}_0^1(\Omega) \\ L_*^2(\Omega) \end{array}$ with finite dimensional subspaces $\begin{array}{c} U_N \subset \boldsymbol{H}_0^1(\Omega) \\ Q_N \subset L_*^2(\Omega) \end{array}$

Discrete linear variational saddle point problem:

$$\begin{array}{l} \mathbf{v}_N \in U_N \\ p_N \in Q_N \end{array} \quad : \quad \begin{array}{lllll} \mathsf{a}(\mathbf{v}_N,\mathbf{w}_N) & + & \mathsf{b}(\mathbf{w}_N,p_N) & = & \ell(\mathbf{w}_N) \quad \forall \mathbf{w}_N \in U_N\,, \\ \mathsf{b}(\mathbf{v}_N,q_N) & & & = & 0 \quad \forall q_N \in Q_N\,. \end{array} \tag{9.3.5}$$

Second step of Galerkin discretization:

Introduce ordered bases $\begin{array}{l}\mathfrak{B}_U := \{\mathbf{b}_N^1,\dots,\mathbf{b}_N^N\},\\ \mathfrak{B}_Q := \{\beta_N^1,\dots,\beta_N^M\}\end{array}$ of $\begin{array}{ll} U_N, & N := \dim U_N,\\ Q_N, & M := \dim Q_N.\end{array}$

$(N+M)\times(N+M)$ linear system of equations symmetric indefinite matrix!

$$\boxed{\begin{pmatrix}\mathbf{A} & \mathbf{B}^T\\ \mathbf{B} & 0\end{pmatrix}\begin{pmatrix}\vec{\nu}\\ \vec{\pi}\end{pmatrix} = \begin{pmatrix}\vec{\varphi}\\ 0\end{pmatrix}}\,, \tag{9.3.6}$$

(9.3.7)

with Galerkin matrices, right hand side vectors

$$\mathbf{A} := \left(\mathsf{a}(\mathbf{b}_N^j,\mathbf{b}_N^i)\right)_{i,j=1}^N = \left(\int_\Omega \mu\, D\mathbf{b}_N^j(\boldsymbol{x}) : D\mathbf{b}_N^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right)_{i,j=1}^N \in \mathbb{R}^{N,N}\,, \tag{9.3.8}$$

$$\mathbf{B} := \left(\mathsf{b}(\mathbf{b}_N^j,\beta_N^i)\right)_{\substack{1\le i\le M\\ 1\le j\le N}} = \left(\int_\Omega \operatorname{div}\mathbf{b}_N^j(\boldsymbol{x})\,\beta_N^i(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right)_{\substack{1\le i\le M\\ 1\le j\le N}} \in \mathbb{R}^{M,N}\,, \tag{9.3.9}$$

$$\vec{\varphi} := \left(\ell(\mathbf{b}_N^j)\right)_{j=1}^N = \left(\int_\Omega \mathbf{f}(\boldsymbol{x})\cdot\mathbf{b}_N^j(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}\right)_{j=1}^N \in \mathbb{R}^N\,, \tag{9.3.10}$$

and basis expansions

$$\mathbf{v}_N = \sum_{j=1}^N \nu_j\mathbf{b}_N^j \quad,\quad p_N = \sum_{j=1}^M \pi_j\beta_N^j\,. \tag{9.3.11}$$

Issue: existence, uniqueness and stability of solutions of (9.3.5)

Existence, uniqueness and stability of solutions of discrete variational saddle point problems cannot be inferred from these properties for the continuous saddle point problem ($\to$ Thm. 9.2.20).
(Unlike in the case of linear variational problems with s.p.d. bilinear forms, *cf.* Thm. 3.2.9)

A simple consideration:

$$M > N \quad\Rightarrow\quad \operatorname{Kern}\mathbf{B} \neq \{0\} \quad\Rightarrow\quad \text{non-uniquenesss of} \quad p_N\,.$$

➢ $\boxed{\dim U_N \ge \dim Q_N \text{ is a } \textit{necessary} \text{ condition for uniqueness of solution } p_N \text{ of (9.3.5)}}$

Some "natural" finite element Galerkin schemes for (9.2.19) $\leftrightarrow$ (9.3.3) fail to meet this condition:

Example 9.3.12 (Unstable P1-P0 finite element pair on triangular mesh)

Notation: (*cf.* $\mathcal{S}_p^0(\mathcal{M})$): $\mathcal{S}_p^{-1}(\mathcal{M})$ discontinuous functions; locally polynomials of degree p , *cf.* $\mathcal{P}_p(\mathbb{R}^d)$

The spaces $\mathcal{S}_p^{-1}(\mathcal{M})$ are the natural finite element spaces for test/trial functions $\in L^2(\Omega)$, because this function space does not enforce any continuity conditions on piecewise smooth functions. Conversely, $H^1(\Omega)$ does, see Thm. 2.3.35.

Regular triangular mesh of $]0,1[^2$ ▶

Finite element spaces for (9.2.19)

$U_N := (\mathcal{S}^0_{1,0}(\mathcal{M}))^2$,
$Q_N := \mathcal{S}_0^{-1}(\mathcal{M}) \cap L^2_*(\Omega)$ ($\mathcal{M}$-piecewise constants) .

$K \in \mathbb{N} \hat{=}$ no. of mesh cells in one coordinate direction,

$$\dim U_N = 2(K-1)^2 \quad , \quad \dim Q_N = 2K^2 - 1 \; .$$

$$\dim Q_N > \dim U_N$$

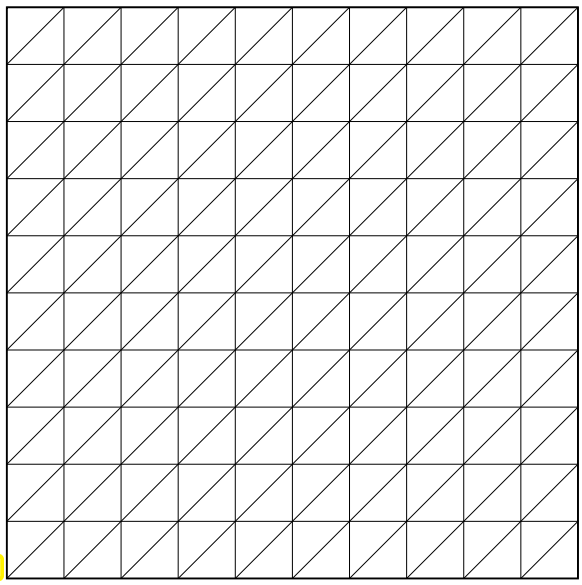


Fig. 481

In this case we end up with a *singular* linear system (9.3.6), which will make the linear solver bail out or produce a pressure solution, which is polluted by "noise" from $\mathrm{Kern}\,\mathbf{B}$.

But $\dim U_N \geq \dim Q_N$ is not enough: even if this condition is satisfied, the pressure may not be unique:

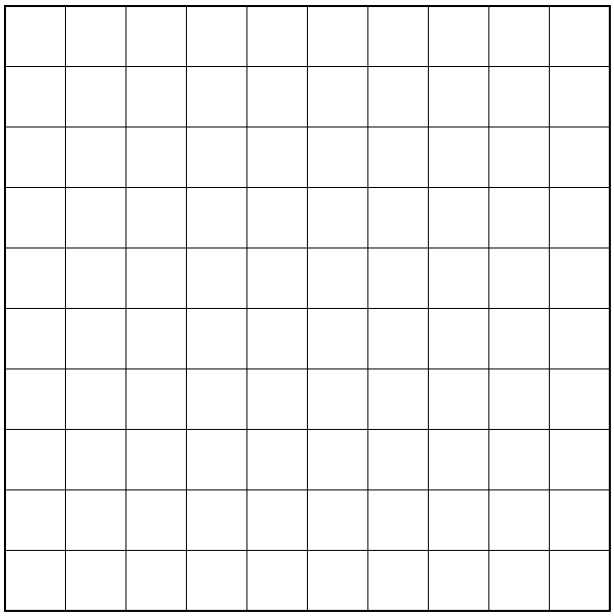
Example 9.3.13 (Checkerboard instability for quadrilateral P1-P0 FE pair → [2, § 6])

- $\mathcal{M}$ = uniform tensor product mesh of $]0,1[^2$ ▷
- velocity space $U_N = (\mathcal{S}^0_{1,0}(\mathcal{M}))^2$
- pressure space $Q_N = \mathcal{S}_0^{-1}(\mathcal{M}) \cap L^2_*(\Omega)$

If $K \in \mathbb{N}$ mesh cells in one coordinate direction,

$$\dim U_N = 2(K-1)^2 \quad , \quad \dim Q_N = K^2 - 1 \; .$$

▶ $\dim Q_N < \dim U_N$ for $K \geq 4$.



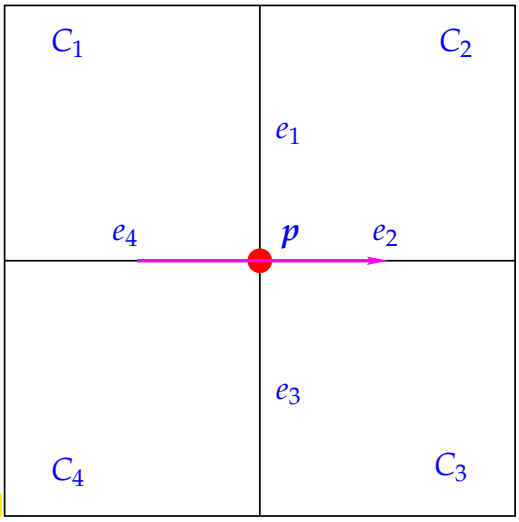


Fig. 482

Consider interior grid point $\boldsymbol{p} = (ih, jh)$, $1 \leq i, j \leq K$, with adjacent quadratic cells C_1, C_2, C_3, C_4, see figure. Denote by p_i the piecewise constant values of p_N on C_i, $i = 1, 2, 3, 4$.

$\mathbf{b}^{\boldsymbol{p}}_{N,1} \mathrel{\hat{=}}$ nodal basis function for x_1 velocity component at vertex $\boldsymbol{p}$: $\mathbf{b}^{\boldsymbol{p}}_{N,1} = b^{\boldsymbol{p}}_N \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, where $b^{\boldsymbol{p}}_N$ is the 2D "tent function" (→ Fig. 92) associated with $\boldsymbol{p}$.

▶
$$\operatorname{supp}(\mathbf{b}^{\boldsymbol{p}}_{N,1}) = C_1 \cup C_2 \cup C_3 \cup C_4$$

Apply Gauss' theorem Thm. 2.5.7 on C_i taking into account that $\mathbf{b}^{\boldsymbol{p}}_N \perp$ normals at e_2, e_4, and $\mathbf{b}^{\boldsymbol{p}}_N \parallel$ normals at e_1, e_3,

$$\begin{aligned}\int\limits_\Omega \operatorname{div} \mathbf{b}^{\boldsymbol{p}}_{N,1}\, p_N \,\mathrm{d}\boldsymbol{x} &= p_1 \int_{e_1} b^{\boldsymbol{p}}_N \,\mathrm{d}\boldsymbol{x} - p_2 \int_{e_1} b^{\boldsymbol{p}}_N \,\mathrm{d}\boldsymbol{x} + p_3 \int_{e_3} b^{\boldsymbol{p}}_N \,\mathrm{d}\boldsymbol{x} - p_4 \int_{e_3} b^{\boldsymbol{p}}_N \,\mathrm{d}\boldsymbol{x} \\ &= \tfrac{1}{2}(p_1 - p_2 + p_3 - p_4) \;.\end{aligned}$$

Similarly, if $\mathbf{b}^{\boldsymbol{p}}_{N,2}$ is the nodal basis function at $\boldsymbol{p}$ for the x_2-component of the velocity $\mathbf{v}_N$, then

$$\int\limits_\Omega \operatorname{div} \mathbf{b}^{\boldsymbol{p}}_{N,2}\, p_N \,\mathrm{d}\boldsymbol{x} = \tfrac{1}{2}(p_1 + p_2 - p_3 - p_4) \;.$$

$$p_1 = 1, p_2 = -1, p_3 = 1, p_4 = -1 \quad \Rightarrow \quad \int\limits_\Omega \operatorname{div} \mathbf{b}^{\boldsymbol{p}}_{N,1}\, p_N \,\mathrm{d}\boldsymbol{x} = \int\limits_\Omega \operatorname{div} \mathbf{b}^{\boldsymbol{p}}_{N,2}\, p_N \,\mathrm{d}\boldsymbol{x} = 0 \;. \tag{9.3.14}$$

Now, realize that the setting is translation invariant!

$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1
-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$
$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1
-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$
$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1
-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$
$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1
-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$
$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1
-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$	-1	$+1$

Fig. 483

By (9.3.14) the discrete pressure with alternating values ± 1 in checkerboard fashion will belong to $\operatorname{Kern} \mathbf{B}$ for this finite element Galerkin method (for odd K). Observation:

$$\{p_N \in Q_N \colon \mathsf{b}(\mathbf{v}_N, p_N) = 0 \quad \forall \mathbf{v}_N \in U_N\} \neq \emptyset \;.$$

= 1-dimensional space of checkerboard modes

◁ p.w. constant checkerboard mode

Example 9.3.15 (P1-P0 quadrilateral finite elements for Stokes problem)

- BVP (9.2.25) on $\Omega =]0,1[^2$, $\mu \equiv 1$, $\mathbf{f} = \cos(\pi x_1)\binom{0}{1}$, see Ex. 9.3.1
- P1-P0 finite element Galerkin discretization on equidistant tensor product quadrilateral mesh, as in Ex. 9.3.13

MATLAB Code 9.3.16: P1-P0 finite difference discretization of augmented Stokes problem

```
function [v1,v2,p] = stokesP1P0FD(N,frc)
% P1-P0 finite element discretization of Stokes problem (9.2.24) on a
% quadrilateral tensor product mesh, see Ex. 9.3.13.
% N: number of mesh cells in one coordinate direction.
% f:  function handle of type symbol64(x1,x2) to right hand side
% force field f
h = 1/N; nv = (N-1)^2; nc = N^2; % meshwidth, ♯V(M), ♯M
% Assemble system matrix from Kronecker products of 1D Galerkin matrices
% Tridiagonal 1D mass matrix for linear finite elements
M = h*spdiags(ones(N-1,3)*diag([1/6 2/3 1/6]),[-1 0 1],N-1,N-1);
% Tridiagonal 1D Galerkin matrix for d^2/dx^2, see (??)
D = spdiags(ones(N-1,3)*diag([-1 2 -1]),[-1 0 1],N-1,N-1)/h;
% 1D Galerkin matrix for p.w. linear/p.w. constant finite elements and the bilinear
% form ∫_0^1 (du/dx) q dx
G = spdiags(ones(N,2)*diag([-1 1]),[-1 0],N,N-1);
% 1D mass matrix for p.w. linear and p.w. constant finite elements
C = 0.5*h*spdiags(ones(N,2),[-1 0],N,N-1);
% constraint on pressure, see Rem. 9.2.22
Delta = kron(M,D)+kron(D,M); % 9-point stencil matrix for discrete Laplacian
divx  = kron(C,G); divy  = kron(G,C); % discrete divergence
% Complete saddle point system matrix including Lagrangian multiplier for enforcing mean zero
A = [     Delta       ,  sparse(nv,nv) ,     divx'      ,  sparse(nv,1) ;...
      sparse(nv,nv)   ,     Delta      ,     divy'      ,  sparse(nv,1) ;...
          divx        ,     divy       , sparse(nc,nc) ,  ones(nc,1); ...
       sparse(1,nv)   , sparse(1,nv)   , ones(1,nc)    ,        0
                 ];
% Assembly of right hand side
phi = zeros(2*nv+nc+1,1); x = h:h:1-h; idx = 1;
for j=1:N-1, for i=1:N-1,
    phi([idx idx+nv]) = h*h*frc(x(i),x(j)); idx = idx+1;
  end, end;
% Direct solve of (singular for even N) linear saddle point system
u = A\phi;
```

```
% Recover velocity and pressure values on the grid
v1 = rot90(reshape(u(1:nv),N-1,N-1));
v2 = rot90(reshape(u(nv+1:2*nv),N-1,N-1));
p = rot90(reshape(u(2*nv+1:end-1),N,N));
```

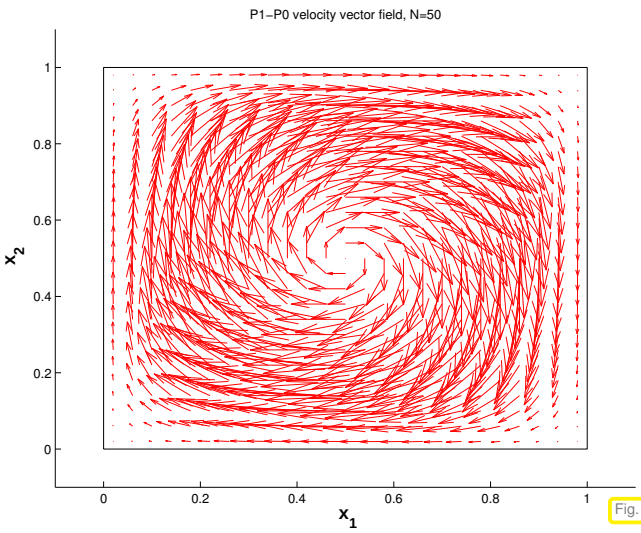


Fig. 484

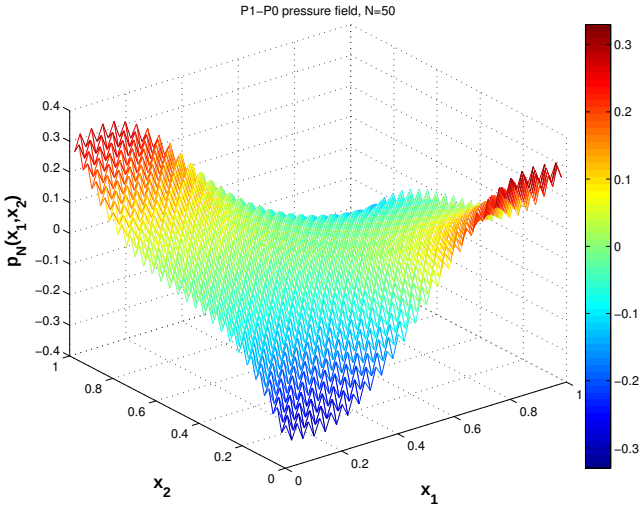


Fig. 485

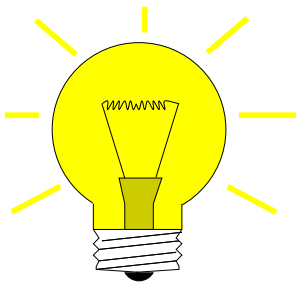
Observation: ☞ pressure solution marred by checkerboard mode
☞ computed velocity field ok!

9.3.2 Stable Galerkin discretization

In the previous examples we found a subspace of Q_N, which dodges $\operatorname{div} \mathbf{v}_N$ in the bilinear form b.

We arrive at the important heuristic insight:

$\operatorname{div} \mathbf{v}_N$ must be "large enough to fix the pressure" $p_N \in Q_N$



Idea: Use larger velocity trial/test spaces $\mathbf{v}_N$

- ➢ larger space $\operatorname{div} \mathbf{v}_N$
- ➢ more control of p_N

How to get a larger trial space for the velocity? Raise polynomial degree!

Example 9.3.17 (P2-P0 finite element scheme for the Stokes problem)

✦ $\Omega =]0,1[^2$, $\mathbf{u}(\boldsymbol{x}) = (\cos(\pi/2(x_1+x_2)), -\cos(\pi/2(x_1+x_2)))^T$, $p(\boldsymbol{x}) = \sin(\pi/2(x_1-x_2))$, $\mathbf{f}$ and inhomogeneous Dirichlet boundary values for $\mathbf{u}$ accordingly

✦ Sequence of (a) uniform triangular meshes, created by regular refinement,
(b) randomly perturbed meshes from (a) (still uniformly shape-regular & quasi-uniform).

✦ "P2-P0-scheme" velocity finite element space $U_N = (\mathcal{S}_{2,0}^0(\mathcal{M}))^2$ (continuous, piecewise quadratic $\rightarrow$ Sect. 3.5.1, Ex. 3.5.3),
pressure finite element space $Q_N = \mathcal{S}_0^{-1}(\mathcal{M}) \cap L_*^2(\Omega)$ (piecewise constant)

Monitored: Error norms $\|\mathbf{u} - \mathbf{u}_N\|_1$, $\|\mathbf{u} - \mathbf{u}_N\|_0$, $\|p - p_N\|_0$

MATLAB Code 9.3.18: LehrFEM driver script P2-P0 finite element method for Stokes problem

```
% LehrFEm driver script for computing solutions of the steady Stokes problem on the unit square
% using piecewise quadratic finite elements for the velocity and piecewise constants for the
% pressure.
  NREFS = 4; % Number of red refinements
  NU = 1;    % Viscosity
  % Dirichlet boundary data
  GD_HANDLE = @(x,varargin)[cos(pi/2*(x(:,1)+x(:,2)))-cos(pi/2*(x(:,1)+x(:,2)))];
  % Right hand side source (force field)
  F_HANDLE = @(x,varargin)[ sin(pi*x(:,1)) zeros(size(x(:,1))) ];

  % Initialize mesh
  Mesh = load_Mesh('Coord_Sqr.dat','Elem_Sqr.dat');
  Mesh.ElemFlag = ones(size(Mesh.Elements,1),1);
  Mesh = add_Edges(Mesh);
  Loc = get_BdEdges(Mesh);
  Mesh.BdFlags = zeros(size(Mesh.Edges,1),1);
  Mesh.BdFlags(Loc) = -1;
  for i = 1:NREFS, Mesh = refine_REG(Mesh); end

  % Assemble Galerkin matrix and load vector
  A = assemMat_Stokes_P2P0(Mesh,@STIMA_Stokes_P2P0,NU,P706());
  L = assemLoad_Stokes_P2P0(Mesh,P706(),F_HANDLE);

  % Incorporate Dirichlet boundary data
  [U,FreeDofs] = assemDir_Stokes_P2P0(Mesh,-1,GD_HANDLE); L = L - A*U;

  % Solve the linear system (direct solver)
  U(FreeDofs) = A(FreeDofs,FreeDofs)\L(FreeDofs);

  % Plot and print solution
  plot_Stokes(U,Mesh,'P2P0');
  title('{\bf Steady Stokes equation (P2 elements)}');
  xlabel(['{\bf # Dofs : ' int2str(size(U,1)) '}']);
  colorbar;
  print('-depsc','func_P2P0.eps')
```

MATLAB Code 9.3.19: Assembly of global Galerkin matrix for P2-P0 finite element method for Stokes problem

```
function varargout = assemMat_Stokes_P2P0(Mesh,EHandle,varargin)
% Assemble Galerking matrix for P2-P0 finite element discretization of Stokes problem
% (9.2.24): piecewise quadratic continuous velocity components and piecewise
% constant pressure approximation.
%mesh LehrFEM mesh data structure, complete with edge information,
%Sect. 3.6.2 The struct MESH must at least contain the following fields:
% COORDINATES M-by-2 matrix specifying the vertices of the mesh.
% ELEMENTS N-by-3 or matrix specifying the elements of the mesh.
% EDGES P-by-2 matrix specifying the edges of the mesh.
% ELEMFLAG N-by-1 matrix specifying additional elementinformation.
% EHandle passes function for computation of element matrix.
% See Sect. 3.6.4 for a discussion of the generic assembly algorithm
nCoordinates = size(Mesh.Coordinates,1);
nElements = size(Mesh.Elements,1);
nEdges = size(Mesh.Edges,1);
% Preallocate memory for the efficient initialization of sparse matrix, Ex. 3.3.37
I = zeros(196*nElements,1); J = zeros(196*nElements,1); A = zeros(196*nElements,1);
% Local assembly: loop over all cells of the mesh
loc = 1:196;
for i = 1:nElements
  % Extract vertex coordinates
  vidx = Mesh.Elements(i,:);
  Vertices = Mesh.Coordinates(vidx,:);
  % Compute 14 × 14 element matrix: there are 6 local shape functions for the finite
  % element space S_2^0(M), and 1 (constant) local shape function for
  % S_0^{-1}(M): 6+6+1=13 local shape functions for the P2-P0 scheme
  Aloc = EHandle(Vertices,Mesh.ElemFlag(i),varargin{:});
  % Add contributions to global Galerkin matrix: the numbering convention is a follows:
  % d.o.f. for x_1-components of the velocity are numbered first, then
  % x_2-components of the velocity, then the pressure d.o.f.
  eidx = [Mesh.Vert2Edge(vidx(1),vidx(2)) ...
          Mesh.Vert2Edge(vidx(2),vidx(3)) ...
          Mesh.Vert2Edge(vidx(3),vidx(1))];
  % Note: entries of an extra last row/column of the Galerkin matrix corresponding to
  % pressure d.o.f. are filled with one to enfore zero mean pressure, see
  % Ex. 9.2.22
  idx = [vidx,eidx+nCoordinates,...
     vidx+nCoordinates+nEdges,eidx+2*nCoordinates+nEdges,...
     i+2*(nEdges+nCoordinates),...
     2*(nEdges+nCoordinates)+nElements+1];
  I(loc) = set_Rows(idx,14); J(loc) = set_Cols(idx,14); A(loc) = Aloc(:);
  loc = loc+196;
  end

  % Assign output arguments for creation of sparse matrix
```

```
  if (nargout > 1), varargout{1} = I; varargout{2} = J;
    varargout{3} = A;
  else, varargout{1} = sparse(I,J,A); end
return
```

MATLAB Code 9.3.20: Computation of element matrix for P2-P0 finite element method for Stokes problem

```
function Aloc =
   STIMA_Stokes_P2P0(Vertices,ElemInfo,nu,QuadRule,varargin)
% Computation of element matrix for P2-P0 finite element discretization
   of 2D Stokes problem
% Vertices passes the location of the vertices of the triangle
% nu is the viscosity parameter
% QuadRule specifies local quadrature rule, see Rem. ??
% The function returns a 14 × 14 dense matrix
Aloc = zeros(14,14); % Preallocate memory
% Compute element mapping
bK = Vertices(1,:); BK = [Vertices(2,:)-bK; Vertices(3,:)-bK];
inv_BK_t = transpose(inv(BK)); det_BK = abs(det(BK));
% Compute gradients element shape functions and their values at
   quadrature points
grad_N = grad_shap_QFE(QuadRule.x);
grad_N(:,1:2) = grad_N(:,1:2)*inv_BK_t;
grad_N(:,3:4) = grad_N(:,3:4)*inv_BK_t;
grad_N(:,5:6) = grad_N(:,5:6)*inv_BK_t;
grad_N(:,7:8) = grad_N(:,7:8)*inv_BK_t;
grad_N(:,9:10) = grad_N(:,9:10)*inv_BK_t;
grad_N(:,11:12) = grad_N(:,11:12)*inv_BK_t;
% The first 6 rows/columns of the element matrix correspond to the
   x_1-component of the
% velocity.  the corresponding block of the element matrix agrees with
   that for −Δ
% discretized by means of quadratic Lagrangian finite elements.  The
   local shape functions are
% described in Ex. 3.5.3.
Aloc(1,1) =
   nu*sum(QuadRule.w.*sum(grad_N(:,1:2).*grad_N(:,1:2),2))*det_BK;
Aloc(1,2) =
   nu*sum(QuadRule.w.*sum(grad_N(:,1:2).*grad_N(:,3:4),2))*det_BK;
Aloc(1,3) =
   nu*sum(QuadRule.w.*sum(grad_N(:,1:2).*grad_N(:,5:6),2))*det_BK;
Aloc(1,4) =
   nu*sum(QuadRule.w.*sum(grad_N(:,1:2).*grad_N(:,7:8),2))*det_BK;
Aloc(1,5) =
   nu*sum(QuadRule.w.*sum(grad_N(:,1:2).*grad_N(:,9:10),2))*det_BK;
Aloc(1,6) =
   nu*sum(QuadRule.w.*sum(grad_N(:,1:2).*grad_N(:,11:12),2))*det_BK;
Aloc(2,2) =
   nu*sum(QuadRule.w.*sum(grad_N(:,3:4).*grad_N(:,3:4),2))*det_BK;
Aloc(2,3) =
```

```
   nu*sum(QuadRule.w.*sum(grad_N(:,3:4).*grad_N(:,5:6),2))*det_BK;
Aloc(2,4) =
   nu*sum(QuadRule.w.*sum(grad_N(:,3:4).*grad_N(:,7:8),2))*det_BK;
Aloc(2,5) =
   nu*sum(QuadRule.w.*sum(grad_N(:,3:4).*grad_N(:,9:10),2))*det_BK;
Aloc(2,6) =
   nu*sum(QuadRule.w.*sum(grad_N(:,3:4).*grad_N(:,11:12),2))*det_BK;
Aloc(3,3) =
   nu*sum(QuadRule.w.*sum(grad_N(:,5:6).*grad_N(:,5:6),2))*det_BK;
Aloc(3,4) =
   nu*sum(QuadRule.w.*sum(grad_N(:,5:6).*grad_N(:,7:8),2))*det_BK;
Aloc(3,5) =
   nu*sum(QuadRule.w.*sum(grad_N(:,5:6).*grad_N(:,9:10),2))*det_BK;
Aloc(3,6) =
   nu*sum(QuadRule.w.*sum(grad_N(:,5:6).*grad_N(:,11:12),2))*det_BK;
Aloc(4,4) =
   nu*sum(QuadRule.w.*sum(grad_N(:,7:8).*grad_N(:,7:8),2))*det_BK;
Aloc(4,5) =
   nu*sum(QuadRule.w.*sum(grad_N(:,7:8).*grad_N(:,9:10),2))*det_BK;
Aloc(4,6) =
   nu*sum(QuadRule.w.*sum(grad_N(:,7:8).*grad_N(:,11:12),2))*det_BK;
Aloc(5,5) =
   nu*sum(QuadRule.w.*sum(grad_N(:,9:10).*grad_N(:,9:10),2))*det_BK;
Aloc(5,6) =
   nu*sum(QuadRule.w.*sum(grad_N(:,9:10).*grad_N(:,11:12),2))*det_BK;
Aloc(6,6) =
   nu*sum(QuadRule.w.*sum(grad_N(:,11:12).*grad_N(:,11:12),2))*det_BK;
% the same for the x2-component of the velocity
Aloc(7,7) = Aloc(1,1);  Aloc(7,8) = Aloc(1,2);  Aloc(7,9) =
   Aloc(1,3);
Aloc(7,10) = Aloc(1,4);  Aloc(7,11) = Aloc(1,5);  Aloc(7,12) =
   Aloc(1,6);
Aloc(8,8) = Aloc(2,2);  Aloc(8,9) = Aloc(2,3);  Aloc(8,10) =
   Aloc(2,4);
Aloc(8,11) = Aloc(2,5);  Aloc(8,12) = Aloc(2,6);  Aloc(9,9) =
   Aloc(3,3);
Aloc(9,10) = Aloc(3,4);  Aloc(9,11) = Aloc(3,5);  Aloc(9,12) =
   Aloc(3,6);
Aloc(10,10) = Aloc(4,4);  Aloc(10,11) = Aloc(4,5);  Aloc(10,12)
   = Aloc(4,6);
Aloc(11,11) = Aloc(5,5);  Aloc(11,12) = Aloc(5,6);  Aloc(12,12)
   = Aloc(6,6);
% Interaction of pressure shape functon (constant ≡1) with velocity:
   evaluation of
% local bilinear form bK.
% First for x1-components, then for
Aloc(1,13) = sum(QuadRule.w.*grad_N(:,1))*det_BK;
Aloc(2,13) = sum(QuadRule.w.*grad_N(:,3))*det_BK;
Aloc(3,13) = sum(QuadRule.w.*grad_N(:,5))*det_BK;
Aloc(4,13) = sum(QuadRule.w.*grad_N(:,7))*det_BK;
```

```
Aloc(5,13) = sum(QuadRule.w.*grad_N(:,9))*det_BK;
Aloc(6,13) = sum(QuadRule.w.*grad_N(:,11))*det_BK;
% Next for x2-components of velocity
Aloc(7,13) = sum(QuadRule.w.*grad_N(:,2))*det_BK;
Aloc(8,13) = sum(QuadRule.w.*grad_N(:,4))*det_BK;
Aloc(9,13) = sum(QuadRule.w.*grad_N(:,6))*det_BK;
Aloc(10,13) = sum(QuadRule.w.*grad_N(:,8))*det_BK;
Aloc(11,13) = sum(QuadRule.w.*grad_N(:,10))*det_BK;
Aloc(12,13) = sum(QuadRule.w.*grad_N(:,12))*det_BK;
% Entry corresponding to zero mean multiplier
Aloc(13,14) = det_BK;
% Fill in lower triangular part
tri = triu(Aloc); Aloc = tri+tril(tri',-1);
return
```

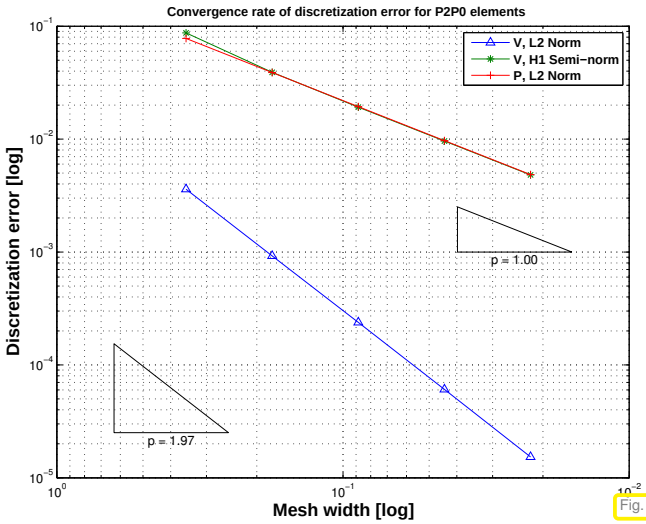


Fig. 486

Structured meshes

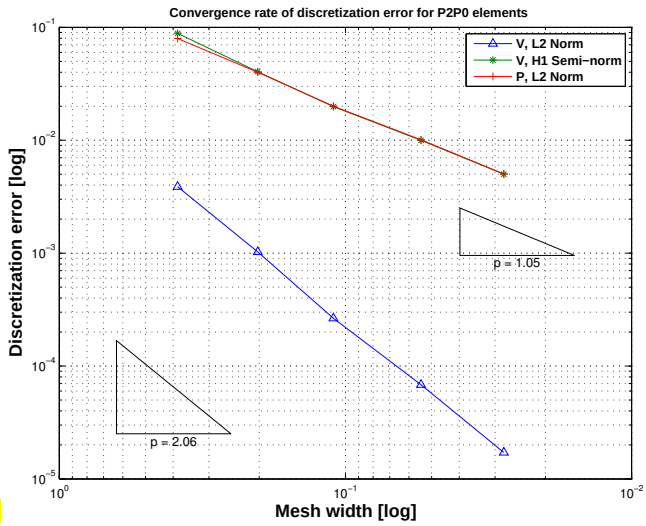


Fig. 487

Randomly perturbed meshes

Raising the polynomial degree has cured the instability!

Oberservation: algebraic convergence
$$\|\mathbf{u}-\mathbf{u}_N\|_1 = O(h_{\mathcal{M}}),$$
$$\|\mathbf{u}-\mathbf{u}_N\|_0 = O(h_{\mathcal{M}}^2),$$
$$\|p-p_N\|_0 = O(h_{\mathcal{M}}).$$

The pair $U_N = \mathcal{S}_{2,0}^0(\mathcal{M})$, $Q_N = \mathcal{S}_0^{-1}(\mathcal{M})$ is the first combination of finite element spaces that we find to provide a **stable** Galerkin discretization of the variational Stokes problem (9.2.19) $\leftrightarrow$ (9.2.14).

Recall the concept of stability/well-posedness for linear problems, see Sect. 2.4.2, "stability estimate" of Thm. 3.2.9,

$$\|\text{solution}\| \leq C\|\text{right hand side}\| \quad \text{for all data,}$$

where relevant norms have to be considered.

For the Stokes problem: relevant norms = norms of Sobolev spaces fitting (9.2.19)

For velocity $\mathbf{v}$: use "energy norm" $\|\cdot\|_{\mathsf{a}} := \mathsf{a}(\cdot,\cdot)^{1/2} \sim \|\cdot\|_{\boldsymbol{H}^1(\Omega)}$, *cf.* Def. 2.2.43

For pressure p : use $\|\cdot\|_{L^2(\Omega)}$.

Definition 9.3.21. Stable finite element pair

A pair of finite element spaces $U_N \subset \boldsymbol{H}_0^1(\Omega)$, $Q_N \subset L_*^2(\Omega)$ is a stable finite element pair, if the solution $(\mathbf{v}_N, p_N)$ of the discrete saddle point problem (9.3.5) satisfies

$$|\ell(\mathbf{w}_N)| \leq C_\ell \|\mathbf{w}_N\|_{\mathsf{a}} \; \forall \mathbf{w}_N \quad \Longrightarrow \quad \exists C > 0\colon \quad \|\mathbf{v}_N\|_{\mathsf{a}} + \|p_N\|_{L^2(\Omega)} \leq C C_\ell \,,$$

where $C > 0$ may depend only on Ω, the coefficient μ, and the shape regularity measure ($\rightarrow$ Def. 5.3.37) of $\mathcal{M}$.

We have already encountered an estimate like

$$|\ell(\mathbf{w}_N)| \leq C_\ell \|\mathbf{w}_N\|_{\mathsf{a}} \quad \forall \mathbf{w}_N \in U_N \,, \tag{9.3.22}$$

when finding that the existence of solutions of quadratic minimization problems ($\rightarrow$ Def. 2.2.32) hinges on the continuity of the involved linear form, see (2.2.55).

Let us embark on a mathematical analysis of the stability issue, which turns out to be much simpler than expected.

Remark 9.3.23 (Stable velocity solution)

Consider (9.2.19) $\leftrightarrow$ (9.2.14), and Galerkin discretization (9.3.5), define the *subspace*

$$\mathcal{N}(\mathsf{b}_N) := \{\mathbf{w}_N \in U_N \colon \mathsf{b}(\mathbf{w}_N, q_N) = 0 \; \forall q_N \in Q_N\} \subset U_N \,. \tag{9.3.24}$$

From 2nd equaton ➢ for any solution $(\mathbf{v}_N, p_N)$ of (9.3.5): $\mathbf{v}_N \in \mathcal{N}(\mathsf{b}_N)$

Test the first equation of (9.3.5) with $\mathbf{w}_N \in \mathcal{N}(\mathsf{b}_N)$

$$\blacktriangleright \quad \mathsf{a}(\mathbf{v}_N, \mathbf{w}_N) = \ell(\mathbf{w}_N) \quad \overset{\mathbf{w}_N := \mathbf{v}_N}{\Longrightarrow} \quad \|\mathbf{v}_N\|_{\mathsf{a}}^2 \leq \ell(\mathbf{v}_N) \overset{(9.3.22)}{\leq} C_\ell \|\mathbf{v}_N\|_{\mathsf{a}} \,.$$

$\blacktriangleright$ perfect stability of *any* velocity Galerkin solution

This explains the observation made in Ex. 9.3.15: reasonable approximation for velocity $\mathbf{v}$ despite pressure instability.

Remark 9.3.25 (Stability of pressure solution: inf-sup condition)

Goal: stability of pressure solution $p_N \in Q_N$ of (9.3.5)

$$\|p_N\|_{L^2(\Omega)} \leq C \sup_{\mathbf{w}_N \in U_N} \frac{\ell(\mathbf{w}_N)}{\|\mathbf{w}_N\|_{\mathsf{a}}} \tag{9.3.26}$$

best constant in (9.3.22)

From the first equation of (9.3.5)

$$\mathsf{a}(\mathbf{v}_N, \mathbf{w}_N) + \mathsf{b}(\mathbf{w}_N, p_N) = \ell(\mathbf{w}_N) \quad \forall \mathbf{w}_N \in U_N \,,$$

and the stability of the velocity solution (→ Rem. 9.3.23) we conclude (9.3.26), once we know

$$\mathsf{b}(\mathbf{w}_N, p_N) = g(\mathbf{w}_N) \quad \forall \mathbf{w}_N \in U_N \quad \Rightarrow \quad \|p_N\|_{L^2(\Omega)} \leq C \sup_{\mathbf{w}_N \in U_N} \frac{|g(\mathbf{w}_N)|}{\|\mathbf{w}_N\|_{\mathsf{a}}} \,. \tag{9.3.27}$$

Theorem 9.3.28. inf-sup condition

The finite element spaces $U_N \subset \boldsymbol{H}_0^1(\Omega)$, $Q_N \subset L_*^2(\Omega)$ provide a stable finite element pair (→ Def. 9.3.21) for the Stokes problem (9.2.19)/(9.2.14) *if there is a constant $\beta > 0$ depending* <u>*only*</u> *on Ω and the shape regularity measure (→ Def. 5.3.37) of $\mathcal{M}$ such that*

$$\sup_{\mathbf{w}_N \in U_N} \frac{|\mathsf{b}(\mathbf{w}_N, q_N)|}{\|\mathbf{w}_N\|_{\mathsf{a}}} \geq \beta \|q_N\|_{L^2(\Omega)} \quad \forall q_N \in Q_N \,. \tag{9.3.29}$$

The estimate (9.3.29) is kown as

inf-sup condition

LBB (Ladyzhenskaya-Babuska-Brezzi) condition

It is the linchpin of the numerical analysis of finite element methods for the Stokes problem, see [6].

9.3.3 Convergence

Abstract considerations (easier this way!):

- $H \hat{=}$ normed vector space, norm $\|\cdot\|$ (think of a function space),
- $\mathsf{c} : H \times H \mapsto \mathbb{R}$ bilinear form on H, *not necessarily s.p.d.* (→ Def. 2.2.40),
- $\ell : H \mapsto \mathbb{R}$ linear form on H,
- Assumption: c is continuous, *cf.* Rem. 7.2.3, (3.2.4)

$$\exists C_c > 0: \quad |\mathsf{c}(u,v)| \leq C_c \|u\| \|v\| \quad \forall u, v \in H \,. \tag{9.3.30}$$

We consider the linear variational problem (→ Rem. **??**)

$$u \in H: \quad \mathsf{c}(u,v) = \ell(v) \quad \forall v \in H \,, \tag{9.3.31}$$

and its Galerkin discretization, based on finite-dimensional subspace $H_N \subset H$, *cf.* (3.2.8),

$$u_N \in H_N: \quad \mathsf{c}(u_N, v_N) = \ell(v_N) \quad \forall v_N \in H_N \,. \tag{9.3.32}$$

Assumption: stability

$$u_N \text{ solves (9.3.32)} \quad \Longrightarrow \quad \exists C_s > 0: \quad \|u_N\| \leq \sup_{w_N \in H_N} \frac{|\ell(w_N)|}{\|w_N\|} \,. \tag{9.3.33}$$

Trick! For *any* $v_N \in H_N$ the difference $u_N - v_N$ (u_N solution of (9.3.32)) solves

$$\mathsf{c}(u_N - v_N, w_N) = \ell(w_N) - \mathsf{c}(v_N, w_N) \quad \forall w_N \in H_N \,.$$

$$\begin{aligned}\overset{(9.3.33)}{\Longrightarrow} \quad \|u_N - v_N\| &\leq C_s \sup_{w_N \in H_N} \frac{|\ell(w_N) - \mathsf{c}(v_N, w_N)|}{\|w_N\|} \\ &\overset{(9.3.31)}{=} C_s \sup_{w_N \in H_N} \frac{|\mathsf{c}(u - v_N, w_N)|}{\|w_N\|} \\ &\overset{(9.3.30)}{\leq} C_c C_s \|u - v_N\| \,. \end{aligned} \tag{9.3.34}$$

"**Trick**" Triangle inequality

$$\|u - u_N\| \leq \|u - v_N\| + \|u_N - v_N\| \overset{(9.3.34)}{\leq} (1 + C_c C_s)\|u - v_N\| \quad \forall v_N \in H_N \,.$$

$$\blacktriangleright \quad \boxed{\|u - u_N\| \leq (1 + C_c C_s) \inf_{v_N \in H_N} \|u - v_N\|} \,. \tag{9.3.35}$$

(9.3.35) is a fundamental insight into the properties of Galerkin discretizations, *cf.* Thm. 5.1.15 that was confined to s.p.d. bilinear forms:

For the Galerkin discretization of linear variational problems:

Stability $\Rightarrow$ Quasi-optimality $(*)$

Terminology: Quasi-optimality of Galerkin solutions: with $C > 0$ *independent* of data and discretization parameters

$$\underbrace{\|u - u_N\|}_{\uparrow} \leq C \underbrace{\inf_{v_N \in H_N} \|u - v_N\|}_{\uparrow} \,, \tag{9.3.36}$$

(norm of) discretization error best approximation error

Application of abstract theory to finite element discretization of Stokes problem (9.2.19):

- $H := \boldsymbol{H}_0^1(\Omega) \times L^2(\Omega)$ (combination of two function spaces!)
- Role of c played by

$$\mathsf{c}\left(\begin{pmatrix}\mathbf{v}\\ p\end{pmatrix}, \begin{pmatrix}\mathbf{w}\\ q\end{pmatrix}\right) := \mathsf{a}(\mathbf{v}, \mathbf{w}) + \mathsf{b}(\mathbf{w}, p) + \mathsf{b}(\mathbf{v}, q) \,. \tag{9.3.37}$$

- Right hand side functional "$\ell\left(\left(\begin{smallmatrix}\mathbf{w}\\ q\end{smallmatrix}\right)\right) = \ell(\mathbf{w})$"
- Galerkin trial/test space $H_N := U_N \times Q_N$.

Then, along the lines of the above abstract considerations, one can show the following a priori error estimate:

Theorem 9.3.38. Convergence of stable FE for Stokes problem

If U_N, Q_N is a stable finite element pair ($\rightarrow$ Def. 9.3.21) for the Stokes problem (9.2.19)*, then the corresponding finite element Galerkin solution $(\mathbf{v}_N, p_N)$ satisfies*

$$\|\mathbf{v}-\mathbf{v}_N\|_{H^1(\Omega)} + \|p-p_N\|_{L^2(\Omega)} \leq C\left(\inf_{\mathbf{w}_N\in U_N}\|\mathbf{v}-\mathbf{w}_N\|_{H^1(\Omega)} + \inf_{q_N\in Q_N}\|p-q_N\|_{L^2(\Omega)}\right),$$

with a constant $C > 0$ that depends <u>only</u> on Ω, μ, and the shape regularity of the finite element mesh.

Note: the a priori error bound of Thm. 9.3.38 involves the *sum* of the best approximation errors for both the velocity and pressure trial/test spaces.

Example 9.3.39 (Convergence of P2-P0 scheme for Stokes equation)

Interpretation of error curves observed in Ex. 9.3.17:

Smooth solutions for both $\mathbf{v}$ and p:

$$\text{Sect. 5.3.5} \quad \rightarrow \quad \begin{aligned} &\inf_{\mathbf{w}_N\in\mathcal{S}^0_{2,0}(\mathcal{M})}\|\mathbf{v}-\mathbf{w}_N\|_{H^1(\Omega)} \leq Ch^2_{\mathcal{M}}\|\mathbf{v}\|_{H^3(\Omega)} \quad \text{(Thm. 5.3.56)},\\ &\inf_{q_N\in\mathcal{S}^{-1}_0(\mathcal{M})}\|p-q_N\|_{L^2(\Omega)} \leq Ch_{\mathcal{M}}\|p\|_{H^1(\Omega)}, \end{aligned}$$

with constants depending *only* on the shape regularity measure ($\rightarrow$ Def. 5.3.37) of triangulation $\mathcal{M}$.

The observed $O(h)$ algebraic convergence in the $\boldsymbol{H}^1(\Omega)$-norm (for $\mathbf{v}_N$) and $L^2(\Omega)$-norm (for p_N) results, because

the larger best aproximation error of $\mathcal{S}_0^{-1}(\mathcal{M})$ dominates.

9.4 The Taylor-Hood element

A: The ultimate cure for instability

chose trial/test space for velocity large enough $\rightarrow$ very large (to play safe).

B: Well, but a large finite element space leads to a large system of linear equations, that is, high computational cost.

A: Never mind, a large space buys good accuracy, which is what we also want!

Remark 9.4.1 (Efficient finite element discretization of Stokes problem)

Thm. 9.3.38, *cf.* discussion in Ex. 9.3.39: the finite element discretization error for a stable finite element pair (U_N, Q_N) ($\rightarrow$ Def. 9.3.21) for the Stokes problem (9.2.19) is the *sum* of approximation errors for the velocity $\mathbf{v}$ in U_N and the pressure p in Q_N.

➢ Excellent approximation of either $\mathbf{v}$ or p alone may not lead to an accurate solution.

Recall similar situation for method of lines, where errors of spatial discretization and timestepping add up, see "Meta-Thms." 6.1.96, 6.2.57.

For the sake of efficiency

$$\text{balance} \quad \inf_{\mathbf{w}_N \in U_N} \|\mathbf{v} - \mathbf{w}_N\|_{H^1(\Omega)} \text{ and } + \inf_{q_N \in Q_N} \|p - q_N\|_{L^2(\Omega)}$$

Too ambitious: we have no chance of guessing the best approximation errors a priori.

Thus we settle for a more modest *asymptotic* balance condition, *cf.* the considerations in Sect. 6.1.6.

Guideline for viable and efficient choice of Galerkin finite element spaces for Stokes problem:

❶ The pair (U_N, Q_N) of finite element spaces must be stable ($\rightarrow$ Def. 9.3.21)

❷ The velocity finite element space U_N should provide the same rate of algebraic convergence of the $\boldsymbol{H}^1(\Omega)$-best approximation error w.r.t. $h_{\mathcal{M}} \to 0$ as the pressure space in $L^2(\Omega)$.

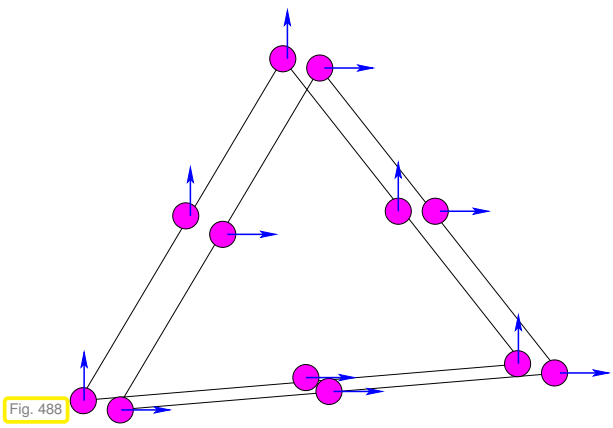
❸ The velocity finite element space U_N should guarantee ❶ and ❷ with as few degrees of freedom as possible.

Note that the stable finite element pair $(\mathcal{S}^0_{2,0}(\mathcal{M}), \mathcal{S}^{-1}_0(\mathcal{M}))$ does not meet the efficiency criterion, because the velocity space offers a better asymptotic rate of convergence than the pressure space, see Ex. 9.3.39.

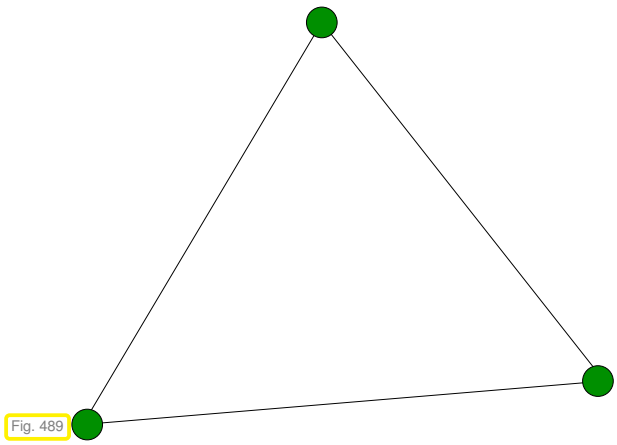
There is a stable, perfectly balanced pair of spaces:

Taylor-Hood finite element method for Stokes problem:

- $\mathcal{M}$: triangular/tetrahedral or rectangular/hexahedral mesh of Ω, may be hybrid, see Sect. 3.4.1
- Velocity space: $U_N := \mathcal{S}^0_{2,0}(\mathcal{M}) \subset \boldsymbol{H}^1_0(\Omega)$
- Pressure space: $Q_N := \mathcal{S}^0_1(\mathcal{M})$ (continuous pressure)



sites of velocity local shape functions



sites of pressure local shape functions

Balanced approximation properties of finite element spaces (for sufficiently smooth velocity and pressure solution):

$$\text{velocity:} \quad \inf_{\mathbf{w}_N \in U_N} \|\mathbf{v} - \mathbf{w}_N\|_{H^1(\Omega)} \leq C h^2_{\mathcal{M}} \|\mathbf{v}\|_{H^3(\Omega)} \quad \text{by Thm. 5.3.56,}$$

$$\text{pressure:} \quad \inf_{q_N \in \mathcal{S}^{-1}_0(\mathcal{M})} \|p - q_N\|_{L^2(\Omega)} \leq C h^2_{\mathcal{M}} \|p\|_{H^2(\Omega)} \quad \text{by Thm. 5.3.38.}$$

Theorem 9.4.2. Stability and convergence of Taylor-Hood finite element → [10]

The Taylor-Hood element provides a stable finite element pair for the Stokes problem (→ Def. 9.3.21) and for sufficiently smooth velocity and pressure solution

$$\|\mathbf{v}-\mathbf{v}_N\|_{H^1(\Omega)} + \|p-p_N\|_{L^2(\Omega)} \leq Ch_{\mathcal{M}}^2\left(\|\mathbf{v}\|_{H^3(\Omega)} + \|p\|_{H^2(\Omega)}\right) ,$$

with a constant $C > 0$ that depends only on Ω, μ, and the shape regularity of the finite element mesh.

Example 9.4.3 (Convergence of Taylor-Hood method for Stokes problem)

- Stokes problem (9.2.24) as in Ex. 9.3.17
- perturbed triangular meshes as in Ex. 9.3.17
- Taylor-Hood finite element Galerkin discretization

Monitored: Error norms $\|\mathbf{u}-\mathbf{u}_N\|_{H^1(\Omega)}$, $\|\mathbf{u}-\mathbf{u}_N\|_{L^2(\Omega)}$, $\|p-p_N\|_{L^2(\Omega)}$

Observation: algebraic convergence

$$\|\mathbf{u}-\mathbf{u}_N\|_{H^1(\Omega)} = O(h_{\mathcal{M}}^2) ,$$
$$\|\mathbf{u}-\mathbf{u}_N\|_{L^2(\Omega)} = O(h_{\mathcal{M}}^3) ,$$
$$\|p-p_N\|_{L^2(\Omega)} = O(h_{\mathcal{M}}^2) .$$

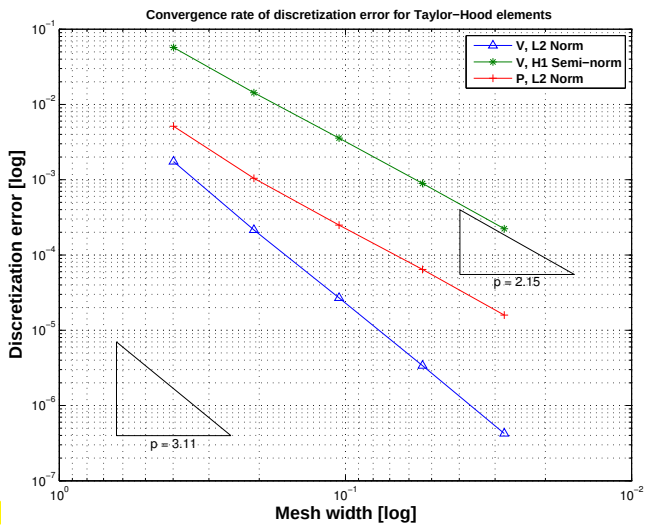


Fig. 490

Bibliography

[1] P. Bochev and R. B. Lehoucq. On the finite element solution of the pure neumann problem. *SIAM Review*, 47(1):50–66, 2005.

[2] D. Braess. *Finite Elements*. Cambridge University Press, 2nd edition, 2001.

[3] C. Brennecke, A. Linke, A. Munk, C. Merdon, and J. Schöberl. Optimal and pressure-independent L^2 velocity error estimates for a modified Crouzeix-Raviart Stokes element with BDM reconstructions. *J. Comput. Math.*, 33(2):191–208, 2015.

[4] S. Brenner and R. Scott. *Mathematical theory of finite element methods*. Texts in Applied Mathematics. Springer–Verlag, New York, 2nd edition, 2002.

[5] A. Burtscher, E. Fonn, P. Meury, and C. Wiesmayr. *LehrFEM - A 2D Finite Element Toolbox*. SAM, ETH Zürich, Zürich, Switzerland, 2010. http://www.sam.math.ethz.ch/ hiptmair/tmp/LehrFEMManual.pdf.

[6] V. Girault and P.A. Raviart. *Finite element methods for Navier–Stokes equations*. Springer, Berlin, 1986.

[7] W. Hackbusch. *Elliptic Differential Equations. Theory and Numerical Treatment*, volume 18 of *Springer Series in Computational Mathematics*. Springer, Berlin, 1992.

[8] R. Hiptmair. Numerical methods for computational science and engineering. Lecture Slides, 2015. http://www.sam.math.ethz.ch/~hiptmair/tmp/NumCSE15.pdf.

[9] L.R. Scott and M. Vogelius. Conforming finite element methods for incompressible and nearly incompressible continua. In B. Engquist, S. Osher, and R. Sommerville, editors, *Large-scale computations in fluid mechanics. Proc. 15th AMS-SIAM Summer Semin. Appl. Math., La Jolla/Calif. 1983*, volume 22 of *Lect. Appl. Math.*, pages 221–243. AMS, Providence, RI, 1985.

[10] R. Stenberg. Error analysis of some finite element methods for the stokes problem. *Math. Comp.*, 54(190):495–508, 1990.

[11] E. Zeidler. *Nonlinear Functional Analysis and its Applications. III: Variational Methods and Optimization*. Springer–Verlag, New York, Berlin, Heidelberg, 1990.

Index

List of Symbols

$C^0(\overline{\Omega}) \hat{=}$ space of functions on domain Ω, continuous up to the boundary $\partial\Omega$, 123
$C_0^2([0,1]) := \{v \in C^2([0,1]) \colon v(0) = v(1) = 0\}$, 42
$C_0^\infty(\Omega) \hat{=}$ smooth functions with support inside Ω, 142
$C^k([a,b]) \hat{=}$ k-times continuously differentiable functions on $[a,b] \subset \mathbb{R}$, 30
$C^k(\overline{\Omega}) \hat{=}$ k-times continuously differentiable funtions up to the boundary of Ω, 123
$C^k_{\mathrm{pw}}([a,b])$, 48
$D^-(\overline{x},\overline{t}) \hat{=}$ maximal analytical domain of dependence of $(\overline{x},\overline{t})$, 625
$D^{\boldsymbol{\alpha}}u \hat{=}$ multiple partial derivatives, 405
$L^2_*(\Omega) := \{q \in L^2(\Omega) \colon \int_\Omega q \,\mathrm{d}\boldsymbol{x} = 0\}$, 665
$M_i \hat{=}$ i-th integrated Legendre polynomial, 65
$O(f(N)) \hat{=}$ Landau-O for $N \to \infty$, 110
$S(z) \hat{=}$ stability function of Runge-Kutta method, 631
$\boldsymbol{n}$, 164
$\boldsymbol{n} \hat{=}$ exterior unit normal vectorfield, 156
$\boldsymbol{\mathcal{H}}_h \hat{=}$ fully discrete evolution operator, 623
$\boldsymbol{\mathcal{L}}_h \hat{=}$ semi-discrete evolution operator doe 1D conservation law, 622
$\mathcal{P}_p(\mathbb{R}) \hat{=}$ space of univariate polynomials of degree $\leq p$, 64
$\mathcal{P}_p(\mathbb{R}^d)$, 217
$\mathcal{P}_p(\mathbb{R}^d) \hat{=}$ space of d-variate polynomials, 217
$\mathcal{Q}_p(\mathbb{R}^d)$, 217
$\mathcal{V}(\mathcal{M}) \hat{=}$ set of vertices of a mesh, 188
$\Delta \hat{=}$ Laplace operator, 158
$\boldsymbol{\Delta} \hat{=}$ vector Laplacian, 667
$\mathsf{D}\, f \hat{=}$ Jacobian of a differentiable function, 16
$\operatorname{div} \mathbf{j} \hat{=}$ divergence of a vector field, 156
I_1, 399
$\Gamma_{\mathrm{in}} \hat{=}$ inflow boundary for advection BVP, 516
$\mathsf{H}\, f \hat{=}$ Hessian of a scalar valued function, 16
$H^m(\Omega) \hat{=}$ m-th order Sobolev space, 405
$\boldsymbol{H}^1_0(\operatorname{div} 0, \Omega) \hat{=}$ componentwise $H^1_0(\Omega)$-vectorfields with vanishing divergence., 662
$\mathcal{S}^0_1(\mathcal{M})$, 192
$\mathsf{I}_1 \hat{=}$ piecewise linear interpolation on finite element mesh, 372
$P_n \hat{=}$ n-th Legendre polynomial, 66
$\mathcal{S}^0_p(\mathcal{M}) \hat{=}$ $H^1(\Omega)$-conforming Lagrangian FE space, 221
$L^\infty(\Omega) \hat{=}$ space of (essentially) bounded functions on Ω, 104
$L^2(\Omega) \hat{=}$ space of square-integrable functions on Ω, 136
$\|\cdot\|_0 \hat{=}$ norm on $L^2(\Omega)$, 136
$\|\cdot\|_\infty \hat{=}$ supremum norm of a function/maximum norm of a vector, 104
$\|u\|_{H^m(\Omega)} \hat{=}$ m-th order Sobolev norm, 405
$\|u\|_{L^\infty(\Omega)} \hat{=}$ supremum norm of $u : \Omega \mapsto \mathbb{R}^n$, 104
$\|\cdot\|_{L^2(\Omega)} \hat{=}$ L^2-norm of a function, 105
$\|\cdot\|_{L^2(\Omega)} \hat{=}$ norm on $L^2(\Omega)$, 136
$\|\cdot\|_0 \hat{=}$ L^2-norm of a function, 105
$\mathcal{V}(\mathcal{M})$, 215
Ω, 120
$\Omega \hat{=}$ spatial domain or parameter domain, 30
Φ^*, 334
$|u|_{H^m(\Omega)}$ m-th order Sobolev semi-norm, 406
$|\cdot|_{H^1(\Omega)} \hat{=}$ H^1-semi-norm of a function, 106
$\mathbf{A}, \mathbf{B}, \mathbf{C}, \ldots$ (matrices), 184
$\mathbf{A} : \mathbf{B} \hat{=}$ componentwise dot product of matrices, 662
$\mathbf{S}^{-T}$ $hat=$ inverse transposed of matrix $\mathbf{S}$, 345
$\approx \hat{=}$ two-sided uniform estimate, 414
$H^1_{\Gamma_D}(\Omega) \hat{=}$ functions in $H^1(\Omega)$ with zero trace on Γ_D, 444
$\mathbb{S}^2 \hat{=}$ unit sphere, 23
$\mathsf{a}_K \hat{=}$ restriction of bilinear form a to cell K, 199
$\cdot \hat{=}$ Euclidean inner product of vectors in $\mathbb{R}^n$, 33
$\chi_I \hat{=}$ characteristic function of an interval $I \subset \mathbb{R}$, 594
$\mathbf{curl} \hat{=}$ rotation/curl of a vector field, 659
$\ddot{u} := \frac{\partial u}{\partial t^2}$, 485
$\dot{u}(t) \hat{=}$ (partial) derivative w.r.t. time, 456
ℓ_K restriction of linear form ℓ to cell K, 209
$\frac{Df}{D\mathbf{v}}(t) \hat{=}$ material derivative w.r.t. velocity field $\mathbf{v}$, 550
$\mathbf{grad} \hat{=}$ gradient of a scalar valued function, 123

$\hat{c}(\xi) \hat{=}$ symbol of a finite difference operator, 628
$\mathbf{1} = (1, \dots, 1)^T$, 631
$\mathrm{d}S \hat{=}$ integration over a surface, 156
$\mathcal{M}$, 215
$\boldsymbol{\nabla} F(\boldsymbol{x}) := \mathbf{grad}\, F(\boldsymbol{x}) \hat{=}$ nabla notation for gradient, 123
$\mathrm{diam}(\Omega) \hat{=}$ diameter of $\Omega \subset \mathbb{R}^d$, 122
nnz, 197
$\partial\Omega \hat{=}$ boundary of domain Ω, 122
$\rho_K \hat{=}$ shape regularity measure of cell K, 404
$\rho_{\mathcal{M}} \hat{=}$ shape regularity measure of a mesh $\mathcal{M}$, 404
$\vec{\boldsymbol{\mu}}, \vec{\boldsymbol{\varphi}}, \vec{\boldsymbol{\xi}}, \dots$ (coefficient vectors), 184
$\mathcal{S}^0_{p,0}(\mathcal{M}) \hat{=}$ Degree p Lagrangian finite element space with zero Dirichlet boundary conditions., 227
$\mathcal{S}^0_{1,0}(\mathcal{M}) \hat{=}$ space of p.w. linear C^0-finite elements, 80
$H^1_0(\Omega)$ Sobolev space, 140
$\boldsymbol{H}^1_0(\Omega) \hat{=}$ componentwise $H^1_0(\Omega)$-vectorfields, 662
$h_{\mathcal{M}} \hat{=}$ mesh width of mesh $\mathcal{M}$, 386
$h_{\mathcal{M}} \hat{=}$ meshwidth of a grid, 80
$x_{j-\frac{1}{2}} := \frac{1}{2}(x_j + x_{j-1}) \hat{=}$ midpoint of cell in 1D, 594
$\|\cdot\| \hat{=}$ Euclidean norm of a vector $\in \mathbb{R}^n$, 31

Examples and Remarks