ETH Lectures 401-0663-00L Numerical Methods for Computer Science
401-2673-00L Numerical Methods for CSE

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2024, Version of November 19, 2025

Link to the current version of this homework collection

0.1 General Information

0.1.1 Weekly Homework Assignments

All problems will be published in this single ".pdf" file. Every week, we publish a list of problem numbers on the lecture Moodle page. These are the current assignments for that week. Details are given on the lecture Moodle page.

0.1.2 Importance of Homework

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

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

Most homework problems come with an estimate for the time required for its solution. These times are what we expect a bright and well-prepared student to require for that problem during an exam.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by handing them to your tutor directly or uploading them via the Moodle upload interface, see the lecture Moodle page for instructions. The assistants will

also be able to examine your codes written in CODEEXPERT. You may submit your solutions even if those are incomplete and/or incorrect, and, in particular, if you have found a solution different from the master solution. Please do not submit solutions you copied from others or from the published solutions.

0.1.4 Codes and templates

For each problem involving coding you will find templates on the platform CODEEXPERT. You can access the platform with your NETHZ username and password. You have to register for the group that belongs to your tutor. Please, do not register to any other group. No local setup is required on your machine and you do not need to install any software or libraries.

Templates must be used as starting point for your solutions. You can create your own `.cpp` files, but you are strongly recommended to avoid it. CODEEXPERT will automatically test your codes and compare your solutions with the expected one. All solutions we provide will be based on templates. A similar workflow will also be used during the exam. All templates come with a `main()` function that cannot be modified: this will call other functions that you have to edit.

More information on the usage of Code Expert can be found here.

0.1.5 Hints and Solutions

Hints and solutions are kept in separate ".pdf" files. Links are supplies in this document. You can also download the ".pdf" files with hints and solutions from the website.

Chapter 0

Introduction

Problem 0-1: Simple operations with vectors and matrices in EIGEN

The problem encourages first steps in C++ coding using the EIGEN template library. The purpose of this exercise is to learn to declare and initialize **Eigen::Matrix** objects, and to get familiar with some handy typedefs and methods.

This problem involves coding in C++ and is related to [Lecture → Section 1.2.1].

The central data type in EIGEN is the matrix, of which vectors are just a special version. Matrix data type fall into three fundamental categories

(I) Variable-size and resizable dense matrices declared with the template argument `Eigen::Dynamic`,

(II) Fixed-size (small) matrices, whose size must be known at compile time,

(III) Variable size sparse matrices that are stored in special formats and which will be treated in [Lecture → Section 2.7.2].

Remark. For all codes in this problem, there are many ways how to meet the specification. Please think about it and discuss alternatives.

(0-1.a) ⚀ (10 min.) Which header file has to be included so that a code can use the (templated) type **Eigen::Matrix**?

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

(0-1.b) ⚂ (15 min.) Write a C++ function

```
Eigen::Matrix<double, 2, 2>
smallTriangular( double a, double b, double c );
```

that returns the 2×2-matrix $\left[\begin{smallmatrix} a & b \\ 0 & c \end{smallmatrix}\right]$.

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

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

(0-1.c) ⚂ (15 min.) Now, again in the file `MatrixClass.hpp`, implement the C++ function

```
Eigen::MatrixXd constantTriangular( int n, double val );
```

that initializes an $n \times n$ upper triangular matrix [Lecture → Def. 1.1.2.3], whose non-zero entries all contain the value passed in the `val` argument.

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

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

(0-1.d) ⚄ (15 min.) Vectors and matrices in EIGEN can also have integer and complex entries. The corresponding types have to be tagged with the `i` and `cd` suffix, respectively, see [Lecture → § 1.2.1.1]. Operations involving matrices/vectors with different types of entries usually entail type casting.

In the file `MatrixClass.hpp` supplement the missing parts of the function `casting()` such that it returns the real part of the Euclidean (standard) scalar product of the vectors `u` and `v`. To that end use EIGEN's dot product accessible via the method `dot()` of **Eigen::Vector**.

SOLUTION for (0-1.d) → 0-1-4-0:s4.pdf ▲

(0-1.e) ⚄ (20 min.) Given $n \in \mathbb{N}$ the matrices $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{B} \in \mathbb{C}^{n,n}$ are defined as follows

$$(\mathbf{A})_{k,\ell} := \begin{cases} 5 & \text{, if } k \geq \ell \text{ ,} \\ 0 & \text{, else} \end{cases} \quad , \quad (\mathbf{B})_{k,\ell} := \frac{k + \imath\ell}{k - \imath\ell} \quad , \quad k,\ell \in \{1,\ldots,n\} \text{ ,}$$

where $\imath \in \mathbb{C}$ is the imaginary unit, $\imath^2 = -1$. In the file `MatrixClass.hpp` realize complete the C++ function

```
Eigen::VectorXcd arithmetics(int n);
```

that evaluates the expression $\mathbf{B}(\mathbf{A} - 5\mathbf{I})[1,2,\ldots,n]^\top \in \mathbb{C}^n$.

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

SOLUTION for (0-1.e) → 0-1-5-1:s3.pdf ▲

End Problem 0-1 , 75 min.

Problem 0-2: EIGEN block operations on matrices

EIGEN matrices offer a range of methods to access sub-matrices (blocks), instead of individual entries at a time. In this exercise we practice their use.

This problem practices certain operations in EIGEN and is connected with [Lecture → § 1.2.1.5], which you should read beforehand.

Remark. The codes given as solutions may leave ample room for improvement. You should have the ambition to do better than the master solution.

(0-2.a) ⚀ (15 min.) In the file `MatrixBlocks.hpp` supplement the missing lines of the function

```
Eigen::MatrixXd
zero_row_col(const Eigen::MatrixXd &A, int p, int q );
```

so that it returns a matrix that arises from `A` by setting to zero the `p`-th row and `q`-th column.

HIDDEN HINT 1 for (0-2.a) → 0-2-1-0:t1.pdf

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

(0-2.b) ⚁ (20 min.) Again in the file `MatrixBlocks.hpp` complete the implementation of the C++ function

```
MatrixXd swap_left_right_blocks(const MatrixXd &A, int p) {
```

that returns (see [Lecture → Section 1.1.1] for explanations concerning the notations; PYTHON users beware!)

$$\left[(\mathbf{A})_{:,p+1:m},(\mathbf{A})_{:,1:p}\right] \quad \text{for} \quad \mathbf{A}\in\mathbb{R}^{n,m}\,,\quad m,n\in\mathbb{N}\,,\quad 1\le p<m\,.$$

The matrix $\mathbf{A}$ is passed through the argument `A`.

HIDDEN HINT 1 for (0-2.b) → 0-2-2-0:t2.pdf

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

(0-2.c) ⚂ (20 min.) Supplement the missing lines of the function

```
MatrixXd tridiagonal(int n, double a, double b, double c);
```

in the file `MatrixBlocks.hpp` that generates a square tridiagonal matrix

$$\begin{bmatrix} b & c & 0 & \dots & & & \dots & 0\\ a & b & c & 0 & & & & \vdots\\ 0 & \ddots & \ddots & \ddots & \ddots & & & \\ \vdots & & & & & & & \\ & & & & & & & \\ & & & & & & \ddots & \vdots\\ & & & & \ddots & \ddots & \ddots & 0\\ \vdots & & & & & a & b & c\\ 0 & \dots & & & \dots & 0 & a & b \end{bmatrix} \in \mathbb{R}^{n,n}\,,$$

with the value $b \in \mathbb{R}$ on the main diagonal, the value $a \in \mathbb{R}$ on the first sub-diagonal, and the value $c \in \mathbb{R}$ on the first super-diagonal.

HIDDEN HINT 1 for (0-2.c) → 0-2-3-0:t3.pdf

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

End Problem 0-2 , 55 min.

Problem 0-3: EIGEN's reduction operations on matrices

EIGEN's matrices have a range of methods that reduce them to a vector or even a single number, see EIGEN documentation. We will also learn about the class template **Eigen::Array**, see EIGEN documentation.

The problem practises the use of EIGEN and can be regarded as a supplement to [Lecture → Section 1.2.1].

Remark. All the coding tasks can be tackled in many different ways. The codes provided as master solution just demonstrate one way.

All functions you will be asked to complete reside in the file `MatrixReduce.hpp`.

(0-3.a) ⚀ (15 min.) Implement the C++ function

```
double average(const MatrixXd &A);
```

that computes the average

$$\frac{1}{mn}\sum_{i=1}^{n}\sum_{j=1}^{m}(\mathbf{A})_{i,j}$$

of the entries of the matrix $\mathbf{A} \in \mathbb{R}^{n,m}$, $n, m \in \mathbb{N}$. That matrix is passed as argument `A`.

HIDDEN HINT 1 for (0-3.a) → 0-3-1-0:s1h1.pdf

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

SOLUTION for (0-3.a) → 0-3-1-2:s1.pdf ▲

(0-3.b) ⚁ (20 min.) Supply the missing parts of the C++ function

```
double percent_zero(const MatrixXd &A);
```

the computes the percentage of (exact) zeros among the entries of the matrix object `A`.

HIDDEN HINT 1 for (0-3.b) → 0-3-2-0:s2h1.pdf

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

SOLUTION for (0-3.b) → 0-3-2-2:s2.pdf ▲

(0-3.c) ⚂ (20 min.) Complete the implementation of the function

```
bool has_zero_column(const MatrixXd &A);
```

that tests whether the matrix passed in `A` has a column that is exactly zero.

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

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

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

(0-3.d) ⚃ (25 min.) Finish the implementation of the C++ function

```
Eigen::MatrixXd columns_sum_to_zero(const Eigen::MatrixXd &A);
```

that returns a matrix $\mathbf{A}_0$ that agrees with the *square* matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, passed in `A` except for the diagonal. Its diagonal entries are set in way that makes all column sums vanish:

$$\sum_{i=1}^{n} (\mathbf{A}_0)_{i,j} = 0 \quad \forall j \in \{1, \ldots, n\} .$$

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

SOLUTION for (0-3.d) → `0-3-4-1:s4.pdf` ▲

End Problem 0-3 , 80 min.

Chapter 1

Computing with Matrices and Vectors

Problem 1-1: Arrow matrix×vector multiplication

Innocent looking linear algebra operation can burn considerable CPU power when implemented carelessly, see [Lecture → Code 1.3.1.11]. In this problem we study operations involving so-called arrow matrices, that is, matrices, for which only a few rows/columns and the diagonal are populated, see the `spy`-plots in [Lecture → Rem. 1.3.1.5].

This problem is related to [Lecture → Section 1.4.3] and contains implementation in C++ based on EIGEN.

Let $n \in \mathbb{N}, n > 0$. An "arrow matrix" $\mathbf{A} \in \mathbb{R}^{n \times n}$ is constructed given two vectors $\mathbf{a} \in \mathbb{R}^n$ and $\mathbf{d} \in \mathbb{R}^n$. The matrix is then squared and multiplied with a vector $\mathbf{x} \in \mathbb{R}^n$. This is implemented in the following C++ function:

C++-code 1.1.1: Computing $\mathbf{A}^2\mathbf{x}$ for an arrow matrix $\mathbf{A}$

```
2  void arrow_matrix_2_times_x(const Eigen::VectorXd &d, const Eigen::VectorXd &a,
3                              const Eigen::VectorXd &x, Eigen::VectorXd &y) {
4    assert(d.size() == a.size() && a.size() == x.size() &&
5           "Vector size must be the same!");
6    const unsigned int n = d.size();
7
8    // In this lines, we extract the blocks used to construct the matrix A.
9    Eigen::VectorXd d_head = d.head(n - 1);
10   Eigen::VectorXd a_head = a.head(n - 1);
11   Eigen::MatrixXd d_diag = d_head.asDiagonal();
12
13   Eigen::MatrixXd A(n, n);
14
15   // We build the matrix A using the "comma initialization": each expression
16   // separated by a comma is a "block" of the matrix we are building. d\_diag is
17   // the top left (n-1)x(n-1) block, a\_head is the top right vertical vector,
18   // a\_head.transpose() is the bottom left horizontal vector,
19   // d(n-1) is a single element (a 1x1 matrix) on the bottom right corner.
20   // This is how the matrix looks like:
21   // A = | D    | a      |
22   //     |------+--------|
23   //     | a\^T | d(n-1) |
24   A << d_diag, a_head, a_head.transpose(), d(n - 1);
25
26   y = A * A * x;
```

```
27 }
```

Get it on GitLab (ArrowMatrix.hpp).

(1-1.a) ⚀ (15 min.) For general vectors $\mathbf{d} = [d_1, \ldots, d_n]^\top$ and $\mathbf{a} = [a_1, \ldots, a_n]^\top$ sketch the pattern of nonzero entries of the matrix $\mathbf{A}$ created in the function `arrow_matrix_2_times_x` in Code 1.1.1. [Lecture → § 1.1.2.2] shows a way to visualize that pattern.

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

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

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

A key concern for the developer of numerical algorithms is their computational cost in case of large problems sizes, recall [Lecture → Section 1.4].

We measure the runtime of the function `arrow_matrix_2_times_x` with respect to the matrix size n and plot the results in Figure 1. (Get it on GitLab (ArrowMatrix.hpp).)
The plot shows timings for `arrow_matrix_2_times_x` (log-log plot).

Machine details: *Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz. Compiled with: gcc 6.2.1 (flags: -O3).* ▷

Give a detailed description of the behavior of the measured runtimes and an explanation for it.

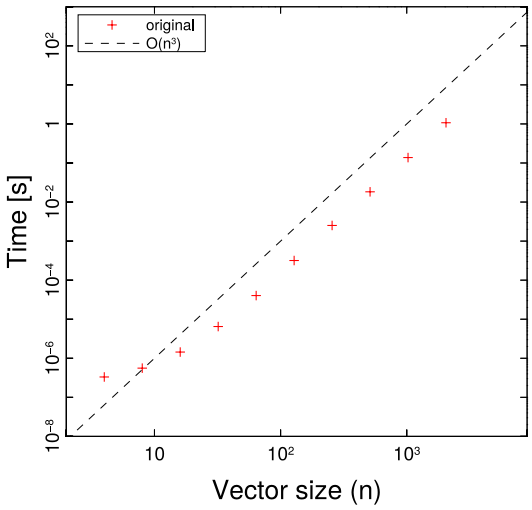


Fig. 1

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

(1-1.c) ⚂ (30 min.) [depends on Sub-problem (1-1.a)]

Write an *efficient* C++ function

```
void efficient_arrow_matrix_2_times_x(const Eigen::VectorXd &d,
                                      const Eigen::VectorXd &a,
                                      const Eigen::VectorXd &x,
                                      Eigen::VectorXd &y)
```

that computes the same product as in Code 1.1.1, but with optimal asymptotic complexity with respect to n. Compute the complexity of your code and explain why you obtain such result.

Here `d` passes the vector $[d_1, \ldots, d_n]^\top$ and `a` passes the vector $[a_1, \ldots, a_n]^\top$.

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

(1-1.d) ⚄ (10 min.) [depends on Sub-problem (1-1.c)]

What is the asymptotic complexity of your algorithm from sub-problem (1-1.c) (with respect to matrix size n)?

SOLUTION for (1-1.d) → 1-1-4-0:arrwce.pdf

▲

(1-1.e) ⚀ (30 min.) [depends on Sub-problem (1-1.c)]

Compare the runtime of your implementation and the implementation given in Code 1.1.1 for $n = 2^5, \ldots, 2^{12}$. Beware, for large n ($n > 2048$) the computations may take a long time.

Output the times in seconds, using 3 decimal digits in scientific notation. You can use **std::setw**, **std::precision(int)** and **std::scientific** from the standard library to output formatted text (include `iomanip`). For example:

```
std::cout << std::setw(8) << 1./3e4
          << std::scientific << std::setprecision(3)
          << std::setw(15) << 1./3e-9;
```

Remark: run your code using optimization flags (`-O3`). With `CMake`, you can achieve this using `-DCMAKE_BUILD_TYPE=Release` as a `CMake` option.

§1.1.6 (Measuring runtimes in a C++ code) In order to measure runtimes you have two options: either you use `std::chrono` or use the **Timer** class.

How to use `Timer`

If you want to time a code, include `timer.h`, create a new `Timer` object, as demonstrated in the following code.

C++-code 1.1.7: Usage of Timer

```
Timer t;
t.start();
// HERE CODE TO TIME
t.stop();

// Now you can get the time passed between start and stop using
t.duration();
// You can start() and stop() again, a number of times
// Ideally:  repeat experiment many times and use min() to obtain the
// fastest run
t.min();
// You can also obtain the mean():
t.mean();
```

All times will be outputted in seconds. `Timer` is simply a wrapper to `std::chrono`.

Advanced user: how to use `std::chrono`

Find the documentation of `chrono` on C++ reference.

First include the `chrono` STL header file (note: this requires C++). The `chrono` header provides the function:

C++-code 1.1.8: `now()` function

```
std::chrono::high_resolution_clock::time_point
    std::chrono::high_resolution_clock::now();
```

that returns the current time using the highest possible precision offered by the machine. For simplicity, we rename the return type:

C++-code 1.1.9: `now()` function

```
using time_point_t = std::chrono::high_resolution_clock::time_point;
// Declare a starting point and a termination time
time_point_t start, end;
```

The difference between two objects of type `time_point_t` (e.g. `end - start`) is an object of type `std::chrono::high_resolution_clock::duration`. In order to convert the `chrono`'s duration type (which, in principle, can be anything: seconds, milliseconds, ...), to a fixed duration (say nanoseconds, `std::chrono::nanoseconds`), use the `duration_cast`:

```
using duration_t = std::chrono::nanoseconds;
duration_t elapsed = std::chrono::duration_cast<duration_t>(end - start);
```

To obtain the actual number (as integer) used to represent `elasped`, use `elapsed.count()`.

Note: the data type used to represent **std**`::chrono::seconds`, **std**`::chrono::milliseconds`, etc. is of integer type. Thus, you cannot obtain fractions of this units. Make sure you use a sufficiently "refined" unit of measure (e.g. `std::chrono::nanoseconds`). ⌟

SOLUTION for (1-1.e) → `1-1-5-0:arrwc.pdf` ▲

End Problem 1-1, 100 min.

Problem 1-2: Gram-Schmidt orthonormalization with EIGEN

In this exercise we rely on EIGEN to implement a fundamental algorithm from linear algebra, Gram-Schmidt orthonormalization, see [NS02, Sect. 4.4] or [Lecture → § 1.5.1.1], in order to practice the use of EIGEN's basic vector and matrix operations.

This problem relies on [Lecture → Ex. 0.3.5.29], involves C++ implementation with EIGEN.

In [Lecture → Ex. 0.3.5.29] and [Lecture → Code 0.3.5.30] you can find an implementation of the famous Gram-Schmidt orthonormalization algorithm from linear algebra. That code makes use of a rather simple (and inefficient) implementation of vector and matrix classes. In this exercise we want to use EIGEN to implement Gram-Schmidt orthonormalization.

(1-2.a) ⚀ (15 min.) Determine the asymptotic complexity of the function `gramschmidt()` from [Lecture → Code 0.3.5.30] in terms of the matrix dimension n, if the argument matrix `A` has size $n \times n$ and no premature termination occurs.

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

(1-2.b) ⚁ (40 min.) Based on the C++ linear algebra library EIGEN [Lecture → Section 1.2.1], implement a function that performs the same computations as [Lecture → Code 0.3.5.30]:

```
Eigen::MatrixXd gram_schmidt(const Eigen::MatrixXd &A);
```

The output vectors should be returned as the columns of a matrix.

HIDDEN HINT 1 for (1-2.b) → 1-2-2-0:gss1h1.pdf

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

(1-2.c) ⚂ (25 min.) [depends on Sub-problem (1-2.b)]

Implement a function

```
bool testGramSchmidt(unsigned int n);
```

that tests your implementation by applying the function `gram_schmidt` to the matrix

$$\mathbf{A} \in \mathbb{R}^{n,n} \quad , \quad (\mathbf{A})_{i,j} := i + 2j \, , \quad 1 \leq i, j \leq n \, .$$

then checks the orthonormality of the columns of the output matrix, returning **true**, if orthonormality is confirmed.

HIDDEN HINT 1 for (1-2.c) → 1-2-3-0:sp2mn.pdf

HIDDEN HINT 2 for (1-2.c) → 1-2-3-1:sp2h2.pdf

HIDDEN HINT 3 for (1-2.c) → 1-2-3-2:sp2QQ.pdf

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

References

[NS02] K. Nipp and D. Stoffer. *Lineare Algebra*. 5th ed. Zürich: vdf Hochschulverlag, 2002 (cit. on p. 13).

End Problem 1-2 , 80 min.

Problem 1-3: Kronecker product

In [Lecture → Def. 1.4.3.7] we learned about the so-called Kronecker product. In this problem we revisit the discussion of [Lecture → Ex. 1.4.3.8].

This problem is connected with [Lecture → Section 1.4.3].

To begin with we recall the definition of the Kronecker product:

Definition [Lecture → Def. 1.4.3.7]. Kronecker product

The Kronecker product $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{l,k}$, $m,n,l,k \in \mathbb{N}$, is the $(ml) \times (nk)$-matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{11}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{ml,nk} .$$

(1-3.a) ⚀ (10 min.) Compute the Kronecker product $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ of the 2×2 matrices

$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} , \quad \mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} . \tag{1.3.1}$$

SOLUTION for (1-3.a) → 1-3-1-0:cmpm.pdf ▲

(1-3.b) ⚁ (20 min.)

Based on EIGEN, in the file `kron.hpp` implement a C++ function

```
Eigen::MatrixXd kron(const Eigen::MatrixXd & A, const
   Eigen::MatrixXd & B);
```

that computes the Kronecker product of the argument matrices `A` and `B` and returns the resulting matrix.

You are not supposed to use EIGEN's built-in Kronecker product EIGEN documentation from the "unsuported" section of the library.

HIDDEN HINT 1 for (1-3.b) → 1-3-2-0:kps2h1.pdf

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

(1-3.c) ⚁ (15 min.) [depends on Sub-problem (1-3.b)]

What is the asymptotic complexity ([Lecture → Def. 1.4.1.1]) in terms of the problem size parameter $n \to \infty$ of your implementation of `kron` from (1-3.b), when we pass $n \times n$ square matrices as arguments.

You may use the *Landau* symbol from [Lecture → Def. 1.4.1.2] to state your answer.

SOLUTION for (1-3.c) → 1-3-3-0:cmpk1.pdf ▲

(1-3.d) ⚂ (45 min.)

In general, computing the entire matrix is unnecessary. Devise an *efficient* implementation of an EIGEN-based C++ function (in the file `kron.hpp`)

```
Eigen::VectorXd kron_mult(
  const Eigen::MatrixXd & A, const Eigen::MatrixXd & B,
  const Eigen::VectorXd & x);
```

for the computation of $\mathbf{y} = (\mathbf{A} \otimes \mathbf{B})\mathbf{x}$ for *square matrices* $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}$. Do not use reshaping through the `Map` method of EIGEN's matrix classes; use access to sub-vectors by the `segment` method instead. The meaning of the arguments of `kron_mult` should be self-explanatory.

HIDDEN HINT 1 for (1-3.d) → 1-3-4-0:h1.pdf

SOLUTION for (1-3.d) → 1-3-4-1:cmpk.pdf ▲

(1-3.e) ⚀ (45 min.) Devise another efficient implementation of a C++ code for the computation of $\mathbf{y} = (\mathbf{A} \otimes \mathbf{B})\mathbf{x}$, $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n,n}$. This time use EIGEN's "matrix reshaping" functions. The function to implement in the file `kron.hpp` is

```
Eigen::VectorXd kron_reshape(
  const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
  const Eigen::VectorXd &x);
```

As its counterpart `kron_mult()` from Sub-problem (1-3.d) it is supposed to compute $\mathbf{y} = (\mathbf{A} \otimes \mathbf{B})\mathbf{x}$.

HIDDEN HINT 1 for (1-3.e) → 1-3-5-0:kph3.pdf

HIDDEN HINT 2 for (1-3.e) → 1-3-5-1:ss3rs.pdf

SOLUTION for (1-3.e) → 1-3-5-2:cmpk.pdf ▲

(1-3.f) ⚀ (45 min.) Compare the runtimes of your implementations in sub-problems (1-3.b), (1-3.d) and (1-3.e). To this end, measure the runtime of the functions `kron`, `kron_mult`, `kron_reshape`, with $n = 2^1, \ldots, 2^8$. Repeat the experiments 10 times and consider only the smallest runtime (skip the computations with `kron` for $n > 2^6$). You can use the class `Timer` or the STL class `std::chrono` C++ reference.

Report the measurements in seconds with scientific notation using 3 decimal digits.

SOLUTION for (1-3.f) → 1-3-6-0:cmpk2.pdf ▲

End Problem 1-3 , 180 min.

Problem 1-4: Fast matrix multiplication with EIGEN

[Lecture → Rem. 1.4.2.2] presents Strassen's algorithm that can achieve the multiplication of two dense square matrices of size $n = 2^k$, $k \in \mathbb{N}$, with an asymptotic complexity better than $O(n^3)$ (which is the complexity of a loop-based matrix multiplication). This problem centers around the implementation of this algorithm in EIGEN.

(1-4.a) ⚄ Using EIGEN , implement a recursive function

```
Eigen::MatrixXd strassenMatMult(const Eigen::MatrixXd &A, const
   Eigen::MatrixXd &B);
```

that uses Strassen's algorithm ([Lecture → Rem. 1.4.2.2]) to multiply the two square matrices $\mathbf{A}$ and $\mathbf{B}$ of size $n = 2^k$, $k \in \mathbb{N}$, and returns the result as output.

SOLUTION for (1-4.a) → 1-4-1-0:fastmi.pdf ▲

(1-4.b) ⚀ Validate the correctness of your code by comparing the result with EIGEN's built-in matrix multiplication.

HIDDEN HINT 1 for (1-4.b) → 1-4-2-0:hs1.pdf

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

(1-4.c) ⚁ Measure the runtime of your function `strassenMatMult` for random matrices of sizes 2^k, $k = 3, \ldots, 6$, and compare with the matrix multiplication offered by the $*$-operator of EIGEN.

You can use the class `Timer` or the STL class `std::chrono` as explained in Problem 1-1, § 1.1.6.

Recommendation: Use the optimization capabilities of your C++ compiler. With `CMake`, this can be done by appending `-DCMAKE_BUILD_TYPE=Release` to the `CMake` invocation. Please note that this also disables various integrity checks.

Warning: For big values of n, the computations may take some time. Consider reducing n while debugging.

SOLUTION for (1-4.c) → 1-4-3-0:fastmt.pdf ▲

End Problem 1-4

Problem 1-5: Householder reflections

This problem is a supplement to [Lecture → Section 1.5.1] and is related to Gram-Schmidt orthogonalization, see [Lecture → Code 1.5.1.4].

For solving this problem, it is useful (but not necessary) to remember the QR-decomposition of a matrix from linear algebra [NS02, Sect. 5.2], see also [Lecture → Thm. 3.3.3.4]. This problem is meant to practise some (advanced) linear algebra skills. It also addresses issues of asymptotic complexity, see [Lecture → Section 1.4.1]. Involves implementation based on EIGEN.

(1-5.a) ⚀ (20 min.) Let $\mathbf{v} \in \mathbb{R}^n$, $n \in \mathbb{N}, n > 0$. Consider the following algorithm given as a pseudocode:

Algorithm 1.5.1: Householder reflection

$\mathbf{w} \leftarrow \mathbf{v}/\|\mathbf{v}\|_2$
$\mathbf{u} \leftarrow \mathbf{w}$
$\mathbf{u}_1 \leftarrow \mathbf{u}_1 + 1$
$\mathbf{q} \leftarrow \mathbf{u}/\|\mathbf{u}\|_2$
$\mathbf{X} \leftarrow \mathbf{I} - 2\mathbf{q}\mathbf{q}^\top$
$\mathbf{Z} \leftarrow (\mathbf{X})_{:,2:n}$

Using the facilities of EIGEN write a C++ function with signature

```
void houserefl(const Eigen::VectorXd &v, Eigen::MatrixXd &Z);
```

that implements the algorithm from Pseudocode 1.5.1.

HIDDEN HINT 1 for (1-5.a) → 1-5-1-0:hrfH1.pdf

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

(1-5.b) ⚁ (15 min.) Show that the matrix $\mathbf{X}$, defined in Code 1.5.1, satisfies:

$$\mathbf{X}^\top\mathbf{X} = \mathbf{I}_n$$

gather* Here $\mathbf{I}_n$ is the identity matrix of size n.

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

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

(1-5.c) ⚂ (15 min.) Show that the first column of $\mathbf{X}$, in Code 1.5.1, is a multiple of the vector $\mathbf{v}$.

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

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

(1-5.d) ⚁ (10 min.) [depends on Sub-problem (1-5.b)]

What property does the set of columns of the matrix $\mathbf{Z}$ have? What is the purpose of the function `houserefl`?

SOLUTION for (1-5.d) → 1-5-4-0:propertyimpl.pdf ▲

(1-5.e) ⚀ (10 min.) [depends on Sub-problem (1-5.a)]

What is the asymptotic complexity of the function `houserefl` as the length n of the input vector $\mathbf{v}$ tends to ∞?

Specify it in leading order using the Landau symbol O.

SOLUTION for (1-5.e) → 1-5-5-0:impl.pdf ▲

References

[NS02] K. Nipp and D. Stoffer. *Lineare Algebra*. 5th ed. Zürich: vdf Hochschulverlag, 2002 (cit. on p. 17).

End Problem 1-5 , 70 min.

Problem 1-6: Matrix powers

This problems studies a (moderately) efficient way to compute large integer powers of matrices in EIGEN.

This problem practises EIGEN and also revisits the concept of asyymptotic complexity, hence is based on [Lecture → Section 1.2.1] and [Lecture → Section 1.4].

This problems deals with computing integer powers of square matrices, defined as:

$$\mathbf{A}^k := \overbrace{\mathbf{A}\cdot\ldots\cdot\mathbf{A}}^{k \text{ times}}, \quad k \in \mathbb{N}$$

for $\mathbf{A} \in \mathbb{C}^{n\times n}$ and $n \in \mathbb{N}$, $n > 0$.

(1-6.a) ⚄ (30 min.) Implement a C++ function

```
Eigen::MatrixXcd matPow(Eigen::MatrixXcd & A, unsigned int k);
```

that, using only basic linear algebra operations (including matrix-vector or matrix-matrix multiplications), computes the k^{th} power of the $n \times n$ matrix $\mathbf{A}$ as efficiently as possible.

Here, `MatrixXcd` is a *complex matrix*. A complex matrix is similar to `MatrixXd`, and differs only by the fact that if contains elements of $\mathbb{C}$.

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

Remark, see also [Lecture → § 1.2.1.1]: a `Eigen::MatrixXcd` is defined internally (in EIGEN), as:

```
using MatrixXcd = Eigen::Matrix<std::complex, Eigen::Dynamic,
  Eigen::Dynamic>;
```

where `Dynamic` is an `enum` type with value -1. A generic matrix can be defined in EIGEN as

```
Eigen::Matrix<T, rows, cols> M;
```

where `rows` and `cols` are integers (number of rows resp. columns) and `T` is the underlying type. If `rows` (resp. `cols`) is -1 (or `Dynamic`), the Matrix will have a variable number of rows (resp. columns), see [Lecture → § 1.2.1.1].

Remark: Matrix multiplication in EIGEN is not affected by aliasing issues, cf. EIGEN documentation. Therefore you can safely write `A = A*A`).

Remark: For code validatioin the EIGEN implementation of Matrix power (`MatrixXcd::pow(unsigned int)`) can be used writing:

```
#include <unsupported/Eigen/MatrixFunctions>
```

SOLUTION for (1-6.a) → 1-6-1-1:powi.pdf ▲

(1-6.b) ⚄ (20 min.) Find (in leading order and state using the Landau symbol O) the asymptotic complexity in terms of $k \to \infty$ (and $n \to \infty$) of your implementation of `matPow()` taking into account that in EIGEN a matrix-matrix multiplication requires a $O(n^3)$ computational effort for $n \to \infty$.

SOLUTION for (1-6.b) → 1-6-2-0:powc.pdf ▲

(1-6.c) ⚄ (30 min.) Compute the runtime of the EIGEN matrix power function (`Eigen::MatrixXxd::pow(int)`) (→ EIGEN documentation) and find out the complexity w.r.t

k. Compare this with the function `matPow()` from Sub-problem (1-6.a). For $n > 0$, use the complex $n \times n$ matrix $\mathbf{A}$ defined by

$$(\mathbf{A})_{j,k} := \frac{1}{\sqrt{n}} \exp\left(\frac{2\pi\imath\, jk}{n}\right), \quad i,j \in \{1,\ldots,n\},$$

to test the two functions ($\imath$ is the complex imaginary unit).

You should use the class **Timer** or the STL library **std**`::chrono` as explained in Problem 1-1.

Compute and output the runtime of the computation of the power for a matrix of size $N = 3$. The runtime should be the minimum runtime of 10 trial runs for $k = 2,\ldots,2^{31}$. Output the time is seconds, using 3 decimal digits in scientific notation. Do so in the C++ function

```
void tabulateRuntime();
```

SOLUTION for (1-6.c) → `1-6-3-0:powr.pdf` ▲

End Problem 1-6, 80 min.

Problem 1-7: Structured matrix–vector product

In [Lecture → Ex. 1.4.3.5] we saw how the particular structure of a matrix can be exploited to compute a matrix-vector product with substantially reduced computational effort. This problem presents a similar case.

Involves a moderate amount of coding in C++ based on EIGEN.

Let $n \in \mathbb{N}, n > 0$ and consider the real $n \times n$ matrix $\mathbf{A}$ defined by

$$(\mathbf{A})_{i,j} = a_{i,j} = \min\{i,j\},\ i,j = 1,\ldots,n. \tag{1.7.1}$$

The matrix-vector product $\mathbf{y} = \mathbf{A}\mathbf{x}$ can be implemented in C++ as

C++11-code 1.7.2: Computing $\mathbf{y} = \mathbf{A}\mathbf{x}$ for $\mathbf{A}$ from (1.7.1)

```
2  Eigen::VectorXd one = Eigen::VectorXd::Ones(n);
3  Eigen::VectorXd linsp = Eigen::VectorXd::LinSpaced(n, 1, n);
4  y = ((one * linsp.transpose()).cwiseMin(linsp * one.transpose())) * x;
```

Get it on GitLab (multAmin.hpp).

(1-7.a) ⚀ (10 min.) What is the asymptotic complexity (for $n \to \infty$) of the evaluation of the C++ code displayed above, with respect to the problem size parameter n?

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

(1-7.b) ⚅ (30 min.) Write an *efficient* C++ function

```
void multAmin(const Eigen::VectorXd &x, Eigen::VectorXd &y);
```

that computes the same multiplication as Code 1.7.2 but with a better asymptotic complexity with respect to n.

You can test your implementation by comparing the returned values with the ones obtained with Code 1.7.2.

HIDDEN HINT 1 for (1-7.b) → 1-7-2-0:smv:h1.pdf

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

(1-7.c) ⚂ (10 min.) What is the asymptotic complexity (in terms of problem size parameter n) of your function `multAmin`?

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

(1-7.d) ⚃ (20 min.) Compare the runtimes of your implementation and the implementation given in Code 1.7.2 for $n = 2^4, \ldots, 2^{10}$.

You can use the class `Timer` or the STL class `std::chrono` as explained in Problem 1-1, § 1.1.6.

Report the measurements in seconds with scientific notation using 3 decimal digits.

SOLUTION for (1-7.d) → 1-7-4-0:cmpk.pdf ▲

(1-7.e) ⚂ (10 min.) Consider the following C++ snippet:

C++11-code 1.7.6: Initializing $\mathbf{B}$

```
2  Eigen::MatrixXd B = Eigen::MatrixXd::Zero(n, n);
3  for (unsigned int i = 0; i < n; ++i) {
```

```
4       B(i, i) = 2;
5       if (i < n - 1) B(i + 1, i) = -1.0;
6       if (i > 0) B(i - 1, i) = -1.0;
7     }
8     B(n - 1, n - 1) = 1.0;
```

Get it on GitLab (multAmin.hpp).

Sketch the matrix $\mathbf{B}$ created by these lines.

SOLUTION for (1-7.e) → 1-7-5-0:cmpk.pdf ▲

(1-7.f) ⚂ (15 min.) With $\mathbf{A}$ from (1.7.1) and $\mathbf{B}$ from (1-7.e) write a short EIGEN-based C++ function

```
Eigen::MatrixXd multABunitv();
```

that, for $n = 10$, computes $\mathbf{ABe}_j$, $\mathbf{e}_j$ the j-th unit vector in $\mathbb{R}^n$, $j = 1,\ldots,n$, and returns the computed vectors as the columns of a matrix in addition to printing that matrix.

Formulate a conjecture based on you observations.

SOLUTION for (1-7.f) → 1-7-6-0:cmpkinv.pdf ▲

End Problem 1-7 , 95 min.

Problem 1-8: Avoiding cancellation

In [Lecture → Section 1.5.4] we saw that the so-called *cancellation phenomenon* is a major cause of numerical instability, *cf.* [Lecture → § 1.5.4.5]. Cancellation is the massive amplification of *relative errors* when subtracting two real numbers in floating point representation of about the same value. Fortunately, expressions vulnerable to cancellation can often be recast in a mathematically equivalent form that is no longer affected by cancellation, see [Lecture → § 1.5.4.16]. There, we studied several examples, and this problem gives some more.

Involves simple implementation in C++.

(1-8.a) ⚀ (15 min.) We consider the function

$$f_1(x_0,h) := \sin(x_0+h) - \sin(x_0)\ . \tag{1.8.1}$$

Derive an *equivalent* expression $f_2(x_0,h) = f_1(x_0,h)$, which no longer involves the difference of return values of trigonometric functions. In other words, f_1 and f_2 give the same values, in exact arithmetic, for any given argument values x_0 and h.

HIDDEN HINT 1 for (1-8.a) → 1-8-1-0:A1h1.pdf

SOLUTION for (1-8.a) → 1-8-1-1:sA1.pdf ▲

(1-8.b) ⚁ (30 min.) [depends on Sub-problem (1-8.a)]

Suggest a formula that avoids cancellation errors for computing the difference quotient approximation

$$f'(x) \approx \frac{f(x_0+h)-f(x_0)}{h} \tag{1.8.3}$$

of the derivative of $f(x) := \sin(x)$ at $x = x_0$.

Write a C++ function

```
void sinederv();
```

that implements

1. the difference quotient (1.8.3) directly, and
2. your new formula

and computes the resulting approximation of $f'(1.2)$, for $h = 1\cdot 10^{-20}, 1\cdot 10^{-19}, \ldots, 1$. Tabulate the relative errors of the results using $\cos(1.2)$ as exact value.

Explain the observed behaviour of the error.

HIDDEN HINT 1 for (1-8.b) → 1-8-2-0:A2h1.pdf

SOLUTION for (1-8.b) → 1-8-2-1:A2s.pdf ▲

(1-8.c) ⚂ (15 min.) Rewrite function $f(x) := \ln(x-\sqrt{x^2-1})$, $x > 1$, into a mathematically equivalent expression that is more suitable for numerical evaluation for any $x > 1$. Explain why, and provide a numerical example, which highlights the superiority of your new formula.

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

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

(1-8.d) ⚂ (15 min.) For the following expressions, state the numerical difficulties that might affect a straightforward implementation for certain values of x (which ones?), and rewrite the formulas in a way that is more suitable for numerical computation.

1. $\sqrt{x+\frac{1}{x}}-\sqrt{x-\frac{1}{x}}$, for $x>1$.
2. $\sqrt{\frac{1}{a^2}+\frac{1}{b^2}}$, for $a,b>0$.

SOLUTION for (1-8.d) → 1-8-4-0:cancsqr.pdf ▲

End Problem 1-8 , 75 min.

Problem 1-9: Complexity of a C++ function

In this problem we recall a concept from linear algebra: the diagonalization of a square matrix. Unless you can still remember what this means, please look up the chapter on "eigenvalues" in your linear algebra lecture notes. This problem also has a subtle relationship with Problem 1-6.

We consider the C++ function defined in `getit.hpp` (cf. Code 1.9.1).

C++11-code 1.9.1: Code for `getit`.

```
2  Eigen::VectorXd getit(const Eigen::MatrixXd& A, const Eigen::VectorXd& x,
3                        unsigned int k) {
4    Eigen::EigenSolver<Eigen::MatrixXd> eig =
5        Eigen::EigenSolver<Eigen::MatrixXd>(A);
6    const Eigen::VectorXcd& V = eig.eigenvalues();
7    const Eigen::MatrixXcd& W = eig.eigenvectors();
8
9    Eigen::VectorXcd cx = x.cast<std::complex<double>>();
10
11   Eigen::VectorXcd ret =
12       W * (V.array().pow(k) * (W.partialPivLu().solve(cx)).array()).matrix();
13
14   return ret.real();
15 }
```

Get it on GitLab (getit.hpp).

See the EIGEN documentation for eigenvalue decompositions in Eigen.

The operator `array()` for $v \in \mathbb{R}^n$ transforms the vector to an EIGEN array, on which operations are performed componentwise. The function `matrix()` is the "inverse" of `array()`, transforming an array to a matrix, on which vector/matrix operations are performed.

The code `W.partialPivLU().solve(cx)` performs a LU decomposition and solves the system `W*x = cx` for `x`. See the EIGEN documentation.

The `MatrixBase::cast<T>()` method applied to a matrix, will perform a componentwise cast of the matrix to a matrix of type `T`. See the EIGEN documentation.

(1-9.a) ⚄ What is the output of `getit`, when A is a diagonalizable $n \times n$ matrix, $x \in \mathbb{R}^n$ and $k \in \mathbb{N}$?

SOLUTION for (1-9.a) → `1-9-1-0:eigpoww.pdf` ▲

(1-9.b) ⚀ Fix $k \in \mathbb{N}$. Discuss (in detail) the asymptotic complexity of `getit` for $n \to \infty$.

You may use the fact that computing eigenvalues and eigenvectors of an $n \times n$ matrix has asymptotic complexity $O(n^3)$ for $n \to \infty$.

SOLUTION for (1-9.b) → `1-9-2-0:eigpowc.pdf` ▲

End Problem 1-9

Problem 1-10: Approximating the Hyperbolic Sine

In this problem, we study how Taylor expansions can be used to avoid cancellation errors in the approximation of the hyperbolic sine, *cf.* the discussion in [Lecture → Ex. 1.5.4.26].

The problem involves a little implementation in C++. The background is provided in [Lecture → Section 1.5.4].

The hyperbolic sine $\sinh()$ is the function $\sinh(x) = \frac{1}{2}(e^x - e^{-x})$, $x \in \mathbb{R}$. The following code implements a C++ function providing $\sinh(x)$ for $x \in \mathbb{R}$.

C++ code 1.10.1: Implementation of sinh

```
2 double sinh_unstable(double x) {
3   const double t = std::exp(x);
4   return .5 * (t - 1. / t);
5 }
```

Get it on GitLab (sinh.hpp).

(1-10.a) ⚀ (30 min.) Explain why the function given in Code 1.10.1 may not give a good approximation of the hyperbolic sine for small values of x, and compute the relative error

$$\epsilon_{rel} := \frac{|\texttt{sinh\_unstable}(x) - \sinh(x)|}{|\sinh(x)|}$$

with $x = 10^{-k}$, $k = 1, 2, \ldots, 10$. To that end implement a C++ function

```
void sinhError();
```

that tabulates those relative errors. As "exact value" use the result of the C++ built-in function `std::sinh` (include `cmath`).

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

(1-10.b) ⚀ (15 min.) Write the Taylor expansion with m terms around $x = 0$ of the function $x \mapsto e^x$. Specify the remainder term as in [Lecture → Ex. 1.5.4.26].

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

(1-10.c) ⚀ (15 min.) Prove that, for every $x \geq 0$ the following inequality holds true:

$$\sinh x \geq x \quad \forall x \geq 0\,. \tag{1.10.4}$$

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

(1-10.d) ⚀ (30 min.) [depends on Sub-problem (1-10.b)]

Based on **Taylor expansion**, find an approximation for $\sinh(x)$, for every $0 \leq x \leq 10^{-3}$, so that the relative approximation error ϵ_{rel} is smaller than 10^{-15}. Follow the considerations of [Lecture → Ex. 1.5.4.26].

SOLUTION for (1-10.d) → 1-10-4-0:sinhty.pdf ▲

End Problem 1-10, 90 min.

Problem 1-11: Complex roots

Universally, operations with complex numbers can be reduced to manipulating their real and imaginary parts. This problem studies how he complex square root can be realized in this way and then has to deal with potentially perilous cancellation.

This problem is related to [Lecture → Section 1.5.4] and involves a little implementation in C++.

Given a complex number $w = u + iv$, $u, v \in \mathbb{R}$ with $v \geq 0$, its root (main branch) $\sqrt{w} = x + \imath y$ can be defined by the following real and imaginary parts

$$x := \sqrt{(\sqrt{u^2+v^2}+u)/2}\,, \tag{1.11.1}$$

$$y := \sqrt{(\sqrt{u^2+v^2}-u)/2}\,, \tag{1.11.2}$$

where $\sqrt{}$ stands for the standard root function defined on $\mathbb{R}_0^+$.

(1-11.a) ⚄ (15 min.) Show that in fact $(x+\imath y)^2 = w$.

SOLUTION for (1-11.a) → 1-11-1-0:sols0cr.pdf ▲

(1-11.b) ⚅ (15 min.) For what $w \in \mathbb{C}$ will the direct implementation of (1.11.1) and (1.11.2) be vulnerable to **cancellation**?

SOLUTION for (1-11.b) → 1-11-2-0:root1.pdf ▲

(1-11.c) ⚀ (5 min.) [depends on Sub-problem (1-11.a)]

Compute xy as an expression of u and v.

SOLUTION for (1-11.c) → 1-11-3-0:root2.pdf ▲

(1-11.d) ⚅ (30 min.) Implement a function

```
std::complex<double> myroot( std::complex<double> w );
```

that computes the root of `w` as given by (1.11.1) and (1.11.2) without the risk of cancellation. You may use only real arithmetic: for instance, you may not apply the function `std::sqrt` with *complex* arguments.

SOLUTION for (1-11.d) → 1-11-4-0:root3.pdf ▲

End Problem 1-11, 65 min.

Problem 1-12: Symmetric Gauss-Seidel iteration

This problem investigates a so-called stationary linear vector iteration from the implementation point of view.

Implementation in C++ based on EIGEN is requested, [Lecture → Section 1.2.1] required

For a square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, define $\mathbf{D_A}, \mathbf{L_A}, \mathbf{U_A} \in \mathbb{R}^{n,n}$ by:

$$(\mathbf{D_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i = j, \\ 0, & i \neq j \end{cases}, (\mathbf{L_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i > j, \\ 0, & i \leq j \end{cases}, (\mathbf{U_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i < j, \\ 0, & i \geq j \end{cases}. \tag{1.12.1}$$

The symmetric Gauss-Seidel iteration associated with the linear system of equations $\mathbf{Ax} = \mathbf{b}$ is defined as

$$\mathbf{x}^{(k+1)} = (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{b} - (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}^{(k)}). \tag{1.12.2}$$

(1-12.a) ⚀ (15 min.) Give a necessary and sufficient condition on $\mathbf{A}$, such that the iteration (1.12.2) is well-defined.

SOLUTION for (1-12.a) → 1-12-1-0:gsitw.pdf ▲

(1-12.b) ⚁ (20 min.) Assume that (1.12.2) is well-defined. Show that a fixed point $\mathbf{x}$ of the iteration (1.12.2) is a solution of the linear system of equations $\mathbf{Ax} = \mathbf{b}$.

SOLUTION for (1-12.b) → 1-12-2-0:sinhex.pdf ▲

(1-12.c) ⚂ (20 min.) Implement a C++ function

```
void GSIt(const Eigen::MatrixXd& A, const Eigen::VectorXd& b,
          Eigen::VectorXd& x, double rtol);
```

solving the linear system $\mathbf{Ax} = \mathbf{b}$ using the iterative scheme (1.12.2). To that end, apply the iterative scheme to an initial guess $\mathbf{x}^{(0)}$ passed trough `x`. The approximated solution given by the final iterate is then stored in `x` as an output.

Use a correction based termination criterion with relative tolerance `rtol` based on the Euclidean vector norm.

SOLUTION for (1-12.c) → 1-12-3-0:gsitf.pdf ▲

(1-12.d) ⚃ (15 min.) Test your implementation ($n = 9$, `rtol` $= 10e-8$) with the linear system given by

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 0 & \dots & 0 \\ 2 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & 2 & 3 \end{bmatrix} \in \mathbb{R}^{n,n}, \mathbf{b} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^n \tag{1.12.11}$$

output the l^2 norm of the residual $\mathbf{b} - \mathbf{A}\widetilde{\mathbf{x}}$ of the approximated solution $\mathbf{x}$. Use $\mathbf{b}$ as your starting vector for the iteration.

To that end, implement a C++ function

```
double testGSIt(unsigned int n);
```

SOLUTION for (1-12.d) → 1-12-4-0:gsitim.pdf ▲

(1-12.e) ⚂ (15 min.) Using the same matrix $\mathbf{A}$ and the same r.h.s. vector $\mathbf{b}$ as above in Subproblem (1-12.d), we have tabulated the quantity $\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$ for $k = 1, \ldots, 20$.

k	$\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$	k	$\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$
1	0.172631	11	0.00341563
2	0.0623049	12	0.00234255
3	0.0464605	13	0.00160173
4	0.0360226	14	0.00109288
5	0.0272568	15	0.000744595
6	0.0200715	16	0.000506784
7	0.0144525	17	0.000344682
8	0.0102313	18	0.000234316
9	0.00715417	19	0.000159234
10	0.00495865	20	0.000108185

Describe qualitatively and quantitatively the convergence of the iterative scheme with respect to the number of iterations k.

SOLUTION for (1-12.e) → 1-12-5-0:gsitc.pdf ▲

End Problem 1-12, 85 min.

Problem 1-13: Structured matrix–vector product II

In this problem, you'll implement a specialized matrix-vector multiplication function, exploiting the special structure of the matrix to obtain a speed-up compared to ordinary matrix-vector multiplication.

This problem practises elementary numerical linear algebra and the use of EIGEN.

Consider the following Matrix-vector multiplication:

$$\mathbf{x} = \mathbf{A}\mathbf{y} \tag{1.13.1}$$

with

$$\mathbf{A} = \begin{bmatrix} a_1 & & & a_n \\ & \ddots & \iddots & \\ & \iddots & \ddots & \\ a_1 & & & a_n \end{bmatrix} \in \mathbb{R}^{n,n}\,, \tag{1.13.2}$$

that is

$$a_{ij} = \begin{cases} a_i & \text{, if } i = j \\ a_j & \text{. if } i = n+1-j \\ 0 & \text{, else.} \end{cases}$$

(1-13.a) ⚀ (15 min.) Implement a C++ function

```
template<class Vector>
void xmatmult(const Vector& a, const Vector& y, Vector& x);
```

that efficiently evaluates (1.13.1) for $\mathbf{A}$ as described above.

HIDDEN HINT 1 for (1-13.a) → 1-13-1-0:template.pdf

SOLUTION for (1-13.a) → 1-13-1-1:xmatmulta.pdf ▲

(1-13.b) ⚁ (10 min.) What is the complexity of your function with respect to the problem size n?

SOLUTION for (1-13.b) → 1-13-2-0:xmatmultb.pdf ▲

(1-13.c) ⚂ (20 min.) Compare the runtime of your solution to that of the matrix-vector multiplication provided by EIGEN when you represent the "X-matrix" through an **Eigen::MatrixXd** object.

Tabulate the runtimes for matrix sizes $n = 2^k$, $k \in \{1, \ldots, 14\}$.

HIDDEN HINT 1 for (1-13.c) → 1-13-3-0:xm3h1.pdf

HIDDEN HINT 2 for (1-13.c) → 1-13-3-1:initialize.pdf

SOLUTION for (1-13.c) → 1-13-3-2:xmatmultc.pdf ▲

End Problem 1-13 , 45 min.

Problem 1-14: Avoiding Cancellation by Rewriting Formulas

This problem studies two situations where careful implementation is necessary to avoid severe error amplification due to cancellation:

1. Blindly applying the well-known analytic formula for the roots of a quadratic polynomial can result in an implementation vulnerable to cancellation.
2. A code unstable for certain arguments will result from straightforwardly implementing the simple formula for the inverse function of the hyperbolic sine.

This problem revisits [Lecture → Ex. 1.5.4.17].

(1-14.a) ⚀ (15 min.) The task is to compute all real roots of the real quadratic polynomial

$$t \mapsto t^2 + \alpha t + \beta \quad \text{for given} \quad \alpha, \beta \in \mathbb{R} .$$

The following C++ code is meant to implement a function

```
Eigen::Vector2d zerosquadpol(double a, double b);
```

that computes the zeros of $t \mapsto t^2 + \alpha t + \beta$ in a stable way. The arguments `a` and `b` pass the coefficients α and β of the polynomial. Supplement the missing parts.

C++ code 1.14.1: Stable computation of real roots of a quadratic polynomial

```
1  Eigen::Vector2d zerosquadpol(double a, double b) {
2    Eigen::Vector2d z(2);
3    double D = a * a - 4 * b;
4    if (D < 0) { throw "no real zeros"; }
5    else {
6      double wD = std::sqrt(D);
7      if (          >= 0) {
8        double t =                          ;
9        z <<          ,          ;
10     } else {
11       double t =                          ;
12       z <<          ,          ;
13     }
14   }
15   return z;
16 }
```

HIDDEN HINT 1 for (1-14.a) → 1-14-1-0:srh1.pdf

HIDDEN HINT 2 for (1-14.a) → 1-14-1-1:srh2.pdf

HIDDEN HINT 3 for (1-14.a) → 1-14-1-2:srh3.pdf

SOLUTION for (1-14.a) → 1-14-1-3:srp1.pdf

▲

(1-14.b) ⚄ (15 min.) The inverse function for the hyperbolic sine $\sinh(x) = \frac{1}{2}(e^x - e^{-x})$ is

$$\operatorname{arsinh}(x) := \log(x + \sqrt{x^2+1}) , \quad x \in \mathbb{R} . \tag{1.14.4}$$

The following C++ code is to supply a *cancellation-free* implementation of $\operatorname{arsinh}(x)$ for all $x \in \mathbb{R}$:

C++ code 1.14.5: Stable evaluation of $\operatorname{arsinh}(x)$

```
1 double arsinh(double x) {
2   if (x > 0) {
3     return                                                    ;
4   } else {
5     return                                                    ;
6   }
7 }
```

HIDDEN HINT 1 for (1-14.b) → 1-14-2-0:strbh1.pdf

SOLUTION for (1-14.b) → 1-14-2-1:srhpb.pdf

▲

End Problem 1-14 , 30 min.

Problem 1-15: Machine numbers and roundoff errors

On conventional digital computers we cannot compute with genuine real numbers. Instead we have to rely on the finite and discrete set $\mathbb{M}$ of machine numbers, usually compliant with the IEEE standard 754 [Lecture → Section 1.5]. This enforces rounding, which can have drastic consequences, unless one takes special care about reorganizing evaluations in $\mathbb{R}$ properly.

This is a theoretical problems, but short C++ code snippets have to be written by hand.

Throughout we take for granted a double precision (**double**) floating point number system according to IEEE standard 754 [Lecture → § 1.5.2.7], correct rounding [Lecture → Def. 1.5.3.6], and the "axiom" of roundoff analysis [Lecture → Ass. 1.5.3.11]. As in [Lecture → Section 1.5.2] we use the symbol $\mathbb{M}$ for the set of machine numbers.

(1-15.a) ⚀ (12 min.) Assess the validity of the following statements:

	Statement	Correct ?
(A)	The spacing of elements of $\mathbb{M}$ is the smaller the larger is their magnitude.	◯ **YES** ◯ **NO**
(B)	The machine precision EPS is the largest lower bound for the relative error introduced by rounding.	◯ **YES** ◯ **NO**
(C)	The order of magnitude of EPS for **double** is 10^{-16}.	◯ **YES** ◯ **NO**
(D)	The expression `(1.0 + 0.1*std::numeric_limits<double>::epsilon() > 1.0)` evaluates to **true**.	◯ **YES** ◯ **NO**

SOLUTION for (1-15.a) → 1-15-1-0:mmss1.pdf ▲

(1-15.b) ⚄ (20 min.) The implementation of the two lambda functions in the following code is prone to cancellation for small arguments $0 < |x| \ll 1$.

C++ code 1.15.1: Problematic implementation

```
2  std::pair<double, double> cancelProne(double x) {
3    auto f = [](double x) -> double {
4      const double t = std::sin(x);
5      return (1.0 - std::cos(x)) / (t * t);
6    };
7    auto g = [](double x) -> double {
8      const double w = std::pow(1 + x * x, 1.0 / 3.0);
9      return (w - 1.0) / (x * x);
10   };
11   return {f(x), g(x)};
12 }
```

Fill the blanks in the following listing so that the completed code implements the lambda functions in a way that

(i) is equivalent to the implementation of Code 1.15.1 in the case of exact arithmetic,

(ii) and avoids amplification of roundoff errors due to cancellation.

```
std::pair<double, double> cancelFree(double x) {
  // Original:  f(x) = (1-cos x)/(sin^2 x)
  auto f = [](double x) -> double {
```

```
    const double t = std::cos(x);
    return
        
  }
  // Original g(x) = ((1+x^2)^(1/3) - 1) / x^2
  auto g = [](double x) -> double {
    const double w = std::pow(1 + x * x, 1.0 / 3.0);
    return
        
  }
  return {f(x), g(x)};
}
```

HIDDEN HINT 1 for (1-15.b) → 1-15-2-0:mnmh2.pdf

SOLUTION for (1-15.b) → 1-15-2-1:mans2.pdf ▲

End Problem 1-15 , 32 min.

Chapter 2

Direct Methods for Linear Systems of Equations

Problem 2-1: Solving a small linear system

In this problem, you will use EIGEN to solve a simple linear algebra exercise.

This problem is meant to practise the solution of linear systems of equations in EIGEN, see [Lecture → § 2.5.0.4].

We have the following information: Daniel buys 3 mangoes, 1 kiwi, 7 lychees and 2 bananas. Peter buys 2 mangoes and 1 pineapple. Julia needs 3 pomegranates and 1 mango for her fruit juice, whereas Ralf needs 5 kiwis and 1 banana for his fruit salad. Marcus buys 1 pineapple and 2 bananas. Manfred buys 20 lychees and 1 kiwi. Daniel pays CHF 11.10, Peter pays CHF 17.00 (and so spends all his pocket money), Julia pays CHF 6.10, Ralf pays CHF 5.25, Marcus pays CHF 12.50 and Manfred pays CHF 7.00.

(2-1.a) ⚀ (15 min.) Study [Lecture → § 2.5.0.8] and EIGEN documentation to learn how to solve linear systems using direct elimination solvers of EIGEN. ▲

(2-1.b) ⚁ (15 min.) Write a C++ function

```
Eigen::VectorXd fruitPrice();
```

that returns the prices of the fruits in the following order:

Mango – Kiwi – Lychee – Banana – Pomegranate – Pineapple

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

End Problem 2-1 , 30 min.

Problem 2-2: Structured linear systems with pivoting

This problem deals with a block structured system and ways to efficiently implement the solution of such system. For this problem, you should have understood Gauss elimination with partial pivoting for the solution of a linear system, see [Lecture → Section 2.3.3].

This problem is connected with [Lecture → Section 2.3.2], [Lecture → Section 2.5], and involves C++ implementation using EIGEN.

We consider a block partitioned linear system of equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}\,, \quad \mathbf{A} = \begin{bmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{C} & \mathbf{D}_2 \end{bmatrix} \in \mathbb{R}^{2n,2n}\,, \tag{2.2.1}$$

where all the $n \times n$-matrices $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{C}$ are *diagonal*. Hence, the matrix $\mathbf{A}$ can be described through three n-vectors $\mathbf{d}_1$, $\mathbf{c}$ and $\mathbf{d}_2$, which provide the diagonals of the matrix blocks. These vectors will be passed as arguments `d1`, `c`, and `d2` to the C++ codes below.

(2-2.a) ⚀ (20 min.) Which permutation of rows and columns converts the matrix into a tridiagonal matrix?

HIDDEN HINT 1 for (2-2.a) → 2-2-1-0:blse1h1.pdf

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

(2-2.b) ⚁ (20 min.) In the file `blocklsepiv.hpp` write an efficient C++ function

```
Eigen::VectorXd multA(
  const Eigen::VectorXd & d1, const Eigen::VectorXd & d2,
  const Eigen::VectorXd & c, const Eigen::VectorXd & x);
```

that returns $\mathbf{y} := \mathbf{A}\mathbf{x}$. The argument x passes a column vector $\mathbf{x} \in \mathbb{R}^{2n}$ and the other arguments the vectors defining the matrix as in (2.2.1).

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

(2-2.c) ⚂ (15 min.) Compute the LU-factors of $\mathbf{A}$ as given in (2.2.1), where you may assume that they exist.

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

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

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

When does an LU-decomposition of $\mathbf{A}$ from (2.2.1) according to [Lecture → Def. 2.3.2.3] exist?

SOLUTION for (2-2.d) → 2-2-4-0:plse2a.pdf ▲

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

Give an example of a matrix $\mathbf{A}$ with the structure as in (2.2.1)

- that possesses an LU-decomposition exists, but fails to be regular,
- for which an LU-decomposition does not exist, though it is invertible.

HIDDEN HINT 1 for (2-2.e) → 2-2-5-0:plseh2b.pdf

SOLUTION for (2-2.e) → 2-2-5-1:plse2ps.pdf ▲

(2-2.f) ⚄ (45 min.) [depends on Sub-problem (2-2.c) and Sub-problem (2-2.a)]

Write an *efficient* C++ function

```
Eigen::VectorXd solveA(
  const Eigen::VectorXd & d1, const Eigen::VectorXd & d2,
  const Eigen::VectorXd & c, const Eigen::VectorXd & b);
```

that solves $\mathbf{Ax} = \mathbf{b}$ with Gaussian elimination. The argument `b` passes the right-hand-side vector $\mathbf{b}$, whereas the other arguments provide information about the matrix $\mathbf{A}$ as defined in (2.2.1).

- You must not use EIGEN's built-in linear solvers for the full matrix.
- Test for "near singularity" of the matrix.

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

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

Remark. Probably the use of EIGEN's *sparse elimination solvers* as presented in [Lecture → Section 2.7.3] also yields an efficient implementation, because that solver should detect the special decoupling of unknowns encoded in $\mathbf{A}$. ▲

(2-2.g) ⚀ (10 min.) State the **asymptotic complexity** of your implementation of `solveA` in term of the problem size parameter $n \to \infty$.

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

(2-2.h) ⚁ Determine the asymptotic complexity of `solveA` in a numerical experiment through measuring runtimes.

As test case use $\mathbf{d}_1 := \mathbf{1}^\top = [1,\ldots,n]^\top = -\mathbf{d}_2$, $\mathbf{c} = \mathbf{1}$ and $\mathbf{b} = \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_1 \end{bmatrix}$. Tabulate and plot the runtimes (in seconds, 3 decimal digit, scientific notation) for $n = 2^\ell$, $\ell = 2,\ldots,20$.

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

End Problem 2-2 , 140 min.

Problem 2-3: Block-wise LU-decomposition

Using block matrix operations and the LU-decomposition as described in [Lecture → Rem. 2.3.2.19], you should implement a function to solve a LSE for a matrix of particular structure.

Moderate EIGEN-based implementation in C++. Connected with [Lecture → Section 2.6]

Let the matrix $\mathbf{A} \in \mathbb{R}^{n+1,n+1}$ be block-partitioned according to

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^\top & 0 \end{bmatrix}$$

where $\mathbf{v}, \mathbf{u} \in \mathbb{R}^n$ and $\mathbf{R} \in \mathbb{R}^{n,n}$ is *upper triangular* and regular.

(2-3.a) ⊡ (15 min.) Refresh yourself on the definition of the LU-decomposition of a matrix [Lecture → Def. 2.3.2.3] and study [Lecture → Rem. 2.3.2.19]. ▲

(2-3.b) ⚂ (10 min.) Find the LU-decomposition of $\mathbf{A}$.

SOLUTION for (2-3.b) → 2-3-2-0:blLUa.pdfHere it is important that $\mathbf{R}$ was assumed to be invertible. ▲

(2-3.c) ⚂ (10 min.) [depends on Sub-problem (2-3.b)]

Show that $\mathbf{A}$ is regular, if and only if $\mathbf{u}^\top \mathbf{R}^{-1}\mathbf{v} \neq 0$.

HIDDEN HINT 1 for (2-3.c) → 2-3-3-0:bh1.pdf

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

(2-3.d) ⚂ (15 min.) [depends on Sub-problem (2-3.b)]

In the file `blockLU.hpp` write a C++ function

```
Eigen::VectorXd solve_LSE(const Eigen::MatrixXd& R, const
   Eigen::VectorXd& v, const Eigen::VectorXd& u, const
   Eigen::VectorXd& b);
```

that solves $\mathbf{Ax} = \mathbf{b}$ for $\mathbf{A}$ described above and returns $\mathbf{x}$. Make use of the block-LU-decomposition you derived in Sub-problem (2-3.b) and only use elementary operations, in particular no built-in solver classes of EIGEN.

HIDDEN HINT 1 for (2-3.d) → 2-3-4-0:<tag>.pdf

SOLUTION for (2-3.d) → 2-3-4-1:blLUc.pdf ▲

End Problem 2-3 , 50 min.

Problem 2-4: Cholesky and QR decomposition

This problem is about the Cholesky and QR decomposition and the relationship between them. The Cholesky decomposition is explained in [Lecture → § 2.8.0.14]. The QR decomposition is explained in [Lecture → Section 3.3.3]. Please study these topics before tackling this problem.

This problem relies on both [Lecture → § 2.8.0.14] and [Lecture → Section 3.3.3]

(2-4.a) ⚀ Show that, for every matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ such that $\mathrm{rank}(\mathbf{A}) = n$, the product matrix $\mathbf{A}^\top\mathbf{A}$ admits a Cholesky decomposition.

HIDDEN HINT 1 for (2-4.a) → 2-4-1-0:CholeskyQR1h.pdf

SOLUTION for (2-4.a) → 2-4-1-1:CholeskyQR1s.pdf ▲

(2-4.b) ⚄ The following C++ functions with EIGEN are given:

C++11-code 2.4.1: QR-decomposition via Cholesky decomposition

```
2  void CholeskyQR(const Eigen::MatrixXd& A, Eigen::MatrixXd& R,
3                  Eigen::MatrixXd& Q) {
4    Eigen::MatrixXd AtA = A.transpose() * A;
5    Eigen::LLT<Eigen::MatrixXd> L = AtA.llt();
6    R = L.matrixL().transpose();
7    Q = R.transpose()
8            .triangularView<Eigen::Lower>()
9            .solve(A.transpose())
10           .transpose();
11   // .triangularView() template member only accesses the triangular
       part
12   // of a dense matrix and allows to easily solve linear problem
13 }
```

Get it on GitLab (choleskyQR.hpp).

C++11-code 2.4.2: Standard way to do QR-decomposition in EIGEN

```
2  void DirectQR(const Eigen::MatrixXd& A, Eigen::MatrixXd& R,
3                Eigen::MatrixXd& Q) {
4    const size_t m = A.rows();
5    const size_t n = A.cols();
6
7    Eigen::HouseholderQR<Eigen::MatrixXd> QR = A.householderQr();
8    Q = QR.householderQ() * Eigen::MatrixXd::Identity(m, std::min(m, n));
9    R = Eigen::MatrixXd::Identity(std::min(m, n), m) *
10       QR.matrixQR().triangularView<Eigen::Upper>();
11   // If A: m x n, then Q: m x m and R: m x n.
12   // If m > n, however, the extra columns of Q and extra rows of R are
       not
13   // needed.  Matlab returns this "economy-size" format calling
       "qr(A,0)", which
14   // does not compute these extra entries.  With the code above, Eigen
       is smart
15   // enough to not compute the discarded vectors.
16 }
```

Get it on GitLab (choleskyQR.hpp).

Prove that, for every matrix $\mathbf{A}$, `CholeskyQR` and `DirectQR` produce the *same* output matrices $\mathbf{Q}$ and $\mathbf{R}$, if there were no roundoff errors (Such functions are called algebraically equivalent).

HIDDEN HINT 1 for (2-4.b) → 2-4-2-0:CholeskyQR2h.pdf

SOLUTION for (2-4.b) → 2-4-2-1:CholeskyQR2s.pdf ▲

(2-4.c) ⚄ Let EPS denote the machine precision. Why does the function `CholeskyQR` from Sub-problem (2-4.b) fail to return the correct result for $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \frac{1}{2}\mathrm{EPS} & 0 \\ 0 & \frac{1}{2}\mathrm{EPS} \end{bmatrix}$?

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

HIDDEN HINT 2 for (2-4.c) → 2-4-3-1:CholeskyQR3h.pdf

SOLUTION for (2-4.c) → 2-4-3-2:CholeskyQR3s.pdf ▲

End Problem 2-4

Problem 2-5: Resistance to impedance map

In [Lecture → § 2.6.0.12], we learned about the Sherman-Morrison-Woodbury update formula [Lecture → Lemma 2.6.0.21], which allows the efficient solution of a linear system of equations after a low-rank update according to [Lecture → (2.6.0.16)], provided that the setup phase of an elimination (→ [Lecture → § 2.3.2.15]) solver has already been done for the system matrix.

This problem is based on [Lecture → Ex. 2.1.0.3] and [Lecture → Ex. 2.6.0.24] and heavily draws on techniques introduced in [Lecture → Section 2.6].

In this problem, we examine the concrete application from [Lecture → Ex. 2.6.0.24], where the update formula is key to efficient implementation. This application is the computation of the impedance of the circuit drawn in Fig. 6 as a function of a variable resistance of a *single* circuit element, the resistor between nodes ♯14 and ♯15.

With the exception of that resistor, the circuit contains only identical linear resistors with the resistance R, that is, the relationship between branch currents and voltages is $I = R^{-1}U$ throughout. Excitation is provided by a voltage V imposed at node ♯16.

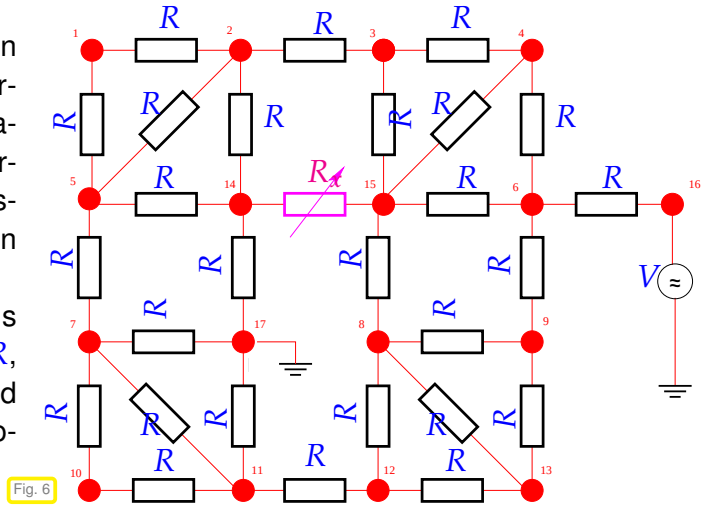


Fig. 6

Here we consider DC operation (stationary setting), that is, all branch currents, voltages, and nodal potentials are real-valued. Moreover, all currents are given in $1\mathrm{A}$, all voltages in $1\mathrm{V}$, and all resistances in $1\Omega = \frac{\mathrm{V}}{\mathrm{A}}$. Thus we can drop the physical units in the following considerations and pretend that we deal with naked numbers even when computing physical quantities.

(2-5.a) ⚀ (15 min.) Study [Lecture → Ex. 2.1.0.3] that explains how to compute voltages and currents in a linear circuit by means of nodal analysis. Understand how this leads to a linear system of equations for the unknown nodal potentials. ▲

(2-5.b) ⚁ (20 min.) Use nodal analysis to derive the 15×17 linear system of equations $\widehat{\mathbf{A}}(R_x)\mathbf{x} = \mathbf{0}$ satisfied by the nodal potentials U_i of the circuit from Fig. 6. Here, $\mathbf{x}$ stands for the vector of unknown nodal potentials: $U_i = (\mathbf{x})_i$ is that at node ♯i, $i \in \{1,\ldots,17\}$. All resistors except for the controlled one (R_x, colored magenta in Fig. 6) have the same resistance R. Use the numbering of nodes indicated in Fig. 6.

SOLUTION for (2-5.b) → 2-5-2-0:.pdf ▲

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

The nodal potential of node ♯16 is fixed to the value V by connecting it to a grounded voltage source. In addition. node ♯17 is grounded, that is, its nodal potential is set to zero.

Taking into account these constraints we arrive at a 15×15 linear system of equations $\mathbf{A}(R_x)\mathbf{x} = \mathbf{b}$ for the unknown nodal potentials $U_1,\ldots,U_{15}$, which are collected in the vector $\mathbf{x} := [U_1,\ldots,U_{15}]^\top$. Determine the system matrix $\mathbf{A} \in \mathbb{R}^{15,15}$ and the right-hand side vector $\mathbf{b} \in \mathbb{R}^{15}$.

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

(2-5.d) ⚂ (20 min.) [depends on Sub-problem (2-5.c)]

Provide an efficient implementation of the constructor and of the evaluation operator of the C++ class

```
class NodalPotentials {
```

```
  public:
  NodalPotentials() = delete;
  NodalPotentials(const NodalPotentials&) = default;
  NodalPotentials(NodalPotentials&&) = default;
  NodalPotentials& operator=(const NodalPotentials&) = default;
  NodalPotentials& operator=(NodalPotentials&&) = default;
  ~NodalPotentials() = default;

  NodalPotentials(double R, double Rx);
  Eigen::VectorXd operator () (double V) const;
  private:
  ...
};
```

This class is meant to simulate the circuit from Fig. 6. Its constructor accepts the resistances R and R_x (in Ω) as arguments. The evaluation operator should return the vector $[U_1,\ldots,U_{15}]^\top \in \mathbb{R}^{15}$ of nodal voltages when the current source excites the circuit with a voltage V.

HIDDEN HINT 1 for (2-5.d) → 2-5-4-0:h2a1.pdf

HIDDEN HINT 2 for (2-5.d) → 2-5-4-1:h2a2.pdf

SOLUTION for (2-5.d) → 2-5-4-2:s2a.pdf ▲

(2-5.e) ⚂ (20 min.) [depends on Sub-problem (2-5.c)]

Characterize the change in the circuit matrix $\mathbf{A}(R_x)$ derived in Sub-problem (2-5.c) induced by a change in the value of R_x as a low-rank modification (→ [Lecture → § 2.6.0.12]) of the circuit matrix $\mathbf{A}(0)$ of the form

$$\mathbf{A}(R_x) = \mathbf{A}(0) + \mathbf{u}\mathbf{u}^\top \quad \text{with suitable} \quad \mathbf{u} \in \mathbb{R}^{15}\,. \tag{2.5.6}$$

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

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

(2-5.f) ⚄ (15 min.) [depends on Sub-problem (2-5.e)]

How can the solution of $\mathbf{A}(R_x) = \mathbf{b}$ be computed relying on

- a solver for linear systems of equations with system matrix $\mathbf{A}(0)$,
- simple vector operations with asymptotic cost proportional to the vector length?

SOLUTION for (2-5.f) → 2-5-6-0:sc3a.pdf ▲

(2-5.g) ⚂ (45 min.) [depends on Sub-problem (2-5.f)]

Based on the EIGEN library, complete the implementation of the C++ class

```
class ImpedanceMap {
  public:
  ImpedanceMap() = delete;
  ImpedanceMap(const ImpedanceMap&) = default;
  ImpedanceMap(ImpedanceMap&&) = default;
  ImpedanceMap& operator=(const ImpedanceMap&) = default;
  ImpedanceMap& operator=(ImpedanceMap&&) = default;
  ~ImpedanceMap() = default;
```

```
  ImpedanceMap(double R, double V);
  double operator () (double Rx) const;
  private:
  ....
};
```

whose evaluation operator **operator()** returns the impedance R_* of the circuit from Fig. 6, when supplied with a concrete value for R_x. The impedance is defined as

$$R_* := \frac{V}{I_V}\,, \quad I_V := \text{current through voltage source.}$$

This function should be implemented efficiently using the result of Sub-problem (2-5.f). The setup phase for the direct solver should be carried out in the constructor, *cf.* [Lecture → § 2.3.2.15], [Lecture → Rem. 2.3.2.16].

HIDDEN HINT 1 for (2-5.g) → 2-5-7-0:h40.pdf

HIDDEN HINT 2 for (2-5.g) → 2-5-7-1:sp4h1.pdf

SOLUTION for (2-5.g) → 2-5-7-2:sc3a.pdf

▲

End Problem 2-5 , 150 min.

Problem 2-6: Banded matrix

Banded matrices are an important class of structured matrices; see [Lecture → Section 2.7.5] and [Lecture → Def. 2.7.5.1]. We will study ways to exploit during computations the knowledge that only some (sub)diagonals of a matrix are nonzero.

Connected with [Lecture → Section 1.4.3] and [Lecture → Section 2.7.5]

For $n \in \mathbb{N}$, consider the following matrix ($a_i, b_i \in \mathbb{R}$):

$$\mathbf{A} := \begin{bmatrix} 2 & a_1 & 0 & \dots & \dots & \dots & 0 \\ 0 & 2 & a_2 & 0 & \dots & \dots & 0 \\ b_1 & 0 & \ddots & \ddots & \ddots & & \vdots \\ 0 & b_2 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & a_{n-1} \\ 0 & 0 & \dots & 0 & b_{n-2} & 0 & 2 \end{bmatrix} \in \mathbb{R}^{n,n} \tag{2.6.1}$$

This matrix is an instance of a **banded matrix**.

(2-6.a) ⚀ (25 min.) In the file `efficientbandmult.hpp` implement an *efficient* C++ function `multAx` for the computation of $\mathbf{y} = \mathbf{A}\mathbf{x}$:

```
Eigen::VectorXd multAx(
  const Eigen::VectorXd & a, const VectorXd & b,
  const Eigen::VectorXd & x);
```

`Eigen::VectorXd a` and `Eigen::VectorXd b` contain the nonzero entries along the subdiagonals of matrix $\mathbf{A}$ from (2.6.1). The function is supposed to return $\mathbf{y} = \mathbf{A}\mathbf{x}$, when `Eigen::VectorXd x` supplies are the vector $\mathbf{x} \in \mathbb{R}^n$.

SOLUTION for (2-6.a) → `2-6-1-0:effbandimpl.pdf` ▲

(2-6.b) ⚄ (30 min.) Show that $\mathbf{A}$ is invertible if $a_i, b_i \in [0,1]$.

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

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

(2-6.c) ⚀ (20 min.) Assume that $b_i = 0$ for all $i = 1, \dots, n-2$. Implement an efficient C++ function solving $\mathbf{A}\mathbf{x} = \mathbf{r}$:

```
Eigen::VectorXd solvelseAupper(const Eigen::VectorXd & a, const
   Eigen::VectorXd & r);
```

`Eigen::VectorXd a` contains the nonzero entries along the upper subdiagonal of matrix $\mathbf{A}$ as defined in (2.6.1). `r` provides the right-hand-side vector of the linear system. The function should return the solution vector.

HIDDEN HINT 1 for (2-6.c) → `2-6-3-0:effbandimplh.pdf`

SOLUTION for (2-6.c) → `2-6-3-1:effbandimplsol.pdf` ▲

(2-6.d) ⚄ (60 min.) For general $a_i, b_i \in [0,1]$, implement an *efficient* C++ function that computes the solution of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{r}$ by Gaussian elimination:

```
Eigen::VectorXd solvelseA(
  const Eigen::VectorXd & a, const Eigen::VectorXd & b,
  const Eigen::VectorXd & r);
```

The vectors `a` and `b` pass information about the matrix $\mathbf{A}$ as in Sub-problem (2-6.c), whereas `r` supplies the right-hand-side vector $\mathbf{R}$. The return value is the solution $\mathbf{x}$.

You must not use any built-in linear solvers of EIGEN.

HIDDEN HINT 1 for (2-6.d) → 2-6-4-0:effbandgaussh.pdf

SOLUTION for (2-6.d) → 2-6-4-1:effbandhausssol.pdf ▲

(2-6.e) ⚀ (10 min.) [depends on Sub-problem (2-6.d)]

What is the **asymptotic complexity** of your implementation of `solvelseA` for $n \to \infty$?

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

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

(2-6.f) ⚀ Implement a C++ function

```
Eigen::VectorXd solvelseASparse(
  const Eigen::VectorXd & a, const Eigen::VectorXd & b,
  const Eigen::VectorXd & r);
```

with exactly the same specification as `solvelseA()` in Sub-problem (2-6.d), but this time using EIGEN's sparse elimination solver as explained in [Lecture → Section 2.7.3].

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

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

End Problem 2-6 , 145 min.

Problem 2-7: Properties of Householder reflections

[Lecture → Section 3.3.3] describes an orthogonal transformation that can be used to map a vector onto another vector of the same length: the Householder transformation [Lecture → (3.3.3.12)]. In this problem we examine properties of the "Householder matrices" and construct some of them.

Simple implementation in C++/EIGEN is requested

Householder transformation are described by square matrices of the form

$$\mathbf{H} = \mathbf{I} - 2\mathbf{v}\mathbf{v}^H \quad \text{with} \quad \mathbf{v}^H\mathbf{v} = 1, \quad \mathbf{v} \in \mathbb{C}^n \,. \tag{2.7.1}$$

We study a few important properties of this class of matrices and delve into the geometric meaning of Householder reflections.

(2-7.a) ⚀ Prove the following properties:

i) $\mathbf{H}\mathbf{H} = \mathbf{I}$.

ii) $\mathbf{H}\mathbf{H}^H = \mathbf{I}$.

iii) $|\det(\mathbf{H})| = 1$.

iv) For every $\mathbf{s} \in \mathbb{C}^n$ perpendicular to $\mathbf{v}$ (i.e. $\mathbf{v}^H\mathbf{s} = 0$): $\mathbf{H}\mathbf{s} = \mathbf{s}$.

v) For every $\mathbf{z} \in \mathbb{C}^n$ collinear to $\mathbf{v}$ (i.e. $\mathbf{z} = c\mathbf{v}$): $\mathbf{H}\mathbf{z} = -\mathbf{z}$.

SOLUTION for (2-7.a) → 2-7-1-0:householder1.pdf ▲

(2-7.b) ⚀

In real space the transformation $\mathbf{H}$ can be understood as a reflection across a hyperplane perpendicular to $\mathbf{v}$. Compute the matrix $\mathbf{H}$ and sketch the corresponding transformation for $\mathbf{v} = \frac{1}{\sqrt{10}}[1,3]^T$, $n = 2$. What is the line of reflection?

SOLUTION for (2-7.b) → 2-7-2-0:householder2.pdf ▲

(2-7.c) ⚀

The matrix

$$\mathbf{C} = \begin{bmatrix} 1 & 4 \\ 2 & 3 \end{bmatrix}$$

should be transformed to an upper triangular matrix using the Householder transformation ($\mathbf{H}\mathbf{C} = \mathbf{R}$).

Compute:

- the vector $\mathbf{v}$ that defines $\mathbf{H}$ (is it unique?);
- the matrix $\mathbf{H}$;
- the images of the columns of $\mathbf{C}$ under the transformation $\mathbf{H}$.

SOLUTION for (2-7.c) → 2-7-3-0:householder3.pdf ▲

(2-7.d) ⚀ Implement a function

```
void applyHouseholder(Eigen::VectorXd &x, const Eigen::VectorXd &v)
```

that efficiently computes $\mathbf{x} \leftarrow \mathbf{H}\mathbf{x}$, $\mathbf{x} \in \mathbb{R}^n$, for the Householder matrix ($\mathbf{v} \in \mathbb{R}^n$)

$$\mathbf{H} = \mathbf{I} - 2\mathbf{w}\mathbf{w}^\top \,, \quad \mathbf{w} := \frac{\mathbf{v}}{\|\mathbf{v}\|_2} \,.$$

What is the asymptotic complexity of your implementation for $n \to \infty$?

SOLUTION for (2-7.d) → 2-7-4-0:householder2.pdf ▲

(2-7.e) ⚀ As we learned in [Lecture → Rem. 3.3.3.22], the sequence of Householder transformations $\mathbf{H}_1, \ldots, \mathbf{H}_{n-1}$ effecting the conversion of an $n \times n$ matrix into upper triangular form (→ [Lecture → § 3.3.3.11]) is represented by the sequence of their defining *unit vectors* $\mathbf{v}_i \in \mathbb{R}^n$, also following the convention that $(\mathbf{v}_i)_i \geq 0$: $\mathbf{H}_i := \mathbf{I} - 2\mathbf{v}_i\mathbf{v}_i^\top$. The vectors $\mathbf{v}_i$ are stored in the *strictly lower triangular* part of another $n \times n$ matrix $\mathbf{V} \in \mathbb{R}^{n,n}$. More precisely, we have

$$\mathbf{v}_i = [0, \ldots, 0, (\mathbf{v}_i)_i, (\mathbf{V})_{i+1:n,i}]^\top \in \mathbb{R}^n \,, \quad i = 1, \ldots, n-1 \,.$$

Write a C++/EIGEN function

```
template <typename Scalar>
void applyHouseholder(Eigen::VectorXd Xd &x,
                      const Eigen::MatrixBase<Scalar> &V)
```

that computes $\mathbf{x} \leftarrow \mathbf{H}_{n-1}^{-1} \ldots \mathbf{H}_1^{-1}\mathbf{x}$ based on Householder vectors stored in $\mathbf{V}$ as described above.

HIDDEN HINT 1 for (2-7.e) → 2-7-5-0:house:mb.pdf

HIDDEN HINT 2 for (2-7.e) → 2-7-5-1:house:h1.pdf

SOLUTION for (2-7.e) → 2-7-5-2:householder2.pdf ▲

End Problem 2-7

Problem 2-8: Lyapunov equation

Any linear system of equations with a finite number of unknowns can be written in the "canonical form" $\mathbf{Ax} = \mathbf{b}$ with a system matrix $\mathbf{A}$ and a right hand side vector $\mathbf{b}$. However, the linear system may be given in a different form and it may not be obvious how to extract the system matrix. We propose an intriguing example and also present an important *matrix equation*, the so-called Lyapunov equation.

Related to [Lecture → Section 2.5] and [Lecture → Section 2.7.2]

Given the square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, we consider the equation

$$\mathbf{AX} + \mathbf{XA}^\top = \mathbf{I}\,, \tag{2.8.1}$$

where the matrix $\mathbf{X} \in \mathbb{R}^{n,n}$ is the unknown.

(2-8.a) ⚀ (10 min.) Show that for a fixed matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ the following mapping is *linear*:

$$L : \begin{cases} \mathbb{R}^{n,n} & \to & \mathbb{R}^{n,n} \\ \mathbf{X} & \mapsto & \mathbf{AX} + \mathbf{XA}^\top \end{cases}$$

HIDDEN HINT 1 for (2-8.a) → 2-8-1-0:ly1h.pdf

SOLUTION for (2-8.a) → 2-8-1-1:ly1s.pdf ▲

In the following let $\mathrm{vec}(\mathbf{M}) \in \mathbb{R}^{n^2}$ denote the column vector obtained by storing the internal coefficient array of a matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ (→ [Lecture → (1.2.3.5)]) in column major format, i.e. a data array with n^2 components [Lecture → Rem. 1.2.3.4]. In PYTHON, MATLAB, $\mathrm{vec}(\mathbf{M})$ would be the column vector obtained by **reshape** `(M,n*n,1)`, see [Lecture → Rem. 1.2.3.6] for the implementation based on EIGEN.

Owing to the observation made in Sub-problem (2-8.a), (2.8.1) is equivalent to the following linear system of equations:

$$\mathbf{C}\,\mathrm{vec}(\mathbf{X}) = \mathbf{b} \tag{2.8.2}$$

The system matrix is $\mathbf{C} \in \mathbb{R}^{n^2,n^2}$ and the right-hand side vector $\mathbf{b} \in \mathbb{R}^{n^2}$.

(2-8.b) ⚀ (10 min.) Recall the notion of *sparse matrix*: see [Lecture → Section 2.7] and, in particular, [Lecture → Notion 2.7.0.1] and [Lecture → Def. 2.7.0.3]. ▲

(2-8.c) ⚁ (15 min.) Determine $\mathbf{C}$ and $\mathbf{b}$ from (2.8.2) for $n = 2$ and

$$\mathbf{A} = \begin{bmatrix} 2 & 1 \\ -1 & 3 \end{bmatrix}$$

HIDDEN HINT 1 for (2-8.c) → 2-8-3-0:ly3h.pdf

SOLUTION for (2-8.c) → 2-8-3-1:ly3s.pdf ▲

(2-8.d) ⚄ (25 min.) Use the Kronecker product $\otimes$ [Lecture → Def. 1.4.3.7] to find a general expression for $\mathbf{C}$ in terms of $\mathbf{A}$.

SOLUTION for (2-8.d) → 2-8-4-0:ly4s.pdf ▲

(2-8.e) ⚄ (30 min.) [depends on Sub-problem (2-8.d)]

In the file `lyapunov.hpp` implement a C++ function that builds the EIGEN matrix $\mathbf{C}$ from $\mathbf{A}$:

```
Eigen::SparseMatrix<double> buildC(const Eigen::MatrixXd& A)
```

Make sure that the initialisation is done efficiently using an intermediate triplet format, see [Lecture → Section 2.7.2].

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

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

(2-8.f) ⚀ (20 min.) In the file `lyapunov.hpp` a C++ function that returns the solution of (2.8.1), i.e. the $n \times n$-matrix $\mathbf{X}$ if $\mathbf{A} \in \mathbb{R}^{n,n}$ is passed in the argument `A`.

```
void solveLyapunov(const Eigen::MatrixXd& A, Eigen::MatrixXd& X)
```

Use a suitable sparse elimination solver provided by EIGEN, see [Lecture → Section 2.7.3].

HIDDEN HINT 1 for (2-8.f) → 2-8-6-0:ly6h.pdf

SOLUTION for (2-8.f) → 2-8-6-1:ly6s.pdf ▲

(2-8.g) ⚁ (10 min.) Write a C++ function

```
bool testLyapunov();
```

that is meant to validate your implementation of `buildC` and `solveLyapunov` for $n = 5$ and:

$$\mathbf{A} = \begin{bmatrix} 10 & 2 & 3 & 4 & 5 \\ 6 & 20 & 8 & 9 & 1 \\ 1 & 2 & 30 & 4 & 5 \\ 6 & 7 & 8 & 20 & 0 \\ 1 & 2 & 3 & 4 & 10 \end{bmatrix} .$$

SOLUTION for (2-8.g) → 2-8-7-0:ly7s.pdf ▲

(2-8.h) ⚂ (15 min.) Give an upper bound (as sharp as possible) for $\mathrm{nnz}(\mathbf{C})$ in terms of $\mathrm{nnz}(\mathbf{A})$ (nnz is the number of nonzero elements). Can $\mathbf{C}$ be legitimately regarded as a sparse matrix for large n even if $\mathbf{A}$ is dense?

SOLUTION for (2-8.h) → 2-8-8-0:ly8s.pdf ▲

End Problem 2-8 , 135 min.

Problem 2-9: Partitioned Matrix

Based on the block view of matrix multiplication presented in [Lecture → § 1.3.1.13], we looked at *block elimination* for the solution of block partitioned linear systems of equations in [Lecture → § 2.6.0.2]. Also of interest are [Lecture → Rem. 2.3.2.19] and [Lecture → Rem. 2.3.2.17] where LU-factorisation is viewed from a block perspective. Closely related to this problem is [Lecture → Ex. 2.6.0.5], which you should study again as warm-up to this problem.

This problem practises block Gaussian elimination and is related to [Lecture → Section 2.6]

Let the matrix $\mathbf{A} \in \mathbb{R}^{n+1,n+1}$ be block partitioned according to

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix}, \tag{2.9.1}$$

where $\mathbf{v} \in \mathbb{R}^n$, $\mathbf{u} \in \mathbb{R}^n$, and $\mathbf{R} \in \mathbb{R}^{n \times n}$ is *upper triangular* and *regular*.

(2-9.a) ⚀ (10 min.) Give a necessary and sufficient condition for the triangular matrix $\mathbf{R}$ to be invertible.

HIDDEN HINT 1 for (2-9.a) → `2-9-1-0:s1.pdf`

SOLUTION for (2-9.a) → `2-9-1-1:partinv.pdf` ▲

(2-9.b) ⚁ (15 min.) Determine expressions for the sub-vector $\mathbf{z} \in \mathbb{R}^n$ and the last component $\xi \in \mathbb{R}$ of the solution vector of the linear system of equations

$$\begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \beta \end{bmatrix} \tag{2.9.2}$$

for arbitrary $\mathbf{b} \in \mathbb{R}^n, \beta \in \mathbb{R}$.

Use block Gaussian elimination as presented in [Lecture → § 2.6.0.2].

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

(2-9.c) ⚁ (10 min.) [depends on Sub-problem (2-9.b)]

Show that $\mathbf{A}$ is regular if and only if $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$.

SOLUTION for (2-9.c) → `2-9-3-0:partareg.pdf` ▲

(2-9.d) ⚂ (25 min.) [depends on Sub-problem (2-9.b)]

Implement the C++ function

```
void solvelse(
  const Eigen::MatrixXd & R,
  const Eigen::VectorXd & v,
  const Eigen::VectorXd & u,
  const Eigen::VectorXd & bb,
  Eigen::VectorXd & x);
```

for the efficient computation of the solution of $\mathbf{Ax} = \mathbf{b}$ (with $\mathbf{A}$ as in (2.9.1)), which is to be returned in the argument `x`. Perform size checks on input matrices and vectors.

Use the decomposition from Sub-problem (2-9.b). You can rely on the `triangularView()` function to inform EIGEN of the triangular structure of $\mathbf{R}$, see [Lecture → Code 1.2.1.6].

SOLUTION for (2-9.d) → `2-9-4-0:partimpl.pdf` ▲

(2-9.e) ⚀ (20 min.) [depends on Sub-problem (2-9.d)]

Code a C++ function

```
bool testSolveLSE(const Eigen::MatrixXd & R,
const Eigen::VectorXd & v, const Eigen::VectorXd & u,
const Eigen::VectorXd & b, Eigen::VectorXd & x);
```

that tests your implementation of `solvelse` by comparison with a standard LU-solver provided by EIGEN. Return **true**, if the results "are the same".

HIDDEN HINT 1 for (2-9.e) → 2-9-5-0:s3h1.pdf

HIDDEN HINT 2 for (2-9.e) → 2-9-5-1:s3h2.pdf

SOLUTION for (2-9.e) → 2-9-5-2:parttest.pdf ▲

(2-9.f) ⚁ (10 min.) [depends on Sub-problem (2-9.d)]

What is the asymptotic complexity of your implementation of `solvelse()` in terms of problem size parameter $n \to \infty$?

SOLUTION for (2-9.f) → 2-9-6-0:partcmp.pdf ▲

End Problem 2-9 , 90 min.

Problem 2-10: Rank-one perturbations

This exercise centers around an application of the Sherman-Morrison-Woodbury formula (see [Lecture → Lemma 2.6.0.21]). Please carefully revise [Lecture → § 2.6.0.12] in the lecture notes.

Linked to [Lecture → Rem. 2.5.0.10] and [Lecture → § 2.6.0.12], related problem Problem 2-5

Consider the following pseudocode:

Algorithm 2.10.1: Code `rankoneinvit`

Require: $\mathbf{d} \in \mathbb{R}^n, tol \in \mathbb{R}, tol > 0$
Ensure: l_{min}

$\mathbf{ev} \leftarrow \mathbf{d}$; $l_{\min} \leftarrow 0$; $l_{\text{new}} \leftarrow \min|\mathbf{d}|$;
while $|l_{\text{new}} - l_{\min}| > tol \cdot l_{\min}$ **do**
  $l_{\min} \leftarrow l_{\text{new}}$;
  $\mathbf{M} \leftarrow \operatorname{diag}(\mathbf{d}) + \mathbf{ev} \cdot \mathbf{ev}^\top$;
  $\mathbf{ev} \leftarrow \mathbf{M}^{-1}\mathbf{ev}$;
  $\mathbf{ev} \leftarrow \mathbf{ev}/|\mathbf{ev}|$;
  $l_{\text{new}} \leftarrow \mathbf{ev}^\top \mathbf{M}\mathbf{ev}$;
end while
return l_{new}

Here diag designates a mapping $\mathbb{R}^n \mapsto \mathbb{R}^{n,n}$ that converts a vector into a diagonal matrix with the vector components as diagonal entries.

We assume that this code does something meaningful in a larger context that we do not care about. It might be taken from an old code that we have to upgrade.

(2-10.a) ⚀ (30 min.) Directly port the function `rankoneinvit` (Algorithm 2.10.1) to C++ using EIGEN. To that end create a function

```
double rankoneinvit(const Eigen::VectorXd &d, const double &tol);
```

The C++ code should perform exactly the same computations. Recall that in EIGEN the `asDiagonal()` method converts a vector into a diagonal matrix, see EIGEN documentation. EIGEN also offers the `normalize()` method for vectors.

HIDDEN HINT 1 for (2-10.a) → `2-10-1-0:s1h1.pdf`

SOLUTION for (2-10.a) → `2-10-1-1:smwps.pdf` ▲

(2-10.b) ⚀ (10 min.) What is the asymptotic complexity of the body of the loop in function `rankoneinvit`?

HIDDEN HINT 1 for (2-10.b) → `2-10-2-0:smwch.pdf`

SOLUTION for (2-10.b) → `2-10-2-1:smwcs.pdf` ▲

(2-10.c) ⚄ (45 min.) [depends on Sub-problem (2-10.a)]

Implement an EIGEN-based C++ function

```
double rankoneinvit_fast(const Eigen::VectorXd &d, const double
  &tol);
```

that is *algebraically equivalent* to `rankoneinvit()` from, but enjoys a way better asymptotic complexity by avoiding any invocation of Gaussian elimination/LU-factorization.

HIDDEN HINT 1 for (2-10.c) → 2-10-3-0:smweh.pdf

SOLUTION for (2-10.c) → 2-10-3-1:smwes.pdf ▲

(2-10.d) ⚀ (10 min.) [depends on Sub-problem (2-10.c)]

What is the asymptotic complexity of the execution of the loop body in your implementation of `rankoneinvit_fast()` from Sub-problem (2-10.c)?

SOLUTION for (2-10.d) → 2-10-4-0:smwecs.pdf ▲

(2-10.e) ⚁ (30 min.) Tabulate the runtimes of the two C++ implementations with different vector sizes $n = 2^p$, with $p = 2,3,\ldots,8$. As test vector `d` use `Eigen::VectorXd::LinSpaced(n,1,2)`. Estimate the laws of the different asymptotic complexities of the two implementations using the measured runtimes.

SOLUTION for (2-10.e) → 2-10-5-0:smwcds.pdf ▲

End Problem 2-10 , 125 min.

Problem 2-11: Sequential linear systems

Consider a sequence of linear systems when all the linear systems share the same matrix $\mathbf{A}$: $\mathbf{A}\mathbf{x} = \mathbf{b}_i$. The computational cost of this problem may be reduced by performing an LU decomposition of $\mathbf{A}$ only once and reusing it for the different systems.

Related to [Lecture → Rem. 2.5.0.10]

Consider the following pseudocode with input data $\mathbf{A} \in \mathbb{R}^{n\times n}$ and $\mathbf{b} \in \mathbb{R}^n$:

Algorithm 2.11.1: Code `solvepermb`
Require: Matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, vector $\mathbf{b} \in \mathbb{R}^n$
Ensure: Matrix $\mathbf{X} = [\mathbf{X}_1, \ldots, \mathbf{X}_n] \in \mathbb{R}^{n\times n}$, $\mathbf{X}_i \in \mathbb{R}^n$
while not all cyclic permutations of $\mathbf{b}$ tested yet **do**
$\mathbf{b} \leftarrow [b_n, b_1, b_2, \ldots, b_{n-1}]^\top$
$\mathbf{X}_i = \mathbf{A}^{-1}\mathbf{b}$
end while

(2-11.a) ⚀ What is the worst case asymptotic complexity of the function `solvepermb` for $n \to \infty$?

HIDDEN HINT 1 for (2-11.a) → 2-11-1-0:permch.pdf

SOLUTION for (2-11.a) → 2-11-1-1:permcs.pdf ▲

(2-11.b) ⚁ Port the function `solvepermb` (Code 2.11.1) to C++ using EIGEN.

HIDDEN HINT 1 for (2-11.b) → 2-11-2-0:permph.pdf

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

(2-11.c) ⚂ Design an efficient C++ implementation of function `solvepermb` using EIGEN with asymptotic complexity $O(n^3)$.

HIDDEN HINT 1 for (2-11.c) → 2-11-3-0:permeh.pdf

SOLUTION for (2-11.c) → 2-11-3-1:permes.pdf ▲

End Problem 2-11

Problem 2-12: Structured linear systems

In this problem we come across the example of a structured matrix, for which a linear system can be solved very efficiently, though this is not obvious.

Related to [Lecture → Section 2.6]

Consider the linear system $\mathbf{Ax} = \mathbf{b}$, where the $n \times n$ matrix $\mathbf{A}$ has the following structure:

$$\mathbf{A} = \begin{bmatrix} a_1 & 0 & 0 & \dots & 0 \\ a_1 & a_2 & 0 & \ddots & \vdots \\ a_1 & a_2 & a_3 & \ddots & \\ \vdots & \vdots & & \ddots & 0 \\ a_1 & a_2 & a_3 & \dots & a_n \end{bmatrix}$$

(2-12.a) ⊡ Give necessary and sufficient conditions for the vector $\mathbf{a} = (a_1, \dots, a_n)^\top \in \mathbb{R}^n$ such that the matrix $\mathbf{A}$ is non-singular.

HIDDEN HINT 1 for (2-12.a) → 2-12-1-0:structuredLSE1h.pdf

SOLUTION for (2-12.a) → 2-12-1-1:structuredLSE1s.pdf ▲

(2-12.b) ⊡ Write a C++ function that builds the matrix $\mathbf{A}$ given the vector $\mathbf{a}$:

```
Eigen::MatrixXd buildA(const Eigen::VectorXd & a);
```

You can use EIGEN classes for your code.

HIDDEN HINT 1 for (2-12.b) → 2-12-2-0:structuredLSE2h.pdf

SOLUTION for (2-12.b) → 2-12-2-1:structuredLSE2s.pdf ▲

(2-12.c) ⊡ Implement a function in C++ which solves the linear system $\mathbf{Ax} = \mathbf{b}$ by means of "structure oblivious" Gaussian elimination, for which EIGEN's dedicated classes and methods should be used.

```
void solveA(const Eigen::VectorXd & a, const Eigen::VectorXd & b,
  Eigen::VectorXd & x);
```

The input is composed of vectors $\mathbf{a}$ and $\mathbf{b}$.

Run this function with $\mathbf{a}$ and $\mathbf{b}$ made of random elements and explore the cases $n = 2^k$, $k = 4, \dots, 12$. What is the asymptotic complexity in n of this naive implementation?

HIDDEN HINT 1 for (2-12.c) → 2-12-3-0:structuredLSE3h.pdf

SOLUTION for (2-12.c) → 2-12-3-1:structuredLSE3s.pdf ▲

(2-12.d) ⊡ Given a generic vector $\mathbf{a} \in \mathbb{R}^n$, compute symbolically (i.e. *by hand*) the inverse of matrix $\mathbf{A}$ and the solution $\mathbf{x}$ of $\mathbf{Ax} = \mathbf{b}$.

HIDDEN HINT 1 for (2-12.d) → 2-12-4-0:structuredLSE4h.pdf

SOLUTION for (2-12.d) → 2-12-4-1:structuredLSE4s.pdf ▲

(2-12.e) ⊡ Implement a function in C++ which efficiently solves the linear system $\mathbf{Ax} = \mathbf{b}$, using EIGEN classes. The input is consists of vectors $\mathbf{a}$ and $\mathbf{b}$.

HIDDEN HINT 1 for (2-12.e) → 2-12-5-0:structuredLSE5h.pdf

SOLUTION for (2-12.e) → 2-12-5-1:structuredLSE5s.pdf ▲

(2-12.f) ⊡ Compare the timing of the naive implementation of (2-12.c) with the efficient solution of (2-12.e).

HIDDEN HINT 1 for (2-12.f) → 2-12-6-0:structuredLSE6h.pdf

SOLUTION for (2-12.f) → 2-12-6-1:structuredLSE6s.pdf ▲

End Problem 2-12

Problem 2-13: Conversion of Triplet (COO) Matrix Format to CRS Format

This exercise is about sparse matrices. Make sure you are prepared on the subject by reading [Lecture → Section 2.7] in the lecture notes. In particular, read through the various *sparse storage formats* discussed in class (cf. [Lecture → Section 2.7.1]).

Focus of the problem is implementation in C++, using EIGEN in one part.

Remark. The ultimate goal is to devise a function that converts a matrix given in **triplet (or COOrdinate) format** (COO, see [Lecture → § 2.7.1.1]) to the **compressed row storage format** (CRS, see [Lecture → § 2.7.1.4]).

You must not rely on EIGEN's sparse matrix functionality to solve this problem. The only allowed EIGEN include is **#include** <Eigen/Dense>!

Recall that the COO format stores a collection of triplets (i,j,v), with $i,j \in \mathbb{N},\ i,j \geq 0$ (the indices) and $v \in \mathbb{R}$ (the value in cell (i,j)). Multiple triplets corresponding to the same cell (i,j) are allowed, meaning that multiple values with the same indices (i,j) should be summed up when fully expressing the matrix.

In C++ the data for a matrix stored in COO format and with entry type **SCALAR** can be stored in the following data structure

```
template <typename SCALAR>
struct TripletMatrix {
  std::size_t n_rows {0}; // Number of rows
  std::size_t n_cols {0}; // Number of columns
  std::vector<std::tuple<std::size_t, std::size_t, SCALAR>> triplets;
};
```

Also remember that the CRS format uses three vectors:

1. `val`, which stores the values of the nonzero cells, one row after the other.
2. `col_ind`, which stores the column indices of the nonzero cells, one row after the other.
3. `row_ptr`, which stores the index of the entry in `val`/`col_ind` containing the first element of each row.

The case of rows composed of only zero elements requires special consideration. The convention is that `row_ptr` stores two consecutive entries with the same values, meaning that in vectors `val` and `col_ind` the next row begins at the same index of the previous row (which implies that the previous row must be empty). This convention should be taken into account when e.g. iterating through `row_ptr`. However, to simplify things, in this problem you may always consider*matrices without all-zero rows*.

A suitable C++ data structure for storing a matrix with entries of type **SCALAR** in CRS format is the following

```
template <typename SCALAR>
struct CRSMatrix {
  std::size_t n_rows {0}; // Number of rows
  std::size_t n_cols {0}; // Number of columns
  std::vector<SCALAR> val; // Value array
  std::vector<std::size_t> col_ind; // Column indices
  std::vector<std::size_t> row_ptr; // Row pointers
};
```

(2-13.a) ⚀ (30 min.) Implement the C++ functions

```
template <typename SCALAR>
Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>
   densify(const TripletMatrix<SCALAR> &M);
template <typename SCALAR>
Eigen::Matrix<SCALAR, Eigen::Dynamic, Eigen::Dynamic>
   densify(const CRSMatrix<SCALAR> &M);
```

that convert the argument matrices into EIGEN's dense matrix storage type.

SOLUTION for (2-13.a) → 2-13-1-0:triplettoCRS3s.pdf ▲

(2-13.b) ⚄ (60 min.) Write a C++ function that converts a matrix $\mathbf{T}$ in COO format to a matrix in CRS format and returns the latter:

```
template <typename SCALAR>
CRSMatrix<SCALAR> tripletToCRS(const TripletMatrix<SCALAR>& T);
```

Your implementation should have (almost, maybe up to a logarithm) optimal complexity, that is, its computational effort should scale linearly with the number of triplets in the COO format.

HIDDEN HINT 1 for (2-13.b) → 2-13-2-0:triplettoCRS4h.pdf

SOLUTION for (2-13.b) → 2-13-2-1:triplettoCRS4s.pdf ▲

(2-13.c) ⚄ (10 min.) What is the worst-case complexity of your function `tripletToCRS` ((2-13.b)) in the number of triplets?

HIDDEN HINT 1 for (2-13.c) → 2-13-3-0:triplettoCRS5h.pdf

SOLUTION for (2-13.c) → 2-13-3-1:triplettoCRS5s.pdf ▲

(2-13.d) ⚄ (20 min.) [depends on Sub-problem (2-13.a)]

Test the correctness of your implementation of `tripletToCRS()` by checking whether the function `densify()` from Sub-problem (2-13.a) yield the same matrix when applied before or after the call to `tripletToCRS()`.

For testing use a **TripletMatrix<double>** object containing an $n \times n$-matrix defined by $5n$ triplets with random indices and value 1. To generate random indices use **std**:`rand()% n`. Implement the test as a C++ function

```
bool testTripletToCRS(std::size_t n);
```

which accepts the matrix dimension as arguments and returns **true**, if the test is passed.

HIDDEN HINT 1 for (2-13.d) → 2-13-4-0:6h1.pdf

SOLUTION for (2-13.d) → 2-13-4-1:triplettoCRS6s.pdf ▲

End Problem 2-13 , 120 min.

Problem 2-14: Sparse matrices in CCS format

Sparse matrices must be stored in special formats that avoid storing the many zero entries. Sometimes one has to manipulate sparse matrices on that basic level. Therefore it is important to be familiar with some details of the storage formats. This exercise focuses on the CCS format.

Related to [Lecture → Section 2.7.1]

In [Lecture → § 2.7.1.4] the so-called *CRS format* (Compressed Row Storage) was introduced. It uses three arrays (`val`, `col_ind` and `row_ptr`) to store a sparse matrix.

EIGEN can also handle the *CCS format* (Compressed Column Storage), which is like CRS for the transposed matrix. It relies on the arrays `val`, `row_ind` and `col_ptr`.

(2-14.a) ⚄ (30 min.) Implement a C++ function which for a given square matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ computes the data describing it in CCS format:

```
std::tuple<Eigen::VectorXd, Eigen::VectorXi, Eigen::VectorXi>
    CCS(const Eigen::MatrixXd& A)
```

The CCS data are returned through a tuple `[val,row_ind, col_ptr]`.

Prohibited. While you can use EIGEN classes such as `Eigen::MatrixXd`, do not use EIGEN methods to directly access `val`, `row_ind` and `col_ptr`. In other words, you must not use EIGEN's class `Eigen::SparseMatrix` and simply run `makeCompressed()`.

You may assume that $\mathbf{A}$ does not have a a column with all elements equal to 0 (Otherwise, it may become problematic to update `col_ptr`.).

In this problem you may test for exact equality with 0.0, because zero matrix entries are not supposed to be results of floating point computations.

SOLUTION for (2-14.a) → `2-14-1-0:smwps.pdf` ▲

(2-14.b) ⚂ (15 min.) [depends on Sub-problem (2-14.a)]

What is the computational complexity of your `CCS` function:

- with respect to the matrix size n, where $\mathbf{A} \in \mathbb{R}^{n\times n}$?
- with respect to nnz, which denotes the number of nonzero elements in $\mathbf{A}$?

SOLUTION for (2-14.b) → `2-14-2-0:smwps.pdf` ▲

End Problem 2-14 , 45 min.

Problem 2-15: Ellpack sparse matrix format

The number of processing cores of high-performance computers has seen an exponential growth. However, the increase in *memory bandwidth*, which is the rate at which data can be read from or written into memory by a processor, has been slower. Hence, memory bandwidth is a bottleneck for several algorithms.
Several papers present storage formats to improve the performance of sparse matrix-vector multiplications. One example is the **Ellpack format**, where the number of nonzero entries per row is bounded and shorter rows are padded with zeros.

Related to [Lecture → Section 2.7.1], which should be studied before tackling this problem.

The C++ class **EllpackMat** whose definition is listed below implements the Ellpack sparse matrix format for a generic matrix $\mathbf{A} \in \mathbb{R}^{m,n}$:

C++ code 2.15.1: Declaration of `EllpackMat` class

```
class EllpackMat {
 public:
  EllpackMat(const Triplets &triplets, index_t m, index_t n);
  double operator()(index_t i, index_t j) const;
  void mvmult(const Vector &x, Vector &y) const;
  void mtvmult(const Vector &x, Vector &y) const;
  index_t get_maxcols() const;

 private:
  std::vector<double> val;  //< Vector containing values
  // corresponding to entries in 'col'
  std::vector<index_t> col;  //< Vector containing column
  // indices of the entries in 'val'.
  // The position of a cell in 'val' and 'col'
  // is determined by its row number and original position in 'triplets'
  index_t maxcols;  //< Number of non-empty columns
  index_t m, n;     //< Number of rows, number of columns
};
```

Get it on GitLab (ellpack.hpp).

In the code above the following typedefs are used

```
using Triplet = Eigen::Triplet<double>;
using Triplets = std::vector<Triplet>;
```

Further, the data member `maxcols` is defined as the maximum number of non-zero entries in a row of the matrix:

$$\texttt{maxcols} := \max\{\#[(i,j) \mid \mathbf{A}_{i,j} \neq 0,\ j = 1,\ldots,n],\ i = 1,\ldots,m\}\ .$$

The entry read-access **operator()** method, which takes the entries $(\mathbf{A})_{i,j}$ row and column indices i and j as arguments, is implemented as follows. It completely "defines" the Ellpack sparse matrix format.

C++11-code 2.15.2: Definition of entry-access operator operator()

```
double EllpackMat::operator()(index_t i, index_t j) const {
  assert(0 <= i && i < m && 0 <= j && j < n && "Index out of bounds!");

  for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
    if (col.at(l) == j) return val.at(l);
```

```
7     }
8     return 0;
9   }
```

Get it on GitLab (ellpack.hpp).

It is well defined for $i = 0, \ldots, m-1$ and for $j = 0, \ldots, n-1$ (C++ indexing!).

(2-15.a) ⚀ (60 min.) Implement an efficient constructor of **EllpackMat** that builds an object of type **EllpackMat** from data forming a matrix in triplet format. In other words, implement the constructor with the following signature:

```
EllpackMat(const Triplets & triplets, index_t m, index_t n);
```

`triplets` is a **std::vector** of EIGEN triplets (see [Lecture → Section 2.7.2]). The arguments m and n represent the number of rows and columns of the matrix $\mathbf{A}$.

- The data in the triplet vector must be compatible with the matrix size provided.
- to the constructor. No assumption is made on the ordering of the triplets.
- Values belonging to multiple occurrences of the same index pair are to be summed up.

Taking care of repeated index pairs requires sophisticated bookkeeping.

HIDDEN HINT 1 for (2-15.a) → 2-15-1-0:ellpack1h.pdf

HIDDEN HINT 2 for (2-15.a) → 2-15-1-1:ellp1h1.pdf

HIDDEN HINT 3 for (2-15.a) → 2-15-1-2:sellh2.pdf

SOLUTION for (2-15.a) → 2-15-1-3:ellpack1s.pdf ▲

(2-15.b) ⚂ (30 min.) Implement an efficient method of class **EllpackMat** that, given an input n-vector $\mathbf{x}$, returns the m-vector $\mathbf{y}$ from the matrix-vector product $\mathbf{Ax}$:

```
void mvmult(const Eigen::VectorXd &x, Eigen::VectorXd &y) const;
```

$\mathbf{A}$ is the matrix represented by $\star$**this**. The implementation must run with the optimal complexity of $O(\mathrm{nnz}(\mathbf{A}))$.

HIDDEN HINT 1 for (2-15.b) → 2-15-2-0:ellpack2h.pdf

SOLUTION for (2-15.b) → 2-15-2-1:ellpack2s.pdf ▲

(2-15.c) ⚂ (20 min.) Parallel to what you did in Sub-problem (2-15.b), implement now the method `mtvmult` of **EllpackMat** that computes the matrix-vector product $\mathbf{A}^\top\mathbf{x}$ with the transposed matrix.

HIDDEN HINT 1 for (2-15.c) → 2-15-3-0:ellpack3h.pdf

SOLUTION for (2-15.c) → 2-15-3-1:ellpack3s.pdf ▲

(2-15.d) ⚀ (10 min.) [depends on Sub-problem (2-15.a)]

Discuss the asymptotic complexity of your implementation of the constructor of **EllpackMat** in terms of growing number of triplets, $\sharp\texttt{triplets} \to \infty$.

SOLUTION for (2-15.d) → 2-15-4-0:sellx.pdf ▲

End Problem 2-15 , 120 min.

Problem 2-16: Grid functions ⚀

This exercise deals with construction of sparse matrices from triplet format, see [Lecture → Section 2.7.2]. This task is relevant for applications like image processing and the numerical solution of partial differential equations.

Connected with [Lecture → Section 2.7]

Consider the following matrix $\mathbf{S} \in \mathbb{R}^{3,3}$:

$$\mathbf{S} := \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

Denote by $s_{i,j}$ the entries of $\mathbf{S}$. Consider the following *linear* operator from $\mathbb{R}^{n,m}$ to $\mathbb{R}^{n,m}$:

$$L : \begin{cases} \mathbb{R}^{n,m} & \to & \mathbb{R}^{n,m} \\ \mathbf{X} & \mapsto & L(\mathbf{X}): \ (L(\mathbf{X}))_{i,j} := \sum_{k,l=1}^{3} s_{k,l} (\mathbf{X})_{i+k-2,j+l-2}, \ i \in \{1,\ldots,n\}, \ m \in \{1,\ldots,m\} \end{cases}$$

where we adopt the convention that in the sum non-existent matrix entries are set to zero.

(2-16.a) ⚀ (10 min.) Show that L is a linear operator.

SOLUTION for (2-16.a) → 2-16-1-0:grfl.pdf ▲

(2-16.b) ⚁ (20 min.) The space $\mathbb{R}^{n,m}$, as a vector-space, is isomorphic to $\mathbb{R}^{n\cdot m}$, via the mapping $\mathbf{X} \mapsto \mathrm{vec}(\mathbf{X})$, see [Lecture → (1.2.3.5)].

From linear algebra, we know that any linear operator over a finite dimensional (real) vector space can be represented by a (real) matrix multiplication. We define the matrix $\mathbf{A} \in \mathbb{R}^{nm,nm}$ s.t.

$$\mathbf{A} \cdot \mathrm{vec}(\mathbf{X}) = \mathrm{vec}(L(\mathbf{X})).$$

Write down the matrix $\mathbf{A}$ for $n = m = 3$.

SOLUTION for (2-16.b) → 2-16-2-0:gfrwa.pdf ▲

(2-16.c) ⚁ (20 min.) Write a C++ function

```
void eval(Eigen::MatrixXd & X, std::function<double(Eigen::Index,
  Eigen::Index)> f);
```

which, given a matrix $\mathbf{X} \in \mathbb{R}^{n,m}$ and a function $f : \mathbb{N}^2 \to \mathbb{R}$, fills the matrix $\mathbf{X}$ according to the rule $(\mathbf{X})_{i,j} := f(i,j)$, $i \in \{1,\ldots,n\}$, $j \in \{1,\ldots,m\}$.

Additionally, inside the auxiliary function `define_f()` define a lambda function f, with signature `double(Eigen::Index,Eigen::Index)`, implementing:

$$f : \begin{cases} \mathbb{N}^2 & \to & \mathbb{R}, \\ (i,j) & \mapsto & \begin{cases} 1 & i > n/4 \wedge i < 3n/4 \wedge j > m/4 \wedge j < 3m/4\,, \\ 0 & \text{otherwise.} \end{cases} \end{cases}$$

SOLUTION for (2-16.c) → 2-16-3-0:gfrwa.pdf ▲

(2-16.d) ⚂ (30 min.) [depends on Sub-problem (2-16.c)]

Implement a function

```
Eigen::SparseMatrix<double> build_matrix(const Eigen::Matrix3d & S,
    Eigen::Index Xn, Eigen::Index Xm);
```

which, given the stencil matrix $\mathbf{S} \in \mathbb{R}^{3,3}$ and the size of the matrix $\mathbf{X}$ (passed through `Xn,Xm`), builds and returns the sparse matrix $\mathbf{A}$ in CRS format. The matrix $\mathbf{A}$ can be built using an intermediate triplet/COO format.

HIDDEN HINT 1 for (2-16.d) → 2-16-4-0:2h1.pdf

SOLUTION for (2-16.d) → 2-16-4-1:gfrwb.pdf ▲

(2-16.e) ⚂ (30 min.) Implement a function

```
Eigen::MatrixXd mult(const Eigen::SparseMatrix<double> & A,
                     const Eigen::MatrixXd & X);
```

which, given a matrix $\mathbf{X}$ and the sparse matrix $\mathbf{A}$, returns the matrix $\mathbf{Y}$ s.t. $\mathrm{vec}(\mathbf{Y}) := \mathbf{A}\,\mathrm{vec}(\mathbf{X})$. You can use objects of type `Eigen::Map` to "reshape" a matrix into a column vector.

SOLUTION for (2-16.e) → 2-16-5-0:gfrwm.pdf ▲

(2-16.f) ⚄ (30 min.)

Implement a function ««««< HEAD

```
void solve(const Eigen::SparseMatrix<double> & A,
           const Eigen::MatrixXd & Y, Eigen::MatrixXd & X);
=
egin{lstlisting}[style=cppsimple, autogobble]
Eigen::MatrixXd solve(const Eigen::SparseMatrix<double> & A,
                      const Eigen::MatrixXd & Y);
> 9e4a6c66d6b28b002633da1fa29d2222f18fc5d9
```

which, given a matrix $\mathbf{Y}$ and the sparse matrix $\mathbf{A}$, returns the matrix $\mathbf{X}$ s.t. $\mathrm{vec}(\mathbf{Y}) := \mathbf{A}\,\mathrm{vec}(\mathbf{X})$. You can use objects of type `Eigen::Map` to "reshape" a matrix into a column vector. Objects of this type are read and write compatible.

SOLUTION for (2-16.f) → 2-16-6-0:gfrwi.pdf ▲

End Problem 2-16 , 140 min.

Problem 2-17: Efficient sparse matrix-matrix multiplication in COO format

The *triplet (or COOrdinate) list format* works very well for defining a matrix or adding/modifying its elements. This is however not so true for matrix operations such as matrix-matrix multiplications, unlike the CSC/CSR storage.
This problem is about (sparse) matrix-matrix multiplication in COO format. We will consider the following items:

1. The worst case of sparse matrix-matrix multiplication, when one wants to use sparse matrix storage formats.
2. The most efficient way to tackle this problem with the COO format.
3. The asymptotic complexity of such algorithm (which is, by definition, the complexity under the worst-case scenario).

Related to [Lecture → Section 2.7.1]

For simplicity, in the following we will only deal with *binary* matrices. Binary matrices are only made of 0 or 1. At the same time, we will allow for duplicates in our implementations of the triplet format (see [Lecture → § 2.7.1.1]). The matrix entry associated to the pair (i,j) is defined to be the sum of all triplets corresponding to it.

For the subproblems involving coding, we define types `Trip = Triplet<double>` and `TripVec = std::vector<trip>`.

(2-17.a) ⚀ Consider multiplications between sparse matrices: $\mathbf{AB} = \mathbf{C}$. Is the product $\mathbf{C}$ guaranteed to be sparse? Make an example of two sparse matrices which, when multiplied, return a dense matrix.

HIDDEN HINT 1 for (2-17.a) → 2-17-1-0:matmatCOO1h.pdf

SOLUTION for (2-17.a) → 2-17-1-1:matmatCOO1s.pdf ▲

(2-17.b) ⚁ Implement a C++ function which returns the input matrix $\mathbf{A} \in \mathbb{R}^{m\times n}$ in COO format:

```
TripVec Mat2COO(const Eigen::MatrixXd &A);
```

You can use EIGEN classes.

SOLUTION for (2-17.b) → 2-17-2-0:matmatCOO2s.pdf ▲

(2-17.c) ⚂ Implement a C++ function which computes the product between two matrices in COO format:

```
TripVec COOprod_naive(const TripVec &A, const TripVec &B);
```

You can use EIGEN classes.

HIDDEN HINT 1 for (2-17.c) → 2-17-3-0:matmatCOO3h.pdf

SOLUTION for (2-17.c) → 2-17-3-1:matmatCOO3s.pdf ▲

(2-17.d) ⚁ What is the asymptotic complexity of your naive implementation in (2-17.c)?

HIDDEN HINT 1 for (2-17.d) → 2-17-4-0:matmatCOO4h.pdf

SOLUTION for (2-17.d) → 2-17-4-1:matmatCOO4s.pdf ▲

(2-17.e) ⚂ Implement a C++ function which computes the product between two matrices in COO format in an efficient way:

```
TripVec COOprod_effic(TripVec &A, TripVec &B);
```

You can use EIGEN classes.

Can you do better than (2-17.c)?

HIDDEN HINT 1 for (2-17.e) → 2-17-5-0:matmatCOO5h.pdf

SOLUTION for (2-17.e) → 2-17-5-1:matmatCOO5s.pdf ▲

(2-17.f) ⚂ What is the asymptotic complexity of your efficient implementation in (2-17.e)?

HIDDEN HINT 1 for (2-17.f) → 2-17-6-0:matmatCOO6h.pdf

SOLUTION for (2-17.f) → 2-17-6-1:matmatCOO6s.pdf ▲

(2-17.g) ⚀ Compare the timing of `COOprod_naive` (2-17.c) and `COOprod_effic` (2-17.e) for random matrices with different dimensions n. Perform the comparison twice: first only for products between sparse matrices, then for any kind of matrix.

HIDDEN HINT 1 for (2-17.g) → 2-17-7-0:matmatCOO7h.pdf

SOLUTION for (2-17.g) → 2-17-7-1:matmatCOO7s.pdf ▲

End Problem 2-17

Problem 2-18: Gauss-Seidel Iteation for a Sparse Linear System of Equations

This problem studies are particular iterative method defined by a matrix and a vector, the so-called **Gauss-Seidel iteration**. We consider its algorithmic aspects when the matrix is given in a raw CRS format.

Familiarity with [Lecture → Section 2.7.1] is esssential for this problem. Some concepts from [Lecture → Section 8.2.2] will also be used. Implementation in C++ based on EIGEN will be requested.

Given the *regular* square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ and the vector $\mathbf{b} \in \mathbb{R}^n$, $n \in \mathbb{N}$, the **Gauss-Seidel iteration** $\mathbf{x}^{(k+1)} := \mathbf{\Phi}(\mathbf{x}^{(k)})$, $\mathbf{\Phi} : \mathbb{R}^n \to \mathbb{R}^n$, is defined as

$$\left(\mathbf{x}^{(k+1)}\right)_i = (\mathbf{A})_{i,i}^{-1}\left((\mathbf{b})_i - \sum_{j=1}^{i-1}(\mathbf{A})_{i,j}\left(\mathbf{x}^{(k+1)}\right)_j - \sum_{j=i+1}^{n}(\mathbf{A})_{i,j}\left(\mathbf{x}^{(k)}\right)_j\right), \quad i = 1,2,\ldots,n \text{ (in this order!)}. \tag{2.18.1}$$

Here sums for which the lower index is larger than the upper are meant to evaluate to zero.

The matrix $\mathbf{A}$ is available in a raw compressed row-storage (CRS) format, encapsulated in the following data type:

```
struct CRSMatrix {
  unsigned int m;                       // number of rows
  unsigned int n;                       // number of columns
  std::vector<double> val;              // value array
  std::vector<unsigned int> col_ind;    // same length as value array
  std::vector<unsigned int> row_ptr;    // length m+1, row_ptr[m] ==
     val.size()
};
```

Refer to [Lecture → § 2.7.1.4] for further explanations on the meaning of the fields of **CRSMatrix**.

(2-18.a) ⚀ (10 min.) For what square matrices $\mathbf{A}$ and vectors $\mathbf{b}$ is the Gauss-Seidel iteration (2.18.1) well-defined?

Iteration (2.18.1) well-defined $\forall \mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$: .

SOLUTION for (2-18.a) → 2-18-1-0:gscrss1.pdf ▲

(2-18.b) ⚁ (45 min.)

In the file `gaussseidelcrs.hpp` implement the C++ function

```
bool GaussSeidelstep_crs(
  const CRSMatrix &A, const Eigen::VectorXd &b,
  Eigen::VectorXd &x);
```

that carries out a single step of the Gauss-Seidel iteration according to (2.18.1). The vector $\mathbf{x}^{(k)}$ is passed through `x` and it should return $\mathbf{x}^{(k+1)}$ also in `x`. **false** is returned in case the iteration is not well-defined, **true** otherwise.

SOLUTION for (2-18.b) → 2-18-2-0:gscrss2.pdf ▲

(2-18.c) ⚄ (20 min.) For given regular $\mathbf{A} \in \mathbb{R}^{n,n}$ and $\mathbf{b} \in \mathbb{R}^n$ assume that the Gauss-Seidel iteration (2.18.1) is well-defined. Characterize the set of all **fixed points** of the Gauss-Seidel iteration (2.18.1) by a compact formula:

$$\{\mathbf{x}^* \in \mathbb{R}^n : \mathbf{x}^* \text{ is fixed point of (2.18.1)}\} = \quad .$$

A fixed point of an iteration $\mathbf{x}^{(k+1)} = F(\mathbf{x}^{(k)})$ is a vector $\mathbf{x}^*$ such that $\mathbf{x}^* = F(\mathbf{x}^*)$.

SOLUTION for (2-18.c) → 2-18-3-0:gss3.pdf ▲

(2-18.d) ⚄ (30 min.) [depends on Sub-problem (2-18.b)]

Develop a C++ function

```
bool GaussSeidel_iteration(
  const CRSMatrix &A, const Eigen::VectorXd &b, Eigen::VectorXd &x,
  double atol = 1.0E-8, double rtol = 1.0E-6,
  unsigned int maxit = 100);
```

which carries out Gauss-Seidel iterations for the data $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$ (arguments `A` and `b`), and with initial guess $\mathbf{x}^{(0)}$ passed in `x`. The same argument is used to return the final iterate. The iteration is to terminate after `maxit` steps. If this happens, **false** should be returned, **true** otherwise.

The function should rely on the correction-based termination criterion [Lecture → (8.5.3.2)] with relative and absolute tolerance passed in `rtol` and `atol`, respectively and based on the Euclidean vector norm.

SOLUTION for (2-18.d) → 2-18-4-0:gss33.pdf ▲

(2-18.e) ⚄ (5 min.) [depends on Sub-problem (2-18.c)]

We apply the Gauss-Seidel iteration for

$$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 2 & 8 & 2 & 0 & 0 \\ 3 & 0 & 1 & 10 & 5 & 0 \\ 0 & 2 & 0 & 3 & 13 & 2 \\ 0 & 4 & 0 & 0 & 2 & 11 \end{bmatrix} \quad , \quad \mathbf{b} = \mathbf{0} \, .$$

and initial guess $\mathbf{x}^{(0)} = [1,2,3,4,5,6]^\top$. The following table displays the progress of the iteration:

k	$\left\|\mathbf{x}^{(k)}\right\|_2$	$\left\|\mathbf{x}^{(k)}-\mathbf{x}^{(k-1)}\right\|$	$\left\|\mathbf{x}^{(k)}\right\|_2 : \left\|\mathbf{x}^{(k-1)}\right\|_2$
1	3.863865e+00	1.127304e+01	
2	8.297690e-01	3.751824e+00	2.147510e-01
3	6.432366e-02	7.857529e-01	7.751996e-02
4	1.476504e-02	5.984439e-02	2.295429e-01
5	4.068577e-03	1.098126e-02	2.755548e-01
6	9.204751e-04	3.155445e-03	2.262401e-01
7	1.880926e-04	7.327086e-04	2.043429e-01
8	3.636497e-05	1.517466e-04	1.933355e-01
9	6.803531e-06	2.956273e-05	1.870902e-01
10	1.246867e-06	5.556761e-06	1.832676e-01
11	2.254491e-07	1.021425e-06	1.808125e-01
12	4.039721e-08	1.850525e-07	1.791855e-01
13	7.194116e-09	3.320315e-08	1.780844e-01
14	1.275722e-09	5.918398e-09	1.773286e-01

Describe qualitatively and quantitatively the observed *asymptotic* convergence of the iteration for this example.

SOLUTION for (2-18.e) → `2-18-5-0:gss5.pdf` ▲

End Problem 2-18 , 110 min.

Problem 2-19: A special Sylvester Equation

A **Sylvester equations** is a *linear matrix equation* of the form

$$\mathbf{AX} + \mathbf{XB} = \mathbf{C} \quad , \quad \mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n,n} \, ,$$

which yields a linear system of equations for the entries of the unknown matrix $\mathbf{X} \in \mathbb{R}^{n,n}$. In this exercise we consider a very special specimen of Sylvester equation and recast it for the sake of efficient direct solution with EIGEN.

Related to various topics from class and also Problem 2-8.

(2-19.a) ⚀ (5 min.) Refresh your knowledge about s.p.d. matrices, see [Lecture → Def. 1.1.2.6] and [Lecture → Lemma 1.1.2.7]. ▲

(2-19.b) ⚁ (5 min.) Recall the "vectorization" $\mathrm{vec}(\mathbf{X}) \in \mathbb{R}^{n^2}$ of a matrix $\mathbf{X} \in \mathbb{R}^{n,n}$ from [Lecture → Rem. 1.2.3.4] and jot down $\mathrm{vec}(\mathbf{X})$ for

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \in \mathbb{R}^{2,2} \, .$$

SOLUTION for (2-19.b) → 2-19-2-0:s1a.pdf ▲

(2-19.c) ⚂ (15 min.) For $\mathbf{S}, \mathbf{T} \in \mathbb{R}^{2,2}$ find the matrix $\mathbf{C} \in \mathbb{R}^{4,4}$ such that

$$\mathrm{vec}(\mathbf{SXT}) = \mathbf{C}\,\mathrm{vec}(\mathbf{X}) \quad \forall \mathbf{X} \in \mathbb{R}^{2,2} \, ,$$

that is, express the entries of $\mathbf{C}$ in terms of the entries $(\mathbf{S})_{i,j}$ and $(\mathbf{T})_{i,j}$ of $\mathbf{S}$ and $\mathbf{T}$.

SOLUTION for (2-19.c) → 2-19-3-0:s1b.pdf ▲

(2-19.d) ⚄ (20 min.) Based on the **Kronecker product** $\otimes$ from [Lecture → Def. 1.4.3.7] write down a matrix $\mathbf{C} \in \mathbb{R}^{n^2,n^2}$ depending on the given square matrices $\mathbf{S}, \mathbf{T} \in \mathbb{R}^{n,n}$ such that

$$\mathrm{vec}(\mathbf{SXT}) = \mathbf{C}\,\mathrm{vec}\,(\mathbf{X}) \, .$$

HIDDEN HINT 1 for (2-19.d) → 2-19-4-0:s1h1.pdf

SOLUTION for (2-19.d) → 2-19-4-1:s2.pdf ▲

Throughout the remainder of this problem let $\mathbf{A} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, stand for a *symmetric, positive definite* (s.p.d.) matrix, whose rows and columns contain at most five non-zero entries. Hence, for large n the matrix $\mathbf{A}$ can be considered *sparse* in the sense of [Lecture → Notion 2.7.0.1].

Then consider the special Sylvester equation:

$$\text{seek} \quad \mathbf{X} \in \mathbb{R}^{n,n} : : \quad \mathbf{XA}^{-1} + \mathbf{AX} = \mathbf{I} \, , \tag{2.19.4}$$

and the equivalent matrix equation:

$$\text{seek} \quad \mathbf{X} \in \mathbb{R}^{n,n} : : \quad \mathbf{X} + \mathbf{AXA} = \mathbf{A} \, . \tag{2.19.5}$$

(2-19.e) ⚂ (5 min.) Both (2.19.4) and (2.19.5) can be recast as an $n^2 \times n^2$ linear system of equations for the unknown vector $\mathrm{vec}(\mathbf{X})$. Based on the result of Sub-problem (2-19.d) express their coefficient matrices by means of the Kronecker product [Lecture → Def. 1.4.3.7].

SOLUTION for (2-19.e) → 2-19-5-0:s3a.pdf ▲

(2-19.f) ⚄ (10 min.) Based on the result of Sub-problem (2-19.e) and in light of the observation made in [Lecture → Ex. 2.7.4.4] give a sharp upper bound for the number of non-zero entries of the coefficient matrices of those linear systems for (2.19.4) and (2.19.5).

HIDDEN HINT 1 for (2-19.f) → 2-19-6-0:s3h1.pdf

SOLUTION for (2-19.f) → 2-19-6-1:s4.pdf ▲

(2-19.g) ⚄ (15 min.) Show that both (2.19.4) and (2.19.5) give rise to linear systems of equations for the entries of $\mathbf{X}$ with *s.p.d.* coefficient matrices.

For your argument you can appeal to the following result:

Theorem 2.19.9. Eigenvalues of Kronecker-product matrices

Assume that both $\mathbf{S},\mathbf{T} \in \mathbb{R}^{n,n}$ can be diagonalized and write $\sigma(\mathbf{S}),\sigma(\mathbf{T}) \subset \mathbb{C}$ for the sets of their eigenvalues. Then the set of eigenvalues of the Kronecker product is given by

$$\sigma(\mathbf{S} \otimes \mathbf{T}) = \{\lambda\mu\colon \lambda \in \sigma(\mathbf{S}), \mu \in \sigma(\mathbf{T})\} .$$

HIDDEN HINT 1 for (2-19.g) → 2-19-7-0:s4h1.pdf

HIDDEN HINT 2 for (2-19.g) → 2-19-7-1:s4h2.pdf

SOLUTION for (2-19.g) → 2-19-7-2:s4.pdf ▲

(2-19.h) ⚅ (15 min.) Temporarily assume that $\mathbf{A}$ is a *diagonal* matrix. Implement a C++ function

```
template <typename Vector>
  Eigen::SparseMatrix<double> solveDiagSylvesterEq(const Vector
     &diagA);
```

that solves (2.19.4) with the diagonal of $\mathbf{A}$ passed through the vector argument `diagA`. The result $\mathbf{X}$ is returned as a sparse matrix. The type **Vector** must supply a `size()` method and component access through **operator** `[]`.

SOLUTION for (2-19.h) → 2-19-8-0:s5.pdf ▲

(2-19.i) ⚅ (45 min.) Without using EIGEN's (beta-status) implementation of the Kronecker product, realize a C++ function

```
Eigen::SparseMatrix<double> sparseKron(const
   Eigen::SparseMatrix<double> &M);
```

that computes the Kronecker product $\mathbf{M} \otimes \mathbf{M}$ for the square sparse matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ and returns the result in sparse-matrix format.

For the sake of efficient initialization you have to use the `reserve()` function and determine the maximum number of non-zero entries per column for $\mathbf{A}$ in advance. To do this use the member functions of **SparseMatrix** that allow direct access to the data vectors of the CCS format, *cf.* [Lecture → § 2.7.1.4]:

- **const double** `*valuePtr()` **const**: returns a pointer to the array of values of non-zero matrix entries,
- **const** `StorageIndex* innerIndexPtr()` **const**: returns a pointer to the array of row indices for every non-zero matrix entry,
- **const** `StorageIndex* outerIndexPtr()` **const**: returns a pointer to the vector of starting positions of columns.

These access methods also permit you to run through all non-zero entries of $\mathbf{A}$ and retrieve their locations.

SOLUTION for (2-19.i) → 2-19-9-0:s6.pdf ▲

(2-19.j) ⚂ (20 min.) [depends on Sub-problem (2-19.i), Sub-problem (2-19.e)]

Write a C++ function

```
Eigen::MatrixXd solveSpecialSylvesterEq(
  const Eigen::SparseMatrix<double> &A);
```

that solves (2.19.4) and returns the solution $\mathbf{X}$.

SOLUTION for (2-19.j) → 2-19-10-0:s7.pdf ▲

References

[Lan70] Peter Lancaster. "Explicit solutions of linear matrix equations". In: *SIAM Rev.* 12 (1970), pp. 544–566. DOI: 10.1137/1012104.

End Problem 2-19 , 155 min.

Problem 2-20: Matrix-Vector Product in CRS format

Sparse matrices have to be stored in special formats in order to save memory and inform algorithms about the position of non-zero entries. This problem will examine the CRS format

This problems is related to [Lecture → § 2.7.1.4] and assumes familiarity with C++.

The following data structure is used to store an $m \times n$-matrix in compressed row-storage (CRS) format [Lecture → § 2.7.1.4]:

```
struct CRSMatrix {
  unsigned int m;                      // number of rows
  unsigned int n;                      // number of columns
  std::vector<double> val;             // value array
  std::vector<unsigned int> col_ind;   // same length as value array
  std::vector<unsigned int> row_ptr;   // length m+1, row_ptr[m] ==
     val.size()
};
```

This format is defined by the relationship ("C++ indexing")

$$\texttt{val}[k] = a_{ij} \quad \Leftrightarrow \quad \begin{cases} \texttt{col\_ind}[k] = j\,, \\ \texttt{row\_ptr}[i] \leq \texttt{k} < \texttt{row\_ptr}[i+1]\,, \end{cases} \qquad 0 \leq k \leq \texttt{val.size()}\,,$$

for $i \in \{0,\ldots,m-1\}$, $j \in \{0,\ldots,n-1\}$.

(2-20.a) ⚀ (40 min.) The function `crsmv` is supposed to return the product of a matrix in CRS format passed through `M` and of a vector given as argument `x`. Supplement the missing parts of the following listing code by writing valid C++ code into the boxes.

```
template <typename VECTORTYPE_I, typename VECTORTYPE_II>
VECTORTYPE_I crsmv(const CRSMatrix &M,
                   const VECTORTYPE_II &x) {
  assert((x.size() == M.[          ])
         && "Size mismatch between x and M");
  VECTORTYPE_I y(M.m);
  for (int k = 0; k < [          ]; ++k) {
    y[k] = 0;
    for (int j = M.[          ]; j < [          ]; ++j) {
      y[[          ]] += M.[          ] * x[[          ]];
    }
  }
  return y;
}
```

SOLUTION for (2-20.a) → 2-20-1-0:crsmv1.pdf ▲

End Problem 2-20, 40 min.

Problem 2-21: Asymptotic Complexity of Numerical Linear Algebra Operations

Most numerical algorithms rely on basic operations from numerical linear algebra. In order to gauge the computational cost of these algorithms, precise knowledge about the asymptotic computational effort involved in these operations is required. This problem asks you to analyze a given algorithm in this respect.

This problem draws on [Lecture → Section 1.4.2] and [Lecture → Section 2.5]. No C++ implementation is requested.

The function

```
std::pair<double, Eigen::VectorXd>
  invit(Eigen::MatrixXd& A, double tol);
```

implements a so-called inverse power iteration. It takes a general (invertible) matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, as argument `A`.

The C++ code for `invit()` is listed in Code 2.21.1.

C++ code 2.21.1: Function implementing inverse power iteration

```
2  std::pair<double, Eigen::VectorXd> invit(Eigen::MatrixXd& A,
3                                           double tol = 1.0E-6) {
4    assert((A.cols() == A.rows()) && "A must be square!");
5    const unsigned int n = A.cols();
6    Eigen::VectorXd y_new = Eigen::VectorXd::Random(n);
7    Eigen::VectorXd y_old(n);
8    double l_new = y_new.transpose() * A * y_new;
9    double l_old;
10   const auto A_lu_dec = A.lu(); //
11   y_new = y_new / y_new.norm();
12   do {
13     l_old = l_new;
14     y_old = y_new; //
15     y_new = A_lu_dec.solve(y_old); //
16     y_new /= y_new.norm(); //
17     l_new = y_new.transpose() * A * y_new; //
18   } while (std::abs(l_new - l_old) > tol * std::abs(l_new));
19   return {l_new, y_new};
20 }
```

(2-21.a) ⚀ (10 min.) What is the asymptotic complexity of the operations in Line 10,

```
const auto A_lu_dec = A.lu();
```

of Code 2.21.1?

$$\text{Cost(Line 10 of Code 2.21.1)} = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty\,.$$

SOLUTION for (2-21.a) → 2-21-1-0:cits1.pdf

▲

(2-21.b) ⚀ (10 min.) What is the asymptotic complexity of the operations in Line 14,

```
y_old = y_new;
```

of Code 2.21.1?

$$\text{Cost(Line 14 of Code 2.21.1)} = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty \,.$$

SOLUTION for (2-21.b) → 2-21-2-0:cits2.pdf

▲

(2-21.c) ⚀ (10 min.) What is the asymptotic complexity of the operations in Line 15,

```
y_new = A_lu_dec.solve(y_old);
```

of Code 2.21.1?

$$\text{Cost(Line 15 of Code 2.21.1)} = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty \,.$$

SOLUTION for (2-21.c) → 2-21-3-0:cits3.pdf

▲

(2-21.d) ⚁ (10 min.) What is the asymptotic complexity of the operations in Line 16,

```
y_new /= y_new.norm();
```

of Code 2.21.1?

$$\text{Cost(Line 16 of Code 2.21.1)} = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty \,.$$

SOLUTION for (2-21.d) → 2-21-4-0:cits4.pdf

▲

(2-21.e) ⚂ (10 min.) What is the asymptotic complexity of the operations in Line 17,

```
l_new = y_new.transpose() * A * y_new;
```

of Code 2.21.1?

$$\text{Cost(Line 17 of Code 2.21.1)} = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty \,.$$

SOLUTION for (2-21.e) → 2-21-5-0:cits5.pdf

▲

End Problem 2-21 , 50 min.

Problem 2-22: Validating LU-decompositions

All libraries for numerical linear algebra offer a built-in function for computing the LU-decomposition [Lecture → Def. 2.3.2.3] of a dense matrix. This small problems studies a way to detect a flawed implementation and demonstrates it for EIGEN's LU-decomposition.

Connected with [Lecture → Section 2.5], [Lecture → Section 1.4] and [Lecture → Section 1.5.3]. Involves "passive C++ coding".

The following code is meant to verify the correctness of EIGEN's computation of the LU-decomposition of a given square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$.

C++ code 2.22.1: Function verifying EIGEN's LU-decomposition

```
2  bool testLU(const Eigen::MatrixXd &A) {
3    Eigen::Index n = A.cols();
4    assert(("Matrix must be square", n == A.rows()));
5    auto ludec = A.lu(); //
6    Eigen::MatrixXd L{ludec.matrixLU().triangularView<Eigen::UnitLower>()}; //
7    Eigen::MatrixXd U{ludec.matrixLU().triangularView<Eigen::Upper>()};
8    L.applyOnTheLeft(ludec.permutationP().inverse());
9    return ((L * U - A).norm() == 0.0); //
10 }
```

(2-22.a) ⚀ (10 min.) Determine the asymptotic computational cost (in terms of matrix size $n \to \infty$) of the instructions in particular lines of Code 2.22.1:

$$\text{cost}(\text{Line 5}) = O(\quad) \quad \text{for} \quad n \to \infty\,,$$

$$\text{cost}(\text{Line 6}) = O(\quad) \quad \text{for} \quad n \to \infty\,,$$

$$\text{cost}(\text{Line 9}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

HIDDEN HINT 1 for (2-22.a) → 2-22-1-0:tluh1.pdf

SOLUTION for (2-22.a) → 2-22-1-1:tlus1.pdf ▲

(2-22.b) ⚀ (10 min.) Unfortunately, for large n, the function `testLU()` of Code 2.22.1 will almost always return **false** and, thus, is not carrying out the validation test correctly.

Rewrite the check of Line 9 so that the test performs as expected.

```
1  bool testLU(const Eigen::MatrixXd &A) {
2    Eigen::Index n = A.cols();
3    assert(("Matrix must be square", n == A.rows()));
4    auto ludec = A.lu();
5    Eigen::MatrixXd L{ludec.matrixLU().triangularView<Eigen::UnitLower>()};
6    Eigen::MatrixXd U{ludec.matrixLU().triangularView<Eigen::Upper>()};
7    L.applyOnTheLeft(ludec.permutationP().inverse());
8    return (                                                    );
```

SOLUTION for (2-22.b) → 2-22-2-0:tlus2.pdf ▲

End Problem 2-22, 20 min.

Problem 2-23: Solving a special near-banded linear system

Banded linear systems of equations, that is, LSEs whose system matrix has only a few filled bands [Lecture → Section 2.7.5], represent an important class of sparse linear systems and often allow a particularly efficient implementation of Gaussian elimination. This is still true, if the system matrix is banded with the exception of a few entries.

This problem is linked to [Lecture → Section 2.6] and [Lecture → Section 2.7.5].

We consider the fallowing near-banded $n \times n$, $n \in \mathbb{N}$, linear system of equations

$$\mathbf{Ax} = \mathbf{b}\,, \quad \mathbf{A} := \begin{bmatrix} 1 & 0 & \dots & \dots & 0 & \alpha \\ 1 & 1 & 0 & \dots & & 0 \\ 0 & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & 1 & 1 & 0 \\ 0 & \dots & \dots & 0 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{n,n} \quad , \quad \alpha \in \mathbb{R}\,, \quad \mathbf{b} \in \mathbb{R}^n\,. \tag{2.23.1}$$

(2-23.a) ⚂ (12 min.) For the case $n = 7$ we consider the **LU-decomposition** of $\mathbf{A}$, $\mathbf{A} = \mathbf{LU}$. Fill in the missing entries of the factors

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \alpha \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} = \mathbf{L} \cdot \mathbf{U}$$

$$= \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \square & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \square & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \square & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & \square & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \square & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & \square & 1 \end{bmatrix}}_{=:\mathbf{L}} \cdot \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & \alpha \\ 0 & 1 & 0 & 0 & 0 & 0 & \square \\ 0 & 0 & 1 & 0 & 0 & 0 & \square \\ 0 & 0 & 0 & 1 & 0 & 0 & \square \\ 0 & 0 & 0 & 0 & 1 & 0 & \square \\ 0 & 0 & 0 & 0 & 0 & 1 & \square \\ 0 & 0 & 0 & 0 & 0 & 0 & \square \end{bmatrix}}_{=:\mathbf{U}} .$$

SOLUTION for (2-23.a) → 2-23-1-0:blss1.pdf ▲

(2-23.b) ⚄ (6 min.) [depends on Sub-problem (2-23.a)]

In the case $n = 7$, for what values of α will $\mathbf{A}$ be invertible?

$$\mathbf{A} \text{ invertible/regular} \quad \Leftrightarrow \quad \alpha \notin \left\{ \square \right\} .$$

SOLUTION for (2-23.b) → 2-23-2-0:dbss2.pdf ▲

(2-23.c) ⚄ (12 min.) [depends on Sub-problem (2-23.a)]

The EIGEN-based C++ function

```
Eigen::VectorXd bandedLSESolve(double alpha,
                               const Eigen::VectorXd& b);
```

is supposed to solve (2.23.1) $\mathbf{Ax} = \mathbf{b}$, where `b` passes the n-vector $\mathbf{b}$ and `alpha` the parameter $\alpha \in \mathbb{R}$.

Complete the following partial implementation of `bandedLSESolve()` so that it meets the specification. Valid C++ code is expected.

```
Eigen::VectorXd bandedLSESolve(double alpha, const
   Eigen::VectorXd& b) {
  const int n = b.size();
  Eigen::VectorXd x(n);
  int sgn = 1;
  x[0] = b[0];
  for (int i = 1; i < n; ++i, sgn *= -1) {
    x[i] =                                          ;
  }
  x[n - 1] /=                                       ;
  for (int j = n - 2; j >= 0; --j, sgn *= -1) {
    x[j] =                                          ;
  }
  return x;
}
```

SOLUTION for (2-23.c) → 2-23-3-0:.pdf ▲

End Problem 2-23 , 30 min.

Problem 2-24: Rearrangement of Sparse Linear System of Equations

In this problem we study a role reversal of some of the components of the vector of unknowns and the right-hand side vector of a sparse linear system of equations. This means, that some entries of the right-hand side vector will be treated as unknowns, while we suppose the same entries of the vector of unknowns to be known.

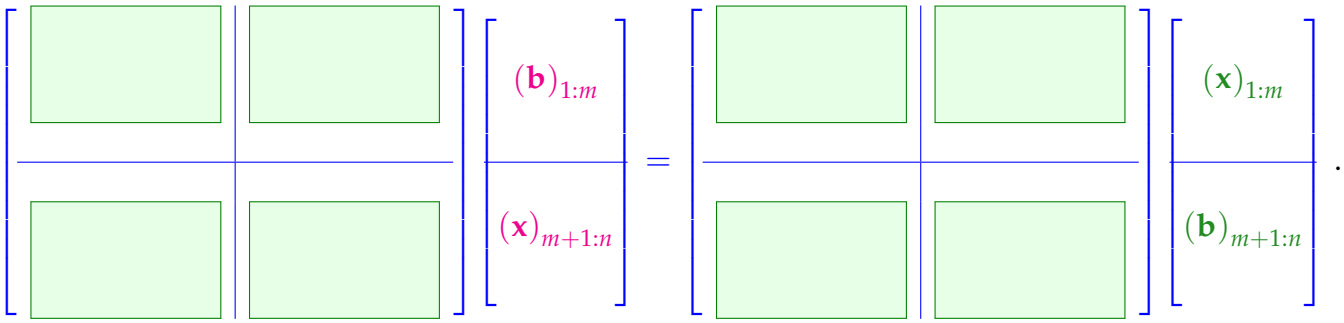
Requires familiarity with [Lecture → Section 2.7.2] and involves substantial C++ coding based on EIGEN.

We consider a large linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{b} \in \mathbb{R}^n$, with a sparse coefficient/system matrix $\mathbf{A}$.

We assume that in $\mathbf{A}\mathbf{x} = \mathbf{b}$
- we know the values of the first m components of $\mathbf{x}$ and the last $n-m$ components of $\mathbf{b}$,
- while the first m components of $\mathbf{b}$ are unknowns, as are the last $n-m$ components of $\mathbf{x}$.

Given: $(\mathbf{x})_{1:m}$, $(\mathbf{b})_{m+1:n}$, $\mathbf{A}\mathbf{x} = \mathbf{b}$ $\leftrightarrow$ Sought: $(\mathbf{x})_{m+1,n}$, $(\mathbf{b})_{1:m}$.

(2-24.a) ⚂ (20 min.) In block-partitioned form state a linear system of equations solved by the vector $\left[(\mathbf{b})_{1:m}, (\mathbf{x})_{m+1:n}\right]^\top \in \mathbb{R}^n$ containing the unknown components of both $\mathbf{x}$ and $\mathbf{b}$.



HIDDEN HINT 1 for (2-24.a) → 2-24-1-0:reah0.pdf

SOLUTION for (2-24.a) → 2-24-1-1:rsls0.pdf ▲

(2-24.b) ⚂ (15 min.) [depends on Sub-problem (2-24.a)]

Assuming $\mathbf{A}\mathbf{x} = \mathbf{b}$, give a sufficient and necessary condition such that, the first m components of $\mathbf{b}$ and the last $n-m$ components of $\mathbf{x}$ are uniquely determined by the last $n-m$ components of $\mathbf{b}$ and the first m components of $\mathbf{x}$.

SOLUTION for (2-24.b) → 2-24-2-0:rsls1.pdf ▲

(2-24.c) ⚄ (60 min.) [depends on Sub-problem (2-24.a)]

In the file `rearrangedsparselse.hpp` write the body of the C++ function

```
bool solveRearrangedLSE(
  const std::vector<Eigen::Triplet<double>> &A_COO,
  Eigen::VectorXd &x, Eigen::VectorXd &b,
  Eigen::Index m);
```

which is supposed to compute $\mathbf{x}, \mathbf{b} \in \mathbb{R}^n$ satisfying $\mathbf{A}\mathbf{x} = \mathbf{b}$ and with $(\mathbf{x})_{1:m}$ given by the top entries of `x` and $(\mathbf{b})_{m+1:n}$ by the bottom entries of `b`. The (sparse) matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ is supplied to the function in triplet/COO format [Lecture → § 2.7.1.1] through the parameter `A_COO`.

The computed unknown entries of $\mathbf{x}$ and $\mathbf{b}$ are returned through the corresponding components of the function arguments `x` and `b`, which are passed by mutable reference.

The function is supposed to report failure (return value **false**) in case a suitably applied Gaussian elimination encounters problems.

SOLUTION for (2-24.c) → `2-24-3-0:rsls2.pdf` ▲

End Problem 2-24 , 95 min.

Problem 2-25: Solving triangular linear system in CRS format

In numerical software sparse matrices are invariably stored in some compressed storage format. In this problem we study how a linear system for its lower triangular part can be solved when the matrix is given in compressed row storage (CRS) format.

This problem assumes knowledge about the CRS compressed matrix format as supplied in [Lecture → § 2.7.1.4].

The following C++ data structure represents a real $m \times n$-matrix in CRS format in a code:

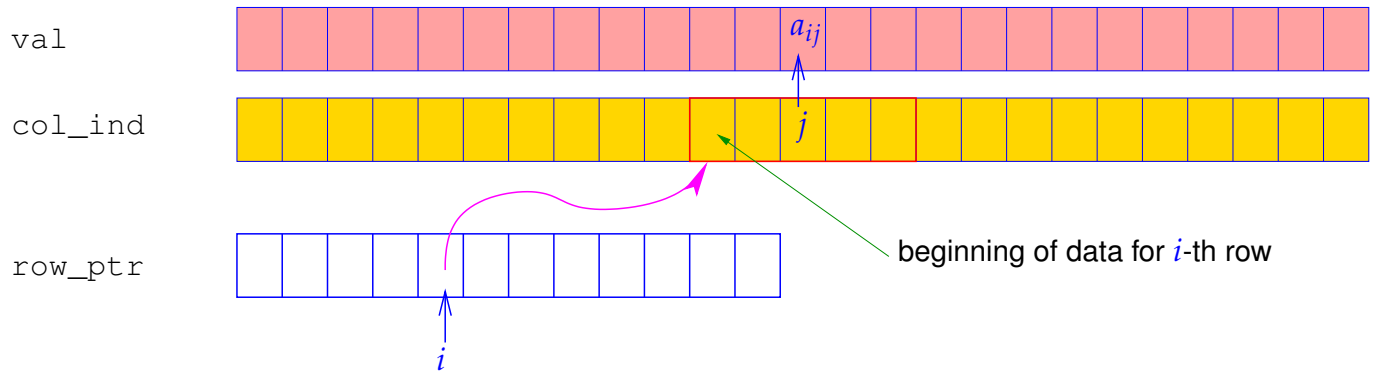
```
struct CRSMat {
  std::size_t m, n; // Numbers of rows and columns
  std::vector<double> val; // size ≥ nnz(A) := #{(i,j) ∈ {1,...,n}², a_ij ≠ 0}
  std::vector<std::size_t> col_ind; // size = val.size()
  std::vector<std::size_t> row_ptr; // size = n+1, w/ sentinel
};
```

As explained in [Lecture → § 2.7.1.4] the CRS format for a sparse matrix $\mathbf{A} = [a_{ij}] \in \mathbb{R}^{n,n}$ organizes the data in three *contiguous* arrays:

`std::vector<double>`	`val`	size $\geq \mathrm{nnz}(\mathbf{A}) := \#\{(i,j) \in \{1,\ldots,n\}^2, a_{ij} \neq 0\}$
`std::vector<size_t>`	`col_ind`	size = `val.size()`
`std::vector<size_t>`	`row_ptr`	size $n+1$ & `row_ptr`$[n+1]$ = `val.size()` (sentinel value)

Access to the matrix entry $a_{ij} := (\mathbf{A})_{ij} \neq 0, 1 \leq i,j \leq n$ ("mathematical indexing") follows the convention:

$$\texttt{val}[k] = a_{ij} \quad \Leftrightarrow \quad \begin{cases} \texttt{col\_ind}[k] = j\,, \\ \texttt{row\_ptr}[i] \leq k < \texttt{row\_ptr}[i+1]\,, \end{cases} \quad 1 \leq k \leq \mathrm{nnz}(\mathbf{A})\,. \tag{2.25.1}$$



A small example is given for illustration:

$$\mathbf{A} = \begin{bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{bmatrix}$$

`val`-vector:

10	-2	3	9	3	7	8	7	3 ... 9	13	4	2	-1

`col_ind`-array:

1	5	1	2	6	2	3	4	1 ... 5	6	2	5	6

`row_ptr`-array:

1	3	6	9	13	17	20

(2-25.a) ⚀ (20 min.) In the file `crstril.hpp` implement a C++ function

```
Eigen::MatrixXd CRSToDense(const CRSMat &A);
```

that converts a matrix given in CRS format by means of a **CRSMat** object `A` into the dense matrix type **Eigen::MatrixXd**.

SOLUTION for (2-25.a) → `2-25-1-0:crss0.pdf` ▲

For a square matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ we define the extraction of its lower triangular part,

$$\mathrm{tril} : \mathbb{R}^{n,n} \to \mathbb{R}^{n,n} \quad , \quad (\mathrm{tril}(\mathbf{A}))_{i,j} = \begin{cases} (\mathbf{A})_{i,j} & \text{, if } i \geq j\,, \\ 0 & \text{, if } i < j\,, \end{cases} \quad 1 \leq i,j \leq n\,.$$

(2-25.b) ⚃ (25 min.) Complete the code for the C++ function

```
std::vector<double> solveTrilA(const CRSMat &A,
                               const std::vector<double> &b);
```

listed below. It is supposed to return the solution $\mathbf{x} := \mathrm{tril}(\mathbf{A})^{-1}\mathbf{b}$ of the linear system of equations

$$\mathrm{tril}(\mathbf{A})\,\mathbf{x} = \begin{bmatrix} a_{11} & 0 & \dots & \dots & \dots & 0 \\ a_{21} & a_{22} & \ddots & & & \vdots \\ a_{31} & a_{32} & a_{33} & \ddots & & \vdots \\ \vdots & & & \ddots & \ddots & \vdots \\ \vdots & & & & \ddots & 0 \\ a_{n1} & a_{n2} & \dots & \dots & a_{n,n-1} & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{bmatrix} = \mathbf{b}\,.$$

The argument `A` passes $\mathbf{A}$ in CRS format, while `b` suppplies the right-hand side vector.

```
std::vector<double> solveTrilA(const CRSMat &A, const std::vector<double> &b) {
  const unsigned int n = A.n;
  assertm((A.n == A.m), "Matrix must be square");
  assertm((A.n == b.size()), "Matrix/vector size mismatch");
  std::vector<double> x = b; // Initialize result vector
  for (unsigned int i = 0; i < n; ++i) {
    double aii = 0.0;  // diagonal element
    for (unsigned int j = A.row_ptr[i]; j < A.row_ptr[i + 1]; ++j) {
      if (A.col_ind[j] <            ) {
        x[i] -=                                        ;
      } else if (A.col_ind[j] == i) {
        aii =                             ;
      }
    }
    if (aii != 0.0) {
                                              ;
    } else {
      throw std::runtime_error("Solve failed: zero diagonal entry encountered!");
    }
  }
  return x;
}
```

SOLUTION for (2-25.b) → `2-25-2-0:crstsx.pdf` ▲

(2-25.c) ⚀ (10 min.) What is the asymptotic complexity of `solveTrilA()` in terms of the size of the input.

$$\mathrm{cost}(\texttt{solveTrilA(A)}) = O\Big(\quad\Big) \quad \text{for} \quad \quad \to \infty\,.$$

SOLUTION for (2-25.c) → 2-25-3-0:crsts1a.pdf ▲

(2-25.d) ⚁ (30 min.) Write a C++ function

```
bool testSolveTrilA(const CRSMat &A, double tol = 1E-10);
```

which tests the proper function of `solveTrilA()` for the sparse matrix passed in `A` and a right-hand side vector whose entries are random numbers, independently uniformly distributed in $[0,1]$. To compute a reference solution for the triangular system, use EIGEN's elimination solver, see [Lecture → Rem. 2.5.0.5].

The function should return **true**, if the norm of the difference of the two results is smaller than `tol` times the norm of the reference solution.

HIDDEN HINT 1 for (2-25.d) → 2-25-4-0:crsth2.pdf

SOLUTION for (2-25.d) → 2-25-4-1:crssthree.pdf ▲

End Problem 2-25, 85 min.

Chapter 3

Direct Methods for Linear Least Squares Problems

Problem 3-1: Hidden linear regression

This problem is about a hidden linear regression: in fact, at first glance it will seem that we are dealing with a nonlinear system of equations. However, we will be able to reduce the system to a linear form.

Linear regression was introduced in [Lecture → Ex. 3.0.1.1] and [Lecture → Ex. 3.0.1.4]. You have to be familiar with normal equations, see [Lecture → Section 3.1.2] and, in particular, [Lecture → Ex. 3.1.2.4]. This problems also addresses the solution of linear least-squares problems in EIGEN, see [Lecture → Code 3.2.0.1] and [Lecture → Code 3.3.4.2].

We consider the function $f(t) = \alpha e^{\beta t}$ with unknown parameters $\alpha > 0$, $\beta \in \mathbb{R}$. Given are measurements (t_i, y_i), $i = 1, \ldots, n$, $2 < n \in \mathbb{N}$. We want to determine $\alpha, \beta \in \mathbb{R}$ such that $f(t)$ fulfills the following condition "to the best of its ability" (in the least square sense):

$$f(t_i) = y_i, \quad \text{with} \quad y_i > 0,\ i = 1, \ldots, n\,. \tag{3.1.1}$$

(3-1.a) ⚀ (10 min.) Which overdetermined system of *nonlinear* equations directly stems from (3.1.1)?

SOLUTION for (3-1.a) → 3-1-1-0:AdaptedLinReg1s.pdf ▲

(3-1.b) ⚁ (15 min.) In which overdetermined system of *linear* equations can you transform the nonlinear system derived in (3-1.a)?

HIDDEN HINT 1 for (3-1.b) → 3-1-2-0:AdaptedLinReg2h.pdf

SOLUTION for (3-1.b) → 3-1-2-1:AdaptedLinReg2s.pdf ▲

(3-1.c) ⚁ (15 min.) Determine the **normal equations** for the overdetermined linear system of (3-1.b).

HIDDEN HINT 1 for (3-1.c) → 3-1-3-0:AdaptedLinReg3h.pdf

SOLUTION for (3-1.c) → 3-1-3-1:AdaptedLinReg3s.pdf ▲

(3-1.d) ⚂ (20 min.) Consider the data $(t_i, y_i) \in \mathbb{R}^2$, $i = 1, \ldots, m$, stored in two vectors `t` and `y` of equal length. In the file `adaptedlinreg.hpp` implement a C++ function

```
Eigen::VectorXd linReg(const Eigen::VectorXd &t, const
   Eigen::VectorXd &y);
```

that uses the *method of normal equations* discussed in [Lecture → Section 3.2] to solve the 1D linear regression problem as described in [Lecture → Ex. 3.0.1.1] in least-squares sense. This function should return the estimated parameters α and β in a vector of length 2.

SOLUTION for (3-1.d) → 3-1-4-0:AdaptedLinReg4s.pdf ▲

(3-1.e) ⚄ (15 min.) [depends on Sub-problem (3-1.d), Sub-problem (3-1.a)]

Now consider the data (t_i, y_i) stored in two vectors `t` and `f` of equal length. Implement a C++ function

```
Eigen::VectorXd expFit(const Eigen::VectorXd &t, const
   Eigen::VectorXd &y);
```

that returns a least squares estimate of α and β, given the nonlinear relationship (3.1.1) between $\mathbf{t}$ and $\mathbf{f}$:

HIDDEN HINT 1 for (3-1.e) → 3-1-5-0:AdaptedLinReg5h.pdf

SOLUTION for (3-1.e) → 3-1-5-1:AdaptedLinReg5s.pdf ▲

End Problem 3-1 , 75 min.

Problem 3-2: Estimating a Tridiagonal Matrix

In [Lecture → Section 3.1] we learned that to determine the least squares solution of an overdetermined linear system of equations $\mathbf{Ax} = \mathbf{b}$ we minimize the residual norm $\|\mathbf{Ax} - \mathbf{b}\|_2$ w.r.t. $\mathbf{x}$. However, we also face a linear least squares problem when minimizing the residual norm w.r.t. the entries of $\mathbf{A}$. This is the unusual situation considered in this problem.

This is an exercise in converting a problem statement into a proper linear least squares problem in the form of an overdetermined linear system of equations as in [Lecture → Section 3.0.1] and then solving it through the normal equations, see [Lecture → Section 3.2].

The following least-squares problem arises in the estimation of material parameters in one-dimensional diffusion problems: Let two vectors $\mathbf{z}, \mathbf{c} \in \mathbb{R}^n$, $n > 2 \in \mathbb{N}$ of measured values be given. The two real numbers α^* and β^* are defined as:

$$(\alpha^*, \beta^*) = \operatorname*{argmin}_{\alpha,\beta\in\mathbb{R}} \left\| \mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} \right\|_2 , \tag{3.2.1}$$

where $\mathbf{T}_{\alpha,\beta} \in \mathbb{R}^{n\times n}$ is the following tridiagonal matrix

$$\mathbf{T}_{\alpha,\beta} = \begin{bmatrix} \alpha & \beta & 0 & \dots & 0 \\ \beta & \alpha & \beta & \ddots & \vdots \\ 0 & \beta & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha & \beta \\ 0 & \dots & 0 & \beta & \alpha \end{bmatrix} . \tag{3.2.2}$$

(3-2.a) ⚀ (20 min.) Reformulate (3.2.1) as a linear least squares problem in standard form according to [Lecture → Def. 3.1.1.1]

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^k} \|\mathbf{Ax} - \mathbf{b}\|_2 \tag{3.2.3}$$

For appropriate choice of $m, k \in \mathbb{N}$ define suitable $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{x} \in \mathbb{R}^k$ and $\mathbf{b} \in \mathbb{R}^m$, with $m, k \in \mathbb{N}$.

HIDDEN HINT 1 for (3-2.a) → 3-2-1-0:TridiagLeastSquares1h.pdf

SOLUTION for (3-2.a) → 3-2-1-1:TridiagLeastSquares1s.pdf ▲

(3-2.b) ⚁ (15 min.) [depends on Sub-problem (3-2.a)]

Explicitly state the normal equations [Lecture → Thm. 3.1.2.1] associated with the linear least-squares problem (3.2.3).

SOLUTION for (3-2.b) → 3-2-2-0:s1b.pdf ▲

(3-2.c) ⚁ (30 min.) [depends on Sub-problem (3-2.b)]

Write a C++ function

```
Eigen::VectorXd lsqEst(const Eigen::VectorXd &z,
                       const Eigen::VectorXd &c);
```

that computes the optimal (in least-squares sense) parameters α^* and β^* according to (3.2.1) from data vectors `z` and `c` (i.e. $\mathbf{z}$ and $\mathbf{c}$ from (3.2.1)). Use the *method of normal equations*.

HIDDEN HINT 1 for (3-2.c) → 3-2-3-0:TridiagLeastSquares2h.pdf

After the matrix $\mathbf{A}$ has been initialized, the solution just copies [Lecture → Code 3.2.0.1].

SOLUTION for (3-2.c) → 3-2-3-1:TridiagLeastSquares2s.pdf ▲

End Problem 3-2 , 65 min.

Problem 3-3: Linear Data Fitting

Abstract linear data fitting and its connection to the linear least squares problem [Lecture → (3.1.3.7)] is discussed in [Lecture → Ex. 3.0.1.1]. In this problem we practise these techniques for a concrete example.

This problem relies on the methods and EIGEN facilities introduced in [Lecture → Section 3.2] and [Lecture → Section 3.3.4].

For certain points in time t_i we have measured data f_i for a physical quantity f whose time-dependence $t \mapsto f(t)$ is only known qualitatively, that is, up to a few unknown parameters, see [Lecture → Ex. 3.0.1.1]:

t_i	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
f_i	100	34	17	12	9	6	5	4	4	2

$f(t)$ is assumed to be a linear combination

$$f(t) = \sum_{j=1}^{4} \gamma_j \phi_j(t) \tag{3.3.1}$$

of the functions

$$\phi_1(t) = \frac{1}{t}, \quad \phi_2(t) = \frac{1}{t^2}, \quad \phi_3(t) = \mathrm{e}^{-(t-1)}, \quad \phi_4(t) = \mathrm{e}^{-2(t-1)} . \tag{3.3.2}$$

We aim for calculating the coefficients γ_j, $j = 1,2,3,4$, such that

$$\sum_{i=1}^{10} (f(t_i) - f_i)^2 \longrightarrow \min . \tag{3.3.3}$$

(3-3.a) ⚀ (30 min.) In the file `linfit.hpp` write a C++ function

```
Eigen::VectorXd data_fit_normal(const Eigen::VectorXd &t, const
   Eigen::VectorXd &f);
```

that computes a least squares estimate for the coefficients γ_i by related to the data (3.3.2) and (3.3.3) by means of solving the *normal equations*.

SOLUTION for (3-3.a) → `3-3-1-0:ldf1.pdf` ▲

(3-3.b) ⚁ (30 min.) Write a C++ function

```
Eigen::VectorXd data_fit_qr(const Eigen::VectorXd &t, const
   Eigen::VectorXd &f);
```

that uses *orthogonal transformation techniques* as introduced in [Lecture → Section 3.3.4] for the same task as in the previous sub-problem.

SOLUTION for (3-3.b) → `3-3-2-0:ldf2.pdf` ▲

(3-3.c) ⚂ (15 min.) Write a C++ function

```
Eigen::VectorXd fitted_function(const Eigen::VectorXd &gamma,
   const Eigen::VectorXd &t_vec)
```

that takes the estimated parameters γ_i, $i = 1,2,3,4$, as argument vector `gamma`, a vector `t_vec` of times τ_j, and returns the vector of values $f(\tau_j)$.

SOLUTION for (3-3.c) → `3-3-3-0:ldf3.pdf` ▲

(3-3.d) ⚄ (15 min.) Using EIGEN write a C++ function

```
Eigen::VectorXd fitting_error(const Eigen::VectorXd &gamma);
```

that returns the vector

$$\left[(f(t_i) - f_i)^2\right]_{i=1}^{10} \in \mathbb{R}^{10}$$

where f is defined according to (3.3.1) using the values for the γ_j-coefficients passed in the `gamma` argument.

SOLUTION for (3-3.d) → `3-3-4-0:s5.pdf` ▲

End Problem 3-3 , 90 min.

Problem 3-4: QR-Factorization of Tridiagonal Matrices

In [Lecture → Rem. 3.3.3.27] we saw that a QR-factorization of a tri-diagonal matrix features an upper triangular factor that is tridiagonal again. In this exercise we exploit this fact for the design of efficient algorithms, in particular for the stable solution of tridiagonal linear systems of equations.

This exercise assume familiarity with the concept of QR-decomposition/factorization [Lecture → Section 3.3.3] and, in particular, Givens rotations [Lecture → § 3.3.3.15] and their compressed storage [Lecture → Rem. 3.3.3.22].

Throughout this problem, in order to be compatible with some software, for generic real-valued square tridiagonal matrices we are supposed to use the data structure

```
#include <Eigen/Dense>
struct TriDiagonalMatrix {
  Eigen::Index n; // Matrix size:  n × n
  Eigen::VectorXd d; // n-vector of diagonal entries
  Eigen::VectorXd l; // n−1-vector, entries of first lower diagonal
  Eigen::VectorXd u; // n−1-vector, entries of first upper diagonal
  // Simple constructor
  TriDiagonalMatrix(const Eigen::VectorXd &d,
  const Eigen::VectorXd &l,
  const Eigen::VectorXd &u)
  : n(d.size()), d(d), l(l), u(u) {
    assert((n-1) == l.size() && (n-1) == u.size());
  }
};
```

Note that in full-fledged C++ implementations, this would be a C++ class with four different constructors and a destructor explicitly defined, private data members, and access member functions. The rudimentary definition above is proposed solely for the sake of simplicity.

Let us write $\mathbf{d} \in \mathbb{R}^n$, $\mathbf{l} \in \mathbb{R}^{n-1}$, and $\mathbf{u} \in \mathbb{R}^{n-1}$ for the vectors stored in `d`, `l`, `u`. Then the matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in an **TriDiagonalMatrix** object is defined by

$$(\mathbf{A})_{i,j} := \begin{cases} (\mathbf{d})_i & \text{for } i = j\,, \\ (\mathbf{u})_i & \text{for } j = i+1\,, \\ (\mathbf{l})_j & \text{for } i = j+1\,, \\ 0 & \text{else,} \end{cases} \qquad i,j \in \{1,\ldots,n\}\,.$$

$$\Leftrightarrow \quad \mathbf{A} = \begin{bmatrix} d_1 & u_1 & & \\ l_1 & d_2 & \ddots & \\ & \ddots & \ddots & u_{n-1} \\ & & l_{n-1} & d_n \end{bmatrix}, \quad \begin{aligned} \mathbf{d} &= [d_1,\ldots,d_n]^\top \in \mathbb{R}^n\,, \\ \mathbf{u} &= [u_1,\ldots,u_{n-1}]^\top \in \mathbb{R}^{n-1}\,, \\ \mathbf{l} &= [l_1,\ldots,l_{n-1}]^\top \in \mathbb{R}^{n-1}\,. \end{aligned}$$

(3-4.a) ⚀ (30 min.) Prove that for a *tridiagonal* invertible matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ a QR-factorization $\mathbf{A} = \mathbf{QR}$, $\mathbf{Q} \in \mathbb{R}^{n,n}$ orthogonal, $\mathbf{R} \in \mathbb{R}^{n,n}$ upper triangular with positive diagonal entries, satisfies

$$(\mathbf{R})_{i,j} = 0 \quad \text{for} \quad j > i+2 \quad \text{and} \quad (\mathbf{Q})_{i,j} = 0 \quad \text{for} \quad i > j+1\,. \tag{3.4.1}$$

HIDDEN HINT 1 for (3-4.a) → `3-4-1-0:s1h1.pdf`

SOLUTION for (3-4.a) → 3-4-1-1:s1.pdf ▲

To store a QR-factorization $\mathbf{A} = \mathbf{QR}$ of a real-valued tridiagonal matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ we use the (again rudimentary) data structure

```
class TriDiagonalQR {
  public:
  explicit TriDiagonalQR(const TriDiagonalMatrix &A);

  template <typename VecType>
  Eigen::VectorXd applyQT(const VecType &x) const;
  template <typename VecType>
  Eigen::VectorXd solve(const VecType &b) const;
  // For debugging purposes:  extract factors as dense matrices
  std::pair<Eigen::MatrixXd,Eigen::MatrixXd> getQRFactors(void)
     const;

  private:
  Eigen::Index n; // size of the square matrix
  Eigen::Matrix B; // Three non-zero upper diagonals of R
  Eigen::VectorXd rho; // Encoded Givens rotations

};
```

Here the non-zero entries of $\mathbf{R}$ are stored in an $n \times 3$-matrix $\mathbf{B}$ adopting the convention

$$(\mathbf{R})_{i,j} := \begin{cases} (\mathbf{B})_{i,1} & \text{, if } i = j\,, \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,2} & \text{, if } j = i+1\,, \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,3} & \text{, if } j = i+2\,, \\ 0 & \text{else,} \end{cases} \qquad i,j \in \{1,\ldots,n\}\,, \tag{3.4.6}$$

that is, $\mathbf{R} = \begin{bmatrix} x_1 & y_1 & z_1 & & & & \\ & x_2 & y_2 & z_2 & & & \\ & & x_3 & y_3 & z_3 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & x_{n-2} & y_{n-2} & z_{n-2} \\ & & & & & x_{n-1} & y_{n-1} \\ & & & & & & x_n \end{bmatrix} \Rightarrow \mathbf{B} = \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_{n-2} & y_{n-2} & z_{n-2} \\ x_{n-1} & y_{n-1} & 0 \\ x_n & 0 & 0 \end{bmatrix}$.

Note that three entries of $\mathbf{B}$ will never be used.

The $\mathbf{Q}$-factor is stored in compressed format through the underlying $n-1$ Givens rotations whose target rows are clear a priori, see [Lecture → Rem. 3.3.3.27]. Every Givens rotation is stored through a single number ρ following the convention described in [Lecture → Rem. 3.3.3.22]. These $n-1$ numbers are collected in the data member `rho`. Its j-th entry corresponds to the Givens rotation acting on rows j & $j+1$.

(3-4.b) ⚄ (20 min.) Implement a C++ function

```
std::tuple<double,double,double>
  compGivensRotation(Eigen::Vector2d a);
```

that computes a Givens rotation mapping the vector $\mathbf{a} \in \mathbb{R}^2$ onto a multiple of the first Cartesian coordinate vector $\mathbf{e}_1 \in \mathbb{R}^2$. Use the algorithm implemented in [Lecture → Code 3.3.3.17]. The

function is to return the triplet (ρ,γ,σ), where $\mathbf{G} = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}$ is the transformation matrix such that $\mathbf{G}^\top \mathbf{a} = (\pm\|\mathbf{a}\|, 0)^\top$, and $\rho \in \mathbb{R}$ encodes it according to the rule set

$$\rho := \begin{cases} 1 & \text{, if } \gamma = 0\,, \\ \frac{1}{2}\operatorname{sign}(\gamma)\sigma & \text{, if } |\sigma| < |\gamma|\,, \\ 2\operatorname{sign}(\sigma)/\gamma & \text{, if } |\sigma| \geq |\gamma|\,, \end{cases} \qquad \text{[Lecture} \rightarrow \text{(3.3.3.23)]}$$

SOLUTION for (3-4.b) → 3-4-2-0:s2.pdf ▲

(3-4.c) ⚅ (45 min.) [depends on [Lecture → Rem. 3.3.3.27]]

Write an *efficient* code for the constructor of **TriDiagonalQR**, which initializes the data members, that is, computes the QR-factorization of the tridiagonal matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ passed to it in `A`.

HIDDEN HINT 1 for (3-4.c) → 3-4-3-0:h2a.pdf

HIDDEN HINT 2 for (3-4.c) → 3-4-3-1:h2b.pdf

SOLUTION for (3-4.c) → 3-4-3-2:s2a.pdf ▲

(3-4.d) ⚅ (30 min.) Provide an *efficient* implementation of the method `applyQT()` of **TriDiagonalQR** that computes $\mathbf{Q}^\top\mathbf{x}$ for a supplied vector $\mathbf{x} \in \mathbb{R}^n$ (passed in `x`).

The type **VecType** must describe a vector with real-valued entries supplying component access through **operator** `[]` **const** and a `size()` method.

HIDDEN HINT 1 for (3-4.d) → 3-4-4-0:sp3h1.pdf

SOLUTION for (3-4.d) → 3-4-4-1:s3.pdf ▲

(3-4.e) ⚅ (30 min.) [depends on Sub-problem (3-4.d) and [Lecture → Rem. 3.3.4.3]]

Devise an efficient implementation of the method `solve()` of **TriDiagonalQR** that accepts a vector $\mathbf{b} \in \mathbb{R}^n$ as arguments and solves the linear systems of equations $\mathbf{Ax} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ the tridiagonal matrix stored in the current **TriDiagonalQR** object, returning the solution $\mathbf{x} \in \mathbb{R}^n$. A **std::runtime_error** exception should be thrown in case the matrix $\mathbf{A}$ is not invertible.

HIDDEN HINT 1 for (3-4.e) → 3-4-5-0:h41.pdf

SOLUTION for (3-4.e) → 3-4-5-1:s4.pdf ▲

(3-4.f) ⚂ (10 min.) [depends on Sub-problem (3-4.b), Sub-problem (3-4.d), Sub-problem (3-4.e)]

What is the asymptotic computational effort for your implementations of the constructor of **TriDiagonalQR** and of the methods `applyQT()` and `solve()` in terms of $n \to \infty$?

SOLUTION for (3-4.f) → 3-4-6-0:s5.pdf ▲

(3-4.g) ⚂ (30 min.) The following templated C++ function implements (in a non-optimal way) a so-called **inverse iteration** [Lecture → Section 9.3.2].

```
template <typename MatrixType>
unsigned int invit(const MatrixType &A,Eigen::VectorXd &x,
double TOL = 1E-6, unsigned int maxit = 100) {
  x.normalize();
  Eigen::VectorXd x_old(x.size());
  int it = 0;
  do {
```

```
    x_old = x;
    x = A.lu().solve(x_old);
    x.normalize();
  }
  while (((x-x_old).norm() > TOL) && (it++ < maxit));
  return it;
}
```

Implement a **specialization** (C++ reference) of this function for **MatrixType = TriDiagonalMatrix**.

HIDDEN HINT 1 for (3-4.g) → 3-4-7-0:s6h1.pdf

SOLUTION for (3-4.g) → 3-4-7-1:.pdf ▲

End Problem 3-4 , 195 min.

Problem 3-5: Conditional minimum with SVD

A *Singular Value Decomposition* (SVD, see [Lecture → Def. 3.4.1.3]) is the generalized version of the diagonalization process for any $m \times n$ matrix. We will see that the diagonal and orthogonal matrices arising from an SVD have certain properties such that it is immediate to identify the minimum of a least squares functional for that matrix. In other words, computing the SVD of a matrix is equivalent to solving its corresponding least square problem.

The solution of this problem is discussed in [Lecture → Section 3.4.4.1]; the algorithm is implemented in [Lecture → Code 3.4.4.2].

Let $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ be the singular value decomposition of $\mathbf{A} \in \mathbb{C}^{m,n}$, $m \geq n$. Consider the following problem:

$$\mathbf{x}^* = \operatorname*{argmin}_{\|\mathbf{x}\|_2=1} \{\|\mathbf{A}\mathbf{x}\|_2\} \tag{3.5.1}$$

(3-5.a) ⚄ (45 min.) Show that the solution of (3.5.1) is:

$$\mathbf{x}^* = \mathbf{v}_n := \mathbf{V}_{:,n} \tag{3.5.2}$$

Show that the following holds:

$$\|\mathbf{A}\mathbf{v}_n\|_2 = \sigma_n := \mathbf{\Sigma}_{n,n}. \tag{3.5.3}$$

HIDDEN HINT 1 for (3-5.a) → 3-5-1-0:MinimumSVD1h.pdf

SOLUTION for (3-5.a) → 3-5-1-1:MinimumSVD1s.pdf ▲

End Problem 3-5, 45 min.

Problem 3-6: Sparse Approximate Inverse (SPAI [GH97])

This problem studies the least squares aspects of the SPAI method, a technique used in the numerical solution of partial differential equations. We encounter an "exotic" sparse matrix technique and rather non-standard least squares problems. Please note that the matrices to which SPAI techniques are applied will usually be huge and extremely sparse, say, of dimension $10^7 \times 10^7$ with only 10^8 non-zero entries. Therefore sparse matrix techniques must be applied.

Requires only familiarity with [Lecture → Section 3.1] and [Lecture → Section 3.2].

Sub-problem (3-6.d) is connected with [Lecture → Section 10.3]. In case this section had not (yet) been covered in the course that sub-problem should be skipped.

Let $\mathbf{A} \in \mathbb{R}^{N,N}$, $N \in \mathbb{N}$, be a regular sparse matrix with at most $n \ll N$ non-zero entries per row and column. We define the space of matrices with the same pattern (on non-zero entries) as $\mathbf{A}$:

$$\mathcal{P}(\mathbf{A}) := \{\mathbf{X} \in \mathbb{R}^{N,N} : \ (\mathbf{A})_{ij} = 0 \ \Rightarrow \ (\mathbf{X})_{ij} = 0\} \ . \tag{3.6.1}$$

The "primitive" SPAI (sparse approximate inverse) $\mathbf{B}$ of $\mathbf{A}$ is defined as

$$\mathbf{B} := \operatorname*{argmin}_{\mathbf{X} \in \mathcal{P}(\mathbf{A})} \|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F \ , \tag{3.6.2}$$

where $\|\cdot\|_F$ stands for the Frobenius norm [Lecture → Def. 3.4.4.17]. The solution of (3.6.2) can be used as a so-called preconditioner for the acceleration of iterative methods for the solution of linear systems of equations, see [Lecture → Chapter 10]. An extended "self-learning" variant of the SPAI method is presented in [GH97].

(3-6.a) ⚀ (20 min.) Show that the columns of $\mathbf{B}$ can be computed independently of each other by solving linear least squares problems. In the statement of these linear least squares problems write $\mathbf{b}_i$ for the columns of $\mathbf{B}$.

HIDDEN HINT 1 for (3-6.a) → 3-6-1-0:spaih1.pdf

SOLUTION for (3-6.a) → 3-6-1-1:spai1.pdf ▲

(3-6.b) ⚅ (60 min.) [depends on Sub-problem (3-6.a)]

Implement an efficient C++ function

```
Eigen::SparseMatrix<double> spai(const
   Eigen::SparseMatrix<double>& A);
```

for the computation of $\mathbf{B}$ according to (3.6.2). You may rely on the normal equations associated with the linear least squares problems for the columns of $\mathbf{B}$ or you may simply invoke the least squares solver of EIGEN, see [Lecture → Code 3.4.3.14].

HIDDEN HINT 1 for (3-6.b) → 3-6-2-0:spai2h.pdf

SOLUTION for (3-6.b) → 3-6-2-1:spai2s.pdf ▲

(3-6.c) ⚂ (15 min.) [depends on Sub-problem (3-6.b)]

What is the total asymptotic computational effort of your implementation of `spai()` in terms of the problem size parameters N and n.

SOLUTION for (3-6.c) → 3-6-3-0:SPAI3.pdf

▲

(3-6.d) ⚄ (45 min.) [depends on Sub-problem (3-6.b)]

For $n \in \mathbb{N}$, consider a sparse s.p.d. [Lecture → Def. 1.1.2.6] matrix $\mathbf{A}$ given by the following C++-code

C++11-code 3.6.5: Initialization of matrix $\mathbf{A}$

```
2  Eigen::SparseMatrix<double> init_A(unsigned int n) {
3    Eigen::SparseMatrix<double> L(n, n);
4    Eigen::SparseMatrix<double> R(n, n);
5    L.reserve(n);
6    R.reserve(n + 2 * (n - 1));
7    L.insert(0, 0) = std::exp(1. / n);
8    R.insert(0, 0) = 2.;
9    for (unsigned int i = 1; i < n; ++i) {
10     L.insert(i, i) = std::exp((i + 1.) / n);
11     R.insert(i, i) = 2.;
12     R.insert(i - 1, i) = -1.;
13     R.insert(i, i - 1) = -1.;
14   }
15   Eigen::SparseMatrix<double> A = kroneckerProduct(L, R);
16   A.makeCompressed();
17   return A;
18 }
```

Get it on GitLab (spai.hpp).

Solve the linear system of equations $\mathbf{Ax} = \mathbf{b}$ using EIGEN's **ConjugateGradient** (EIGEN documentation) iterative solver with initial guess $\mathbf{0}$ and default tolerance. Do this without a preconditioner and using the preconditioner $\frac{1}{2}(\mathbf{B} + \mathbf{B}^T)$, where $\mathbf{B}$ solves (3.6.2). Use $\mathbf{b} = [1,1,\ldots,1]^\top$. Write a C++ function

```
std::vector<std:pair<unsigned int, unsigned int>>
testSPAIPrecCG(unsigned int L);
```

that returns the number of CG iterations (with and without the SPAI preconditioner) versus the system size $N = n^2$ for $n = 2^{1,\ldots,L}$.

HIDDEN HINT 1 for (3-6.d) → 3-6-4-0:spaih4.pdf

HIDDEN HINT 2 for (3-6.d) → 3-6-4-1:spaih4.pdf

SOLUTION for (3-6.d) → 3-6-4-2:spai4.pdf

▲

References

[GH97] Marcus J. Grote and Thomas Huckle. "Parallel preconditioning with sparse approximate inverses". In: *SIAM J. Sci. Comput.* 18.3 (1997), pp. 838–853. DOI: 10.1137/S1064827594276552 (cit. on p. 94).

End Problem 3-6, 140 min.

Problem 3-7: Shape identification

This problem deals with pattern recognition, a central problem in image processing (e.g. in aerial photography, self-driving cars, and recognition of pictures of cats). We will deal with a very simplistic problem.

This problem practises the solution of a linear least squares problem based on the normal-equation method, see [Lecture → Section 3.2].

A routine within the program of a self driving-car is tasked with the job of identifying road signs. The task is the following: given a collection of points $\mathbf{p}_i \in \mathbb{R}^2, i = 1, \ldots, n$, we have to decide whether those point represent a stop sign, or a "priority road sign". In this exercise we consider $n = 8$.

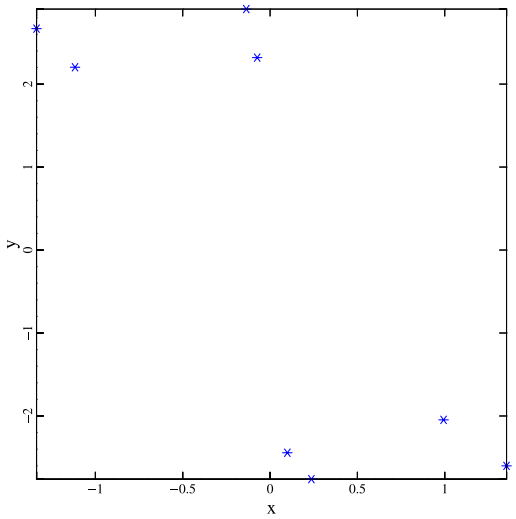


Fig. 11

A set of possible input points $\mathbf{p}^i$: corners detected in the photo.

Which traffic sign had been in the focus of the camera?

Fig. 12

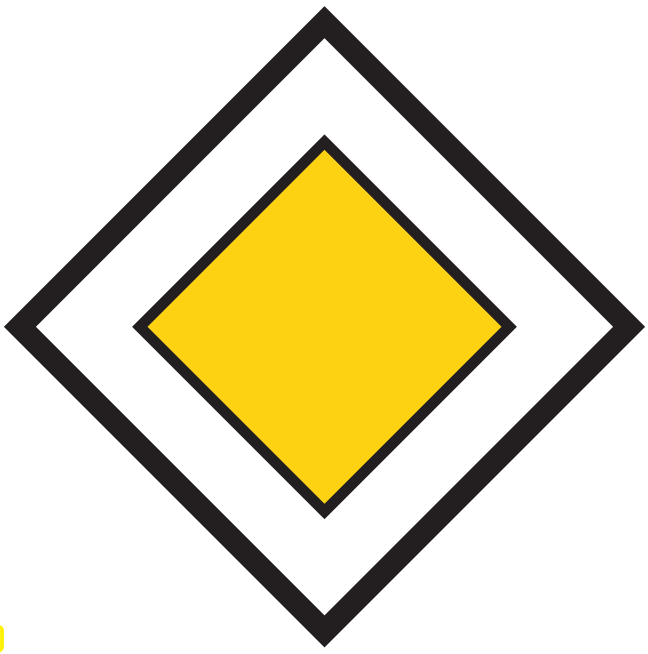


Fig. 13

The shape of the sign can be represented by a 2×8 matrix with the 8 coordinates in $\mathbb{R}^2$ defining the shape of the sign. We assume that the stop sign resp. the priority road sign are defined by the "model" points (known a priori) $\mathbf{x}^i_{stop} \in \mathbb{R}^2$ resp. $\mathbf{x}^i_{priority} \in \mathbb{R}^2$ for $i = 1, \ldots, 8$.

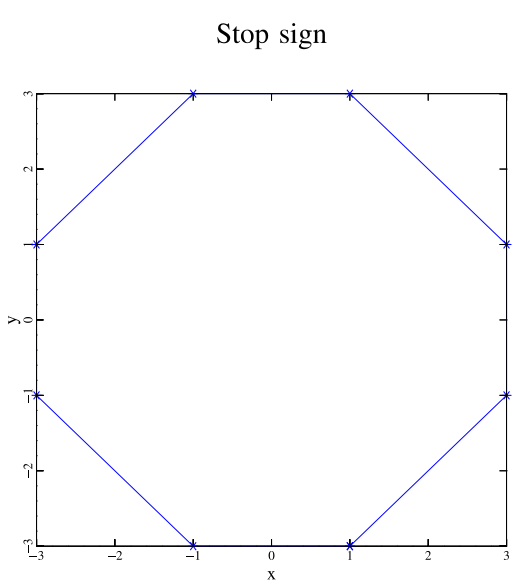


Fig. 14

The 8-points $\mathbf{x}^i_{stop}$ defining the model of a stop sign.

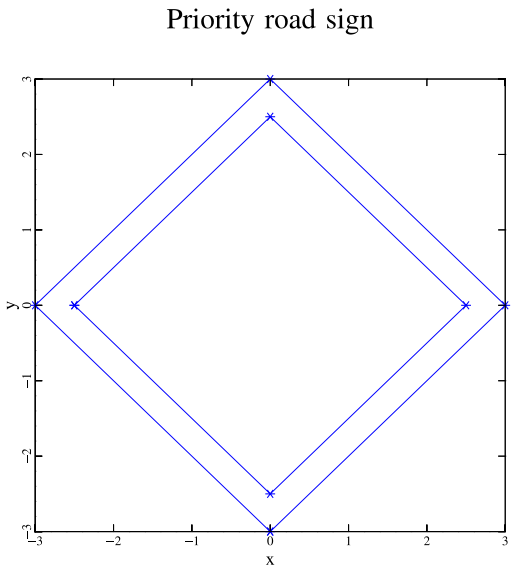


Fig. 15

The 8-points $\mathbf{x}^i_{priority}$ defining the model of a priority road sign.

However, in a real case scenario, one can imagine that the photographed shape is not exactly congruent to the one specified by the "model" points. Instead, one can assume that the points on the photo ($\mathbf{p}^i \in \mathbb{R}^2$) are the result of a *linear transformation* of the original points $\mathbf{x}^i \in \mathbb{R}^2$ for $i = 1, \ldots, 8$; i.e. we can assume that there exists a matrix $\mathbf{A} \in \mathbb{R}^{2,2}$, s.t.

$$\mathbf{p}^i = \mathbf{A}\mathbf{x}^i, i = 1, \ldots, n \,. \tag{3.7.1}$$

We do not know whether $\mathbf{x}^i = \mathbf{x}^i_{stop}$ or $\mathbf{x}^i = \mathbf{x}^i_{priority}$.

Moreover, we have some error in the data, i.e. our points do not represent exactly one of the linearly transformed shapes (it is plausible to imagine that there is some measurement error, and that the photographed shape is not exactly the same as our "model" shape), i.e. (3.7.1) is satisfied only "approximately".

With this problem, we will try to use the least square method to find the matrix $\mathbf{A}$ and to *classify* our points, i.e. to decide whether they represent a stop or a priority road sign.

(3-7.a) ⚄ (20 min.) For the moment, assume we know the points $\mathbf{x}^i$ (i.e. we know the shape of our sign). Explicitly write (3.7.1) as an overdetermined linear system:

$$\mathbf{B}\mathbf{v} = \mathbf{w} \,,$$

whose least-squares solution $\mathbf{v}^*$ will allow to determine the "best" linear transformation $\mathbf{A}$ (in the least-squares sense). How is the vector of unknowns related to $\mathbf{A}$?

SOLUTION for (3-7.a) → 3-7-1-0:ShapeIdent1.pdf ▲

(3-7.b) ⚅ (10 min.) When does the matrix $\mathbf{B}$ have full rank? Give a geometric interpretation of this condition.

SOLUTION for (3-7.b) → 3-7-2-0:si2s.pdf ▲

(3-7.c) ⚅ (15 min.) [depends on Sub-problem (3-7.a)]

In the file `shape_ident.hpp` implement a C++ function

```
Eigen::MatrixXd shape_ident_matrix(const Eigen::MatrixXd & X);
```

that returns the matrix $\mathbf{B}$. Pass the vectors $\mathbf{x}^i$ in the columns of a $2 \times n$ EIGEN matrix `X`.

SOLUTION for (3-7.c) → 3-7-3-0:si3s.pdf ▲

(3-7.d) ⚄ (30 min.) [depends on Sub-problem (3-7.c)]

In the file `shape_ident.hpp` implement a C++ function

```
double solve_lsq(const Eigen::MatrixXd & X, const Eigen::MatrixXd
    & P,
                 Eigen::MatrixXd & A);
```

that computes the matrix $\mathbf{A}$ by solving the least squares problem based on the normal equations, see [Lecture → Section 3.2]. It should return the Euclidean norm of the residual of the least square solution. Pass the vectors $\mathbf{x}^i$ and the vectors $\mathbf{p}^i$ as a $2 \times n$ EIGEN matrix `X` resp. `P`.

SOLUTION for (3-7.d) → 3-7-4-0:si4s.pdf ▲

(3-7.e) ⚄ (30 min.) [depends on Sub-problem (3-7.d)]

Explain how the norm of the residual of the least square solution can be used to identify the shape defined by the points $\mathbf{p}^i$. Implement a function

```
enum Shape { Stop, Priority };
Shape identify(const Eigen::MatrixXd Xstop,
               const Eigen::MatrixXd Xpriority,
               const Eigen::MatrixXd & P,
               Eigen::MatrixXd & A);
```

that identifies the shape (either Stop or Priority sign) of the input points $\mathbf{p}^i$. The function returns an `enum` that classifies the shape of points specified by `P`. Return the linear transformation in the matrix `A`. The "model points" $\mathbf{x}^i_{stop}$ resp. $\mathbf{x}^i_{priority}$ are passed through `Xstop` resp. `Xpriority`.

SOLUTION for (3-7.e) → 3-7-5-0:si5s.pdf ▲

End Problem 3-7, 105 min.

Problem 3-8: Low rank approximation of matrices

As explained in the course, large $m \times n$ matrices of low rank $k \ll \min\{m,n\}$ can be stored using only $k(m+n)$ real numbers when using their SVD factorization/representation as sum of tensor products [Lecture → (3.4.1.9)]. Thus, low rank matrices are of considerable interest for *matrix compression* (see [Lecture → Ex. 3.4.4.24]). Unfortunately, adding two low rank matrices usually leads to an increase of the rank and entails "recompression" by computing a low-rank best approximation of the sum. This problems demonstrates an efficient approach to recompression.

Study [Lecture → Section 3.4.4.2] to prepare for this exercise. The algorithms have to be implemented based on EIGEN and they rely on the SVD [Lecture → Section 3.4.2] and QR factorization [Lecture → Section 3.3.3.4].

(3-8.a) ⚀ (10 min.) Show that for a matrix $\mathbf{X} \in \mathbb{R}^{m,n}$ the following statements are equivalent:

(i) $\operatorname{rank}(\mathbf{X}) = k$

(ii) $\mathbf{X} = \mathbf{A}\mathbf{B}^\top$ for matrices $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{B} \in \mathbb{R}^{n,k}$, $k \le \min\{m,n\}$, *both of full rank.*

HIDDEN HINT 1 for (3-8.a) → `3-8-1-0:LowRankRep1h.pdf`

SOLUTION for (3-8.a) → `3-8-1-1:LowRankRep1s.pdf` ▲

(3-8.b) ⚁ (30 min.) [depends on Sub-problem (3-8.a)]

In the file `lowrankrep.hpp` write an EIGEN-based C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
factorize_X_AB(const Eigen::MatrixXd& X, unsigned int k);
```

that factorizes the matrix $\mathbf{X} \in \mathbb{R}^{m,n}$ with $\operatorname{rank}(\mathbf{X}) = k$ into $\mathbf{A}\mathbf{B}^\top$, as in Sub-problem (3-8.a), and returns the two matrices $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{B} \in \mathbb{R}^{n,k}$ in this order.

The function should issue a warning in case $\operatorname{rank}(\mathbf{X}) = k$ is in doubt.

HIDDEN HINT 1 for (3-8.b) → `3-8-2-0:LowRankRep2h.pdf`

SOLUTION for (3-8.b) → `3-8-2-1:LowRankRep2s.pdf` ▲

(3-8.c) ⚂ (30 min.) Let $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{B} \in \mathbb{R}^{n,k}$, with $k \le \min\{m,n\}$. Based on the economical QR-factorization [Lecture → Thm. 3.3.3.4] of both $\mathbf{A}$ and $\mathbf{B}$

$$\mathbf{A} = \mathbf{Q}_A\mathbf{R}_A \;,\; \mathbf{Q}_A \in \mathbb{R}^{m,k} \text{ with } \mathbf{Q}_A^\top\mathbf{Q}_A = \mathbf{I} \;,\; \mathbf{R}_A \in \mathbb{R}^{k,k} \text{ upper triangular ,}$$
$$\mathbf{B} = \mathbf{Q}_B\mathbf{R}_B \;,\; \mathbf{Q}_B \in \mathbb{R}^{n,k} \text{ with } \mathbf{Q}_B^\top\mathbf{Q}_B = \mathbf{I} \;,\; \mathbf{R}_B \in \mathbb{R}^{k,k} \text{ upper triangular ,}$$

devise a way, how the economical singular value decomposition $\mathbf{A}\mathbf{B}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}$ [Lecture → (3.4.1.5)] of $\mathbf{A}\mathbf{B}^\top$ can be computed using only the singular value decomposition of a $k \times k$-matrix.

SOLUTION for (3-8.c) → `3-8-3-0:s4.pdf` ▲

(3-8.d) ⚃ (30 min.) [depends on Sub-problem (3-8.c)]

Given $\mathbf{A} \in \mathbb{R}^{m,k}$, $\mathbf{B} \in \mathbb{R}^{n,k}$, with $k \ll m,n$, write an efficient C++ function

```
std::tuple<Eigen::MatrixXd, Eigen::MatrixXd, Eigen::MatrixXd>
svd_AB(const Eigen::MatrixXd& A, const Eigen::MatrixXd& B);
```

that calculates a singular value decomposition (SVD) [Lecture → Thm. 3.4.1.1] of the product $\mathbf{A}\mathbf{B}^\top = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$, where $\mathbf{U} \in \mathbb{R}^{m,k}, \mathbf{V} \in \mathbb{R}^{n,k}$ have orthogonal columns and $\mathbf{\Sigma} \in \mathbb{R}^{k,k}$ is a diagonal matrix. The SVD-factors are returned as a triple $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V})$.

HIDDEN HINT 1 for (3-8.d) → 3-8-4-0:LowRankRep3h.pdf

SOLUTION for (3-8.d) → 3-8-4-1:LowRankRep3s.pdf ▲

(3-8.e) ⚀ (10 min.) [depends on Sub-problem (3-8.d)]

What is the asymptotic computational cost of the function `svd_AB()` for (fixed) small k and $m = n \to \infty$? Discuss the effort required by the different steps of your algorithm.

SOLUTION for (3-8.e) → 3-8-5-0:LowRankRep4s.pdf ▲

(3-8.f) ⚁ (10 min.) If $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m,n}$ satisfy $\operatorname{rank}(\mathbf{Y}) = \operatorname{rank}(\mathbf{X}) = k$, show that $\operatorname{rank}(\mathbf{Y} + \mathbf{X}) \leq 2k$.

SOLUTION for (3-8.f) → 3-8-6-0:LowRankRep5s.pdf ▲

(3-8.g) ⚂ (10 min.) Consider $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$, $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$, $\mathbf{X} = \mathbf{A}_X \mathbf{B}_X^\top$ and $\mathbf{Y} = \mathbf{A}_Y \mathbf{B}_Y^\top$. Find a factorization of the sum $\mathbf{X} + \mathbf{Y}$ as $\mathbf{X} + \mathbf{Y} = \mathbf{A}\mathbf{B}^\top$, with $\mathbf{A} \in \mathbb{R}^{m,2k}$ and $\mathbf{B} \in \mathbb{R}^{n,2k}$.

SOLUTION for (3-8.g) → 3-8-7-0:LowRankRep6s.pdf ▲

(3-8.h) ⚃ (30 min.) [depends on Sub-problem (3-8.d), Sub-problem (3-8.g)]

Rely on the previous subproblems to write an *efficient* C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
rank_k_approx(const Eigen::MatrixXd& Ax, const Eigen::MatrixXd& Ay,
              const Eigen::MatrixXd& Bx, const Eigen::MatrixXd&
                 By);
```

that returns two matrix factors $\mathbf{A}_Z \in \mathbb{R}^{m,k}$ and $\mathbf{B}_Z \in \mathbb{R}^{n,k}$ whose product $\mathbf{Z} = \mathbf{A}_Z \mathbf{B}_Z^\top$ yields the rank-k best approximation of the matrix sum $\mathbf{X} + \mathbf{Y} = \mathbf{A}_X \mathbf{B}_X^\top + \mathbf{A}_Y \mathbf{B}_Y^\top$:

$$\mathbf{Z} := \operatorname*{argmin}_{\substack{\mathbf{M} \in \mathbb{R}^{m,n} \\ \operatorname{rank}(\mathbf{M}) \leq k}} \left\| \mathbf{A}_X \mathbf{B}_X^\top + \mathbf{A}_Y \mathbf{B}_Y^\top - \mathbf{M} \right\|_F$$

Here the matrices $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$ and $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$ are given and passed to the function through the arguments `Ax`, `Ay`, `Bx`, and `By`. You may only consider the case where $2k < m, n$.

HIDDEN HINT 1 for (3-8.h) → 3-8-8-0:LowRankRep7h.pdf

SOLUTION for (3-8.h) → 3-8-8-1:LowRankRep7s.pdf ▲

(3-8.i) ⚂ (10 min.) [depends on Sub-problem (3-8.h)]

What is the asymptotic computational cost of the function `rank_k_approx` for a small k and $m = n \to \infty$?

SOLUTION for (3-8.i) → 3-8-9-0:LowRankRep8s.pdf ▲

End Problem 3-8 , 170 min.

Problem 3-9: Matrix Fitting by Constrained Least Squares

In this problem we look at a particular *constrained* linear least squares problem, as they have been discussed in [Lecture → Section 3.6]. In particular, we will study the augmented normal equation approach to solve such type of least squares problem, see [Lecture → Section 3.6.1]. In this context we refresh our understanding of the Lagrange multiplier technique.

Related to [Lecture → Section 3.6]; involves implementation with EIGEN.

Consider the following problem of finding the minimal-norm matrix fitting a linear system of equations with given solution and right-hand side vector:

$$\text{Given } n \in \mathbb{N},\ \mathbf{z} \in \mathbb{R}^n,\ \mathbf{g} \in \mathbb{R}^n, \quad \text{find} \quad \mathbf{M}^* = \operatorname*{argmin}_{\mathbf{M}\in\mathbb{R}^{n,n},\ \mathbf{Mz}=\mathbf{g}} \|\mathbf{M}\|_F \ , \tag{3.9.1}$$

where $\|\cdot\|_F$ denotes the Frobenius norm of a matrix as introduced in [Lecture → Def. 3.4.4.17].

Definition [Lecture → Def. 3.4.4.17]. Frobenius norm

The Frobenius norm of $\mathbf{A} \in \mathbb{K}^{m,n}$ is defined as

$$\|\mathbf{A}\|_F^2 := \sum_{i=1}^{m}\sum_{j=1}^{n} |a_{ij}|^2 \ .$$

(3-9.a) ⚀ (30 min.) Reformulate the problem as an equivalent standard linearly constrained least squares problem [Lecture → (3.6.0.1)]

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^N,\ \mathbf{Cx}=\mathbf{d}} \|\mathbf{Ax}-\mathbf{b}\|_2 \ , \tag{3.9.2}$$

for suitable matrices $\mathbf{A}$, $\mathbf{C}$ and vectors $\mathbf{b}$ and $\mathbf{d}$, see [Lecture → (3.6.0.1)]. These matrices and vectors have to be specified based on $\mathbf{z}$ and $\mathbf{g}$.

SOLUTION for (3-9.a) → 3-9-1-0:lsqfrobe.pdf ▲

(3-9.b) ⚁ (15 min.) [depends on Sub-problem (3-9.a)]

State a necessary and sufficient condition for the matrix $\mathbf{C}$ found in Sub-problem (3-9.a) to possess full rank.

SOLUTION for (3-9.b) → 3-9-2-0:lsqf1.pdf ▲

(3-9.c) ⚂ (15 min.) [depends on Sub-problem (3-9.a)]

State the **augmented normal equations** [Lecture → (3.6.1.6)] corresponding to the constrained linear least squares problem found in Sub-problem (3-9.a) and give necessary and sufficient conditions on $\mathbf{g}$ and $\mathbf{z}$ that ensure existence and uniqueness of solutions.

HIDDEN HINT 1 for (3-9.c) → 3-9-3-0:lsqfh1.pdf

SOLUTION for (3-9.c) → 3-9-3-1:.pdf ▲

(3-9.d) ⚂ (30 min.) [depends on Sub-problem (3-9.c)]

Write a C++ function

```
Eigen::MatrixXd min_frob(const Eigen::VectorXd &z, const
   Eigen::VectorXd &g);
```

that computes the solution $\mathbf{M}^*$ of the minimization problem (3.9.1) given the vectors $\mathbf{z}, \mathbf{g} \in \mathbb{R}^n$. Use the augmented normal equations [Lecture → (3.6.1.6)] derived in ((3-9.c)).

It is convenient to use EIGEN's built-in Kronecker product class **Eigen::KroneckerProduct** (**#include** `<unsupported/Eigen/KroneckerProduct>` required).

SOLUTION for (3-9.d) → 3-9-4-0:lsqfrobi.pdf ▲

(3-9.e) ⚀ (15 min.) [depends on Sub-problem (3-9.d)]

Using the test vectors $\mathbf{z} = [1, 2, \ldots, n]^\top \in \mathbb{R}^n$ and $\mathbf{g} = [1, -1, 1, \ldots, 1, -1]^\top \in \mathbb{R}^n$, implement a C++ function

```
bool testMformula(unsigned int n);
```

that checks *in a numerical experiment* whether

$$\mathbf{M} = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2} \tag{3.9.7}$$

is a solution to (3.9.1). To that end call the function `min_frob` implemented in Sub-problem (3-9.d), conduct the test and return **true** if it succeeds.

HIDDEN HINT 1 for (3-9.e) → 3-9-5-0:lsqfrobh31.pdf

SOLUTION for (3-9.e) → 3-9-5-1:lsqfrobc.pdf

▲

(3-9.f) ⚀ (30 min.) [depends on Sub-problem (3-9.c)]

Now, give a rigorous proof that (3.9.7) gives a solution of (3.9.1), provided that $\mathbf{z} \neq \mathbf{0}$.

SOLUTION for (3-9.f) → 3-9-6-0:lsqfrx.pdf ▲

So far, we have simply applied the formulas from [Lecture → Section 3.6.1] to (3.9.1) and its reformulation as a constrained linear least squares problem. Now we take a closer look at the method of Lagrangian multipliers, which was used to derive these equations in class.

(3-9.g) ⚀ (20 min.) As in [Lecture → (3.6.1.2)] and [Lecture → (3.6.1.3)] from the lecture notes, find a **Lagrange functional** $L : \mathbb{R}^{n,n} \times \mathbb{R}^n \longrightarrow \mathbb{R}$ such that:

$$\mathbf{M}^* = \operatorname*{argmin}_{\mathbf{M} \in \mathbb{R}^{n,n}} \left\{ \sup_{\mathbf{m} \in \mathbb{R}^n} [L(\mathbf{M}, \mathbf{m})] \right\}. \tag{3.9.10}$$

The matrix $\mathbf{M}^*$ should provide the solution of (3.9.1).

HIDDEN HINT 1 for (3-9.g) → 3-9-7-0:LSQFrobLagr1h.pdf

SOLUTION for (3-9.g) → 3-9-7-1:LSQFrobLagr1s.pdf ▲

(3-9.h) ⚀ (15 min.) Find the derivative $\mathbf{grad}\,\Phi \in \mathbb{R}^{n,n}$ of the function Φ defined as:

$$\Phi : \mathbb{R}^{n,n} \to \mathbb{R}, \quad \Phi(\mathbf{X}) = \|\mathbf{X}\|_F^2 \tag{3.9.12}$$

HIDDEN HINT 1 for (3-9.h) → 3-9-8-0:LSQFrobLagr2h.pdf

SOLUTION for (3-9.h) → 3-9-8-1:LSQFrobLagr2s.pdf ▲

(3-9.i) ⚀ (20 min.) [depends on Sub-problem (3-9.g), Sub-problem (3-9.h)]

Use the result of Sub-problem (3-9.h) to derive **saddle point equations** for $\mathbf{grad}\, L(\mathbf{M},\mathbf{m}) = 0$, for the L obtained in Sub-problem (3-9.g).

HIDDEN HINT 1 for (3-9.i) → 3-9-9-0:LSQFrobLagr3h.pdf

SOLUTION for (3-9.i) → 3-9-9-1:LSQFrobLagr3s.pdf ▲

(3-9.j) ⚀ (15 min.) [depends on Sub-problem (3-9.i)]

Solve the saddle point equations obtained in Sub-problem (3-9.i) in symbolic form.

HIDDEN HINT 1 for (3-9.j) → 3-9-10-0:LSQFrobLagr4h.pdf

SOLUTION for (3-9.j) → 3-9-10-1:LSQFrobLagr4s.pdf ▲

End Problem 3-9 , 205 min.

Problem 3-10: Fitting of (relative) Point Locations from Distances

Given *all* pairwise distance of points on a line, this redundant information can be used to determine their positions through least-squares fitting. Pertinent algorithms are discussed in this problem.

This problem addresses the algorithmic details of the least-squares solution of the sparse overdetermined linear system of equations described in [Lecture → Ex. 3.0.1.9]. The focus is on normal equation methods as introduced in [Lecture → Section 3.2].

The numbers $x_1, \ldots, x_n \in \mathbb{R}$ describe the location of n points lying on a line that is identified with the real axis $\mathbb{R}$. We know measured values for the $m := \binom{n}{2} = \frac{1}{2}n(n-1)$ signed distances $d_{i,j} := x_j - x_i \in \mathbb{R}$, $1 \le i < j \le n$, from which we have to determine the x_i up to a shift. To fix the shift, we set $x_n := 0$.

This amounts to solving the overdetermined linear system of equations

$$\begin{cases} x_j - x_i = d_{i,j}\,, & 1 \le i < j \le n-1\,, \\ -x_i = d_{i,n}\,, & i \in \{1,\ldots,n-1\}\,. \end{cases} \tag{3.10.1}$$

Collecting the unknown positions in the vector $\mathbf{x} := [x_1, \ldots, x_{n-1}]^\top \in \mathbb{R}^{n-1}$, the equations (3.10.1) can be recast as an $m \times (n-1)$ linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, where $\mathbf{b} \in \mathbb{R}^m$ contains each of the distances $d_{i,j}$, $1 \le i < j \le n$, exactly once.

(3-10.a) ⚁ (30 min.) How many real numbers and integers have to be stored at least in order to represent the matrix $\mathbf{A}$, regarded as a matrix with real entries, in

- COO/triplet format and
- CRS format?

SOLUTION for (3-10.a) → 3-10-1-0:s1.pdf ▲

(3-10.b) ⚂ (25 min.) Based on EIGEN implement (in the file `disfitting.hpp`) an *efficient* C++ function for the initialization of a sparse-matrix data structure for the matrix $\mathbf{A}$:

```
Eigen::SparseMatrix<double> initA(unsigned int n);
```

It goes without saying that the parameter `n` passes the number n of points.

SOLUTION for (3-10.b) → 3-10-2-0:s4.pdf ▲

(3-10.c) ⚂ (45 min.) [depends on Sub-problem (3-10.b)]

According to [Lecture → § 3.2.0.7] the **extended normal equations** for the generic overdetermined linear system $\mathbf{M}\mathbf{x} = \mathbf{c}$, $\mathbf{M} \in \mathbb{R}^{m,\ell}$, $\mathbf{c} \in \mathbb{R}^m$, $m \ge \ell$, boil down to the linear system of equations

$$\begin{bmatrix} -\mathbf{I} & \mathbf{M} \\ \mathbf{M}^\top & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}. \tag{3.10.5}$$

Implement an efficient C++ function

```
Eigen::VectorXd solveExtendedNormalEquations(
  const Eigen::MatrixXd &D);
```

that uses a sparse direct solver provided by EIGEN to solve the extended normal equations for the position fitting problem (3.10.1) and returns its least-squares solution $\mathbf{x} \in \mathbb{R}^{n-1}$. The argument `D` passes an $n \times n$-matrix $\mathbf{D}$ whose strict upper triangular part contains the distances

$$[\mathbf{D}]_{i,j} := d_{i,j}\,, \quad 1 \le i < j \le n\,.$$

HIDDEN HINT 1 for (3-10.c) → 3-10-3-0:h3tt.pdf

SOLUTION for (3-10.c) → 3-10-3-1:s3.pdf ▲

(3-10.d) ⚃ (20 min.) [depends on Sub-problem (3-10.b)]

The coefficient matrix $\mathbf{M} \in \mathbb{R}^{n-1,n-1}$ of the **normal equations** [Lecture → (3.1.2.2)] associated with the overdetermined linear system of equations (3.10.1) is of the form

$$\mathbf{M} = \mathbf{D} - \mathbf{z}\mathbf{z}^\top \quad \text{with} \quad \begin{array}{l} \text{a diagonal matrix} \quad \mathbf{D} \in \mathbb{R}^{n-1,n-1}\,, \\ \text{and a column vector} \quad \mathbf{z} \in \mathbb{R}^{n-1}\,. \end{array} \tag{3.10.8}$$

Compute $\mathbf{D}$ and $\mathbf{z}$, and find a simple expression for the inverse $\mathbf{M}^{-1}$.

HIDDEN HINT 1 for (3-10.d) → 3-10-4-0:smw.pdf

SOLUTION for (3-10.d) → 3-10-4-1:s4.pdf ▲

(3-10.e) ⚄ (45 min.) [depends on Sub-problem (3-10.b) and Sub-problem (3-10.d)]

Write an *efficient* C++ function

```
Eigen::VectorXd solveNormalEquations(
  const Eigen::MatrixXd &D);
```

that returns the least-squares solution of (3.10.1) by solving the corresponding normal equations. The signature is the same as for `solveExtendedNormalEquations()` from Sub-problem (3-10.c).

HIDDEN HINT 1 for (3-10.e) → 3-10-5-0:h4iA.pdf

SOLUTION for (3-10.e) → 3-10-5-1:s5.pdf ▲

(3-10.f) ⚄ (15 min.) [depends on Sub-problem (3-10.c) and Sub-problem (3-10.e)]

Write a C++ function

```
bool testNormalEquations(const MatrixXd &D);
```

that calls both `solveExtendedNormalEquations()` and `solveNormalEquations()` and returns **true**, if the results agree.

HIDDEN HINT 1 for (3-10.f) → 3-10-6-0:dfh5a1.pdf

SOLUTION for (3-10.f) → 3-10-6-1:df5as.pdf ▲

(3-10.g) ⚃ (10 min.) [depends on Sub-problem (3-10.e)]

What is the asymptotic complexity of your implementation of `solveNormalEquations()` in Sub-problem (3-10.e) for $n \to \infty$?

SOLUTION for (3-10.g) → 3-10-7-0:s6.pdf ▲

End Problem 3-10 , 190 min.

Problem 3-11: A Class Representing Low-Rank Matrices

Low-rank matrices play a central role in modern algorithms for data compression and the efficient handling of non-local operations. In this problem we devise a dedicated data type for efficiently storing and manipulating low-rank matrices.

This problem is connected with [Lecture → Section 1.4.3] and [Lecture → Section 3.4.4]. It is closely related to Problem 3-8.

Recall the following result from elementary linear algebra:

Theorem 3.11.1. Representation of low-rank matrices

Every matrix $\mathbf{M} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, *with rank* $r := \operatorname{rank}(\mathbf{M}) > 0$ *can be factored as*

$$\mathbf{M} = \mathbf{A}\mathbf{B}^\top \quad \text{with} \quad \begin{array}{l} \mathbf{A} \in \mathbb{R}^{m,r} \,, \\ \mathbf{B} \in \mathbb{R}^{n,r} \,. \end{array}$$

This factorization is used in the design of the following EIGEN-based C++ class (File `matrixlowrank.hpp`) meant to store real matrices $\mathbf{M} \in \mathbb{R}^{m,n}$ with maximal rank $r < \min\{m,n\}$ and to provide special operations on them:

```
class MatrixLowRank {
public:
  MatrixLowRank(unsigned int m, unsigned int n, unsigned int r);
  MatrixLowRank(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B);

  Eigen::Index rows() const { return _m; };
  Eigen::Index cols() const { return _n; };
  Eigen::Index rank() const { return _r; };

  Eigen::MatrixXd operator*(const Eigen::MatrixXd &)const;
  MatrixLowRank &operator*=(const Eigen::MatrixXd &);
  MatrixLowRank &addTo(const MatrixLowRank &, double tol = 1E-6);
private:
  unsigned int _m;     // num.  of rows
  unsigned int _n;     // num.  of columns
  unsigned int _r;     // maximal rank, =1 for zero matrix
  Eigen::MatrixXd _A; // factor matrix A
  Eigen::MatrixXd _B; // factor matrix B
};
```

The convention for the case $\mathbf{M} = \mathbf{O}$ is that we set `_r = 1` and the low-rank factors to zero.

(3-11.a) ⚀ (15 min.)

Give a proof of Thm. 3.11.1.

HIDDEN HINT 1 for (3-11.a) → 3-11-1-0:h1.pdf

SOLUTION for (3-11.a) → 3-11-1-1:s1.pdf ▲

(3-11.b) ⚀ (15 min.) In the file `matrixlowrank.hpp` implement the method

```
Eigen::MatrixXd MatrixLowRank::operator *
  (const Eigen::MatrixXd &X) const;
```

which is supposed to return the product of the $m \times n$ low-rank matrix stored in the **MatrixLowRank** object (left factor) with a generic dense matrix $\mathbf{X} \in \mathbb{R}^{n,k}$ (right factor). Take care, that your code is efficient.

SOLUTION for (3-11.b) → 3-11-2-0:s3.pdf ▲

(3-11.c) ⚀ (5 min.) [depends on Sub-problem (3-11.b)]

What is the asymptotic complexity of your implementation of `MatrixLowRank::operator *` from Sub-problem (3-11.b) in terms of $m, n, k \to \infty$, $r := \operatorname{rank}(\mathbf{M})$ fixed. Here, the size of the argument matrix is $n \times k$, $k \in \mathbb{N}$, and the **MatrixLowRank** object stores $\mathbf{M} \in \mathbb{R}^{m,n}$.

SOLUTION for (3-11.c) → 3-11-3-0:s3.pdf ▲

(3-11.d) ⚄ (10 min.) Provide an efficient implementation of the method (File `matrixlowrank.hpp`)

```
MatrixLowRank &operator *= (const Eigen::MatrixXd &X);
```

for the *in-situ (right) multiplication* of a low-rank $m \times n$ matrix stored in a **MatrixLowRank** object with a generic matrix $\mathbf{X} \in \mathbb{R}^{n,k}$. "In-situ" means that after the operation the **MatrixLowRank** object stores the result of the matrix multiplication. The "internal rank" `_r` of the **MatrixLowRank** object should not change.

SOLUTION for (3-11.d) → 3-11-4-0:s4.pdf ▲

(3-11.e) ⚅ (40 min.) With an eye on efficiency, in the file `matrixlowrank.hpp` complete the implementation of the method

```
MatrixLowRank &MatrixLowRank::addTo(
  const MatrixLowRank &X, double atol, double rtol);
```

that replaces the matrix $\mathbf{M} \in \mathbb{R}^{m,n}$ stored in the **MatrixLowRank** object with $\operatorname{trunc}_{\mathtt{tol}}(\mathbf{M} + \mathbf{X})$, where

- $\mathbf{X} \in \mathbb{R}^{m,n}$ is the matrix passed through the argument `X`,
- and, for $\mathbf{Y} \in \mathbb{R}^{m,n}$,

$$\operatorname{trunc}_{\mathtt{tol}}(\mathbf{Y}) := \operatorname{argmin}\left\{ \operatorname{rank}(\mathbf{R}) : \mathbf{R} \in \mathbb{R}^{m,n}, \begin{array}{c} \|\mathbf{Y}-\mathbf{R}\|_2 \le \mathrm{rtol}\|\mathbf{Y}\|_2 \,, \\ \textbf{or} \\ \|\mathbf{Y}-\mathbf{R}\|_2 \le \mathrm{atol} \end{array} \right\},$$

where $\|\cdot\|_2$ designates the Euclidean matrix norm. The "internal rank" `_r` must be updated.

Your implementation may assume that

$$\operatorname{rank}(\mathbf{M}) + \operatorname{rank}(\mathbf{X}) \le \min\{m, n\} \,.$$

HIDDEN HINT 1 for (3-11.e) → 3-11-5-0:h5a.pdf

HIDDEN HINT 2 for (3-11.e) → 3-11-5-1:h5t.pdf

SOLUTION for (3-11.e) → 3-11-5-2:s4.pdf ▲

End Problem 3-11 , 85 min.

Problem 3-12: Polar Decomposition of Matrices

In class we have seen various product decompositions/factorizations of matrices like the LU-decomposition [Lecture → Section 2.3.2], the QR-decomposition [Lecture → Section 3.3.3], and the singular-value decomposition (SVD) [Lecture → Section 3.4]. In this problems we study another factorization, which is sometimes used in numerical methods, though it is not as important as the decompositions listed before.

This problem is related to [Lecture → Section 3.3.3.2] and [Lecture → Section 3.4.2] and requires familiarity with the EIGEN-based C++ implementation discussed in those sections.

Theorem 3.12.1. Polar decomposition

For every matrix $\mathbf{X} \in \mathbb{R}^{m,n}$, $m \geq n$, there is a matrix $\mathbf{Q} \in \mathbb{R}^{m,n}$ with orthonormal columns, $\mathbf{Q}^\top \mathbf{Q} = \mathbf{I}_n$, and a symmetric positive semi-definite [Lecture → Def. 1.1.2.6] matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ such that $\mathbf{X} = \mathbf{Q}\mathbf{M}$.

The matrix factorization postulated in Thm. 3.12.1 is called **polar decomposition** of $\mathbf{X}$.

(3-12.a) ⚄ (30 min.) Give a proof of Thm. 3.12.1.

HIDDEN HINT 1 for (3-12.a) → 3-12-1-0:h1p.pdf

SOLUTION for (3-12.a) → 3-12-1-1:s1.pdf ▲

Based on the data types of EIGEN, the polar decomposition of a "tall/slim" real matrix is to be implemented as the following C++ class.

```
class PolarDecomposition {
 public:
  explicit PolarDecomposition(const Eigen::MatrixXd &X) { initialize(X); }
  PolarDecomposition(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B);
  PolarDecomposition(const PolarDecomposition &) = default;
  ~PolarDecomposition() = default;

  // Left multiplication of M with the Q-factor of the polar decomposition
  void applyQ(Eigen::MatrixXd &Y) const { Y.applyOnTheLeft(Q_); }
  // Left multiplication of M with the M-factor of the polar decomposition
  void applyM(Eigen::MatrixXd &Y) const { Y.applyOnTheLeft(M_); }

 private:
  void initialize(const Eigen::MatrixXd &X);
  Eigen::MatrixXd Q_;  // factor Q
  Eigen::MatrixXd M_;  // factor M
};
```

The following specification of the class if given:

- The constructor

  ```
  PolarDecomposition(const Eigen::MatrixXd &X);
  ```

  should compute the polar decomposition factors $\mathbf{Q}$ and $\mathbf{M}$ according to Thm. 3.12.1 of the matrix passed in `X` and store them in the data members `Q_` and `M_`.

- The constructor

```
PolarDecomposition(const Eigen::MatrixXd &A, const
    Eigen::Matrix &B);
```

is supposed the initialize the data members `Q_` and `M_` with the polar decomposition factors $\mathbf{Q} \in \mathbb{R}^{m,n}$ and $\mathbf{M} \in \mathbb{R}^{n,n}$ of the matrix $\mathbf{AB}^\top \in \mathbb{R}^{m,n}$, where the matrices $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{B} \in \mathbb{R}^{n,k}$ are passed through the arguments `A` and `B`.

- The methods `applyQ()` and `applyM()` realize the operations

$$\mathbf{Y} \leftarrow \mathbf{QY} \quad , \quad \mathbf{Y} \leftarrow \mathbf{MY} \, ,$$

where $\mathbf{Q}$ and $\mathbf{M}$ are the factors of the polar decomposition stored in the **PolarDecomposition** object.

(3-12.b) ⚀ (15 min.) Regardless of the implementation, what is the *minimal* asymptotic computational cost, that is, a *sharp lower bound* of the asymptotic computational effort, for a call of the second constructor

```
PolarDecomposition(const Eigen::MatrixXd &A, const Eigen::Matrix
    &B);
```

of a **PolarDecomposition** object for a $m \times n$-matrix, $m \geq n$, for $m, n \to \infty$, and small fixed k?

Minimal asymptotic cost $= O\Big($ ☐ $\Big)$ for $m, n \to \infty$.

SOLUTION for (3-12.b) → 3-12-2-0:.pdf ▲

(3-12.c) ⚁ (30 min.) [depends on Sub-problem (3-12.a)]

In the file `polardecomposition.hpp` implement the method

```
void PolarDecomposition::initialize(const Eigen::MatrixXd &X);
```

that sets the data members `_Q` and `_M` of the class **PolarDecomposition**. These data members store the factors $\mathbf{Q}$ and $\mathbf{M}$ of the polar decomposition of the argument matrix `X` according to Thm. 3.12.1.

HIDDEN HINT 1 for (3-12.c) → 3-12-3-0:pds2h.pdf

SOLUTION for (3-12.c) → 3-12-3-1:s2.pdf ▲

(3-12.d) ⚂ (120 min.) [depends on Sub-problem (3-12.b)]

Write a code for the constructor

```
PolarDecomposition(const Eigen::MatrixXd &A,
                   const Eigen::Matrix &B);
```

which is supposed to initialize the data members `Q_` and `M_` with the polar decomposition factors $\mathbf{Q} \in \mathbb{R}^{m,n}$ and $\mathbf{M} \in \mathbb{R}^{n,n}$ of the matrix $\mathbf{X} := \mathbf{AB}^\top \in \mathbb{R}^{m,n}$, where the matrices $\mathbf{A} \in \mathbb{R}^{m,k}$ and $\mathbf{B} \in \mathbb{R}^{n,k}$ are passed through the arguments `A` and `B`. We assume that k is small and fixed and $k \leq n < m$. Your code should have *optimal complexity* with respect to $m, n \to \infty$.

HIDDEN HINT 1 for (3-12.d) → 3-12-4-0:pcs3h1.pdf

SOLUTION for (3-12.d) → 3-12-4-1:s2.pdf ▲

End Problem 3-12 , 195 min.

Problem 3-13: QR-Iteration

We have learned about the QR-decomposition/factorization as a tool for solving linear systems of equations [Lecture → Rem. 3.3.4.3] and linear least-squares problems [Lecture → Section 3.3.4]. Yet, this matrix factorization has many more important applications. For instance, it is a crucial ingredient for the so-called QR-algorithm, an iteration for solving algebraic eigenvalue problems [GV13, Sections 7.5 & 8.3]. This problem will examine some algorithmic aspects of that algorithm.

You are supposed to know the contents of [Lecture → Section 3.3.3] and implementation in EIGEN.

We recall the QR-decomposition of matrices:

Theorem [Lecture → Thm. 3.3.3.4]. QR-decomposition

For any matrix $\mathbf{A} \in \mathbb{K}^{n,k}$ *with* $\operatorname{rank}(\mathbf{A}) = k$ *there exists*

(i) a unique *Matrix* $\mathbf{Q}_0 \in \mathbb{R}^{n,k}$ *that satisfies* $\mathbf{Q}_0^{\mathrm{H}}\mathbf{Q}_0 = \mathbf{I}_k$, *and a* unique *upper triangular Matrix* $\mathbf{R}_0 \in \mathbb{K}^{k,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\ldots,k\}$, *such that*

$$\mathbf{A} = \mathbf{Q}_0 \cdot \mathbf{R}_0 \qquad (\textit{"economical" QR-decomposition}) ,$$

(ii) a unitary Matrix $\mathbf{Q} \in \mathbb{K}^{n,n}$ *and a* unique *upper triangular* $\mathbf{R} \in \mathbb{K}^{n,k}$ *with* $(\mathbf{R})_{i,i} > 0$, $i \in \{1,\ldots,n\}$, *such that*

$$\mathbf{A} = \mathbf{Q} \cdot \mathbf{R} \qquad (\textit{full QR-decomposition}) .$$

If $\mathbb{K} = \mathbb{R}$ *all matrices will be real and* $\mathbf{Q}$ *is then orthogonal.*

(3-13.a) ⚀ (30 min.) Prove the following fact:

Lemma 3.13.1. Lower bandwidth of Q-factor

Let $\mathbf{T} = \mathbf{QR}$ *be a QR-decomposition of a regular tridiagonal matrix* $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$. *Then* $\mathbf{Q}$ *has lower bandwidth 1, that is,*

$$i,j \in \{1,\ldots,n\} , \quad i > j+1 \quad \Rightarrow \quad (\mathbf{Q})_{i,j} = 0 .$$

HIDDEN HINT 1 for (3-13.a) → 3-13-1-0:s1h1.pdf

SOLUTION for (3-13.a) → 3-13-1-1:s1.pdf ▲

(3-13.b) ⚁ (30 min.) Give a proof of the following result:

Lemma 3.13.3. QR-based similarity transform

If $\mathbf{T} = \mathbf{QR}$ *is the QR-decomposition of a regular, symmetric, and tridiagonal matrix* $\mathbf{T} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, *then* $\mathbf{T}' := \mathbf{RQ}$ *is also symmetric and tridiagonal.*

HIDDEN HINT 1 for (3-13.b) → 3-13-2-0:s2h1.pdf

SOLUTION for (3-13.b) → 3-13-2-1:s2.pdf ▲

(3-13.c) ⚂ (120 min.) A *symmetric* and *tridiagonal* matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ can be encoded by two

vectors $\mathbf{d} = \mathbf{d}(\mathbf{T}) \in \mathbb{R}^n$ and $\mathbf{u} = \mathbf{u}(\mathbf{T}) \in \mathbb{R}^{n-1}$ when setting

$$(\mathbf{T})_{i,j} = \begin{cases} (\mathbf{d})_i & \text{, if } i = j\,, \\ (\mathbf{u})_i & \text{, if } j = i+1\,, \\ (\mathbf{u})_j & \text{, if } i = j+1\,, \\ 0 & \text{else.} \end{cases} \qquad i,j \in \{1,\ldots,n\}:$$

$$\blacktriangleright \quad \mathbf{T} = \begin{bmatrix} (\mathbf{d})_1 & (\mathbf{u})_1 & 0 & \ldots & & & \ldots & 0 \\ (\mathbf{u})_1 & (\mathbf{d})_2 & (\mathbf{u})_2 & 0 & & & & 0 \\ 0 & (\mathbf{u})_2 & (\mathbf{d})_3 & (\mathbf{u})_3 & 0 & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & & \\ & & & \ddots & \ddots & \ddots & & \\ & & & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & & 0 & (\mathbf{u})_{n-2} & (\mathbf{d})_{n-1} & (\mathbf{u})_{n-1} \\ 0 & \ldots & & & & 0 & (\mathbf{u})_{n-1} & (\mathbf{d})_n \end{bmatrix}.$$

In the file `qriteration.hpp` implement a C++ function

```
std::pair<Eigen::VectorXd, Eigen::VectorXd>
  qrStep(const Eigen::VectorXd &dT,
         const Eigen::VectorXd &uT);
```

that takes the vectors $\mathbf{d}(\mathbf{T}) \in \mathbb{R}^n$ and $\mathbf{u}(\mathbf{T}) \in \mathbb{R}^{n-1}$ describing a symmetric, tridiagonal matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ as arguments, and

- returns the tuple $(\mathbf{d}(\mathbf{T}'), \mathbf{u}(\mathbf{T}'))$ of the defining vectors $\mathbf{d}(\mathbf{T}')$ and $\mathbf{u}(\mathbf{T}')$ of the symmetric, tridiagonal matrix $\mathbf{T}' := \mathbf{Q}^\top \mathbf{T}\mathbf{Q}$, where $\mathbf{Q}$ is the Q-factor of the QR-decomposition $\mathbf{T} = \mathbf{Q}\mathbf{R}$ of $\mathbf{T}$, *cf.* Lemma 3.13.1,
- and features an asymptotic complexity of $O(n)$ for $n \to \infty$.

HIDDEN HINT 1 for (3-13.c) → 3-13-3-0:s3hgiv.pdf

HIDDEN HINT 2 for (3-13.c) → 3-13-3-1:s3hbd.pdf

SOLUTION for (3-13.c) → 3-13-3-2:.pdf ▲

References

[GV13] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Fourth. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 2013, pp. xiv+756 (cit. on p. 110).

End Problem 3-13 , 180 min.

Problem 3-14: Economical Singular-Value Decomposition

For most applications requiring the singular-value decomposition (SVD) of matrix, its economical (thin) variant is sufficient. This problems examines some of its features.

This problems is based on the contents of [Lecture → Section 3.4.1].

Given a matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, $\mathbf{A} \neq \mathbf{O}$, we write $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ for its **economical (thin) SVD**.

(3-14.a) ⚀ (12 min.) Fill in the correct matrix sizes

$\mathbf{U} \in \mathbb{R}^{\square,\square}$,

$\mathbf{\Sigma} \in \mathbb{R}^{\square,\square}$,

$\mathbf{V} \in \mathbb{R}^{\square,\square}$.

HIDDEN HINT 1 for (3-14.a) → 3-14-1-0:ecosvdh1.pdf

SOLUTION for (3-14.a) → 3-14-1-1:ecoqrs1.pdf ▲

(3-14.b) ⚁ (24 min.) Decide, whether the following statements are true for *every* $\mathbf{A} \in \mathbb{R}^{m,n} \setminus \{\mathbf{O}\}$, $m,n \in \mathbb{N}$.

1. The matrix $\mathbf{U}$ is an orthogonal matrix.

   ○ TRUE ○ FALSE

2. The set of all columns of $\mathbf{U}$ is an orthonormal basis of $\mathcal{R}(\mathbf{A})$.

   ○ TRUE ○ FALSE

3. $\mathbf{V}$ has orthogonal rows.

   ○ TRUE ○ FALSE

4. $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$.

   ○ TRUE ○ FALSE

5. $\operatorname{nnz}(\mathbf{\Sigma}) := \sharp\{(i,j) : (\mathbf{\Sigma})_{i,j} \neq 0\} \leq n$.

   ○ TRUE ○ FALSE

6. $\mathbf{U}\mathbf{V}^\top = \mathbf{I}$.

○ TRUE ○ FALSE

SOLUTION for (3-14.b) → 3-14-2-0:ecoqrs2.pdf ▲

(3-14.c) ⚄ (12 min.) What is the asymptotic computational effort for computing the economical SVD of a dense matrix $\mathbf{A} \in \mathbb{R}^{m,n}$, $m, n \in \mathbb{N}$?

$$\text{cost}(\text{economical SVD of } \mathbf{A} \in \mathbb{R}^{m,n}) = O(\quad)$$

for $m, n \to \infty$.

SOLUTION for (3-14.c) → 3-14-3-0:ecoqrs3.pdf ▲

End Problem 3-14 , 48 min.

Problem 3-15: Low-rank Approximation of Concatenated Matrices

Low-rank approximation is a widely used classical technique for the compression of data sets. In this problem we study the recompression of two already compressed data sets after they have been merged. In linaer-algebra terms this boils down to computing the economical singular-value decomposition of a matrix arising from concatenating two low-rank factorized matrices.

This problem requires the concepts, techniques and results presented in [Lecture → Section 3.4.2] and [Lecture → Section 3.4.4.2].

Given are two **low-rank matrices** in factorized form

$$\left.\begin{aligned} \mathbf{X}_1 &:= \mathbf{A}_1\mathbf{B}_1^\top \\ \mathbf{X}_2 &:= \mathbf{A}_2\mathbf{B}_2^\top \end{aligned}\right\}, \qquad \begin{aligned} &\mathbf{A}_1,\mathbf{A}_2 \in \mathbb{R}^{m,k}, \\ &\mathbf{B}_1,\mathbf{B}_2 \in \mathbb{R}^{n,k}, \end{aligned} \tag{3.15.1}$$

for $m,n \in \mathbb{N}$, $k \le \min\{m,n\}$. We "concatenate" the two matrices $\mathbf{X}_1$ and $\mathbf{X}_2$ into

$$\mathbf{X} := [\mathbf{X}_1\ \mathbf{X}_2] \in \mathbb{R}^{m,2n}. \tag{3.15.2}$$

Our ultimate goal is the efficient low-rank compression of $\mathbf{X}$, given $\mathbf{A}_1,\mathbf{A}_2,\mathbf{B}_1,\mathbf{B}_2$.

(3-15.a) ⚀ (15 min.) Give a *sharp bound* for the maximal rank of the matrix $\mathbf{X}$ defined in (3.15.2)?

$$\operatorname{rank}(\mathbf{X}) \le \boxed{} .$$

SOLUTION for (3-15.a) → 3-15-1-0:cmlrcs1.pdf ▲

(3-15.b) ⚄ (90 min.) In the sequel we assume $2k \le \min\{m,n\}$. In the file `concatmatlrc.hpp` implement a C++ function

```
std::tuple<Eigen::MatrixXd, Eigen::VectorXd, Eigen::MatrixXd>
  eco_svd_concat(
    const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
    const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2);
```

which accepts four matrices $\mathbf{A}_1,\mathbf{B}_1,\mathbf{A}_2,\mathbf{B}_2$ with sizes $\mathbf{A}_1,\mathbf{A}_2 \in \mathbb{R}^{m,k}$ and $\mathbf{B}_1,\mathbf{B}_2 \in \mathbb{R}^{n,k}$, see (3.15.1), as arguments. The function is supposed to compute the factors

(i) $\mathbf{U} \in \mathbb{R}^{m,2k}$ with $\mathbf{U}^\top\mathbf{U} = \mathbf{I}_{2k}$, (3.15.3a)

(ii) $\mathbf{\Sigma} = \operatorname{diag}(\sigma_1,\sigma_2,\ldots,\sigma_{2k}) \in \mathbb{R}^{2k,2k}$ with $\sigma_1 \ge \sigma_2 \ge \cdots \ge \sigma_{2k} \ge 0$, (3.15.3b)

(iii) $\mathbf{V} \in \mathbb{R}^{2n,2k}$ with $\mathbf{V}^\top\mathbf{V} = \mathbf{I}_{2k}$, (3.15.3c)

of the **economical/thin SVD** $\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top$ of the concatenation $\mathbf{X}$, $\mathbf{X} := \left[\mathbf{A}_1\mathbf{B}_1^\top\ \mathbf{A}_2\mathbf{B}_2^\top\right]$ as defined in (3.15.2). The function should return the tuple $(\mathbf{U},[\sigma_1,\ldots,\sigma_{2k}]^\top,\mathbf{V})$.

The asymptotic complexity of your implementation for $m,n \to \infty$ and for small, fixed k, $2k \le \min\{m,n\}$, must be *optimal*.

HIDDEN HINT 1 for (3-15.b) → 3-15-2-0:clrm2h2.pdf

SOLUTION for (3-15.b) → 3-15-2-1:cmlrcs2.pdf ▲

(3-15.c) ⚂ (30 min.) Complete the C++ function

```
bool test_eco_svd_concat(
    const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
    const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2);
```

that verifies that for the given argument matrices the implemented function `eco_svd_concat()` complies with the specifications, that is, that its return values provide a factorization of $\mathbf{X}$ from (3.15.2) and meet the requirements (3.15.3). Return **true**, if this is the case.

HIDDEN HINT 1 for (3-15.c) → 3-15-3-0:lrcmh3.pdf

SOLUTION for (3-15.c) → 3-15-3-1:lrcms3.pdf ▲

(3-15.d) ⚂ (30 min.)

Based on `eco_svd_concat()` as specified in Sub-problem (3-15.b) in the file `concatmatlrc.hpp` implement a C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
  concat_low_rank_best(
    const Eigen::MatrixXd &A1, const Eigen::MatrixXd &B1,
    const Eigen::MatrixXd &A2, const Eigen::MatrixXd &B2,
    double tol);
```

that computes a matrix $\mathbf{M} \in \mathbb{R}^{m,2n}$ of *minimal rank* $r \in \mathbb{N}$ such that

$$\|\mathbf{X} - \mathbf{M}\|_2 \leq \mathrm{tol} \cdot \|\mathbf{X}\|_2 \,, \tag{3.15.11}$$

where $\mathbf{X}$ is defined in (3.15.2), $\|\cdot\|_2$ stands for the Euclidean matrix norm, and the tolerance $\mathrm{tol} \in]0,1[$ is passed as an argument.

The matrix $\mathbf{M}$ should be returned in low-rank factorized form

$$\mathbf{M} = \mathbf{A}_M \mathbf{B}_M^\top \quad , \quad \mathbf{A}_M \in \mathbb{R}^{m,r} \,, \quad \mathbf{B}_M \in \mathbb{R}^{2n,r} \,, \tag{3.15.12}$$

as tuple $(\mathbf{A}_M, \mathbf{B}_M)$.

SOLUTION for (3-15.d) → 3-15-4-0:crm4s.pdf ▲

End Problem 3-15, 165 min.

Problem 3-16: Compact Storage Format for QR-Factorization

It is essential for the efficiency of orthogonalization techniques that orthogonal matrices, which commonly occur as factors in matrix products. are stored as products of elementary orthogonal transformations like Householder reflections or Givens rotations, see [Lecture → Rem. 3.3.3.22]. This approach is adopted in all numerical libraries including EIGEN. This problem addresses some related algorithmic issues.

This problem assumes familiarity with orthogonal matrices [Lecture → Section 3.3.2] and Householder reflections [Lecture → § 3.3.3.11].

A legacy FORTRAN-77 numerical library routine returns a raw array (type **double** $*$) of n^2 **double** floating-point numbers, $n \in \mathbb{N}$, which represents an $n \times n$-matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ according to the following specification:

(I) The array encodes an $n \times n$ matrix $\mathbf{M} \in \mathbb{R}^{n,n}$ in column-major format [Lecture → Section 1.2.3].

(II) The column vectors $\mathbf{u}_j \in \mathbb{R}^n$, $j = 1, \ldots, n-1$, defined as[1]

$$(\mathbf{u}_j)_k := \begin{cases} 0 & \text{for } k < j\,, \\ \sqrt{1 - \sum_{\ell=k+1}^{n} (\mathbf{M})_{\ell,j}^2} & \text{for } k = j\,, \\ (\mathbf{M})_{k,j} & \text{for } k > j\,, \end{cases} \qquad j = 1, \ldots, n-1\,, \tag{3.16.1}$$

are the building blocks for the product matrix

$$\mathbf{Q} = (\mathbf{I}_n - 2\mathbf{u}_1\mathbf{u}_1^\top)(\mathbf{I}_n - 2\mathbf{u}_2\mathbf{u}_2^\top) \cdot \cdots \cdot (\mathbf{I}_n - 2\mathbf{u}_{n-1}\mathbf{u}_{n-1}^\top)\,. \tag{3.16.2}$$

(III) The matrix $\mathbf{A}$ is given as $\mathbf{A} = \mathbf{QR}$, where $\mathbf{R} \in \mathbb{R}^{n,n}$ is the *upper triangular* part of $\mathbf{M}$:

$$(\mathbf{R})_{k,j} := \begin{cases} (\mathbf{M})_{k,j} & \text{for } k \le j\,, \\ 0 & \text{else,} \end{cases} \qquad k, j \in \{1, \ldots, n\}\,. \tag{3.16.3}$$

Your task is to write an EIGEN-based wrapper class for the raw data that offers certain operations with the matrix $\mathbf{A}$. The class definition is a follows:

C++ code 3.16.4: Class definition of CompactStorageQR

```
2  class CompactStorageQR {
3  public:
4   /**
5    * \brief Construct a new CompactStorageQR object
6    *
7    * \param data raw array which has to be persistent
8    * \param n number of rows/columns of encoded matrix
9    */
10  CompactStorageQR(const double *data, unsigned int n) : n_(n), M_(data, n, n) {
11    for (unsigned int k = 0; k < n_ - 1; ++k) {
12      double sn{0};
13      for (unsigned int j = k + 1; j < n_; ++j) {
14        const double t{M_(j, k)};
15        sn += t * t;
16      }
```

[1] A sum whose lower summation bound is larger than the upper summation bound is understood to evaluate to zero.

```
17        if (sn > 1.0) {
18          throw std::runtime_error(
19              "CompactStorageQR: Illegal subdiagonal column norm!");
20        }
21      }
22    }
23
24    /**
25     * \brief Determinant of the matrix
26     *
27     * \return double the determinant
28     */
29    double det() const;
30
31    /**
32     * \brief  Right multiplication with another matrix
33     *
34     * \param X matrix to multiply
35     * \return Eigen::MatrixXd product
36     */
37    Eigen::MatrixXd matmult(const Eigen::MatrixXd &X) const;
38
39    /**
40     * \brief Solution of linear systems of equations
41     *
42     * \param B right hand side
43     * \return Eigen::MatrixXd solution matrix
44     */
45    Eigen::MatrixXd solve(const Eigen::MatrixXd &B) const;
46
47   private:
48    unsigned int n_;                        // Matrix dimensions
49    Eigen::Map<const Eigen::MatrixXd> M_;  // Raw data wrapped into a matrix
50  };
```

Get it on GitLab (compactstorageqr.hpp).

(3-16.a) ⚀ (45 min.) In the file `compactstorageqr.hpp` implement the method

```
Eigen::MatrixXd CompactStorageQR::matmult(
  const Eigen::MatrixXd &X) const;
```

which returns the matrix product $\mathbf{A} \cdot \mathbf{X}$ of the matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the **CompactStorageQR** object and of a matrix $\mathbf{X} \in \mathbb{R}^{n,k}$, $k \in \mathbb{N}$, passed in `X`. Your implementation must not incur asymptotic computational cost worse than $O(n^2k)$ for $n,k \to \infty$.

SOLUTION for (3-16.a) → `3-16-1-0:.pdf` ▲

(3-16.b) ⚀ (30 min.) In the file `compactstorageqr.hpp` supply the missing parts of the implementation of

```
Eigen::MatrixXd CompactStorageQR::solve(const Eigen::MatrixXd &B)
    const;
```

which is to compute $\mathbf{A}^{-1}\mathbf{B}$ for $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the current **CompactStorageQR** object and $\mathbf{B} \in \mathbb{R}^{n,k}$ contained in the argument `B`. Again, the asymptotic complexity of your implementation must be $O(n^2k)$ for $n,k \to \infty$.

HIDDEN HINT 1 for (3-16.b) → `3-16-2-0:csqrs2h1.pdf`

SOLUTION for (3-16.b) → `3-16-2-1:.pdf` ▲

(3-16.c) ⚂ (30 min.) In the file `compactstorageqr.hpp` write the code for the method

```
double CompactStorageQR::det() const;
```

which gives the **determinant** $\det \mathbf{A}$ of the matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ stored in the current **CompactStorageQR** object.

HIDDEN HINT 1 for (3-16.c) → `3-16-3-0:cqrdeth1.pdf`

SOLUTION for (3-16.c) → `3-16-3-1:csqr3.pdf` ▲

End Problem 3-16 , 105 min.

Problem 3-17: Extended Normal Equations

In [Lecture → § 3.2.0.7] we learned that sparsity of the system matrix of a large overdetermined linear system of equations, which is lost when forming the regular normal equations, can be preserved by switching to the so-called extended normal equations. This problem reviews those and examines the initialization of the corresponding system matrix.

The problem is based on [Lecture → Section 3.1.2], [Lecture → § 3.2.0.7], and requires familiarity with [Lecture → Section 2.7.2].

Throughout this problem we consider a large *sparse* overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, with $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$ given, $m \gg n \gg 1$. The matrix $\mathbf{A}$ is assumed to have full rank.

(3-17.a) ⚀ (15 min.) By introducing the residual $\mathbf{r} := \mathbf{A}\mathbf{x} - \mathbf{b}$ as additional auxiliary variable the normal equations [Lecture → (3.1.2.2)] for $\mathbf{A}\mathbf{x} = \mathbf{b}$ can be converted into the **extended normal equations**

$$\begin{bmatrix} \square & \square \\ \square & \square \end{bmatrix} \begin{bmatrix} \mathbf{r}_e \\ \mathbf{x}_e \end{bmatrix} = \begin{bmatrix} \square \\ \mathbf{0} \end{bmatrix} . \tag{3.17.1}$$

Complete the blocks of the system matrix and of the right-hand-side vector such that the solution of (3.17.1) provides the (unique) least-squares solution of $\mathbf{A}\mathbf{x} = \mathbf{b}$ in $\mathbf{x}_e$ and the residual $\mathbf{r} := \mathbf{A}\mathbf{x} - \mathbf{b}$ in $\mathbf{r}_e$.

SOLUTION for (3-17.a) → 3-17-1-0:extneqs1.pdf

▲

(3-17.b) ⚀ (5 min.) [depends on Sub-problem (3-17.a)]

What is the number of non-zero entries of the system matrix $\mathbf{A}_e$ of the extended normal equation (3.17.1) in terms of the number $\mathrm{nnz}(\mathbf{A})$ of non-zero entries of $\mathbf{A}$?

$$\mathrm{nnz}(\mathbf{A}_e) = \square ,$$

SOLUTION for (3-17.b) → 3-17-2-0:extneqs2.pdf

▲

(3-17.c) ⚀ (15 min.) [depends on Sub-problem (3-17.a)]

The function

```
TripletMatrix
  extNeqSysMatCOO(
    unsigned int m, unsigned int n,
    const TripletMatrix &A_coo);
```

creates the matrix $\mathbf{A}_e$ of the extended normal equations (3.17.1) in triplet (COO) format when given the matrix $\mathbf{A} \in \mathbb{R}^{m,n}$ in triplet format through the argument `A`, and its dimensions through `m` and `n`. The following typedefs have been used:

```
using Triplet = Eigen::Triplet<double>;
using TripletMatrix = std::vector<Triplet>;
```

Complete the implementation of `extNeqSysMatCOO()` given in Code 3.17.3.

C++ code 3.17.3: Initialization of $\mathbf{A}_e$ in triplet format

```
using Triplet = Eigen::Triplet<double>;
using TripletMatrix = std::vector<Triplet>;
TripletMatrix extNeqSysMatCOO(unsigned int m, unsigned int n,
                              const TripletMatrix &A_coo) {
  TripletMatrix A_ext_coo{};
  for (int j = 0; j < (int)m; ++j) {
    A_ext_coo.push_back(
      Triplet{ ,  ,  });
  }
  for (const auto &t : A_coo) {
    A_ext_coo.push_back(
      Triplet{ ,  ,  });
    A_ext_coo.push_back(
      Triplet{ ,  ,  });
  }
  return A_ext_coo;
}
```

HIDDEN HINT 1 for (3-17.c) → 3-17-3-0:extneqh1.pdf

SOLUTION for (3-17.c) → 3-17-3-1:extneqs3.pdf

▲

End Problem 3-17, 35 min.

Problem 3-18: Least-Squares Solution of Low-Rank Linear Systems

For any linear system of equations (LSE) a generalized solution can be defined, see [Lecture → Def. 3.1.3.1]. In this problem we study efficient algorithms for computing the generalized solution of an LSE whose system matrix has known low rank.

This problem depends on [Lecture → Section 3.1.3], [Lecture → Section 3.4.1], [Lecture → Section 3.4.2], and, crucially, on [Lecture → Section 3.4.3].

Every matrix $\mathbf{M} \in \mathbb{R}^{m,n}$, $m,n \in \mathbb{N}$, with $\operatorname{rank}(\mathbf{M}) = p$, $0 \leq p \leq \min\{m,n\}$, can be represented as a matrix product $\mathbf{M} = \mathbf{A}\mathbf{B}^\top$ with $\mathbf{A} \in \mathbb{R}^{m,p}$ and $\mathbf{B} \in \mathbb{R}^{n,p}$. This is of particular interest if $p \ll m,n$.

The following data type stores a low-rank matrix in the mentioned factored representation:

```
struct LowRankMatrix {
  LowRankMatrix(const Eigen::MatrixXd &A, const Eigen::MatrixXd &B)
      : A_(A), B_(B) {
    if (A.cols() != B.cols()) throw std::runtime_error("A,B size
      mismatch");
  }
  operator Eigen::MatrixXd() const { return A_ * (B_.transpose()); }
  Eigen::MatrixXd A_;  // First matrix factor
  Eigen::MatrixXd B_;  // Transpose of second matrix factor
};
```

(3-18.a) ⚀ (15 min.) Based on EIGEN the numerical rank of a matrix can be computed by the following function, see also [Lecture → Code 3.4.2.5]:

C++ code 3.18.1: Computation of numerical rank of a dense matrix in EIGEN

```
2  unsigned int rank(const Eigen::MatrixXd &A,
3                    double tol = std::numeric_limits<double>::epsilon()) {
4    if (A.norm() == 0) return 0;
5    Eigen::JacobiSVD<Eigen::MatrixXd> svd_object(A);
6    const Eigen::VectorXd sv = svd_object.singularValues();
7    const unsigned int n = sv.size();
8    unsigned int r = 0;
9    while ((r < n) && (sv(r) >= sv(0) * A.rows() * A.cols() * tol)) r++;
10   return r;
11 }
```

The overloaded C++ function

```
unsigned int rank(const LowRankMatrix &M,
   double tol = std::numeric_limits<double>::epsilon());
```

is meant to efficiently compute the numerical rank of a matrix stored in a **LowRankMatrix** object using a singular value decomposition, as done in `rank()` from Code 3.18.1 for a generic dense matrix. Fill in the blanks in the following undocumented listing so that the function serves this purpose.

C++ code 3.18.2: Incomplete code for `rank()`

```
unsigned int rank(const LowRankMatrix &M,
                  double tol = std::numeric_limits<double>::epsilon()) {
  unsigned int p = M.A_.cols();
  if ((M.A_.norm() == 0.0) || (M.B_.norm() == 0.0)) return 0;

  Eigen::HouseholderQR<Eigen::MatrixXd> qrA(M.A_);
  Eigen::HouseholderQR<Eigen::MatrixXd> qrB(M.B_);
  const Eigen::MatrixXd RA =
      qrA.matrixQR().block(
          ).template triangularView<Eigen::Upper>();

  const Eigen::MatrixXd RB =
      qrB.matrixQR().block(
          ).template triangularView<Eigen::Upper>();
  Eigen::JacobiSVD<Eigen::MatrixXd> svd_object(                    );

  const Eigen::VectorXd sv = svd_object.singularValues();
  const unsigned int n = sv.size();
  unsigned int r = 0;
  while ((r < n) && (sv(r) >= sv(0) *                    )) r++;

  return r;
}
```

SOLUTION for (3-18.a) → 3-18-1-0:.pdf ▲

(3-18.b) ⚄ (40 min.) [depends on Sub-problem (3-18.a)]

In the file `lowranklsqsol.hpp` provide an *efficient* implementation of the C++ function

```
Eigen::VectorXd lowRankLsqSol(
    const LowRankMatrix &M, const Eigen::VectorXd &b,
    double tol = std::numeric_limits<double>::epsilon());
```

that computes the generalized solution of the linear system of equations $\mathbf{M}\mathbf{x} = \mathbf{b}$ ($\mathbf{M}$ supplied through `M`, $\mathbf{b}$ through `b`) in the sense of

Definition [Lecture → Def. 3.1.3.1]. Generalized solution of a linear system of equations

The generalized solution $\mathbf{x}^\dagger \in \mathbb{R}^n$ of a linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, is defined as

$$\mathbf{x}^\dagger := \operatorname{argmin}\{\|\mathbf{x}\|_2 : \mathbf{x} \in \operatorname{lsq}(\mathbf{A}, \mathbf{b})\} . \tag{3.18.6}$$

Here $\operatorname{lsq}(\mathbf{A}, \mathbf{b})$ is the set of least-squares solutions of $\mathbf{A}\mathbf{x} = \mathbf{b}$, *cf.* [Lecture → (3.1.1.2)].

Moreover, the implementation should take the result of `rank(M)` (`rank()` as in Code 3.18.2) as the true rank of $\mathbf{M}$.

HIDDEN HINT 1 for (3-18.b) → 3-18-2-0:lrlsqbh1.pdf

HIDDEN HINT 2 for (3-18.b) → 3-18-2-1:lrlsqbh2.pdf

SOLUTION for (3-18.b) → 3-18-2-2:.pdf ▲

(3-18.c) ⚄ (10 min.) [depends on Sub-problem (3-18.b)]

The function `lowRankLsqSol()` from Sub-problem (3-18.b) is called with an **LowRankMatrix** object `M`, which represents an $m \times n$-matrix of rank atmost p.

Assuming $p \ll m,n$ small and fixed, what is the asymptotic complexity of your implementation of `LowRankLsqSol()` for $m,n \to \infty$.

$$\text{cost}(\texttt{LowRankLsqSol()}) = O\left(\quad\right) \quad \text{for} \quad m,n \to \infty\,.$$

SOLUTION for (3-18.c) → 3-18-3-0:.pdf ▲

End Problem 3-18, 65 min.

Problem 3-19: Least Squares problems via four approaches

In this problem, you are asked to solve the arising least squares problem of linear fitting via four different approaches. This exercise serves as an overview of the basic algorithms we learned.

This question requires knowledge about the techniques introduced in [Lecture → Section 3.2], [Lecture → Section 3.3.4], and [Lecture → Section 3.4.3].

For m points in time t_i, we have measured data f_i for a time-dependent physical quantity $f(t)$. Assume that $m > n$ and $f(t)$ is a linear combination

$$f(t) = \sum_{j=1}^{n} \gamma_j \phi_j(t), \quad \gamma_j \in \mathbb{R}, j = 1, \ldots, n$$

where the expressions of $\phi_j, j = 1, \ldots, n$ are known. We aim to calculate the coefficients $\gamma_j, j = 1, \ldots, n$ being the solution of the following regularized least squares problem

$$\sum_{i=1}^{m} (f(t_i) - f_i)^2 \longrightarrow \text{min.} \tag{3.19.1}$$

(3-19.a) ⚀ (10 min.) First, we need to recast the least square problem (3.19.1) into an overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ where $\mathbf{x} := [\gamma_1, \gamma_2, \ldots, \gamma_n]^\top$ and $\mathbf{b} := [f_1, f_2, \ldots, f_m]^\top$. Note the size of $\mathbf{A}$ and the expression of entities in $\mathbf{A}$.

$$\mathbf{A} \in \mathbb{R}^{\square,\square}, \quad (\mathbf{A})_{i,j} = \square.$$

SOLUTION for (3-19.a) → 3-19-1-0:odeexas.pdf ▲

For the rest questions, suppose that we have in hand a function `make_A(t)` providing the correct matrix $\mathbf{A}$ required in Sub-problem (3-19.a) where `t` is the vector containing $t_1, t_2, \ldots, t_n$. Assume that matrix $\mathbf{A}$ is of full rank, namely, $\text{rank}(\mathbf{A}) = n$.

(3-19.b) ⚀ (10 min.) If we would like to use the normal equation method to solve the least squares problem $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 \rightarrow \text{min}$, write below the square linear system of equations for calculating the solution $\mathbf{x}$.

$$\square \mathbf{x} = \square$$

SOLUTION for (3-19.b) → 3-19-2-0:odeexbs.pdf ▲

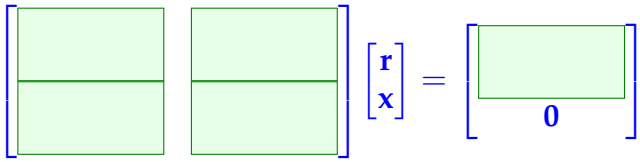
(3-19.c) ⚀ (20 min.) In the file `datafit.hpp` write a C++ function

```
Eigen::VectorXd data_fit_normal(const Eigen::VectorXd &t,
                                const Eigen::VectorXd &f);
```

that solves the least squares problem (3.19.1) based on the normal equation method.

SOLUTION for (3-19.c) → 3-19-3-0:odeexbs.pdf ▲

(3-19.d) ⚂ (20 min.) By introducing the residual $\mathbf{r} := \mathbf{A}\mathbf{x} - \mathbf{b}$ as additional auxiliary variable the least squares problem $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 \to \min$ can be solved by solving the following extended normal equations



where $\mathbf{x}$ is the least squares solution. Please complete the missing blocks.

SOLUTION for (3-19.d) → 3-19-4-0:odeexbs.pdf ▲

(3-19.e) ⚂ (20 min.) In the file `datafit.hpp` write a C++ function

```
Eigen::VectorXd data_fit_xnormal(const Eigen::VectorXd &t,
                                 const Eigen::VectorXd &f);
```

that solves the least squares problem (3.19.1) based on the extended normal equation method in Sub-problem (3-19.d).

SOLUTION for (3-19.e) → 3-19-5-0:odeexbs.pdf ▲

(3-19.f) ⚂ (15 min.) If we would like to use the QR-decomposition to solve the least squares problem $\|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 \to \min$, illustrate and justify the algorithm *step by step*.

SOLUTION for (3-19.f) → 3-19-6-0:odeexbs.pdf ▲

(3-19.g) ⚂ (20 min.) [depends on Sub-problem (3-19.f)]

In the file `datafit.hpp` write a C++ function

```
Eigen::VectorXd data_fit_qr(const Eigen::VectorXd &t,
                            const Eigen::VectorXd &f);
```

that solves the least squares problem (3.19.1) based on the QR-decomposition.

SOLUTION for (3-19.g) → 3-19-7-0:odeexbs.pdf ▲

(3-19.h) ⚂ (15 min.) If we would like to use the SVD to solve the least squares problem (3.19.1), illustrate and justify the algorithm *step by step*.

SOLUTION for (3-19.h) → 3-19-8-0:odeexbs.pdf ▲

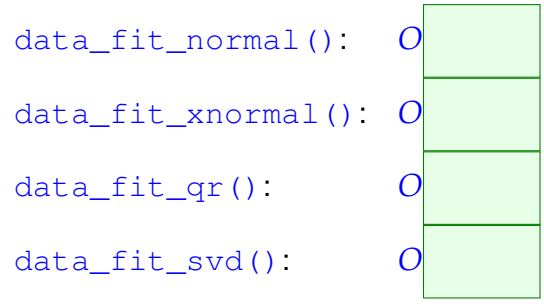
(3-19.i) ⚂ (20 min.) In the file `datafit.hpp` write a C++ function

```
Eigen::VectorXd data_fit_svd(const Eigen::VectorXd &t,
                             const Eigen::VectorXd &f);
```

that solves the least squares problem (3.19.1) based on the SVD.

SOLUTION for (3-19.i) → 3-19-9-0:odeexbs.pdf ▲

(3-19.j) ⚂ (30 min.) What is the asymptotic complexity of each C++ function for fixed n and $m \to \infty$?



SOLUTION for (3-19.j) → 3-19-10-0:odeexbs.pdf ▲

End Problem 3-19 , 180 min.

Problem 3-20: Computing with Kronecker Products

In [Lecture → Ex. 1.4.3.8] we could catch a glimpse of the fact that matrix operations can be carried out with significantly reduced cost, if (some of) the involved matrices have Kronecker product structure.

This problem is connected to [Lecture → Ex. 1.4.3.8] and [Lecture → Section 3.4.4.2]. It requests implementation in C++ based on EIGEN.

We recall the definition of the Kronecker product

Definition [Lecture → Def. 1.4.3.7]. Kronecker product

The **Kronecker product** $\mathbf{A} \otimes \mathbf{B}$ of two matrices $\mathbf{A} \in \mathbb{K}^{m,n}$ and $\mathbf{B} \in \mathbb{K}^{\ell,k}$, $m,n,\ell,k \in \mathbb{N}$, is the $(m\ell) \times (nk)$-matrix

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} (\mathbf{A})_{1,1}\mathbf{B} & (\mathbf{A})_{1,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{1,n}\mathbf{B} \\ (\mathbf{A})_{2,1}\mathbf{B} & (\mathbf{A})_{2,2}\mathbf{B} & & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & & \vdots \\ (\mathbf{A})_{m,1}\mathbf{B} & (\mathbf{A})_{m,2}\mathbf{B} & \dots & \dots & (\mathbf{A})_{m,n}\mathbf{B} \end{bmatrix} \in \mathbb{K}^{m\ell,nk} .$$

and the concept of the **vectorization** of a matrix by stacking its columns on top of each other:

$$\mathrm{vec} : \mathbb{K}^{n,m} \to \mathbb{K}^{n\cdot m} \quad , \quad \mathrm{vec}(\mathbf{A}) := \begin{bmatrix} (\mathbf{A})_{:,1} \\ (\mathbf{A})_{:,2} \\ \vdots \\ (\mathbf{A})_{:,m} \end{bmatrix} \in \mathbb{R}^{n\cdot m} . \qquad [\text{Lecture} \to (1.2.3.5)]$$

When solving the sub-problems you can rely on the following properties of the Kronecker product

Lemma 3.20.1. Properties of matrix Kronecker products

- *The Kronecker product according to [Lecture → Def. 1.4.3.7] is a bilinear mapping* $\mathbb{R}^{m,n} \times \mathbb{R}^{\ell,k} \to \mathbb{R}^{m\ell,nk}$.
- *If* $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ *are matrices of such size that the matrix products* $\mathbf{AC}$ *and* $\mathbf{BD}$ *can be formed, then*

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}) . \tag{3.20.2}$$

- *For all* $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{\ell,k}$, $m,n,\ell,k \in \mathbb{N}$, $\mathbf{X} \in \mathbb{R}^{k,n}$,

$$(\mathbf{A} \otimes \mathbf{B})\,\mathrm{vec}(\mathbf{X}) = \mathrm{vec}\left(\mathbf{B}\mathbf{X}\mathbf{A}^\top\right) . \tag{3.20.3}$$

- *For all* $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{B} \in \mathbb{R}^{\ell,k}$, $m,n,\ell,k \in \mathbb{N}$,

$$(\mathbf{A} \otimes \mathbf{B})^\top = \mathbf{A}^\top \otimes \mathbf{B}^\top . \tag{3.20.4}$$

(3-20.a) ⚀ (5 min.) Let $\mathbf{A} \in \mathbb{R}^{m,m}$ and $\mathbf{B} \in \mathbb{R}^{n,n}$ be regular/invertible matrices. Then

$$(\mathbf{A} \otimes \mathbf{B})^{-1} = \square \otimes \square .$$

SOLUTION for (3-20.a) → 3-20-1-0:kprs1.pdf ▲

(3-20.b) ⚂ (20 min.) Given $\mathbf{A} \in \mathbb{R}^{m,n}$ and $\mathbf{B} \in \mathbb{R}^{\ell,k}$, $m,n,\ell,k \in \mathbb{N}$, with $m \geq n$ and $\ell \geq k$, let

$$\mathbf{A} = \mathbf{Q}_A \mathbf{R}_A\,, \quad \begin{matrix}\mathbf{Q}_A^\top \mathbf{Q}_A = \mathbf{I}_n \\ \mathbf{R}_A \in \mathbb{R}^{n,n} \text{ upper triangular}\end{matrix}\,, \quad \mathbf{B} = \mathbf{Q}_B \mathbf{R}_B\,, \quad \begin{matrix}\mathbf{Q}_B^\top \mathbf{Q}_B = \mathbf{I}_k \\ \mathbf{R}_B \in \mathbb{R}^{k,k} \text{ upper triangular,}\end{matrix}$$

be economical/thin **QR-factorizations** of $\mathbf{A}$ and $\mathbf{B}$, respectively. Then an economical QR-factorization of the Kronecker product $\mathbf{A} \otimes \mathbf{B}$ is

$$\mathbf{A} \otimes \mathbf{B} = \underbrace{\qquad\qquad}_{\mathbf{Q}\text{-factor}} \cdot \underbrace{\qquad\qquad}_{\mathbf{R}\text{-factor}}\,.$$

SOLUTION for (3-20.b) → 3-20-2-0:kprs2.pdf ▲

(3-20.c) ⚂ (60 min.) [depends on Sub-problem (3-20.a)]

In the file `kronprodmisc.hpp` implement an *efficient* C++ function

```
Eigen::VectorXd solveAkpBSys(
  const Eigen::MatrixXd &A, const Eigen::MatrixXd &B,
  const Eigen::VectorXd &y);
```

which solves the linear system of equations

$$(\mathbf{A} \otimes \mathbf{B})\mathbf{x} = \mathbf{y}$$

for given regular matrices $\mathbf{A} \in \mathbb{R}^{m,m}$, $\mathbf{B} \in \mathbb{R}^{n,n}$, and given right-hand side vector $\mathbf{y} \in \mathbb{R}^{m\cdot n}$, and returns the solution vector $\mathbf{x} \in \mathbb{R}^{m\cdot n}$.

HIDDEN HINT 1 for (3-20.c) → 3-20-3-0:kpmh3.pdf

HIDDEN HINT 2 for (3-20.c) → 3-20-3-1:kpmh32.pdf

SOLUTION for (3-20.c) → 3-20-3-2:kprsol3.pdf ▲

(3-20.d) ⚂ (15 min.) [depends on Sub-problem (3-20.c)]

What is the asymptotic complexity of your implementation of `solveAkpBSys()` from Sub-problem (3-20.c) for $m,n \to \infty$. Give a sharp bound for general densely populated matrices.

$$\texttt{cost(solveAkpBSys())} = O\Big(\qquad\qquad\Big) \quad \text{for} \quad m,n \to \infty\,.$$

SOLUTION for (3-20.d) → 3-20-4-0:kps4.pdf ▲

Following [VP93] we study the best approximation of matrices by Kronecker products of smaller matrices. The key idea in [VP93] relies on the **rearrangement operator**

$$\mathrm{rar}_{\ell,k} : \mathbb{R}^{m\ell,nk} \to \mathbb{R}^{mn,\ell k}\,, \quad \mathrm{rar}_{\ell,k}(\mathbf{A}) := \begin{bmatrix}\widehat{\mathbf{A}}_1 \\ \vdots \\ \widehat{\mathbf{A}}_n\end{bmatrix}\,, \quad \widehat{\mathbf{A}}_j = \begin{bmatrix}\mathrm{vec}(\mathbf{B}_{1,j})^\top \\ \vdots \\ \mathrm{vec}(\mathbf{B}_{m,j})^\top\end{bmatrix}\,, \tag{3.20.9}$$

based on the block decomposition

$$\mathbf{A} = \begin{bmatrix} \mathbf{B}_{1,1} & \mathbf{B}_{1,2} & \dots & \mathbf{B}_{1,n} \\ \mathbf{B}_{2,1} & \mathbf{B}_{2,2} & \dots & \mathbf{B}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{B}_{m,1} & \mathbf{B}_{m,2} & \dots & \mathbf{B}_{m,n} \end{bmatrix}, \quad \mathbf{B}_{i,j} \in \mathbb{R}^{\ell,k} . \tag{3.20.10}$$

The authors of [VP93] establish the following remarkable identity.

Lemma 3.20.11. Frobenius distance to Kronecker product

Let $\mathbf{A} \in \mathbb{R}^{m\cdot\ell,n\cdot k}$ be given. If $\mathbf{X} \in \mathbb{R}^{m,n}$ and $\mathbf{Y} \in \mathbb{R}^{\ell,k}$, then

$$\|\mathbf{A} - \mathbf{X} \otimes \mathbf{Y}\|_F = \left\| \operatorname{rar}_{\ell,k}(\mathbf{A}) - \operatorname{vec}(\mathbf{X}) \operatorname{vec}(\mathbf{Y})^\top \right\|_F ,$$

where $\|\cdot\|_F$ stands for the ***Frobenius norm*** *[Lecture → Def. 3.4.4.17] of a matrix.*

(3-20.e) ⚄ (75 min.) Based on Lemma 3.20.11 implement (in the file `kronprodmisc.hpp`) a C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
  kronprodBestApprox(const Eigen::MatrixXd &A,
                     Eigen::Index l, Eigen::Index k);
```

that solves the following best approximation problem:

Given $\mathbf{A} \in \mathbb{R}^{m\cdot\ell,n\cdot k}$ find $\mathbf{X} \in \mathbb{R}^{m,n}$ and $\mathbf{Y} \in \mathbb{R}^{\ell,k}$ such that

$$\|\mathbf{A} - \mathbf{X} \otimes \mathbf{Y}\|_F = \min\left\{ \|\mathbf{A} - \mathbf{X}' \otimes \mathbf{Y}'\|_F : \mathbf{X}' \in \mathbb{R}^{m,n}, \mathbf{Y}' \in \mathbb{R}^{\ell,k} \right\} .$$

The argument `A` passes the matrix $\mathbf{A}$ and `l` and `k` give the size of the second Kronecker factor. They must divide the numbers of rows and columns, respectively, of $\mathbf{A}$.

HIDDEN HINT 1 for (3-20.e) → `3-20-5-0:kp5h1.pdf`

SOLUTION for (3-20.e) → `3-20-5-1:kpsol5.pdf` ▲

References

[VP93] C. F. Van Loan and N. Pitsianis. "Approximation with Kronecker products". In: *Linear algebra for large scale and real-time applications (Leuven, 1992)*. Vol. 232. NATO Adv. Sci. Inst. Ser. E: Appl. Sci. Kluwer Acad. Publ., Dordrecht, 1993, pp. 293–314 (cit. on pp. 128, 129).

End Problem 3-20, 175 min.

Problem 3-21: Normal Equations and Pseudoinverse

The easiest way to solve linear least-squares problems with full-rank coefficient matrix is via the associated normal equations. In case of rank deficiency a generalized least squares solution can be obtained via the pseudoinverse. This problem recalls these concepts.

This problem assumes familiarity with [Lecture → Section 3.1] and [Lecture → Section 3.2].

The **normal equations** [Lecture → Thm. 3.1.2.1] for a linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{b} \in \mathbb{R}^m$, $m, n \in \mathbb{N}$, read $\mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}$. The set of all least squares solutions of $\mathbf{A}\mathbf{x} = \mathbf{b}$ is the affine space

$$\mathrm{lsq}(\mathbf{A},\mathbf{b}) := \left\{\mathbf{x} \in \mathbb{R}^n : \ \mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b}\right\} . \tag{3.21.1}$$

The **pseudoinverse** $\mathbf{A}^\dagger \in \mathbb{R}^{n,m}$ of $\mathbf{A} \in \mathbb{R}^{m,n}$ is defined as selecting the shortest least-squares solution,

$$\mathbf{A}^\dagger\mathbf{b} := \operatorname{argmin}\{\|\mathbf{x}\|_2 : \ \mathbf{x} \in \mathrm{lsq}(\mathbf{A},\mathbf{b})\} . \tag{3.21.2}$$

An equivalent characterization is given in the following theorem.

Theorem [Lecture → Thm. 3.1.3.6]. Formula for generalized least-squares solution

Given $\mathbf{A} \in \mathbb{R}^{m,n}$, $\mathbf{b} \in \mathbb{R}^m$, the generalized solution $\mathbf{x}^\dagger$ of the linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$ is given by

$$\mathbf{x}^\dagger = \mathbf{A}^\dagger\mathbf{b} = \mathbf{V}(\mathbf{V}^\top\mathbf{A}^\top\mathbf{A}\mathbf{V})^{-1}\mathbf{V}^\top\mathbf{A}^\top\mathbf{b} ,$$

where $\mathbf{V}$ is any *matrix whose columns form a basis of $\mathcal{N}(\mathbf{A})^\perp$ (= orthogonal complement of the nullspace of $\mathbf{A}$).*

(3-21.a) ⚂ (10 min.) Let $\mathbf{A} \in \mathbb{R}^{m,n}$ and $\mathbf{b} \in \mathbb{R}^m$ be given. Complete the statement of the **extended normal equations**.

$$\begin{array}{c} \mathbf{x} \in \mathbb{R}^n \quad \text{solves the normal equations} \quad \mathbf{A}^\top\mathbf{A}\mathbf{x} = \mathbf{A}^\top\mathbf{b} \\ \Updownarrow \\ \exists \mathbf{r} \in \mathbb{R}^m\text{:} \quad (\mathbf{x},\mathbf{r}) \quad \text{solve} \quad \begin{bmatrix} \mathbf{I}_m & \square \\ \square & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \mathbf{0} \end{bmatrix} \end{array} \tag{3.21.3}$$

SOLUTION for (3-21.a) → 3-21-1-0:nepis1.pdf ▲

(3-21.b) ⚂ (10 min.) Compute the pseudoinverse of a non-zero *column vector* $\mathbf{a} \in \mathbb{R}^m$, that is, compute $\mathbf{A}^\dagger$ for $\mathbf{A} = \mathbf{a} \in \mathbb{R}^{m,1}$.

$$\mathbf{a}^\dagger = \square .$$

HIDDEN HINT 1 for (3-21.b) → 3-21-2-0:nepih2.pdf

SOLUTION for (3-21.b) → 3-21-2-1:nepis2.pdf ▲

(3-21.c) ⚄ (15 min.) Compute the pseudoinverse of a non-zero *row vector* $\mathbf{a}^\top \in \mathbb{R}^{1,n}$ ($\mathbf{a} \in \mathbb{R}^n$), that is, compute $\mathbf{A}^\dagger$ for $\mathbf{A} = \mathbf{a}^\top \in \mathbb{R}^{1,n}$.

$$\left(\mathbf{a}^\top\right)^\dagger = \quad .$$

HIDDEN HINT 1 for (3-21.c) → 3-21-3-0:nepih2.pdf

SOLUTION for (3-21.c) → 3-21-3-1:nepis2.pdf ▲

End Problem 3-21 , 35 min.

Chapter 4

Filtering Algorithms

Problem 4-1: Autofocus with FFT

In this problem, we will use 2D frequency analysis to find the "best focused" image among a collection of out-of-focus photos. To that end, we will implement an algorithm based on 2D DFT as introduced in [Lecture → Section 4.2.5].

This problem takes for granted that you know about the 2D discrete Fourier transform and its connection with discrete convolutions, see [Lecture → Section 4.2.2] and [Lecture → Section 4.2.5].

Let a grey-scale image consisting of $n \times m$ pixels be given as a matrix $\mathbf{P} \in \mathbb{R}^{n,m}$ as in [Lecture → Ex. 4.2.5.17]; each element of the matrix indicates the gray-value of the pixel as a number between 0 and v_{max}. This image is regarded as the "perfect image".

If a camera is poorly focused due to an inadequate arrangement of the lenses, it will record a blurred image $\mathbf{B} \in \mathbb{R}^{n,m}$. As before [Lecture → (4.2.5.18)], the blurring operation can be modeled through the 2D (periodic) discrete convolution and can be undone provided that the point spread function (PSF) is known. However, in this problem we must not assume any knowledge of the PSF.

Assume that the only data available to us is the "black-box" C++ function (declared in `autofocus.hpp`):

```
Eigen::MatrixXd set_focus(double f);
```

which returns the potentially blurred image $\mathbf{B}(f)$, when the focus parameter f is set to a particular value. The actual operation of `set_focus` is utterly obscure. The focus parameter can be read as the distance of the lenses of a camera that can be changed through turning on and off a stepper motor. The following sequence of images illustrates the impact of setting different focus parameters:

Fig. 16

Fig. 17

Fig. 18

Fig. 19

The problem of *autofocusing* is to determine the value of the focus parameter f, which yields an image $\mathbf{B}(f)$ with the least blur. The idea of the autofocusing algorithm is the following: *The less the image is marred by blur, the larger its "high frequency content"*. The latter can be found out based on the discrete Fourier transform, more precisely, its 2D version.

To translate the autofocus idea into an algorithm we have to give a quantitative meaning to the notion of "high frequency content" of a (blurred) image $\mathbf{C}$, which is done by looking at the second moments of its 2D discrete Fourier transform:

$$V(\mathbf{C}) = \sum_{k_1=0}^{n-1} \sum_{k_2=0}^{m-1} \left(\left(\frac{n}{2} - \left| k_1 - \frac{n}{2} \right| \right)^2 + \left(\frac{m}{2} - \left| k_2 - \frac{m}{2} \right| \right)^2 \right) |\hat{\mathbf{C}}_{k_1,k_2}|^2 \,. \tag{4.1.1}$$

Here, $\hat{\mathbf{C}} \in \mathbb{C}^{n,m}$ stand for the 2D DFT of the image $\mathbf{C}$. Hence, the autofocusing policy can be rephrased as

$$\text{find } f \in \mathbb{R} \text{ such that } V(\mathbf{B}(f)) \text{ becomes maximal.}$$

(4-1.a) ⚀ (20 min.) This sub-problem supplies us with a tool to write an image file from a C++ code. Write a C++ function

```
void save_image(double focus);
```

that saves the image $\mathbf{B}(f)$ returned by `set_focus()` for $f = 0, 1, 2, 3$ in **P**ortable **G**ray**m**ap (PGM) format to the file `./cx_out/image_focus<f>.png`, where `<f>` is a string containing the argument `focus`.

For this task you can use objects of type `PGMObject` (found in NumCSE code repository "`Utils/pgm.hpp`"). You can find example of usages of this class in "`Utils/examples/pgm_example.cpp`".

SOLUTION for (4-1.a) → `4-1-1-0:render.pdf` ▲

(4-1.b) ⚁ (30 min.) Note that the EIGEN DFT module (by M. Saladin) used in this course supports the two-dimensional discrete Fourier transform, see the documentation for usage. Write a C++ function

```
void plot_freq(double focus);
```

that creates 2D color plots of the (modulus of the) 2D DFTs of the images obtained in sub-problem (4-1.a). Clamp the data between 0 and 8000, this means when a value exceeds 8000, then set it to 8000.

HIDDEN HINT 1 for (4-1.b) → `4-1-2-0:fft2h1.pdf`

SOLUTION for (4-1.b) → `4-1-2-1:plotfft2.pdf` ▲

(4-1.c) ⚀ (15 min.) [depends on Sub-problem (4-1.b)]

In the plots generated in Sub-problem (4-1.b), mark (or explain) the regions corresponding to the high and low frequencies.

HIDDEN HINT 1 for (4-1.c) → `4-1-3-0:afh1.pdf`

SOLUTION for (4-1.c) → `4-1-3-1:explain.pdf` ▲

(4-1.d) ⚁ (30 min.) Write a C++ function

```
double high_frequency_content(const Eigen::MatrixXd & C);
```

that returns the value $V(\mathbf{C})$ for a given matrix $\mathbf{C}$. Write a C++ function

```
void plotV();
```

that uses MATPLOTLIBCPP (function `plot()`) and plots the function $V(\mathbf{B}(f))$ in terms of the focus parameter f. Use 100 equidistantly spaced sampling points in the interval $[0,5]$.

SOLUTION for (4-1.d) → `4-1-4-0:plotV.pdf` ▲

(4-1.e) ⚂ (60 min.) Write an *efficient* (you want the auto-focus procedure to take not longer than a few milliseconds!) C++ function

```
double autofocus();
```

that numerically determines optimal focus parameter $f_0 \in [0,5]$ for which the 2nd moment $V(\mathbf{B}(f))$ is maximized. What is the resulting focus parameter f_0?

Two particular challenges have to be tackled in the implementation of `autofocus()`:

(I) To locate the change of slope of $V(\mathbf{B}(f))$ in $[0,5]$ you should use the **bisection algorithm** [Lecture → Section 8.4.1] for finding a zero of the derivative $f \mapsto \frac{dV(\mathbf{B}(f))}{df}$, which will, hopefully, mark the location of the maximum of $V(f)$.

The bisection algorithm for finding a zero of a continuous function $\varphi : [a,b] \to \mathbb{R}$ with $\varphi(a) \cdot \varphi(b) < 0$ is rather simple and lucidly demonstrated in [Lecture → Code 8.4.1.2]. Study that sample code first.

(II) Of course, the derivative is not available, so that we have to approximate it crudely by means of a difference quotient

$$\frac{dV(f)}{df} \approx \frac{V(\mathbf{B}(f+\delta f)) - V(\mathbf{B}(f-\delta f))}{2\delta f} .$$

You can assume that the smallest step of the auto-focus mechanism is 0.05 and so the natural choice for δf is $\delta f = 0.05$. This maximal resolution also tells you a priori how many bisection steps should be performed.

Exploit the structure of the function $V(\mathbf{B}(f))$ that you observed in sub-problem (4-1.d). Use as few evaluations of $V(\mathbf{B}(f))$ as possible!

SOLUTION for (4-1.e) → `4-1-5-0:bisect.pdf` ▲

End Problem 4-1 , 155 min.

Problem 4-2: FFT and least squares

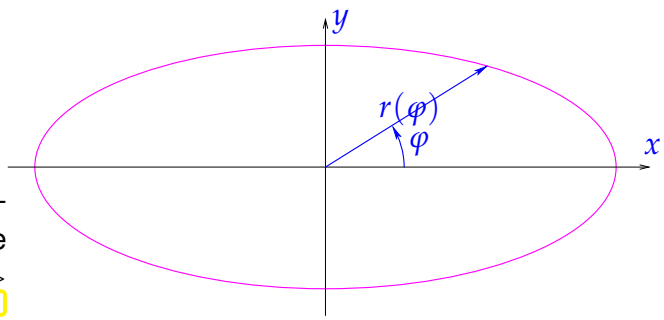
This problem deals with an application of fast Fourier transformation in the context of a least squares (data fitting) problems.

You should be familiar with both DFT [Lecture → Section 4.2], the idea of the least-squares solution of fitting problems [Lecture → Section 3.1.1], and the normal equation method [Lecture → Section 3.2].

The orbit of a planet around the sun is a closed, planar curve $\mathcal{C} \subset \mathbb{R}^2$, which can be described by what is called a polar representation, closely related to polar coordinates:

$$\mathcal{C} := \left\{ \begin{bmatrix} d(\varphi)\cos\varphi \\ d(\varphi)\sin\varphi \end{bmatrix} : 0 \le \varphi < 2\pi \right\},$$

where $d : [0, 2\pi] \to \mathbb{R}^+$ is a given function of the polar angle φ providing the distance of a point on the curve from the origin in the direction of that angle. ▷ Fig. 25



We put the sun in the origin of the coordinate system. For equispaced polar angles (→ Fig. 25) $\varphi_j = 2\pi\frac{j}{n}$, $j = 0, \ldots, n-1$, $n \in \mathbb{N}$ the distance of the planet from the sun is measured and found to be $d_j \in \mathbb{R}$ $j = 0, \ldots, n-1$. We now discuss a numerical method providing the planet's path in polar representation.

Write $\mathcal{P}_m^C$, $m \in \mathbb{N}$, for the space of real **trigonometric polynomials** of degree $\le m$. These are 2π-periodic functions $\mathbb{R} \to \mathbb{R}$ of the form

$$p(t) = \sum_{k=0}^{m} c_k \cos(kt), \quad c_k \in \mathbb{R}. \tag{4.2.1}$$

The idea is to approximate the distance $d(\varphi)$ of the planet from the sun as a function of the angle by a trigonometric polynomial $p^* \in \mathcal{P}_m^C$, which minimized the sum of squares of "distance mismatches":

$$p^* = \operatorname*{argmin}_{p \in \mathcal{P}_m^C} \sum_{j=0}^{n-1} |p(\varphi_j) - d_j|^2. \tag{4.2.2}$$

Throughout, we assume that the number of distance data is more than twice as big as the degree of p^*: $2m < n$.

(4-2.a) ⚀ (30 min.) Recast this task as a standard linear least squares problem

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x} \in \mathbb{R}^?} \|\mathbf{A}\mathbf{x} - \mathbf{b}\|_2, \tag{4.2.3}$$

with a suitable matrix $\mathbf{A} \in \mathbb{R}^{?,?}$ and vectors $\mathbf{b}, \mathbf{x} \in \mathbb{R}^?$.

SOLUTION for (4-2.a) → 4-2-1-0:relsq.pdf ▲

(4-2.b) ⚁ (30 min.) [depends on Sub-problem (4-2.a)]

State the **normal equations** [Lecture → Thm. 3.1.2.1] for the linear least-squares problem found in Sub-problem (4-2.a) in explicit form.

HIDDEN HINT 1 for (4-2.b) → 4-2-2-0:spnh1.pdf

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

(4-2.c) ⚂ (20 min.) [depends on Sub-problem (4-2.a), Sub-problem (4-2.b)]

Write a short EIGEN-based C++ function

```
bool testNormEqMatrix(unsigned int n, unsigned int m);
```

that tests, whether the formula for the matrix product $\mathbf{A}^\top\mathbf{A}$ that you have found in Sub-problem (4-2.b) is correct.

To that end, form the dense matrix $\mathbf{A}$ as an object of type **Eigen::MatrixXd**, compute $\mathbf{A}^\top\mathbf{A}$ and compare with the matrix given by your formula. Of course, the comparison has to be done in a numerically sound way following the advice of [Lecture → Rem. 1.5.3.15].

SOLUTION for (4-2.c) → 4-2-3-0:ts.pdf ▲

(4-2.d) ⚄ (45 min.) [depends on Sub-problem (4-2.b)]

Write a C++ function

```
Eigen::VectorXd find_c(const Eigen::VectorXd &d, unsigned int m);
```

that computes the coefficients c_k, $k = 0,\ldots,m$, of p^* in the form (4.2.1) for arbitrary inputs $d_j \in \mathbb{R}$, $j = 0,\ldots,n-1$. These are passed as vector `d`.

We aim for an efficient implementation: the function must not have asymptotic complexity worse than $O(n\log n)$ in terms of the problem size parameters $m,n \to \infty$.

HIDDEN HINT 1 for (4-2.d) → 4-2-4-0:imph1.pdf

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

(4-2.e) ⚄ (60 min.) [depends on Sub-problem (4-2.d)]

Since Kepler we know that the orbits of planets around the sun are ellipses. Choose $n = 100$ and sample the distance data form the polar form of an ellipse:

$$d(\varphi) = \frac{1}{\sqrt{1-c^2\cos^2\varphi}}\,, \quad \varphi \in [0,2\pi]\,, \quad c = 0.8\,. \tag{4.2.11}$$

1. Using your implementation of `find_c` compute and tabulate the coefficients c_k of p^* for $m = 1,2,3$.
2. Using MATPLOTLIBCPP's `plot()` function create a plot of the ellipse given by (4.2.11).
3. Into the same plot insert the curves described by the trigonometric polynomials p^* of degrees $m = 1,2,3$.

This should be accomplished by a C++ function

```
void fitEllipse();
```

HIDDEN HINT 1 for (4-2.e) → 4-2-5-0:sxh1.pdf

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

End Problem 4-2, 185 min.

Problem 4-3: Multiplication and division of polynomials based on DFT

In [Lecture → § 4.1.3.6] we saw that the formula of discrete convolution [Lecture → Def. 4.1.3.3] also gives the coefficients of products of polynomials. Thus, in light of the realization of discrete convolutions by means of DFT [Lecture → Section 4.2.2], the Fast Fourier Transform (FFT) becomes relevant for efficiently multiplying polynomials of very high degree.

This problem requires knowledge about DFT and discrete convolutions, see [Lecture → Section 4.1.3] and [Lecture → Section 4.2.2]. For the connection of DFT, FFT, and polynomial multiplication also see the YouTube Video.

Polynomials. We call a polynomial in x of degree $N \in \mathbb{N}$ a function $p : \mathbb{K} \to \mathbb{K}$ of the form

$$p(x) = \sum_{j=0}^{N} \gamma_j x^j \quad \text{with coefficients} \quad \gamma_j \in \mathbb{K}\,, \tag{4.3.1}$$

also called monomial representation. Under pointwise addition and multiplication the polynomials of degree N form a vector space $\mathcal{P}_N$ of dimension $N+1$. Associating the vector $\mathbf{c} := [\gamma_0, \ldots, \gamma_N] \in \mathbb{K}^{N+1}$ to every $p \in \mathcal{P}_N$ as given in (4.3.1) induces an isomorphism of the vector spaces $\mathcal{P}_N$ and $\mathbb{K}^{N+1}$. In other words, we can identify $p \in \mathcal{P}_N$ with its coefficient vector $\mathbf{c}$.

Thus, when we have to represent polynomials $\in \mathcal{P}_N$ in a C++ code we can do this via objects of type **Eigen::VectorXd** that contain their coefficient vectors. This convention is adopted throughout this problem.

Let two large numbers $m, n \gg 1$ and two polynomials of degrees $m-1$ and $n-1$, respectively, in monomial representation

$$u(x) = \sum_{j=0}^{m-1} \alpha_j x^j\,, \quad v(x) = \sum_{j=0}^{n-1} \beta_j x^j\,, \quad \alpha_j, \beta_j \in \mathbb{C}\,, \tag{4.3.2}$$

be given.

The tasks that we are going to tackle in this problem are the following.

1. *Polynomial multiplication*: Efficiently compute the coefficients of the polynomial $p = uv$ (point wise product, a polynomial of degree $\leq m+n-2$).
2. *Polynomial division*: Given the polynomials p and u in terms of their coefficients, the degree of p at least as bis as that of u, find a polynomial v, if it exists, such that $p = uv$.

(4-3.a) ⚀ (25 min.) Write a C++ function

```
Eigen::VectorXd polyMult_naive(const Eigen::VectorXd & u, const
   Eigen::VectorXd & v);
```

that "naively", i.e. using a simple loop-based implementation, computes and returns the vector of coefficients of the polynomial $p = uv$. The coefficient vectors for the polynomials u and v are passed in `u` and `v`.

Also determine the asymptotic complexity of your algorithm for $m, n \to \infty$ (separately).

HIDDEN HINT 1 for (4-3.a) → 4-3-1-0:PolyDiv1h.pdf

SOLUTION for (4-3.a) → 4-3-1-1:PolyDiv1s.pdf ▲

(4-3.b) ⚁ (30 min.) Write a C++ function

```
Eigen::VectorXd polyMult_fast(const Eigen::VectorXd & u, const
  Eigen::VectorXd & v);
```

that *efficiently* computes the vector of monomial coefficients of the product polynomial $p = uv$ of the polynomials u and v, passed through their respective coefficient vectors. To that end use the DFT functionality of EIGEN, see [Lecture → Code 4.2.1.22].

HIDDEN HINT 1 for (4-3.b) → 4-3-2-0:PolyDiv2h.pdf

SOLUTION for (4-3.b) → 4-3-2-1:PolyDiv2s.pdf ▲

(4-3.c) ⚂ (10 min.) Determine the complexity of your implementation of algorithm in (4-3.b) making use of the fact that EIGEN's DFT relies on the FFT algorithm [Lecture → Section 4.3] and thus can be carried out with an asymptotic effort $O(n \log n)$ for vector length $n \to \infty$.

HIDDEN HINT 1 for (4-3.c) → 4-3-3-0:PolyDiv3h.pdf

SOLUTION for (4-3.c) → 4-3-3-1:PolyDiv3s.pdf ▲

(4-3.d) ⚄ (20 min.) Give an interpretation of the entries of the vector $\mathrm{DFT}_n \mathbf{u}$ obtained by applying a discrete Fourier transform to the coefficient vector $\mathbf{u}$ of a polynomial u of degree $n-1$? What is an equivalent operation that can be performed for the polynomial u which leads to the same result?

HIDDEN HINT 1 for (4-3.d) → 4-3-4-0:PolyDiv4h.pdf

SOLUTION for (4-3.d) → 4-3-4-1:PolyDiv4s.pdf ▲

The following sub-problems deal with polynomial division. Polynomial division with remainder can be carried out using Euclid's algorithm and is captured by the following theorem:

Theorem 4.3.6. Polynomial devision with remainder

For every $p \in \mathcal{P}_N$ and $u \in \mathcal{P}_m$, $m \leq N$, there exist unique polynomials $q \in \mathcal{P}_{N-m}$ and $r \in \mathcal{P}_{m-1}$ such that

$$p = uq + r\,.$$

Using the notations of the theorem, we are interested in determining, whether the polynomial p can be divided by u exactly, that is, whether $r \equiv 0$. In this case we also want to compute the quotient polynomial q.

(4-3.e) ⚄ (45 min.) [depends on Sub-problem (4-3.b)]

Write a C++ function

```
Eigen::VectorXd polyDiv(const Eigen::VectorXd & p, const
  Eigen::VectorXd & u);
```

that takes the coefficient vectors `p` and `u` of two polynomials p and u, respectively, as arguments, where the degree of p must be at least as large as that of u. The function is to return the coefficient vector of a q, such that $p = uv$, if it exists. If not, a vector of length zero should be returned.

HIDDEN HINT 1 for (4-3.e) → 4-3-5-0:PolyDiv6h.pdf

HIDDEN HINT 2 for (4-3.e) → 4-3-5-1:pdxh2.pdf

SOLUTION for (4-3.e) → 4-3-5-2:PolyDiv6s.pdf ▲

End Problem 4-3 , 130 min.

Problem 4-4: Solving Triangular Toeplitz Linear Systems

In [Lecture → Section 4.5] we learned about Toeplitz matrices, the class of matrices with constant diagonals [Lecture → Def. 4.5.1.7]. Obviously, $m \times n$ Toeplitz matrices are *data-sparse* in the sense that it takes only $m+n-1$ steps to encode them completely. Therefore, operations with Toeplitz matrices can often be done with asymptotic computational cost significantly lower than that of the same operations for a generic matrix of the same size. In [Lecture → Section 4.5.2] we have seen this for FFT-based matrix×vector multiplication for Toeplitz matrices.
In this problem we study an efficient FFT-based algorithm for solving triangular linear systems of equations whose coefficient matrix is Toeplitz. Such linear systems are faced, for instance, when inverting a finite, linear, time-invariant, causal channel (LT-FIR) as introduced in [Lecture → Section 4.1].

This problem is connected with [Lecture → Section 4.5.1] and assumes familiarity with DFT as covered in [Lecture → Section 4.2].

In [Lecture → Section 4.1.2] we found that the output operator of a causal finite linear time-invariant filter with impulse response $\mathbf{h} = [h_0,\dots,h_{n-1}]^\top \in \mathbb{R}^n$ can be obtained by discrete convolution [Lecture → (4.1.2.4)]; see also [Lecture → Def. 4.1.3.3]. Given an input signal $\mathbf{x} := [x_0,\dots,x_{n-1}]^\top \in \mathbb{R}^n$, the first n components $y_0,\dots,y_{n-1}$ of the output are obtained by:

$$\begin{bmatrix} y_0 \\ \vdots \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} h_0 & 0 & \cdots & & \cdots & 0 \\ h_1 & h_0 & 0 & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & 0 \\ h_{n-1} & \cdots & & \cdots & h_1 & h_0 \end{bmatrix}}_{=:\mathbf{H}_n \in \mathbb{R}^{n,n}} \begin{bmatrix} x_0 \\ \vdots \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{4.4.1}$$

Note that the matrix $\mathbf{H}_n \in \mathbb{R}^{n,n}$ is a lower triangular matrix with constant diagonals (linear algebra indexing)

$$(\mathbf{H})_{l,j} = \begin{cases} h_{l-j} & \text{if } l \geq j\,, \\ 0 & \text{else,} \end{cases} \qquad l,j \in \{1,\dots,n\}$$

This matrix is clearly not circulant: see [Lecture → Def. 4.1.4.12]. At the same time, it still belongs to the class of **Toeplitz matrices**, see [Lecture → Def. 4.5.1.7], which generalizes the class of circulant matrices. (Recall [Lecture → Def. 4.5.1.7]: $\mathbf{T} \in \mathbb{K}^{m,n}$ is a Toeplitz matrix if there is a sequence/array $\mathbf{u} = (u_{-m+1},\dots,u_{n-1}) \in \mathbb{K}^{m+n-1}$ such that $t_{ij} = u_{j-i}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$.)

If you want to recover the input signal from the output (*deconvolution*), you will face the following crucial task for digital signal processing: *solve a lower triangular LSE with a Toeplitz system matrix*. Of course, forward elimination, see [Lecture → Section 2.3], can achieve this with asymptotic complexity $\mathcal{O}(n^2)$: see [Lecture → (2.3.1.6)]. However, since a Toeplitz matrix merely has an information content of $\mathcal{O}(n)$ numbers, there might be a more efficient way – which ought to be explored by you in this problem.

(4-4.a) ⚀ (15 min.) Given a Toeplitz matrix $\mathbf{T} \in \mathbb{R}^{n,n}$, find another matrix $\mathbf{S} \in \mathbb{R}^{n,n}$ such that the following block matrix is a *circulant* matrix:

$$\mathbf{C} = \begin{bmatrix} \mathbf{T} & \mathbf{S} \\ \mathbf{S} & \mathbf{T} \end{bmatrix} \in \mathbb{R}^{2n,2n}\,.$$

HIDDEN HINT 1 for (4-4.a) → 4-4-1-0:Toeplitz1h.pdf

SOLUTION for (4-4.a) → 4-4-1-1:Toeplitz1s.pdf ▲

(4-4.b) ⚀ (15 min.) [depends on Sub-problem (4-4.a)]

Write the following C++ function:

```
Eigen::MatrixXd toeplitz(const Eigen::VectorXd &c, const
    Eigen::VectorXd &r)
```

This function should take as input column vectors $\mathbf{c} \in \mathbb{R}^m$ and $\mathbf{r} \in \mathbb{R}^n$ and return a Toeplitz matrix $\mathbf{T} \in \mathbb{R}^{m,n}$ [Lecture → Def. 4.5.1.7] such that (linear-algebra indexing)

$$(\mathbf{T})_{i,j} = \begin{cases} (c)_{i-j+1} & \text{, if } i \geq j \text{ ,} \\ (r)_{j-i+1} & \text{, if } j > i \text{ ,} \end{cases} \quad 1 \leq i \leq m,\ 1 \leq j \leq n \text{ .}$$

SOLUTION for (4-4.b) → 4-4-2-0:tplcpp.pdf ▲

(4-4.c) ⚀ (20 min.) [depends on Sub-problem (4-4.a)]

Show that the following two C++ functions realize the same linear mapping $\mathbf{x} \mapsto \mathbf{y}$, when supplied with the same arguments.

C++ code 4.4.3: Function `toepmatmult()`

```
2  Eigen::VectorXd toepmatmult(const Eigen::VectorXd &c, const Eigen::VectorXd &r,
3                              const Eigen::VectorXd &x) {
4    assert(c.size() == r.size() && c.size() == x.size() &&
5           "c, r, x have different lengths!");
6    return toeplitz(c, r) * x;
7  }
```

Get it on GitLab (toeplitz.hpp).

C++ code 4.4.4: Function `toepmult()`

```
2  Eigen::VectorXd toepmult(const Eigen::VectorXd &c, const Eigen::VectorXd &r,
3                           const Eigen::VectorXd &x) {
4    assert(c.size() == r.size() && c.size() == x.size() &&
5           "c, r, x have different lengths!");
6    const unsigned int n = c.size();
7
8    // Complex arithmetic required here
9    Eigen::VectorXcd cr_tmp = c.cast<std::complex<double>>();
10   Eigen::VectorXcd x_tmp = x.cast<std::complex<double>>();
11
12   // Prepare vector encoding circulant matrix C
13   cr_tmp.conservativeResize(2 * n);
14   cr_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
15   cr_tmp.tail(n - 1).real() = r.tail(n - 1).reverse();
16
17   // Zero padding
18   x_tmp.conservativeResize(2 * n);
19   x_tmp.tail(n) = Eigen::VectorXcd::Zero(n);
20
21   // Periodic discrete convolution from [Lecture → Code 4.2.2.4]
22   Eigen::VectorXd y = pconvfft(cr_tmp, x_tmp).real();
23   y.conservativeResize(n);
24
```

```
25   return y;
26 }
```

Get it on GitLab (toeplitz.hpp).

The function `pconvfft` is defined in [Lecture → Section 4.2.2], [Lecture → Code 4.2.2.4], and `toeplitz()` is the simple C++ function from Sub-problem (4-4.b).

HIDDEN HINT 1 for (4-4.c) → `4-4-3-0:Toeplitz1h.pdf`

SOLUTION for (4-4.c) → `4-4-3-1:Toeplitz2s.pdf` ▲

(4-4.d) ⚀ (10 min.) What is the asymptotic complexity of `toepmatmult()` (→ Code 4.4.3) and `toepmult()` (→ Code 4.4.4)?

SOLUTION for (4-4.d) → `4-4-4-0:Toeplitz3s.pdf` ▲

(4-4.e) ⚄ (45 min.) Explain in detail what the following C++ functions are meant for and why they are algebraically equivalent (i.e. equivalent when ignoring roundoff errors), provided that `h.size()=2^l`.

C++ code 4.4.6: Function `ttmatsolve`

```
2  Eigen::VectorXd ttmatsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y) {
3    assert(h.size() == y.size() && "h and y have different lengths!");
4    const unsigned int n = h.size();
5
6    Eigen::VectorXd h_tmp = Eigen::VectorXd::Zero(n);
7    h_tmp(0) = h(0);
8
9    Eigen::MatrixXd T = toeplitz(h, h_tmp);
10
11   Eigen::VectorXd x = T.fullPivLu().solve(y);
12
13   return x;
14 }
```

Get it on GitLab (toeplitz.hpp).

C++ code 4.4.7: Function `ttrecsolve`

```
2  Eigen::VectorXd ttrecsolve(const Eigen::VectorXd &h, const Eigen::VectorXd &y,
3                             int l) {
4    assert(h.size() == y.size() && "h and y have different lengths!");
5    // Result vector
6    Eigen::VectorXd x;
7    // Trivial case of asn 1x1 LSE
8    if (l == 0) {
9      x.resize(1);
10     x(0) = y(0) / h(0);
11   } else {
12     int n = std::pow(2, l);
13     int m = n / 2;
14
15     // Check matching length of vectors
16     assert(h.size() == n && y.size() == n &&
17            "h and y have length different from 2^l!");
18
19     Eigen::VectorXd x1 = ttrecsolve(h.head(m), y.head(m), l - 1);
20     Eigen::VectorXd y2 =
```

```
21          y.segment(m, m) -
22          toepmult(h.segment(m, m), h.segment(1, m).reverse(), x1);
23      Eigen::VectorXd x2 = ttrecsolve(h.head(m), y2, l - 1);
24
25      x.resize(n);
26      x.head(m) = x1;
27      x.tail(m) = x2;
28    }
29
30    return x;
31  }
```

Get it on GitLab (toeplitz.hpp).

HIDDEN HINT 1 for (4-4.e) → 4-4-5-0:Toeplitz4h.pdf

SOLUTION for (4-4.e) → 4-4-5-1:Toeplitz4s.pdf ▲

(4-4.f) ⊡ (5 min.) Why is the algorithm implemented in `ttrecsolve()` called a "divide & conquer" method?

SOLUTION for (4-4.f) → 4-4-6-0:Toeplitz5s.pdf ▲

(4-4.g) ⚂ (45 min.) Perform a runtme comparison of `toepmatmult` vs `toepmult` and `ttmatsolve` vs `ttrecsolve` for

```
h = Eigen::VectorXd::LinSpaced(n,1,n).cwiseInverse();
```

and $n = 2^l, l = 3, \ldots, 9$. Plot the timing results for both functions in doubly logarithmic scale. Describe your observations.

HIDDEN HINT 1 for (4-4.g) → 4-4-7-0:tp6h1.pdf

SOLUTION for (4-4.g) → 4-4-7-1:Toeplitz6s.pdf ▲

(4-4.h) ⚄ (20 min.) Derive the asymptotic complexity of both functions `ttmatsolve()` and `ttrecsolve()` in terms of the problem size parameter $n \to \infty$.

SOLUTION for (4-4.h) → 4-4-8-0:Toeplitz7s.pdf ▲

(4-4.i) ⚄ (45 min.) For the case that `n = h.size()` is **not** a power of 2, implement a *wrapper function*

```
Eigen::VectorXd ttsolve(const Eigen::VectorXd &h, const
   Eigen::VectorXd &y);
```

for `ttrecsolve` that, in the absence of roundoff errors, would behave exactly like `ttmatsolve`, i.e. `ttsolve(h,y)== ttmatsolve(h,y)`. The term "wrapper function" indicates that `ttsolve()` is supposed to call `ttrecsolve()` after preparing suitable input vectors.

HIDDEN HINT 1 for (4-4.i) → 4-4-9-0:tplhext.pdf

SOLUTION for (4-4.i) → 4-4-9-1:Toeplitz8s.pdf ▲

End Problem 4-4, 220 min.

Problem 4-5: Implementing Discrete Convolution

[Lecture → Section 4.2.2] discussed in detail the implementation of discrete convolutions of signals stored in vectors by means of the discrete Fourier transform (DFT). This problem revisits these techniques and adds a new twist to implementation.

This problem assumes solid knowledge of [Lecture → Section 4.2.2].

From [Lecture → Section 4.1.3] we recall the definitions

Definition [Lecture → Def. 4.1.3.3]. Discrete convolution

Given $\mathbf{x} = [x_0,\ldots,x_{m-1}]^\top \in \mathbb{K}^m$, $\mathbf{h} = [h_0,\ldots,h_{n-1}]^\top \in \mathbb{K}^n$ their **discrete convolution** (DCONV) $\mathbf{h} * \mathbf{x}$ is the vector $\mathbf{y} = [y_0,\ldots,y_{m+n-2}]^\top \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j \,, \quad k = 0,\ldots,m+n-2 \,, \qquad \text{[Lecture → (4.1.3.4)]}$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$.

and

Definition *cf.* [Lecture → Def. 4.1.4.7]. Discrete periodic convolution (of vectors)

The **discrete periodic convolution** $\mathbf{p} *_n \mathbf{x}$ of two vectors $\mathbf{p},\mathbf{x} \in \mathbb{K}^n$ is the vector $\mathbf{y} \in \mathbb{K}^n$ with components (C++ indexing)

$$(\mathbf{y})_k := \sum_{j=0}^{n-1} (\mathbf{p})_{(k-j) \bmod n} (\mathbf{x})_j \,, \quad k \in \{0,\ldots,n-1\} \,.$$

($(k-j) \bmod n \in \{0,\ldots,n-1\} \;\hat{=}\;$ `(k-j)%n` in C++ syntax.)

(4-5.a) ⚀ (30 min.) For given $\mathbf{h} \in \mathbb{K}^n$, $\mathbf{x} \in \mathbb{K}^m$, specify the vectors $\widetilde{\mathbf{h}},\widetilde{\mathbf{x}}$, the *minimal* vector length $N \in \mathbb{N}$, and the missing index range `? :?` in the formula

$$\mathbf{h} * \mathbf{x} = \left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_{?:?} \,. \tag{4.5.1}$$

C++ indexing has to be used and the notational convention is $(\mathbf{v})_{r:s} = [(\mathbf{v})_r,\ldots,(\mathbf{v})_s] \in \mathbb{K}^{s-r+1}$ for $r,s \in \mathbb{N}$, $r \leq s$. Note the different discrete convolutions on the left-hand and right-hand side of (4.5.1).

SOLUTION for (4-5.a) → `4-5-1-0:ptdcs1s.pdf` ▲

Let us assume that the discrete Fourier transform in EIGEN relies on a backend library, which offers a very efficient implementation of DFT with asymptotic computational cost $O(n \log n)$ *only* for vectors of length $n := 2^\ell$, $\ell \in \mathbb{N}$, but may not be fast for vectors of other lengths. To avoid degraded performance of a numerical code, the following rule is imposed throughout this problem:

EIGEN's built-in DFT implementations may be invoked only for vectors of length 2^ℓ for some $\ell \in \mathbb{N}$!

(4-5.b) ⚀ (90 min.) Based on EIGEN, in the file `discreteconvolution.hpp` implement an *efficient* function

```
Eigen::VectorXd pconv_fast(const Eigen::VectorXd &p,
                           const Eigen::VectorXd &x);
```

that computes the discrete periodic convolution $\mathbf{p} *_n \mathbf{x}$ as defined in [Lecture → Def. 4.1.4.7] of two real vectors $\mathbf{p}, \mathbf{x} \in \mathbb{R}^n$ of the same length.

HIDDEN HINT 1 for (4-5.b) → 4-5-2-0:dcv2h1.pdf

SOLUTION for (4-5.b) → 4-5-2-1:dcsol1.pdf ▲

End Problem 4-5 , 120 min.

Problem 4-6: Asymptotic Cost of EIGEN-Based Functions

This short problem is about reading off the asymptotic computational cost of some C++ functions performing numerical linear algebra tasks based on EIGEN. It relies on information provided in [Lecture → Section 1.4.2], [Lecture → § 2.5.0.4], [Lecture → § 3.3.3.38], and [Lecture → Section 4.3].

Just reading of C++ code is required.

(4-6.a) ⚀ (8 min.) The function `slvtriag()` listed in Code 4.6.1 expects two vector arguments of equal length $n \in \mathbb{N}$. What is its asymptotic computational complexity for $n \to \infty$?

$$\mathrm{cost}(\texttt{slvtriag}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

C++ code 4.6.1: Function `slvtriag()`

```
2 Eigen::VectorXd slvtriag(const Eigen::VectorXd &u, const Eigen::VectorXd &v) {
3   assert((u.size() == v.size()) && "Size mismatch");
4   const Eigen::MatrixXd A{u * v.transpose()};
5   return A.triangularView<Eigen::Upper>().solve(u);
6 }
```

SOLUTION for (4-6.a) → 4-6-1-0:s1.pdf ▲

(4-6.b) ⚀ (8 min.) What is the asymptotic complexity of the C++ function `slvsymrom()` listed as Code 4.6.2 in terms of the length n of its argument vector `u`?

$$\mathrm{cost}(\texttt{slvsymrom}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

A sharp bound is expected.

C++ code 4.6.2: Function `slvsymrom()`

```
2 Eigen::VectorXd slvsymrom(const Eigen::VectorXd &u) {
3   const unsigned int n = u.size();
4   return (Eigen::MatrixXd::Identity(n, n) + u * u.transpose()).lu().solve(u);
5 }
```

SOLUTION for (4-6.b) → 4-6-2-0:s2.pdf ▲

(4-6.c) ⚀ (12 min.) Code 4.6.3 lists the C++ function `getcompbas()`, whose argument `A` is a densely populated matrix $\mathbf{A} \in \mathbb{R}^{n,k}$, $k < n$. Give a sharp asymptotic bound for the asymptotic computational cost of `getcompbas()` for $n \to \infty$ assuming k to be small and fixed.

$$\mathrm{cost}(\texttt{getcompbas}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

C++ code 4.6.3: Function `getcompbas()`

```
2 Eigen::MatrixXd getcompbas(const Eigen::MatrixXd &A) {
3   const int n = A.rows();
4   const int k = A.cols();
5   Eigen::HouseholderQR<Eigen::MatrixXd> qr(A);
```

```
  return qr.householderQ() *
         (Eigen::MatrixXd(n, n - k) << Eigen::MatrixXd::Zero(k, n - k),
          Eigen::MatrixXd::Identity(n - k, n - k))
             .finished();
}
```

HIDDEN HINT 1 for (4-6.c) → 4-6-3-0:asch31.pdf

SOLUTION for (4-6.c) → 4-6-3-1:s3.pdf ▲

(4-6.d) ⚄ (12 min.) Code 4.6.4 shows the implementation of the C++ function `fft2_square()`, which requires a complex $n \times n$-matrix $\mathbf{Y} \in \mathbb{C}^{n,n}$ as argument `Y`. What is a sharp bound on the asymptotic complexity of `fft2_square()` as $n \to \infty$?

$$\text{cost}(\texttt{fft2\_square}) = O(\quad) \quad \text{for} \quad n \to \infty\,.$$

C++ code 4.6.4: Function `fft2_square()`

```
Eigen::MatrixXcd fft2_square(const Eigen::MatrixXcd &Y) {
  const unsigned int n = Y.rows();
  assert((n == Y.cols()) && "Matrix Y must be square");
  Eigen::MatrixXcd tmp(n, n);
  // Note:  The updated Eigen::FFT Module allows for
  // 2D Transforms but for the sake of the
  // exercise we do it explicitly with 1D
  // Transforms instead.
  Eigen::FFT fft;
  for (int k = 0; k < n; ++k) {
    tmp.row(k) = fft.fwd(Y.row(k)).transpose();
  }
  for (int k = 0; k < n; ++k) {
    tmp.col(k) = fft.fwd(tmp.col(k));
  }
  return tmp;
}
```

SOLUTION for (4-6.d) → 4-6-4-0:s4.pdf ▲

End Problem 4-6, 40 min.

Problem 4-7: Modified Cosine Transform

Many so-called trigonometric transformations can be reduced to the discrete Fourier transform (DFT). One such example is studied in this problem.

Assumes familiarity with the discrete Fourier transform, C++, and the complex exponential

For a given vector $\mathbf{a} = [a_0, \ldots, a_{n-1}]^\top \in \mathbb{R}^n$, $n \in \mathbb{N}$, we want to compute another vector $\mathbf{f} := [f_0, \ldots, f_{n-1}]^\top \in \mathbb{R}^n$ according to the formula

$$f_k := \sum_{j=0}^{n-1} a_j \cos\left(\pi \frac{jk}{n}\right), \quad k = 0, \ldots, n-1 . \tag{4.7.1}$$

Using the identity $\cos x = \frac{1}{2}(\exp(-\imath x) + \exp(\imath x))$, we find the alternative formula

$$f_k = \sum_{\ell=0}^{n-1} \tfrac{1}{2} a_\ell \exp\left(-2\pi\imath \frac{k\ell}{2n}\right) + \exp\left(\pi\imath \frac{k(n-1)}{n}\right) \cdot \sum_{\ell=0}^{n-1} \tfrac{1}{2} a_{n-1-\ell} \exp\left(-2\pi\imath \frac{k\ell}{2n}\right) \tag{4.7.2}$$

for $k = 0, \ldots, n-1$.

Recall the definition of the discrete Fourier transform (DFT) as multiplication of a vector with the Fourier matrix $\mathbf{F}_n$:

Definition [Lecture → Def. 4.2.1.18].

The linear map $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathrm{DFT}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform** (DFT), i.e. for $[c_0, \ldots, c_{n-1}] := \mathrm{DFT}_n(\mathbf{y})$

$$c_k := \sum_{j=0}^{n-1} y_j \exp\left(-2\pi\imath \frac{kj}{n}\right), \quad k = 0, \ldots, n-1 . \qquad [\text{Lecture} \rightarrow (4.2.1.19)]$$

(4-7.a) ⚀ (30 min.) Taking the cue from (4.7.2) we can write ("C++ indexing" throughout)

$$\mathbf{f} = (\mathrm{DFT}_{2n}(\mathbf{x}))_{0:n-1} + \mathbf{D}(\mathrm{DFT}_{2n}(\mathbf{y}))_{0:n-1} , \tag{4.7.3}$$

with suitable vectors $\mathbf{x}, \mathbf{y} \in \mathbb{C}^{2n}$ and a diagonal matrix $\mathbf{D} \in \mathbb{C}^{n,n}$. Characterize the vectors $\mathbf{x}, \mathbf{y}$ (in terms of $\mathbf{a}$) and the matrix $\mathbf{D}$.

$$(\mathbf{x})_\ell = \begin{cases} \square & \text{for } \ell \in \{\square\}, \\ \square & \text{for } \ell \in \{\square\}, \end{cases} \quad , \quad (\mathbf{y})_\ell = \begin{cases} \square & \text{for } \ell \in \{\square\}, \\ \square & \text{for } \ell \in \{\square\}, \end{cases} \quad ,$$

$$(\mathbf{D})_{k,k} = \square , \quad k = 0, \ldots, n-1 .$$

SOLUTION for (4-7.a) → 4-7-1-0:mcts1.pdf ▲

(4-7.b) ⚄ (30 min.) The C++ function `modcostrf()` realizes the mapping $\mathbf{a} \to \mathbf{f}$ according to (4.7.1). Supplement the missing parts of the following listing by writing valid C++ code in the boxes.

```
Eigen::VectorXd modcostrf(const Eigen::VectorXd &a) {
  using Comp = std::complex<double>;
  unsigned int n = a.size();
```

```
  Eigen::VectorXd f(n);
  Eigen::FFT<double> fft;
  Eigen::VectorXcd tmp(          );
  tmp << ____________ , Eigen::VectorXcd::Zero(n);
  Eigen::VectorXcd v1 = fft.____________;
  tmp << 0.5 * a.reverse(), ____________________________;
  Eigen::VectorXcd v2 = fft.____________;
  Comp fac = std::exp(((n - 1) * M_PI * Comp(0, 1)) / (double)n);
  Comp d(1.0, 0.0);
  for (int k = 0; k < n; ++k) {
    f[k] = (v1[      ] + ________ * v2[      ]).real();
    ______ *= ______;
  }
  return f;
}
```

The constant `M_PI` is the number π.

HIDDEN HINT 1 for (4-7.b) → 4-7-2-0:mcth1.pdf

HIDDEN HINT 2 for (4-7.b) → 4-7-2-1:mcth2.pdf

SOLUTION for (4-7.b) → 4-7-2-2:.pdf ▲

End Problem 4-7 , 60 min.

Problem 4-8: Discrete Fourier Transform and Applications

The discrete Fourier transform (DFT) is a fundamental tool is signal theory and processing. This problem studies some of its properties and the connection with periodic convolution.

This problem is based upon [Lecture → Section 4.2.1] and [Lecture → Section 4.2.2].

Recall the discrete Fourier transform (DFT)

Definition [Lecture → Def. 4.2.1.18]. discrete Fourier transform

The linear map $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$, $\mathrm{DFT}_n(\mathbf{y}) := \mathbf{F}_n\mathbf{y}$, $\mathbf{y} \in \mathbb{C}^n$, is called **discrete Fourier transform** (DFT), i.e. for $[c_0, \ldots, c_{n-1}] := \mathrm{DFT}_n(\mathbf{y})$

$$c_k = \sum_{j=0}^{n-1} y_j \omega_n^{kj} = \sum_{j=0}^{n-1} y_j \exp\left(-2\pi\imath \frac{kj}{n}\right) \quad , \quad k = 0, \ldots, n-1 \, . \qquad [\text{Lecture} \rightarrow (4.2.1.19)]$$

Here $\mathbf{F}_n$ is the Fourier matrix

$$\mathbf{F}_n = \left[\exp\left(-2\pi\imath \frac{\ell j}{n}\right)\right]_{\ell,j=0}^{n} \in \mathbb{C}^{n,n} \, , \quad n \in \mathbb{N} \, .$$

about which we known

Lemma [Lecture → Lemma 4.2.1.14]. Properties of Fourier matrices

The scaled Fourier-matrix $\frac{1}{\sqrt{n}}\mathbf{F}_n$ is unitary (→ [Lecture → Def. 6.3.1.2]) : $\mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^{\mathrm{H}} = \frac{1}{n}\overline{\mathbf{F}}_n$.

In EIGEN the discrete Fourier transform of a vector is available through the helper class **Eigen::FFT** and its member function `fwd()`. The inverse DFT can be carried out by calling the member function `inv()`, see [Lecture → Code 4.2.1.22].

(4-8.a) ⚂ (10 min.) Code 4.8.1 displays the incomplete listing of a C++ function

```
Eigen::VectorXcd inverseDFT(const Eigen::VectorXcd &c);
```

that implements `Eigen::FFT::inv()` **only** based on calls to `Eigen::FFT::fwd()`. Fill in the missing code parts so that the implementation conforms with the specification.

C++ code 4.8.1: Function `inverseDFT()`

```
1 Eigen::VectorXcd inverseDFT(const Eigen::VectorXcd &c) {
2   const Eigen::VectorXcd::Index n = c.size();
3   Eigen::FFT<double> fft;
4   Eigen::VectorXcd tmp = ;
5   Eigen::VectorXcd y = fft.fwd(tmp);
6   return ;
7 }
```

HIDDEN HINT 1 for (4-8.a) → `4-8-1-0:fftmha1.pdf`

SOLUTION for (4-8.a) → `4-8-1-1:fftmas.pdf` ▲

(4-8.b) ⚄ (15 min.) Based on

Lemma [Lecture → Lemma 4.2.1.16]. Diagonalization of circulant matrices

For any circulant matrix $\mathbf{C} \in \mathbb{K}^{n,n}$, $c_{ij} = u_{i-j}$, $(u_k)_{k\in\mathbb{Z}}$ an n-periodic sequence, holds true

$$\mathbf{C}\overline{\mathbf{F}}_n = \overline{\mathbf{F}}_n \operatorname{diag}(d_0,\ldots,d_{n-1}) \quad , \quad [d_0,\ldots,d_{n-1}]^\top = \mathbf{F}_n[u_0,\ldots,u_{n-1}]^\top .$$

the C++ function

```
Eigen::VectorXcd circmatvec(const Eigen::VectorXcd &u,
const Eigen::VectorXcd &x);
```

realizes the left-multiplication of the n-vector passed in `x` with a circulant $n \times n$-matrix defined by the n-vector `u` (periodically extended to an n-periodic sequence (u_n)).

Complete the listing given in Code 4.8.3.

C++ code 4.8.3: Function `circmatvec()`

```
1 Eigen::VectorXcd circmatvec(const Eigen::VectorXcd &u,
2 const Eigen::VectorXcd &x) {
3   Eigen::FFT<double> fft;
4   Eigen::VectorXcd tmp = (fft.      (      )).
5   cwiseProduct(fft.      (      ));
6   return fft.      (tmp);
7 }
```

SOLUTION for (4-8.b) → 4-8-2-0:fftm1.pdf ▲

End Problem 4-8, 25 min.

Problem 4-9: SVD of circulant matrices

Circulant matrices play an important role in signal processing and in efficient algorithms for finite-difference methods. This problem studies how the singular-value decomposition of a general complex circulant matrix can be computed efficiently.

This problem is based on [Lecture → Section 4.2.1]

In this problem we rely on a slightly generalized notion of a singular value decomposition (SVD):

Definition 4.9.1. Proto-SVD

For $\mathbf{C} \in \mathbb{C}^{n,n}$, $n \in \mathbb{N}$, the matrix factorization $\mathbf{C} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{H}}$ is called a proto-singular-value decomposition (proto-SVD) of $\mathbf{C}$, if

(i) $\mathbf{D} \in \mathbb{R}^{n,n}$ is a *diagonal matrix* with real, non-negative diagonal entries, and

(ii) $\mathbf{U} \in \mathbb{C}^{n,n}$ and $\mathbf{V} \in \mathbb{C}^{n,n}$ are *unitary matrices* [Lecture → Def. 6.3.1.2].

Note that in contrast to the classical SVD from [Lecture → Def. 3.4.1.3] a proto-SVD does not impose any ordering on the diagonal elements of $\mathbf{D}$.

Next recall that the vector $\mathbf{u} \in \mathbb{C}^n$, $n \in \mathbb{N}$, induces a circulant matrix [Lecture → Def. 4.1.4.12] $\mathbf{C} = \operatorname{circul}(\mathbf{u})$ according to (C++ indexing!)

$$(\mathbf{C})_{\ell,k} = (\mathbf{u})_{\ell-k \mod n}, \quad \ell,k \in \{0,\ldots,n-1\}. \tag{4.9.2}$$

(4-9.a) ⚄ (45 min.) Implement an *efficient* (that is, enjoying *optimal asymptotic complexity*) C++ function

```
std::tuple<Eigen::MatrixXcd, Eigen::VectorXd, Eigen::MatrixXcd>
psvdCirculant(const Eigen::VectorXcd &u);
```

that computes a proto-SVD $\mathbf{C} = \mathbf{U}\mathbf{D}\mathbf{V}^{\mathrm{H}}$ of $C := \operatorname{circul}(\mathbf{u}) \in \mathbb{C}^{n,n}$, $\mathbf{u} \in \mathbb{C}^n$ passed in `u`, and returns the 3-tuple $(\mathbf{U}, \mathbf{d}, \mathbf{V})$, where $\mathbf{d} \in \mathbb{R}^n$ is a vector containing the diagonal entries of $\mathbf{D}$.

HIDDEN HINT 1 for (4-9.a) → `4-9-1-0:svdcah1.pdf`

HIDDEN HINT 2 for (4-9.a) → `4-9-1-1:svdcah2.pdf`

SOLUTION for (4-9.a) → `4-9-1-2:.pdf` ▲

(4-9.b) ⚂ (15 min.) In the file `svdcirculant.hpp` complete the code of the C++ function

```
bool testSvdCirculant(const Eigen::VectorXcd &u,
                      double tol = 1.0E-10);
```

which checks whether `psvdCirculant()` called with the argument `u` really delivers a proto-SVD of $\operatorname{circul}(\mathbf{u})$ according to Def. 4.9.1. Make wise use of the `tol` argument.

HIDDEN HINT 1 for (4-9.b) → `4-9-2-0:psvdcbh1.pdf`

SOLUTION for (4-9.b) → `4-9-2-1:.pdf` ▲

End Problem 4-9, 60 min.

Problem 4-10: Computing with sums of random variables

Convolutions play an important role in computational probability, because they are the right tool to obtain the distributions of sums of random variables. This problem studies the use of fast numerical algorithms in this context.

This problem draws on [Lecture → Section 4.1.3], [Lecture → Section 4.1.4], and [Lecture → Section 4.2].

Let $X_1, \ldots, X_N$ be a collection of $N \in \mathbb{N}$ *independent*, identically distributed (i.i.d.) discrete random variables $X_j : \Omega \to \mathbb{Z}$ defined on some probability space Ω. Their distribution is given by a vector $\mathbf{p} \in [0,1]^n$, $n \in \mathbb{N}$, with $\sum_{\ell=0}^{n-1} (\mathbf{p})_\ell = 1$ (C++ indexing!) according to

$$\mathbb{P}(X_j = k) = \begin{cases} 0 & \text{, if } \quad k < 0 \quad \text{or} \quad k \geq n \,, \\ (\mathbf{p})_k \in [0,1] & \text{, if } \quad k \in \{0, \ldots, n-1\} \,, \end{cases} \qquad k \in \mathbb{Z} \,, \tag{4.10.1}$$

where we write $\mathbb{P}$ for the probability of an event.

The sum of the X_js is the random variable

$$S := \sum_{j=1}^{N} X_j \,. \tag{4.10.2}$$

The following result of probability theory can be used.

Theorem 4.10.3. Distribution of sum of two independent discrete random variables

Let Ω be a probability space and $X : \Omega \to \mathbb{Z}$, $Y : \Omega \to \mathbb{Z}$ two independent discrete random variables. Then their sum is distributed according to

$$\mathbb{P}(X + Y = k) = \sum_{\ell \in \mathbb{Z}} \mathbb{P}(X = \ell) \cdot \mathbb{P}(Y = k - \ell) \,, \quad k \in \mathbb{Z} \,,$$

which is a convolution of the distributions of X and Y.

(4-10.a) ⚀ (15 min.) We write $*$ for the discrete convolution of two bi-infinite sequences [Lecture → Def. 4.1.2.7] and $*_M$ for the periodic convolution [Lecture → Def. 4.1.4.7] of two M-periodic bi-infinite sequences, $M \in \mathbb{N}$.

Given $n \in \mathbb{N}$ and any sequence $(p_\ell)_{\ell \in \mathbb{Z}}$ satisfying

$$p_\ell = 0 \quad \text{for} \quad \ell < 0 \text{ or } \ell \geq n \,, \quad \text{that is} \quad (p_\ell)_{\ell \in \mathbb{Z}} = (\ldots, 0, p_0, p_1, \ldots, p_{n-1}, 0, \ldots) \,, \tag{4.10.4}$$

find the *minimal* $M \in \mathbb{N}$ and an M-*periodic* sequence $(q_\ell)_{\ell \in \mathbb{Z}}$ such that the $N-1$-fold discrete convolution of $(p)_\ell$ with itself, $N \in \mathbb{N}$, can be expressed through an $N-1$-fold periodic discrete convolution of $(q_\ell)_{\ell \in \mathbb{Z}}$:

$$\underbrace{\big((p_\ell) * (p_\ell) * \cdots * (p_\ell)\big)_k}_{N-1 \text{ convolutions}} = \underbrace{\big((q_\ell) *_M (q_\ell) *_M \cdots *_M (q_\ell)\big)_k}_{N-1 \text{ periodic convolutions}} \,. \tag{4.10.5}$$

for all indices k, for which a term of the $N-1$-fold convolution $(p_\ell) * (p_\ell) * \cdots * (p_\ell)$ need not vanish.

$M =$,

$q_k =$, $k = 0, \ldots, M-1$.

HIDDEN HINT 1 for (4-10.a) → 4-10-1-0:srvcassoc.pdf

HIDDEN HINT 2 for (4-10.a) → 4-10-1-1:srvah1.pdf

SOLUTION for (4-10.a) → 4-10-1-2:srv0s.pdf ▲

(4-10.b) ⚄ (45 min.) [depends on Sub-problem (4-10.a)]

In the file `sumrandomvariables.hpp` implement an *efficient* C++ function

```
Eigen::VectorXd probDistSum(const Eigen::VectorXd &p, unsigned int
    N);
```

which returns a vector $\mathbf{s}$ containing the probability distribution of the sum random variable $S := \sum_{i=1}^{N} X_i$, that is

$$\mathbb{P}(S = k) = (\mathbf{s})_k \,, \quad k \in \{0, \ldots, L\} \,,$$

where L is the largest possible value that S can attain. The argument `p` passes the vector $\mathbf{p} \in [0,1]^n$, $\mathbf{1}^\top \mathbf{p} = 1$, describing the probability distribution of the X_is according to (4.10.1).

HIDDEN HINT 1 for (4-10.b) → 4-10-2-0:srvM.pdf

HIDDEN HINT 2 for (4-10.b) → 4-10-2-1:srvah2.pdf

SOLUTION for (4-10.b) → 4-10-2-2:srvas.pdf ▲

End Problem 4-10 , 60 min.

Problem 4-11: DFT and FFT: A Repetition

This problem is meant for the active recall of different aspects of the **discrete Fourier transform** (DFT).

This problems draws on material covered in [Lecture → Section 4.1], [Lecture → Section 4.2], and [Lecture → Section 4.3]

(4-11.a) ⚀ (10 min.) The **discrete Fourier transform** (DFT) $\mathrm{DFT}_n : \mathbb{C}^n \mapsto \mathbb{C}^n$ of length $n \in \mathbb{N}$ can be defined as multiplication of a vector with the **Fourier matrix** $\mathbf{F}_n \in \mathbb{C}^{n,n}$,

$$\mathrm{DFT}_n(\mathbf{y}) := \mathbf{F}_n \mathbf{y} , \quad \mathbf{y} \in \mathbb{C}^n , \tag{4.11.1}$$

$$\mathbf{F}_n := \begin{bmatrix} \omega_n^0 & \omega_n^0 & \cdots & \omega_n^0 \\ \omega_n^0 & \omega_n^1 & \cdots & \omega_n^{n-1} \\ \omega_n^0 & \omega_n^2 & \cdots & \omega_n^{2n-2} \\ \vdots & \vdots & & \vdots \\ \omega_n^0 & \omega_n^{n-1} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix} = \left[\omega_n^{\ell j}\right]_{\ell,j=0}^{n-1} , \quad \omega_n := \exp(-2\pi\imath/n) . \tag{4.11.2}$$

Complete the sum formula for DFT using C++ indexing:

$$(\mathrm{DFT}_n(\mathbf{y}))_k = \sum_{j=\square}^{\square} (\mathbf{y})_j \exp(\square) , \quad k = 0, \ldots, n-1 .$$

SOLUTION for (4-11.a) → 4-11-1-0:.pdf ▲

(4-11.b) ⚁ (10 min.) We consider time-discrete signals sampled at equidistant points in time $t_k := k\Delta t$, $\Delta t > 0$, $k \in \mathbb{Z}$. Assume that $(\ldots, 0, h_0, h_1, \ldots, h_{m-1}, 0, \ldots)$, $h_j \neq 0$ for $j \in \{0, \ldots, m-1\}$, is the impulse response [Lecture → Def. 4.1.1.12] of a causal, finite, linear, time-invariant filter. We feed it a signal of total finite duration $n\Delta t$, $m \in \mathbb{N}$. What will be the total duration $X > 0$ of the output signal?

$$X = \square .$$

SOLUTION for (4-11.b) → 4-11-2-0:dfts1.pdf ▲

(4-11.c) ⚁ (10 min.) Recall the discrete convolution of two vectors (C++ indexing used throughout):

Definition 4.11.3. Discrete convolution of vectors

Given $\mathbf{x} = [x_0, \ldots, x_{m-1}]^\top \in \mathbb{K}^m$, $\mathbf{h} = [h_0, \ldots, h_{n-1}]^\top \in \mathbb{K}^n$ their **discrete convolution** $\mathbf{h} * \mathbf{x}$ is the vector $\mathbf{y} = [y_0, \ldots, y_{m+n-2}]^\top \in \mathbb{K}^{m+n-1}$ with components

$$y_k = \sum_{j=0}^{m-1} h_{k-j} x_j , \quad k = 0, \ldots, m+n-2 , \tag{4.11.4}$$

where we have adopted the convention $h_j := 0$ for $j < 0$ or $j \geq n$.

Given $\mathbf{h} \in \mathbb{C}^n$, $\mathbf{x} \in \mathbb{C}^m$, and writing $*_\ell$ for the ℓ-periodic convolution of sequences, determine the *minimal* vector length $N \in \mathbb{N}$ such that

$$\mathbf{h} * \mathbf{x} = \left(\widetilde{\mathbf{h}} *_N \widetilde{\mathbf{x}}\right)_{0:m+n-2} ,$$

where $\widetilde{\mathbf{h}} = \begin{bmatrix} \mathbf{h} \\ \mathbf{0} \end{bmatrix} \in \mathbb{C}^N$, $\widetilde{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \mathbf{0} \end{bmatrix} \in \mathbb{C}^N$ are *zero-padded* vectors.

$$N = \boxed{} ,$$

SOLUTION for (4-11.c) → 4-11-3-0:dfts2.pdf ▲

(4-11.d) ⚄ (20 min.) According to [Lecture → Section 4.2.5] the **two-dimensional discrete Fourier transform** (2D DFT) of the matrix $\mathbf{Y} \in \mathbb{C}^{m,n}$ is defined as

$$\mathrm{DFT}_{m,n}(\mathbf{Y}) := \sum_{j_1=0}^{m-1} \sum_{j_2=0}^{n-1} (\mathbf{Y})_{j_1,j_2} (\mathbf{F}_m)_{:,j_1} (\mathbf{F}_n)_{:,j_2}^\top \quad \Leftrightarrow \quad \boxed{\mathrm{DFT}_{m,n}(\mathbf{Y}) = \mathbf{F}_m (\mathbf{F}_n \mathbf{Y}^\top)^\top = \mathbf{F}_m \mathbf{Y} \mathbf{F}_n} .$$

[Lecture → (4.2.5.5)]

The following C++ function is supposed to compute *efficiently* and for any $n \in \mathbb{N}$ and $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$ the 2D DFT

$$\mathrm{DFT}_{n,n}(\mathbf{Y}) \quad \text{for} \quad \mathbf{Y} = [(\mathbf{u})_k (\mathbf{v})_\ell]_{k,\ell=1}^n = \mathbf{u}\mathbf{v}^\top \in \mathbb{C}^{n,n} .$$

```
1 Eigen::MatrixXcd dft2dlr(const Eigen::VectorXcd &u, const Eigen::VectorXcd &v) {
2   Eigen::Index n = u.size();
3   assertm((n == v.size()), "Vector legnths must agree");
4   // Object for carrying out DFT via the fwd() method
5   Eigen::FFT fft;
6   return                                                          ;
7 }
```

SOLUTION for (4-11.d) → 4-11-4-0:dfts3.pdf ▲

(4-11.e) ⚂ (10 min.) [depends on Sub-problem (4-11.d)]

What is the asymptotic complexity of your completed implementation of the C++ function `dft2dlr()` from Sub-problem (4-11.d) in terms of vector length $n \to \infty$?

$$\mathrm{cost}(\texttt{dft2dlr()}) = O\left(\boxed{}\right) \quad \text{for} \quad n \to \infty .$$

SOLUTION for (4-11.e) → 4-11-5-0:.pdf ▲

End Problem 4-11 , 60 min.

Chapter 5

Data Interpolation and Data Fitting in 1D

Problem 5-1: Evaluating the derivatives of interpolating polynomials

In [Lecture → Ex. 5.1.0.8] we learned about the importance of data interpolation for obtaining functor representations of constitutive relationships $t \mapsto f(t)$. Numerical methods like Newton's method [Lecture → Code 8.4.2.2] often require information about the derivative f' as well. Therfore, we need efficient ways to evaluate the derivatives of interpolants. In this problem we discuss this issue for polynomial interpolation for (i) monomial representation and (ii) "update-friendly" point evaluation. We generalize the Horner scheme [Lecture → § 5.2.1.5] and the Aitken-Neville algorithm [Lecture → § 5.2.3.8].

This problem is connected with [Lecture → Section 5.2.3], simple coding in C++/EIGEN is requested.

We first deal with polynomials in monomial representation [Lecture → Rem. 5.2.1.4].

(5-1.a) ⚄ (20 min.) Using the Horner scheme [Lecture → § 5.2.1.5], in the file `eval_deriv.hpp` write an efficient C++ implementation of a template function which returns the pair $(p(x), p'(x))$, where p is a polynomial with coefficients in `c`:

```
std::pair<double,double> evaldp(const CoeffVec &c, double x);
```

Vector `c` contains the coefficient of the polynomial in the monomial basis, following the PYTHON/MATLAB convention (leading coefficient in `c[0]`), that is,

$$p(t) = \texttt{c[0]}\,t^k + \texttt{c[1]}\,t^{k-1} + \cdots + \texttt{c[k-1]}\,t + \texttt{c[k]}\,, \quad t \in \mathbb{R}\,. \tag{5.1.1}$$

As indicated above, the type **CoeffVec** must provide component access via **operator** `[](int) const` and a `size()` method.

SOLUTION for (5-1.a) → `5-1-1-0:Horner1s.pdf` ▲

(5-1.b) ⚄ (15 min.) For the sake of testing, write a naive C++ implementation of the evaluation requested in (5-1.a)

```
std::pair<double,double> evaldp_naive(const CoeffVec &c, double x);
```

which returns the same pair $(p(x), p'(x))$. However, this time $p(x)$ and $p'(x)$ should be calculated by means of simply summing the contributions of the monomials $t \mapsto t^k$.

SOLUTION for (5-1.b) → `5-1-2-0:Horner2s.pdf` ▲

(5-1.c) ⚀ (5 min.) [depends on Sub-problem (5-1.a), Sub-problem (5-1.b)]

What are the asymptotic complexities of the two functions you implemented in (5-1.a) and (5-1.b) in terms of raising the polynomial degree?

SOLUTION for (5-1.c) → 5-1-3-0:Horner3s.pdf ▲

(5-1.d) ⚁ (15 min.) [depends on Sub-problem (5-1.a), Sub-problem (5-1.b)]

Implement a function

```
bool polyTestTime(const unsigned int d);
```

that

(i) validates your implementation of the two functions `evaldp()`/`evaldp_naive()` from (5-1.a)/(5-1.b) by comparing their results for $x = 1.23$ and a polynomial of degree n with monomial coefficients $c_0 = 1, c_1 = 2, \ldots, c_n = n$ and $n = 2, 4, 8, \ldots, 2^d$. Return **false** in case of a discrepancy.

(ii) measures the runtimes of the two functions for every degree and tabulates them. To that end use the **Timer** class defined in `timer.h`:

```
Timer t;
t.start(); /* do something */ t.stop();
double time = t.duration();
```

HIDDEN HINT 1 for (5-1.d) → 5-1-4-0:h4h1.pdf

SOLUTION for (5-1.d) → 5-1-4-1:Horner4s.pdf ▲

Now we revisit polynomial interpolation. In [Lecture → Section 5.2.3.2] we learned about an efficient and "update-friendly" scheme for evaluating Lagrange interpolants at a *single or a few* points. This so-called **Aitken-Neville algorithm**, see [Lecture → Code 5.2.3.10], can be extended to return the derivative value of the polynomial interpolant as well. This will be explored next.

(5-1.e) ⚂ (30 min.) In the file `eval_deriv.hpp` write an efficient C++ function

```
Eigen::VectorXd dipoleval(const Eigen::VectorXd & t,
                          const Eigen::VectorXd & y,
                          const Eigen::VectorXd & x);
```

that returns the row vector $[p'(x_1), \ldots, p'(x_m)]^\top$, when the argument `x` passes $[x_1, \ldots, x_m]$, $m \in \mathbb{N}$ small. Here, p' denotes the *derivative* of the polynomial $p \in \mathcal{P}_n$ interpolating the data points (t_i, y_i), $i = 0, \ldots, n$, for pairwise different $t_i \in \mathbb{R}$ and data values $y_i \in \mathbb{R}$.

To that end differentiate the recursion formula [Lecture → (5.2.3.9)] and devise an algorithm in the spirit of the Aitken-Neville algorithm implemented in [Lecture → Code 5.2.3.10].

HIDDEN HINT 1 for (5-1.e) → 5-1-5-0:hornxh1.pdf

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

(5-1.f) ⚂ (30 min.) For validation purposes devise an alternative, less efficient, implementation of `dipoleval()`

```
Eigen::VectorXd dipoleval_alt(const Eigen::VectorXd &t,
                              const Eigen::VectorXd &y,
                              const Eigen::VectorXd &x);
```

based on the following algorithm:

1. Use the function

```
Eigen::VectorXd polyfit(const Eigen::VectorXd& t,
                        const Eigen::VectorXd& y,
                        unsigned int degree);
```

(supplied in `polyfit.hpp`) to compute the monomial coefficients of the Lagrange interpolant of the data points.

2. Compute the monomial coefficients of the derivative.

3. Use the function

```
void polyval(const Eigen::VectorXd& p,
             const Eigen::VectorXd& x,
             Eigen::VectorXd& y);
```

(defined in `polyval.hpp`) to evaluate the derivative at a number of points. This function evaluates the polynomial with monomial coefficients passed in `p` at all points in `x`, *cf.* [Lecture → Code 5.2.1.7].

Use `dipoleval_alt()` to verify the correctness of your implementation of `dipoleval()` by computing the derivative of the degree-10 interpolating polynomial for the values $(t_i, sin(t_i))$, where t_i are equidistant points in $[0,3]$. This test should be conducted within the function

```
bool testDipolEval(void);
```

Return **false** in case the test fails, **true** otherwise.

SOLUTION for (5-1.f) → 5-1-6-0:test.pdf

▲

(5-1.g) ⚀ (20 min.) Based on your version of `dipoleval()` write a C++ function

```
void plotPolyInterpolant(const std::string &filename);
```

that saves plots the derivative of the polynomial interpolant for the test data as specified in Subproblem (5-1.f) in the file `<filename>.png`. To that end evaluate the polynomial in 100 equidistant points in the interval $[0,3]$ and, based on the obtained value, use the `plot()` function of MATPLOTLIBCPP to draw the polynomials. Do not forget to add axis labels "t" and "$p'(t)$", and a title.

HIDDEN HINT 1 for (5-1.g) → 5-1-7-0:hx1.pdf

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

End Problem 5-1, 135 min.

Problem 5-2: Lagrange Polynomials

This short problem studies a particular aspect of the **Lagrange polynomials** introduced in [Lecture → § 5.2.2.3].

This exercise just requires basic knowledge of [Lecture → Section 5.2.2]

As in [Lecture → (5.2.2.4)] we denote by L_i the i-th Lagrange polynomial for given nodes $t_j \in \mathbb{R}$, $j = 0,\ldots,n$, $t_i \neq t_j$, if $i \neq j$. Then the polynomial Lagrange interpolant p through the data points $(t_i,y_i)_{i=0}^n$ has the representation

$$p(x) = \sum_{i=0}^{n} y_i L_i(x) \quad \text{with} \quad L_i(x) := \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-t_j}{t_i-t_j} \,. \tag{5.2.1}$$

(5-2.a) ⚀ (10 min.) Show that

$$\sum_{i=0}^{n} L_i(t) = 1 \quad \forall t \in \mathbb{R} \,. \tag{5.2.2}$$

HIDDEN HINT 1 for (5-2.a) → 5-2-1-0:glgh1.pdf

SOLUTION for (5-2.a) → 5-2-1-1:glgs1.pdf ▲

(5-2.b) ⚁ (15 min.) Show that

$$\sum_{i=0}^{n} L_i(0) t_i^j = \begin{cases} 1 & \text{for } j = 0 \,, \\ 0 & \text{for } j = 1,\ldots,n \,. \end{cases} \tag{5.2.3}$$

(Here, read t_i^j as t_i raised to the power j.)

HIDDEN HINT 1 for (5-2.b) → 5-2-2-0:glgh1.pdf

SOLUTION for (5-2.b) → 5-2-2-1:glgs1.pdf ▲

(5-2.c) ⚂ (15 min.) Show that $p(x)$ can also be written as

$$p(x) = \omega(x) \sum_{i=0}^{n} \frac{y_i}{(x-t_i)\omega'(t_i)} \quad \text{with} \quad \omega(t) := \prod_{j=0}^{n} (t-t_j) \,. \tag{5.2.4}$$

Here ω' is the derivative of ω.

HIDDEN HINT 1 for (5-2.c) → 5-2-3-0:LagrangePoly1h.pdf

SOLUTION for (5-2.c) → 5-2-3-1:LagrangePoly1s.pdf ▲

(5-2.d) ⚀ (10 min.) The **leading coefficient** of an uni-variate polynomial p is the coefficient multiplying the highest power of the independent variable. What is the leading coefficient of the Lagrange polynomial L_i as defined above?

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

(5-2.e) ⚂ (10 min.) [depends on Sub-problem (5-2.d)]

Recall from [Lecture → § 5.2.3.26] that the **divided difference** $y[t_0,\ldots,t_n]$ for a sequence $(t_0,y_0),\ldots,(t_n,y_n)$ of $n+1$ data points $\in \mathbb{R}^2$ is defined as the leading coefficient of the (unique)

Lagrange interpolation polynomial of degree n [Lecture → Thm. 5.2.2.7] through those data points. Show that

$$y[t_0,\ldots,t_n] = \sum_{i=0}^{n} \frac{y_i}{\prod_{\substack{j=0\\ j\neq i}}^{n}(t_i - t_j)} \,. \tag{5.2.6}$$

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

(5-2.f) ⚂ (10 min.) Based on (5.2.6) show that changing ordering of the interpolation points does not affect the value of the associated divided differences.

SOLUTION for (5-2.f) → 5-2-6-0:.pdf ▲

End Problem 5-2 , 70 min.

Problem 5-3: Generalized Lagrange polynomials for Hermite interpolation

[Lecture → Rem. 5.2.2.14] addresses a particular generalization of the Lagrange polynomial interpolation considered in [Lecture → Section 5.2.2]: **Hermite interpolation**, where beside the usual "nodal" interpolation conditions also the derivative of the interpolating polynomials is prescribed in the nodes, see [Lecture → (5.2.2.15)] (for $l_j = 1$ throughout). The associated generalized Lagrange polynomials were defined in [Lecture → Def. 5.2.2.17] and in this problem we aim to find concrete formulas for them in the spirit of [Lecture → (5.2.2.4)].

This is a purely theoretical problem.

Let $n+1$ different nodes t_i, $i = 0,\ldots,n$, $n \in \mathbb{N}$, be given. For numbers $y_j, c_j \in \mathbb{R}$, $j = 0,\ldots,n$, Hermite interpolation seeks a polynomial $p \in \mathcal{P}_{2n+1}$ satisfying $p(t_i) = y_i$ and $p'(t_i) = c_i$, $i = 0,\ldots,n$. Here p' designates the derivative of p.

The **generalized Lagrange polynomials** for Hermite interpolation $L_i, K_i \in \mathcal{P}_{2n+1}$ satisfy

$$L_i(t_j) = \delta_{ij}\,, \qquad L_i'(t_j) = 0\,, \tag{5.3.1a}$$

$$K_i(t_j) = 0\,, \qquad K_i'(t_j) = \delta_{ij}\,, \tag{5.3.1b}$$

for $i, j \in \{0,\ldots,n\}$.

(5-3.a) ⚀ (15 min.) Explicitly state the linear system of equations for the basis expansion coefficients of the Hermite interpolant with respect to the *monomial basis* $\{x \mapsto x^k,\ k = 0,\ldots,2n+1\}$ of $\mathcal{P}_{2n+1}$.

HIDDEN HINT 1 for (5-3.a) → 5-3-1-0:gagh0.pdf

SOLUTION for (5-3.a) → 5-3-1-1:glgs1.pdf ▲

(5-3.b) ⚁ (15 min.) [depends on Sub-problem (5-3.a)]

Give concrete formulas for K_0, K_1, L_0, L_1 for $n = 1$, $t_0 = -1$, $t_1 = 1$.

HIDDEN HINT 1 for (5-3.b) → 5-3-2-0:glgh1.pdf

,

SOLUTION for (5-3.b) → 5-3-2-1:glgs1.pdfNote that $L_0(t) = L_1(-t)$, $K_0(t) = K_1(-t)$, which reflects the symmetry of the interval $[-1,1]$. ▲

(5-3.c) ⚁ (10 min.) Find the general formula describing all polynomials in t

1. of degree 2 that have a double zero in $t = a$, $a \in \mathbb{R}$,
2. of degree $n > 2$ that have a double zero in $t = a$, $a \in \mathbb{R}$.

A polynomial p is said to have a double zero in $t = a$, if $p(a) = 0$ *and* $p'(a) = 0$.

SOLUTION for (5-3.c) → 5-3-3-0:glgs1.pdf ▲

(5-3.d) ⚁ (15 min.) Let $g_1,\ldots,g_m$, $m \in \mathbb{N}$, be functions in $C^1(I)$ (space of continuously differentiable functions). Find a formula for the derivative of $h(t) = \prod_{k=1}^{m} g_k(t)$.

SOLUTION for (5-3.d) → 5-3-4-0:glgs1.pdf

▲

(5-3.e) ⚂ (30 min.) [depends on Sub-problem (5-3.c), Sub-problem (5-3.d)]

Find general formulas for the polynomials L_i, K_i as defined through (5.3.1). Of course, these formulas will involve the nodes t_i.

HIDDEN HINT 1 for (5-3.e) → 5-3-5-0:glgh1.pdf

HIDDEN HINT 2 for (5-3.e) → 5-3-5-1:glghx.pdf

SOLUTION for (5-3.e) → 5-3-5-2:glgs1.pdf

▲

End Problem 5-3 , 85 min.

Problem 5-4: Piecewise linear interpolation

[Lecture → Ex. 5.1.0.15] introduced piecewise linear interpolation as a simple linear interpolation scheme. It finds an interpolant in the space spanned by the so-called tent functions, which are *cardinal basis functions*. Formulas are given in [Lecture → (5.1.0.17)].

Study [Lecture → Ex. 5.1.0.15] before tackling this problem. Problem involves coding in C++.

(5-4.a) ⚄ (20 min.)

Realize a C++ class

```
class LinearInterpolant {
public:
  LinearInterpolant(const Eigen::VectorXd &t, const
     Eigen::VectorXd &y);
  double operator(double x) const;
private:
  // Necessary data members here
};
```

representing the piecewise linear interpolant of the data points (t_i, y_i) passed to the constructor through two vectors `t` and `y` of equal length. In addition the class should provide the evaluation operator **double operator(double x) const**, which returns the value of the piecewise linear interpolant at $x \in \mathbb{R}$. Pay attention to efficient implementation.

Note that you must not assume that the nodes passed in `t` are sorted.

SOLUTION for (5-4.a) → 5-4-1-0:impl.pdf

▲

(5-4.b) ⚄ (10 min.) Discuss the asymptotic effort involved in the constructor and the evaluation operator of the class **LinearInterpolant** as the number n of data points becomes large.

SOLUTION for (5-4.b) → 5-4-2-0:scost.pdf ▲

End Problem 5-4, 30 min.

Problem 5-5: Cardinal basis for trigonometric interpolation

In [Lecture → Section 5.6] we learned about interpolation into the space $\mathcal{P}_{2n}^T$ or trigonometric polynomials from [Lecture → Def. 5.6.1.1]. In this task we investigate the cardinal basis functions for this interpolation operator and will also obtain a result about its stability.

Requires everything from [Lecture → Section 5.6] and involves coding in C++.

The (ordered) real trigonometric basis of $\mathcal{P}_{2n}^T$ is

$$\mathcal{B}_{\mathrm{Re}} := \Big\{ C_0 := t \mapsto 1, S_1 := t \mapsto \sin(2\pi t), C_1 := t \mapsto \cos(2\pi t), S_2 := t \mapsto \sin(4\pi t), \\ C_2 := t \mapsto \cos(4\pi t), \ldots, S_n := t \mapsto \sin(2n\pi t), C_n := t \mapsto \cos(2n\pi t) \Big\} . \tag{5.5.1}$$

Another (ordered) basis of $\mathcal{P}_{2n}^T$ is

$$\mathcal{B}_{\exp} := \{ B_k := t \mapsto \exp(2\pi k \imath t) : k = -n, \ldots, n \} . \tag{5.5.2}$$

(Strictly speaking, we have to take the real part of these functions.) Both bases have $2n+1$ elements, which agrees with the dimension of $\mathcal{P}_{2n}^T$ [Lecture → Cor. 5.6.1.6].

(5-5.a) ⚀ (15 min.) Give the matrix $\mathbf{S} \in \mathbb{C}^{2n+1,2n+1}$ that effects the transformation of a coefficient representation with respect to $\mathcal{B}_{\mathrm{Re}}$ into a coefficient representation with respect to $\mathcal{B}_{\exp}$.

HIDDEN HINT 1 for (5-5.a) → 5-5-1-0:trh1.pdf

SOLUTION for (5-5.a) → 5-5-1-1:trisol1.pdf ▲

(5-5.b) ⚁ (10 min.) The shift operator $\mathsf{S}_\tau : C^0(\mathbb{R}) \to C^0(\mathbb{R})$ acts on a function according to $\mathsf{S}_\tau f(t) = f(t-\tau)$, for $\tau \in \mathbb{R}$. If $\mathbf{c} \in \mathbb{C}^{2n+1}$ is the coefficient vector of $p \in \mathcal{P}_{2n}^T$ with respect to $\mathcal{B}_{\exp}$, what is the coefficient vector $\tilde{\mathbf{c}} \in \mathbb{C}^{2n+1}$ of $\mathsf{S}_\tau p$?

HIDDEN HINT 1 for (5-5.b) → 5-5-2-0:tch2.pdf

SOLUTION for (5-5.b) → 5-5-2-1:trisol2.pdf ▲

Now we consider the set of interpolation nodes

$$\mathcal{T}_n := \left\{ t_j := \frac{j+1/2}{2n+1} : j = 0, \ldots, 2n \right\} . \tag{5.5.3}$$

The **cardinal basis functions** $b_0, \ldots, b_{2n} \in \mathcal{P}_{2n}^T$ of $\mathcal{P}_{2n}^T$ with respect to $\mathcal{T}_n$ are defined by

$$b_j(t_k) = \delta_{kj} \quad \text{[Kronecker symbol]} , \quad j,k = 0, \ldots, 2n \tag{5.5.4}$$

(5-5.c) ⚂ (30 min.) Use the function `trigpolyvalequid`, see [Lecture → Code 5.6.3.9] to plot the cardinal basis function (implement plot_basis in trigipcardfn.hpp) $t \mapsto b_0(t)$ in $[0,1]$ for $n = 5$. To that end evaluate $b_0(t)$ in 1000 equidistant points $\frac{k}{1000}$, $k = 0, \ldots, 999$.

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

(5-5.d) ⚀ Show that

$$b_{k+1} = \mathsf{S}_\delta b_k \quad \text{with} \quad \delta := \frac{1}{2n+1} . \tag{5.5.6}$$

HIDDEN HINT 1 for (5-5.d) → 5-5-4-0:h1.pdf

SOLUTION for (5-5.d) → 5-5-4-1:trisol1.pdf ▲

(5-5.e) ⚁ (15 min.) Describe the entries of the "interpolation matrix", that is, the system matrix of [Lecture → (5.1.0.23)], for the trigonometric interpolation problem with node set $\mathcal{T}_n$ and with respect to the basis $\mathcal{B}_{\exp}$ of $\mathcal{P}_{2n}^T$.

HIDDEN HINT 1 for (5-5.e) → 5-5-5-0:tklh1.pdf

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

SOLUTION for (5-5.e) → 5-5-5-2:trisol1.pdf ▲

(5-5.f) ⚁ (15 min.) [depends on Sub-problem (5-5.e)]

What is the inverse of the interpolation matrix found in the previous problem?

HIDDEN HINT 1 for (5-5.f) → 5-5-6-0:h1.pdf

SOLUTION for (5-5.f) → 5-5-6-1:trisol1.pdf

▲

(5-5.g) ⚂ (45 min.) Give a closed-form expression for the cardinal basis functions b_k defined in (5.5.4).

HIDDEN HINT 1 for (5-5.g) → 5-5-7-0:h1.pdf

SOLUTION for (5-5.g) → 5-5-7-1:trisol1.pdf ▲

(5-5.h) ⚂ (30 min.) Write a function

```
double trigIpL(std::size_t n);
```

that approximately computes the **Lebesgue constant** $\lambda(n)$, see [Lecture → § 5.2.4.8], for trigonometric interpolation based on the node set $\mathcal{T}_n$. The Lebesgue constant is available through the formula

$$\lambda(n) = \sup_{\mathbf{y}\in\mathbb{C}^{2n+1}\setminus\{\mathbf{0}\}} \frac{\|\mathsf{I}_n\mathbf{y}\|_{L^\infty([0,1])}}{\|\mathbf{y}\|_\infty} = \sup_{t\in[0,1]} \sum_{k=0}^{2n} |b_k(t)| \,, \tag{5.5.11}$$

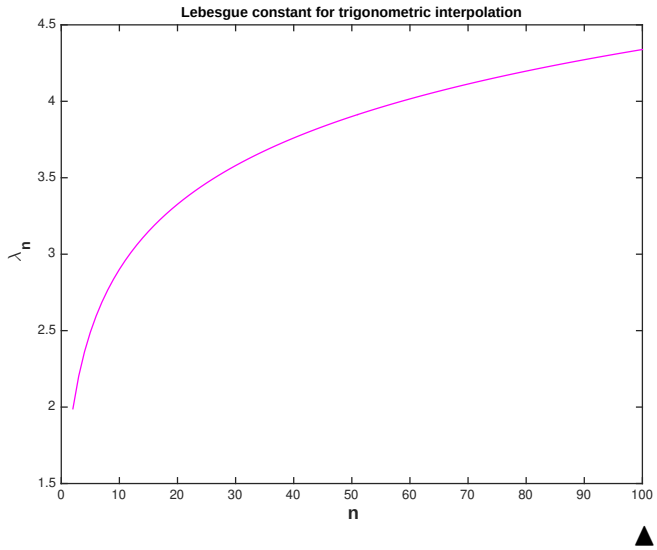
where $\mathsf{I}_n : \mathbb{C}^{2n+1} \to C^0(\mathbb{R})$ is the trigonometric interpolation operator.

Use sampling in 10^4 equidistant points to obtain a good guess for the supremum norm of a function on $[0,1]$.

Note. Efficiency of the implementation need not be a concern in this numerical experiment.

SOLUTION for (5-5.h) → 5-5-8-0:trisol1.pdf

Graph of $n \mapsto \lambda(n)$



▲

(5-5.i) ⚂ (5 min.) The following table gives the values $\lambda(n)$ for $n = 2^k$, $k = 2,3,\ldots,8$.

2^k	$\lambda(2^k)$
4	2.7
8	3.07
16	3.46
32	3.89
64	4.32
128	4.75
256	5.18
512	5.62

From these numbers predict the asymptotic behavior of $\lambda(n)$ for $n \to \infty$. Is it $O(\log n)$, $O(n)$, $O(n\log n)$, or $O(n^2)$?

SOLUTION for (5-5.i) → `5-5-9-0:xtrsol1.pdf` ▲

End Problem 5-5 , 165 min.

Problem 5-6: Quadratic Splines

[Lecture → Def. 5.4.1.1] introduces spline spaces $\mathcal{S}_{d,\mathcal{M}}$ of any degree $d \in \mathbb{N}_0$ on a node set $\mathcal{M} \subset \mathbb{R}$. [Lecture → Section 5.4.2] discusses interpolation by means of cubic splines, which is the most important case. In this problem we practise spline interpolation for quadratic splines in order to understand the general principles. We will see a situation, where knot sets and node sets have to be different.

Familiarity with [Lecture → Section 5.4.2] is required. This exercise involves substantial implementation in C++ and EIGEN.

Consider a 1-periodic function $f : \mathbb{R} \to \mathbb{R}$, that is, $f(t+1) = f(t)$ for all $t \in \mathbb{R}$, and a set of knots

$$\mathcal{M} := \{0 = t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = 1\} \subset [0,1] .$$

We want to approximate f using a *1-periodic* quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$, which interpolates f *in the midpoints* of the intervals $[t_{j-1}, t_j]$, $j = 0, \ldots, n$:

$$s\left(\tfrac{1}{2}(t_{j-1}+t_j)\right) = y_j := f\left(\tfrac{1}{2}(t_{j-1}+t_j)\right) , \quad j = 1, \ldots, n . \tag{5.6.1}$$

In analogy to the local representation of a cubic spline function according to [Lecture → (5.4.2.6)], we locally in each knot interval parameterize a quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$ according to

$$s|_{[t_{j-1},t_j]}(t) = d_j\, \tau^2 + c_j\, 4\, \tau(1-\tau) + d_{j-1}\, (1-\tau)^2 , \quad \tau := \frac{t - t_{j-1}}{t_j - t_{j-1}} , \quad j = 1, \ldots, n , \tag{5.6.2}$$

with $c_j, d_k \in \mathbb{R}$, $j = 1, \ldots, n$, $k = 0, \ldots, n$.

Remark. In this problem we see a case of spline interpolation for which the **knots** t_j, $j = 0, \ldots, n$, do not coincide with the interpolation **nodes**, which are the first coordinates of the data points to be interpolated.

(5-6.a) ⚀ (10 min.) How are the coefficients d_j, c_j, and d_{j-1} in (5.6.2) linked to the values of $s|_{[t_{j-1},t_j]}$ in the points $t = t_{j-1}, t_j, \frac{1}{2}(t_{j-1}+t_j)$?

HIDDEN HINT 1 for (5-6.a) → 5-6-1-0:s0h1.pdf

SOLUTION for (5-6.a) → 5-6-1-1:s0.pdf ▲

(5-6.b) ⚁ (10 min.) What is the dimension of the subspace of *1-periodic* spline functions in $\mathcal{S}_{2,\mathcal{M}}$?

HIDDEN HINT 1 for (5-6.b) → 5-6-2-0:s1h1.pdf

SOLUTION for (5-6.b) → 5-6-2-1:s1.pdf ▲

(5-6.c) ⚂ (10 min.) [depends on Sub-problem (5-6.a)]

What kind of continuity is already guaranteed by the mere use of the representation (5.6.2)?

SOLUTION for (5-6.c) → 5-6-3-0:s2.pdf ▲

(5-6.d) ⚃ (60 min.) Derive a $n \times n$ linear system of equations (system matrix and right hand side) whose solution provides the coefficients c_j, $j = 1, \ldots, n$, in the representation (5.6.2) based on the function values $y_j := f(\frac{1}{2}(t_{j-1}+t_j))$, $j = 1, \ldots, n$.

HIDDEN HINT 1 for (5-6.d) → 5-6-4-0:s3h1.pdf

SOLUTION for (5-6.d) → 5-6-4-1:s3.pdf ▲

(5-6.e) ⚁ (15 min.) [depends on Sub-problem (5-6.d)]

Show that the system matrix found in Sub-problem (5-6.d) can be written as a rank-1 modification of a tridiagonal matrix.

SOLUTION for (5-6.e) → 5-6-5-0:s3a.pdf ▲

(5-6.f) ⚁ (15 min.) [depends on Sub-problem (5-6.d) and Sub-problem (5-6.e)]

Outline an efficient algorithm for the solution of the linear system of equations found in Sub-problem (5-6.d). What is the asymptotic complexity of the algorithm in terms of the number n of data values for $n \to \infty$

HIDDEN HINT 1 for (5-6.f) → 5-6-6-0:qsph6.pdf

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

(5-6.g) ⚂ (45 min.) [depends on Sub-problem (5-6.d)]

Implement a C++ function

```
Eigen::VectorXd compute_c(const Eigen::VectorXd &t,
                          const Eigen::VectorXd &y);
```

that returns the coefficient vector $[c_1,\ldots,c_n]^\top \in \mathbb{R}^n$ when supplied a sorted knot vector `t` (of length $n-1$, because $t_0 = 0$ and $t_n = 1$ will be taken for granted), an n-vector `y` of data values y_j, $j = 1,\ldots,n$, that specify the function values of the quadratic spline s at the midpoints of the knot intervals.

You can immediately use a suitable sparse direct solver provided by EIGEN, which is "smart" enough to solve the linear system (**??**) with optimal computational effort. You need not employ the techniques of [Lecture → § 2.6.0.12].

SOLUTION for (5-6.g) → 5-6-7-0:s3b.pdf ▲

(5-6.h) ⚁ (20 min.) [depends on Sub-problem (5-6.d)]

Realize a C++ function

```
Eigen::VectorXd compute_d ( const Eigen::VectorXd &c, const
   Eigen::VectorXd &t);
```

that determines the coefficients d_j, $j = 0,\ldots,n$, in the local representation (5.6.2) assuming that the values c_j, $j = 1,\ldots,n$, have already been computed. The argument `t` is to pass the variable knot positions t_j, $j = 1,\ldots,n-1$.

SOLUTION for (5-6.h) → 5-6-8-0:3c.pdf ▲

(5-6.i) ⚁ (30 min.) [depends on Sub-problem (5-6.e), Sub-problem (5-6.g), Sub-problem (5-6.h)]

Based on `compute_c()` from Sub-problem (5-6.g) implement an efficient C++ function

```
Eigen::VectorXd quadspline(const Eigen::VectorXd &t,
                           const Eigen::VectorXd &y,
                           const Eigen::VectorXd &x);
```

which takes as input a (sorted) knot vector `t` (of length $n-1$, because $t_0 = 0$ and $t_n = 1$ will be taken for granted), an n-vector `y` containing the values $y_j \in \mathbb{R}$ of a function f at the midpoints $\frac{1}{2}(t_{j-1}+t_j)$, $j = 1,\ldots,n$, and a *sorted* N-vector `x` of evaluation points in $[0,1]$.

The function is to return the values of the interpolating quadratic spline s at the positions `x`.

SOLUTION for (5-6.i) → 5-6-9-0:s4.pdf ▲

(5-6.j) ⚂ (30 min.) [depends on Sub-problem (5-6.j)]

Based on MATPLOTLIBCPP write a C++ function

```
void plotquadspline(const std::string &filename);
```

that saves a plot (in `<filename>.png`) of the interpolating 1-periodic quadratic spline s for $f(t) := \exp(\sin(2\pi t))$, $n = 10$, and $\mathcal{M} = \left\{\frac{j}{n}\right\}_{j=0}^{10}$, that is, the spline s is to fulfill $s(\frac{1}{2}(t_j + t_{j-1})) = f(\frac{1}{2}(t_j + t_{j-1}))$, $j = 1, \ldots, 10$. To that end evaluate the spline in 200 equidistant points in the interval $[0,1]$ and, based on the obtained value, use the `plot()` function of MATPLOTLIBCPP to draw it. Do not forget to add axis labels "t" and "$s(t)$", and a suitable title.

SOLUTION for (5-6.j) → 5-6-10-0:s5.pdf ▲

(5-6.k) ⚄ (30 min.) [depends on Sub-problem (5-6.i)]

Write a C++ function

```
std::vector<double> qsp_error(unsigned int q);
```

that approximately computes the $L^\infty([0,1])$-norm of the error of the approximation of $f(t) := \exp(\sin(2\pi t))$ by its 1-periodic quadratic spline interpolant s_n based on the equidistant knot set $\mathcal{M}_n = \{j/n\}_{j=0}^n$:

$$E_n := \|f - s_n\|_{L^\infty([0,1])} := \sup_{x\in[0,1]} |f(x) - s_n(x)| \,.$$

To approximate the $L^\infty([0,1])$-norm you can evaluate the difference $|f(x) - s_n(x)|$ at $N \gg n$ equidistant points in $[0,1]$ and then take the maximum; choose $N = 10000$.

The function is supposed to return the error norms E_n for $n = 2, 4, 8, \ldots, 2^q$, where q is passed in the argument `q`.

Optional (Assumes familiarity with [Lecture → § 6.2.2.5] and [Lecture → § 6.2.2.9]). Describe quantitatively and qualitatively the convergence of the error norms in terms of $n \to \infty$.

HIDDEN HINT 1 for (5-6.k) → 5-6-11-0:s7h1.pdf

SOLUTION for (5-6.k) → 5-6-11-1:s7.pdf ▲

End Problem 5-6, 275 min.

Problem 5-7: Linear monotonicity-preserving C^1-interpolation scheme

In Sections [Lecture → Section 5.3.2], [Lecture → Section 5.3.3.2], [Lecture → Section 5.3], different shape preserving operator are introduced. In the first case (piecewise linear functions) the interpolating function is continuous but not C^1, in the two remaining cases (Hermite polynomials and splines) the interpolating operator is not linear. In all these examples the monotonicity is preserved, i.e., monotonic data yield monotonic interpolants. This problem shows that monotonicity preserving interpolators that are both C^1 and linear present an unpleasant and unexpected feature.

A purely theoretical problem. As preparation study [Lecture → § 5.1.0.21] and [Lecture → Def. 5.1.0.25] together with [Lecture → Section 5.3.3.2].

The goal of this problem is to prove the following theorem.

Theorem 5.7.1. Linear monotonicity preserving interpolation operator

Given the node set $\mathcal{T} = \{t_0, \ldots, t_n\} \subset [t_0, t_n]$, with $t_0 < t_1 < \ldots < t_n$, consider an interpolation operator $I_{\mathcal{T}} : \mathbb{R}^{n+1} \to C^0([t_0, t_n])$, for which, by definition, we have $\big(I_{\mathcal{T}}(\mathbf{y})\big)(t_j) = y_j$ for every $j = 0, \ldots, n$ and every vector of data values $\mathbf{y} = [y_0, \ldots, y_n]^\top \in \mathbb{R}^{n+1}$. Assume that the operator satisfies

i) smoothness: $I_{\mathcal{T}}(\mathbf{y})$ belongs to $C^1([t_0, t_n])$ for any $\mathbf{y} \in \mathbb{R}^{n+1}$,

ii) linearity: $I_{\mathcal{T}}(\alpha\mathbf{y} + \beta\mathbf{z}) = \alpha I_{\mathcal{T}}(\mathbf{y}) + \beta I_{\mathcal{T}}(\mathbf{z})$, for every $\alpha, \beta \in \mathbb{R}, \mathbf{y}, \mathbf{z} \in \mathbb{R}^{n+1}$,

iii) (positive) monotonicity preserving: if $y_j \le y_{j+1}$, $\forall\, j = 0, \ldots, n-1$, then $I_{\mathcal{T}}(\mathbf{y})$ is a monotonic non-decreasing function.

Then, for any $\mathbf{y} \in \mathbb{R}^{n+1}$, the derivative of $I_{\mathcal{T}}(\mathbf{y})$ in the interior nodes vanishes:

$$\big(I_{\mathcal{T}}(\mathbf{y})\big)'(t_j) = 0\,, \qquad \forall j \in \{1, \ldots, n-1\}\,, \quad \forall \mathbf{y} \in \mathbb{R}^{n+1}\,. \tag{5.7.2}$$

(5-7.a) ⚀ (15 min.) Show that the assertion (5.7.2) holds for the special data vectors

$$\mathbf{y} = \mathbf{y}^{(j)} := [\underbrace{0, \ldots, 0}_{j \text{ times}}, 1, \ldots, 1]^\top\,, \quad j = 0, \ldots, n\,.$$

HIDDEN HINT 1 for (5-7.a) → 5-7-1-0:s1h1.pdf

SOLUTION for (5-7.a) → 5-7-1-1:s1.pdf ▲

(5-7.b) ⚁ (10 min.) Prove that $\operatorname{span}\{\mathbf{y}^{(j)},\, j = 0, \ldots, n\} = \mathbb{R}^{n+1}$.

SOLUTION for (5-7.b) → 5-7-2-0:s1a.pdf ▲

(5-7.c) ⚂ (15 min.) [depends on Sub-problem (5-7.a) and Sub-problem (5-7.b)]

Invoke the insight gained in Sub-problem (5-7.a) to complete the proof of the theorem.

SOLUTION for (5-7.c) → 5-7-3-0:s2.pdf ▲

End Problem 5-7, 40 min.

Problem 5-8: Approximating π by extrapolation

In [Lecture → Section 5.2.3.3] we learned about the approximation of a limit $\lim_{h\to 0}\psi(h)$ based on the evaluation of the function ψ "far away" from 0. In this exercise we will practice the underlying technique of "Lagrangian polynomial extrapolation" for a geometric approximation problem.

Study [Lecture → Section 5.2.3.3] and, in particular [Lecture → Code 5.2.3.19], before your tackle this problem.

In this problem we encounter the situation that a quantity of interest is defined as a limit of a sequence

$$x^* = \lim_{n\to\infty} T(n)\ ,$$

where the function $T : \{n_{\min}, n_{\min}+1, \ldots\} \mapsto \mathbb{R}$ may be given in procedural form as `double T(int n)` only. However, invoking `T(Inf)` will usually result in an error and for very large arguments n the implementation of `T()` may not yield reliable results, *cf.* [Lecture → Ex. 1.5.4.7] and [Lecture → § 5.2.3.16].

The idea of **extrapolation** is, firstly, to compute a few values $T(n_0), T(n_1), \ldots, T(n_k)$, $k \in \mathbb{N}$, and to consider them as the values $g(1/n_0), g(1/n_1), \ldots, g(1/n_k)$ of a continuous function

$$g : (0, 1/n_{\min}] \mapsto \mathbb{R}, \quad \text{for which, obviously} \quad x^* = \lim_{h\to 0} g(h)\ .$$

Secondly, the function g is approximated by an interpolating polynomial $p_k \in \mathcal{P}_k$ with $p_k(n_j^{-1}) = T(n_j)$, $j = 0, \ldots, k$. In many cases we can expect that $p_k(0)$ will provide a good approximation for x^*.

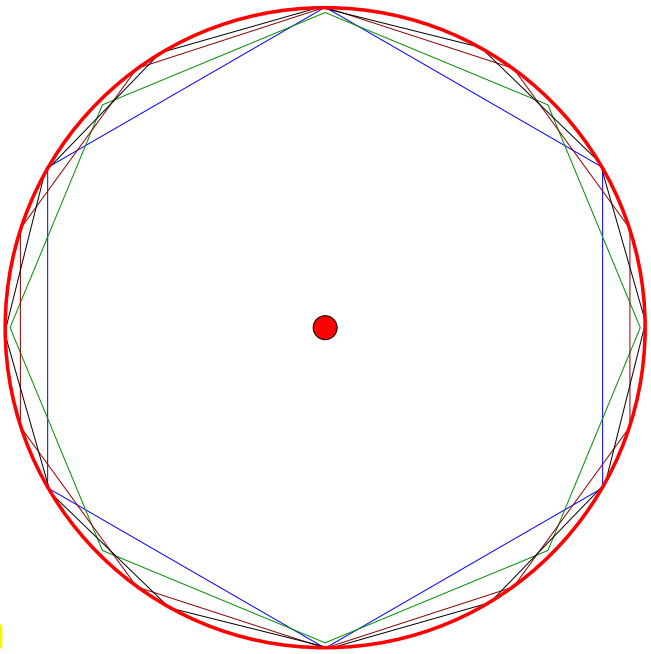


Fig. 36

The unit circle can be approximated by inscribed regular polygons with k edges. Thus, the length of half circumference (i.e., π) can be approximated by the half the length of the perimeters s_k of such polygons. These values $s_k/2$ can be calculated by elementary trigonometry and a closed form formula is

$$\tfrac{1}{2}s_k = k\sin(\pi/k)\ . \tag{5.8.1}$$

(5-8.a) ⚄ (10 min.) Show that for any $L \in \mathbb{N}$ and for $k \to \infty$

$$\tfrac{1}{2}s_k = \pi + \sum_{\ell=1}^{L} c_\ell k^{-2\ell} + O(k^{-2(L+1)}) \quad \text{for some} \quad c_\ell \in \mathbb{R} \quad \text{independent of} \quad L\ . \tag{5.8.2}$$

We say that s_k has an asymptotic expansion in k^{-2}.

Remark. The heuristics behind polynomial extrapolation in this example is that the existence of an asymptotic expansion (5.8.2) suggests that $\frac{1}{2}s_\infty$ can be well approximated by $p(0)$, where p is an interpolating polynomial in k^{-1} or k^{-2}.

HIDDEN HINT 1 for (5-8.a) → 5-8-1-0:s0h1.pdf

SOLUTION for (5-8.a) → 5-8-1-1:s0.pdf ▲

(5-8.b) ⚄ (45 min.) Implement a C++ function

```
double extrapolate_to_pi(const unsigned int k);
```

that uses the *Aitken-Neville scheme*, see [Lecture → Code 5.2.3.10], to approximate π by extrapolation from the data points $(j^{-1}, \frac{1}{2}s_j)$, for $j = 2, \ldots, k$. Use the values $\frac{1}{2}s_j$ as given in (5.8.1).

SOLUTION for (5-8.b) → 5-8-2-0:s1.pdf ▲

(5-8.c) ⚄ (30 min.) Write a C++ function

```
void plotExtrapolationError(const unsigned int kmax);
```

that generates a *linear-logarithmic* plot of the extrapolation errors

$$\mathrm{err}(k) := |\pi - p_k(0)| \,, \quad k = 2, \ldots, k_{\max} \,,$$

versus k and also tabulates the errors. To create the plot use the functions of MATPLOTLIBCPP. Do not forget axis annotations and a meaningful title for the plot.

In `main.cpp`, `plotExtrapolationError(10)` is called for $k_{\max} = 10$.

SOLUTION for (5-8.c) → 5-8-3-0:s2.pdf ▲

(5-8.d) ⚄ (15 min.) [depends on Sub-problem (5-8.c)]

Which kind of convergence of $\mathrm{err}(k) \to 0$ for $k \to \infty$ can you conclude from the data generated in Sub-problem (5-8.c)? The main types of asymptotic convergence to 0 are introduced in [Lecture → Def. 6.2.2.7].

HIDDEN HINT 1 for (5-8.d) → 5-8-4-0:s3h1.pdf

SOLUTION for (5-8.d) → 5-8-4-1:s3.pdf ▲

(5-8.e) ⚄ (15 min.) In `main.cpp`, `plotExtrapolationError(30)` is called which plots and tabulates err_k for k up to 30. Describe and explain your observations of the plot found in `cx_out/pi_error_30.png`.

SOLUTION for (5-8.e) → 5-8-5-0:s4.pdf ▲

End Problem 5-8 , 115 min.

Problem 5-9: Spaces of Piecewise Polynomial Functions

Piecewise polynomial functions, most prominently splines, are widely used for data interpolation and functions approximation. In this problem we examine some aspects of particular specimens.

A purely theoretical problem connected with [Lecture → Section 5.4].

For a mesh $\mathcal{M} := \{a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b\}$ we consider the following three spaces of piecewise polynomials:

$$V_{-1} := \left\{ v : [a,b] \to \mathbb{R} : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}, \tag{5.9.1}$$

$$V_0 := \left\{ v \in C^0([a,b]) : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}, \tag{5.9.2}$$

$$V_1 := \left\{ v \in C^1([a,b]) : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}, \tag{5.9.3}$$

$$V_2 := \left\{ v \in C^2([a,b]) : \ v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\ j = 1,\ldots,n \right\}. \tag{5.9.4}$$

Here, $\mathcal{P}_d$ stands for the space of polynomials $\mathbb{R} \to \mathbb{R}$ of degree $\leq d$.

(5-9.a) ⚄ (20 min.) Determine the dimensions of the four spaces defined above:

$$\dim V_{-1} = \boxed{},$$
$$\dim V_0 = \boxed{},$$
$$\dim V_1 = \boxed{},$$
$$\dim V_2 = \boxed{}.$$

SOLUTION for (5-9.a) → 5-9-1-0:s1.pdf ▲

(5-9.b) ⚄ (10 min.) Which of the spaces V_k, $k = -1, 0, 1$, coincides with one of the spline spaces $\mathcal{S}_{d,\mathcal{M}}$ introduced in [Lecture → Def. 5.4.1.1] in class?

$$V_{\boxed{}} = \mathcal{S}_{\boxed{},\mathcal{M}}.$$

SOLUTION for (5-9.b) → 5-9-2-0:s2.pdf ▲

(5-9.c) ⚄ (20 min.) Which of the following conditions for a function $v \in V_1$ are *sufficient* and which are *necessary* for v being non-decreasing, that is, for

$$a \leq x \leq y \leq b \quad \Rightarrow \quad v(x) \leq v(y) \ ?$$

(i) $v(t_{j-1}) \leq v(t_j)$ for all $j = 1, \ldots, n$

○ necessary ○ sufficient ○ neither

(ii) $v'(x) \geq 0$ for all $x \in [a, b]$

○ necessary ○ sufficient ○ neither

(iii) $v'(\frac{1}{2}(t_{j-1} + t_j)) \geq 0$ for all $j \in \{1, \ldots, n\}$

○ necessary ○ sufficient ○ neither

(iv) $v'(t_j) \geq 0$ for all $j \in \{0, \ldots, n\}$

○ necessary ○ sufficient ○ neither

SOLUTION for (5-9.c) → 5-9-3-0:s3.pdf ▲

In the sequel we consider the mesh (knot set) $\mathcal{M} = \{0, 1, 2\}$.

(5-9.d) ⚄ (10 min.) Determine the parameters $a, b \in \mathbb{R}$ in the following function definition

$$v(x) := \begin{cases} ax + b & \text{for } 0 \leq x < 1\,, \\ 1 - x^2 & \text{for } 1 \leq x \leq 2\,, \end{cases}$$

such that $v \in V_1$.

$$a = \boxed{} \quad , \quad b = \boxed{}\,.$$

SOLUTION for (5-9.d) → 5-9-4-0:s4.pdf ▲

(5-9.e) ⚄ (10 min.) Write down a $v \in V_1$ such that

$$v(0) = 0 \quad , \quad v(1) = 1 \quad , \quad v(2) = 0\,.$$

$$v(x) = \begin{cases} \boxed{} & \text{for } 0 \leq x < 1\,, \\ \boxed{} & \text{for } 1 \leq x \leq 2\,. \end{cases}$$

SOLUTION for (5-9.e) → 5-9-5-0:s5.pdf ▲

End Problem 5-9, 70 min.

Problem 5-10: Adaptive Interpolation of Closed Curves

Often, for the sake of efficiency, closed curves (loops) need to be approximated by piecewise polynomials. This problem presents a simple adaptive algorithm for constructing such an approximation.

Related to [Lecture → Section 5.3.3]; the idea of adaptive approximation is borrowed from [Lecture → Section 7.6].

A *1-periodic* continuously differentiable function $\mathbf{c} : \mathbb{R} \to \mathbb{R}^2$ parameterizes a closed curve in the plane.

For instance, the "kite-shaped" curve displayed beside is described by

$$\mathbf{c}(t) = \begin{bmatrix} \cos(2\pi t) + \frac{2}{3}\cos(4\pi t) \\ \frac{3}{2}\sin(2\pi t) \end{bmatrix}. \tag{5.10.1}$$

▷

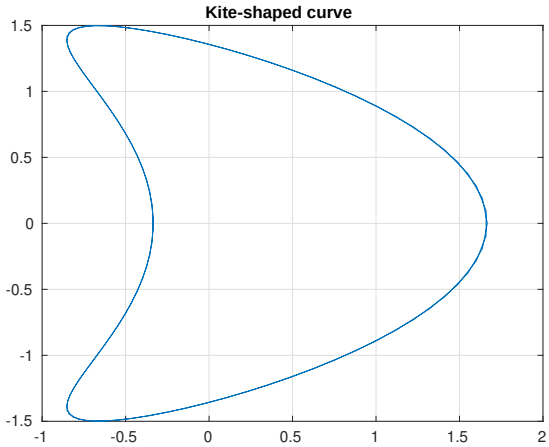


Fig. 40

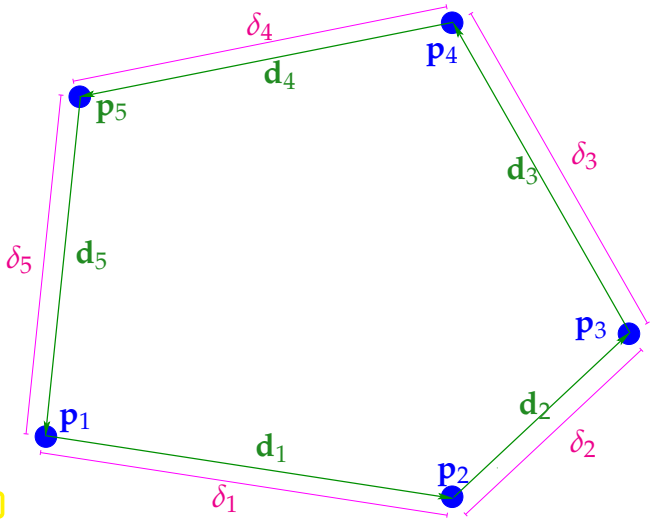


Fig. 41

Notations. Based on a sequence of $n \geq 2$ mutually distinct points $(\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_n)$ in the plane ($\mathbf{p}_k \in \mathbb{R}^2$) the following notations are defined:

- $\delta_i := \|\mathbf{p}_{i+1} - \mathbf{p}_i\|$, $i = 1, \ldots, n-1$,
  $\delta_n := \|\mathbf{p}_1 - \mathbf{p}_n\|$,
- $\lambda_0 := 0$,
  $\lambda_k := \sum_{j=1}^{k} \delta_j$, $k = 1, \ldots, n$,
- $\mathbf{d}_i := \mathbf{p}_{i+1} - \mathbf{p}_i$, $i = 1, \ldots, n-1$,
  $\mathbf{d}_n := \mathbf{p}_1 - \mathbf{p}_n$,
- $\mathbf{t}_i := \dfrac{\mathbf{d}_i}{\|\mathbf{d}_i\|}$, $i = 1, \ldots, n$.

We study two ways to connect points in the plane by a piecewise polynomial closed curve.

Definition 5.10.2. Closed polygonal interpolant

Given a sequence $\Sigma := (\mathbf{p}_1, \ldots, \mathbf{p}_n)$, $n \in \mathbb{N}$, of mutually distinct points $\in \mathbb{R}^2$, their **interpolating closed polygon** $\mathbf{\Pi}_\Sigma$ is a function $\mathbf{\Pi}_\Sigma : [0, \lambda_n] \to \mathbb{R}^2$ that fulfills:

(i) $\mathbf{\Pi}_\Sigma(\lambda_k) = \mathbf{p}_{k+1}$, $k = 1, \ldots, n-1$, and $\mathbf{\Pi}_\Sigma(0) = \mathbf{\Pi}_\Sigma(\lambda_n) = \mathbf{p}_1$.

(ii) $\mathbf{\Pi}_\Sigma|_{[\lambda_{k-1}, \lambda_k]}$, $k = 1, \ldots, n$, is componentwise an affine linear function, that is, $\mathbf{\Pi}_\Sigma|_{[\lambda_{k-1}, \lambda_k]} \in (\mathcal{P}_1)^2$.

Definition 5.10.3. Interpolating closed cubic Hermite curve

Given a sequence $\Sigma := (\mathbf{p}_1, \ldots, \mathbf{p}_n)$, $n \in \mathbb{N}$, of mutually distinct points $\in \mathbb{R}^2$, their **interpolating closed cubic Hermite curve** $\mathbf{H}_\Sigma$ is a function $\mathbf{H}_\Sigma : [0, \lambda_n] \to \mathbb{R}^2$ that satisfies:

(i) $\mathbf{H}_\Sigma(\lambda_k) = \mathbf{p}_{k+1}$, $k = 1, \ldots, n-1$, and $\mathbf{H}_\Sigma(0) = \mathbf{H}_\Sigma(\lambda_n) = \mathbf{p}_1$.

(ii) $\mathbf{H}'(\lambda_k) = \dfrac{\mathbf{s}_k}{\|\mathbf{s}_k\|}$, $\mathbf{s}_k := \begin{cases} \frac{\delta_{k+1}}{\delta_k+\delta_{k+1}}\mathbf{t}_k + \frac{\delta_k}{\delta_k+\delta_{k+1}}\mathbf{t}_{k+1} & \text{, if } k = 1, \ldots, n-1 \text{ ,} \\ \frac{\delta_n}{\delta_1+\delta_n}\mathbf{t}_1 + \frac{\delta_1}{\delta_1+\delta_n}\mathbf{t}_n & \text{, if } k = 0, n \text{ .} \end{cases}$

($\mathbf{H}'$ denotes the derivative of $\mathbf{H}$.)

(iii) $\mathbf{H}_\Sigma|_{[\lambda_{k-1},\lambda_k]}$, $k = 1, \ldots, n$, is componentwise a cubic polynomial, that is $\mathbf{H}_\Sigma|_{[\lambda_{k-1},\lambda_k]} \in (\mathcal{P}_3)^2$.

(5-10.a) ⚂ (20 min.)

Code an efficient C++ function

```
std::vector<Eigen::Vector2d>
  closedPolygonalInterpolant(
    std::vector<Eigen::Vector2d> &Sigma,
    const Eigen::VectorXd &x);
```

that returns the sequence of coordinate vectors $\mathbf{\Pi}_\Sigma(x_k\lambda_n)$, $k = 1, \ldots, M$, M the length of the vector `x` with *sorted* entries $0 \le x_1 < x_2 < \ldots x_M < 1$. $\mathbf{\Pi}_\Sigma$ is the interpolating closed polygon for a set Σ of points passed through the vector argument `Sigma`.

SOLUTION for (5-10.a) → 5-10-1-0:s1.pdf ▲

(5-10.b) ⚄ (20 min.) [depends on Sub-problem (5-10.a)]

Implement an efficient C++ function

```
std::vector<Eigen::Vector2d>
  closedHermiteInterpolant(
    std::vector<Eigen::Vector2d> &Sigma,
    const Eigen::VectorXd &x);
```

that returns the M vectors $\mathbf{H}_\Sigma(x_k\lambda_n) \in \mathbb{R}^2$, $k = 1, \ldots, M$, M the length of the vector `x` with *sorted* entries $0 \le x_1 < x_2 < \cdots < x_M \le 1$. In this case $\mathbf{H}_\Sigma$ is the interpolating closed cubic Hermite curve according to Def. 5.10.3 for the points passed in the vector `Sigma`.

You may use the auxiliary function

```
double hermloceval(double t,
    double t1, double t2,
    double y1, double y2,
    double c1, double c2);
```

provided in the file `hermloceval.hpp` , which implements the formula [Lecture → (5.3.3.4)] for evaluating a cubic Hermite interpolating polynomial, see also [Lecture → Code 5.3.3.6].

HIDDEN HINT 1 for (5-10.b) → 5-10-2-0:h2.pdf

SOLUTION for (5-10.b) → 5-10-2-1:s1.pdf ▲

Now, given a 1-periodic function $\mathbf{c} : \mathbb{R} \to \mathbb{R}^2$ providing the parameterization of a closed curve, we pursue the following policy aiming for its adaptive piecewise polynomial approximation.

Pseudocode 5.10.6: Algorithm for adaptive approximation of closed curve

Given: $\texttt{tol} > 0$, $n_{\min} \in \mathbb{N} \setminus \{1\}$.
$\mathcal{M} := \{t_0 := 0, t_1 := \frac{1}{n_{\min}}, \ldots, t_{n-1} := \frac{n_{\min}-1}{n_{\min}}\}$ // An ordered set
repeat {
  $n := \sharp\mathcal{M}$; // Notation: $\mathcal{M} := \{t_0, \ldots, t_{n-1}\}$
  $\Sigma := (\mathbf{c}(t))_{t\in\mathcal{M}}$; // sequence of points; also defines λ_k
  // $\Pi_\Sigma/\mathbf{H}_\Sigma$ according to Def. 5.10.2, Def. 5.10.3
  $\sigma_k := \left\| (\mathbf{H}_\Sigma - \Pi_\Sigma)(\frac{1}{2}(\lambda_{k-1} + \lambda_k)) \right\|$, $k = 1, \ldots, n$;
  $\alpha := \frac{1}{n}\sum_{k=1}^{n} \sigma_k$;
  for $j = 1, \ldots, n$ **do** {
    if $(\sigma_j > 0.9\alpha)$ $\mathcal{M} \leftarrow \mathcal{M} \cup \{\frac{1}{2}(t_{j-1} + t_j)\}$; // Convention $t_n := 1$
    // $\mathcal{M}$ is always assumed to be sorted!
  }
}
until $(\max_{k=1,\ldots,n} \sigma_k \le \texttt{tol})$;

(5-10.c) ⚃ (30 min.) [depends on Sub-problem (5-10.a) and Sub-problem (5-10.b)]

Based on the C++ functions `closedPolygonalInterpolant()` and `closedHermiteInterpolant()` implement a C++ function

```
template <typename CurveFunctor>
  std::pair<std::vector<Eigen::Vector2d>,
            std::vector<Eigen::Vector2d> >
    adaptedHermiteInterpolant(
      CurveFunctor &&c,
      unsigned int nmin,
      const Eigen::VectorXd &x, double tol = 1.0E-3);
```

that performs the interpolation of a closed curve using the above algorithm from Code 5.10.6. The 1-periodic continuous function $\mathbf{c} : \mathbb{R} \to \mathbb{R}$ is passed through a functor object `c`, which is to feature an evaluation operator

```
Eigen::Vector2d operator () (double t);
```

The argument `tol` supplies the tolerance for the termination of the adaptive algorithm. The function is to return

1. the vectors $\mathbf{H}_\Sigma(x_k L)$, $k = 1, \ldots, M$, M the length of the vector `x` with *sorted* entries $0 \le x_1 < x_2 < \cdots < x_M \le 1$. In this case $\mathbf{H}_\Sigma$ is the interpolating closed cubic Hermite curve according to Def. 5.10.3 for the set Σ at the termination of the algorithm,
2. the coordinates of the points in the sequence Σ as a second sequence of 2-vectors.

SOLUTION for (5-10.c) → `5-10-3-0:s3.pdf` ▲

(5-10.d) ⚃ (30 min.) [depends on Sub-problem (5-10.c)]

Using MATPLOTLIBCPP and based on your implementation of `adaptedHermiteInterpolant()`, write a C++ function

```
void plotKite(const char *filename, double tol = 1.0e-3);
```

that, for the kite-shaped curve $\mathbf{c} : [0,1] \to \mathbb{R}^2$ as defined in (5.10.1), and with the choice $n_{\min} = 6$, generates a rendering of the closed polygonal interpolant (→ Def. 5.10.2) and the interpolating closed cubic Hermite curve (→ Def. 5.10.3) produced by a call to the function `adaptedHermiteInterpolant()` with the given tolerance `tol`. The plots should be created using $N := 10000$ equidistant sampling points $x_k := \frac{k}{N}$, $k = 0, \ldots, N-1$.

The graphics is to be saved in PNG format in a file `<filename>.png`.

SOLUTION for (5-10.d) → `5-10-4-0:s4.pdf` ▲

Figures of kite-shaped curves produced by the code above code:

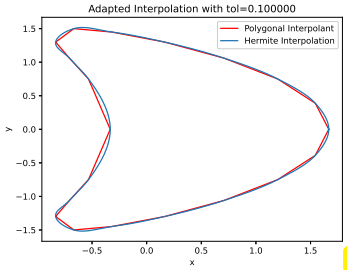


Fig. 45

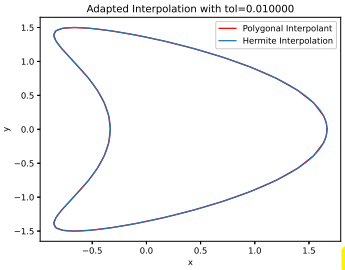


Fig. 46

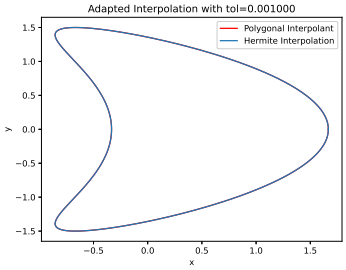


Fig. 47

End Problem 5-10 , 100 min.

Problem 5-11: A cubic spline

Since they mimic the behavior of an elastic rod pinned at fixed points, see [Lecture → § 5.4.3.1], cubic splines are very popular for creating "aesthetically pleasing" interpolating functions. However, in this problem we look at a cubic spline from the perspective of its defining properties, see [Lecture → Def. 5.4.1.1], in order to become more familiar with the concept of spline function and the consequences of the smoothness required by the definition.

Related to [Lecture → Section 5.4.1]

(5-11.a) ⚄ (15 min.) For parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ we define the function

$$s(t) = \begin{cases} -1 & \text{for} \quad -1 \leq t < 0\,, \\ \alpha t^3 + \beta t^2 + \gamma t + \delta & \text{for} \quad 0 \leq t < 1\,, \\ -t^3 + \sigma t^2 + \rho t + 2 & \text{for} \quad 1 \leq t \leq 2\,. \end{cases}$$

Determine the parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ such that s is a cubic spline with respect to the knot set $\mathcal{M} := \{-1, 0, 1, 2\}$.

$$\alpha = \boxed{}\,, \quad \beta = \boxed{} \quad \gamma = \boxed{}\,,$$

$$\delta = \boxed{}\,, \quad \sigma = \boxed{} \quad \rho = \boxed{}\,.$$

SOLUTION for (5-11.a) → 5-11-1-0:sx1.pdf ▲

(5-11.b) ⚄ (15 min.) For parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ we define the function

$$s(t) = \begin{cases} 2 & \text{for} \quad -1 \leq t < 0\,, \\ \sigma t^3 + \rho t^2 + \gamma t + \delta & \text{for} \quad 0 \leq t < 1\,, \\ -t^3 + \alpha t^2 + \beta t - 1 & \text{for} \quad 1 \leq t \leq 2\,. \end{cases}$$

Determine the parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ such that s is a cubic spline with respect to the knot set $\mathcal{M} := \{-1, 0, 1, 2\}$.

$$\alpha = \boxed{}\,, \quad \beta = \boxed{} \quad \gamma = \boxed{}\,,$$

$$\delta = \boxed{}\,, \quad \sigma = \boxed{} \quad \rho = \boxed{}\,.$$

SOLUTION for (5-11.b) → 5-11-2-0:sx2.pdf ▲

(5-11.c) ⚄ (15 min.) For parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ we define the function

$$s(t) = \begin{cases} 1 & \text{for} \quad -1 \leq t < 0\,, \\ \alpha t^3 + \beta t^2 + \gamma t + \delta & \text{for} \quad 0 \leq t < 2\,, \\ -t^3 + \rho t^2 + \sigma t - 7 & \text{for} \quad 2 \leq t \leq 3\,. \end{cases}$$

Determine the parameters $\alpha, \beta, \gamma, \delta, \sigma, \rho \in \mathbb{R}$ such that s is a cubic spline with respect to the knot set $\mathcal{M} := \{-1, 0, 2, 3\}$.

$\alpha =$ ☐ , $\beta =$ ☐ $\gamma =$ ☐ ,

$\delta =$ ☐ , $\sigma =$ ☐ $\rho =$ ☐ .

SOLUTION for (5-11.c) → 5-11-3-0:sx3.pdf ▲

End Problem 5-11 , 45 min.

Problem 5-12: Not-a-knot Cubic Splines

As we have seen in [Lecture → Section 5.4.2] cubic splines can be used for one-dimensional data interpolation, which is a popular technique in computer aided geometric design (CAGD). Unfortunately, extra conditions have to be imposed to render the cubic-spline interpolant unique as discussed in [Lecture → § 5.4.2.11]. Not-a-knot cubic splines are another way to arrive at a unique cubic-spline interpolant.

This problem builds on [Lecture → Section 5.4.2] and contents of [Lecture → § 5.4.2.5] should be studied in detail before tackling this problem.

Recall what is a cubic spline interpolant,

Definition [Lecture → Def. 5.4.2.3]. Cubic-spline interpolant

Given a node set/knot set $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$, $n \in \mathbb{N}$, and data values $y_0, \ldots, y_n \in \mathbb{R}$, an associated **cubic spline interpolant** is a function $s \in \mathcal{S}_{3,\mathcal{M}}$ that complies with the interpolation conditions

$$s(t_j) = y_j \quad , \quad j = 0, \ldots, n \, . \tag{5.12.1}$$

and, more fundamentally, the definition of spline functions as special piecewise polynomial functions:

Definition [Lecture → Def. 5.4.1.1]. Splines

Given an interval $I := [a,b] \subset \mathbb{R}$ and a **knot sequence** $\mathcal{M} := \{a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b\}$, $n \in \mathbb{N}$, the vector space $\mathcal{S}_{d,\mathcal{M}}$ of the **spline functions** of degree d (or order $d+1$), $d \in \mathbb{N}_0$, is defined by

$$\mathcal{S}_{d,\mathcal{M}} := \{s \in C^{d-1}(I) \colon s_j := s_{|[t_{j-1},t_j]} \in \mathcal{P}_d \; \forall j = 1, \ldots, n\} \, .$$

Now we study a specialization of [Lecture → Def. 5.4.2.3]:

Definition 5.12.2. Not-a-knot cubic spline

Given a node set/knot set $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$, $n \in \mathbb{N}$, $n \geq 2$, and data values $y_0, \ldots, y_n \in \mathbb{R}$, an associated **not-a-knot cubic spline interpolant** is a cubic spline interpolant s for the data points (t_i, y_i), $i = 0, \ldots, n$, that satisfies the extra conditions that

the *third derivative* of s is *continuous at* t_1 *and* t_{n-1}.

The following theorem of De Boor confirms that the not-a-knot constructions resolves the non-uniqueness of cubic spline interpolants.

Theorem 5.12.3. Existence and uniqueness of not-a-knot cubic spline interpolant

For any $n \in \mathbb{N}$, $n > 2$, and any node set/knot set $\mathcal{M} := \{t_0 < t_1 < \cdots < t_n\}$ and data values $y_i \in \mathbb{R}$, $i = 0, \ldots, n$, there exists a unique not-a-knot cubic spline interpolant according to Def. 5.12.2.

(5-12.a) ⚂ (15 min.) Why is the assumption $n > 2$ necessary in the statement of Thm. 5.12.3?

SOLUTION for (5-12.a) → 5-12-1-0:knks1.pdf ▲

You are supposed to implement a C++ class

```
class NotAKnotCubicSpline {
  public:
  NotAKnotCubicSpline(Eigen::VectorXd t,
  Eigen::VectorXd y);
  ~NotAKnotCubicSpline() = default;
  [[nodiscard]] double eval(double t) const;
  private:
  [[nodiscard]] int getPtIdx(double t) const;
  Eigen::VectorXd t_;   // knot sequence
  Eigen::VectorXd y_;  // data values
  Eigen::VectorXd c_;  // slopes
};
```

The public member function `eval()` is supposed to evaluate the not-a-knot cubic spline interpolant at a single point $t \in [t_0, t_n]$.

Fortunately, you found a complete implementation of a class **NaturalCubicSpline** with a 100% analogous definition for natural cubic spline interpolation. You stored it in the files `notaknotcubicspline.hpp` and ran a text replacement **NaturalCubicSpline** → **NotAKnotCubicSpline** on it. And this is where you are standing.

(5-12.b) ⚁ (45 min.) Each of the continuity requirements for the third derivative of the cubic spline interpolant at the knots t_1 and t_{n-1} (assume $n > 2$, see Def. 5.12.2 for notations) induces one linear relationship between the components of the vectors `y_` and `c_`. Derive and state those relationships.

SOLUTION for (5-12.b) → `5-12-2-0:.pdf` ▲

(5-12.c) ⚄ (45 min.) [depends on Sub-problem (5-12.b)]

In the file `notaknotcubicspline.hpp` write *efficient* code for the constructor

```
NotAKnotCubicSpline(Eigen::VectorXd t,
                    Eigen::VectorXd y);
```

that is supposed to initialize the data members `y_` and `c_` in a way that does not break the already implemented `eval()` member function.

HIDDEN HINT 1 for (5-12.c) → `5-12-3-0:knk2h1.pdf`

SOLUTION for (5-12.c) → `5-12-3-1:.pdf` ▲

(5-12.d) ⚁ (30 min.) An annoying client asks you to extend the functionality of **NotAKnotCubicSpline** to include a member function

```
double evalderivative(double t) const;
```

that returns the value of the first derivative of the not-a-knot cubic spline interpolant at a single point $t \in]t_0, t_n[$.

Add this member function to the class definition and provide an implementation in the file `notaknotcubicspline.hpp`.

SOLUTION for (5-12.d) → `5-12-4-0:s3ss.pdf` ▲

End Problem 5-12, 135 min.

Problem 5-13: Various Aspects of Interpolation by Global Polynomials

[Lecture → Chapter 5] teaches the mathematical foundations and algorithmic realizations of interpolation by means of global polynomials. This problems reviews some of these aspects.

This is a purely theoretical problem connected with [Lecture → Section 5.2]. It should be solved without resorting to the lecture document or tablet notes.

Throughout this problem we write $\mathcal{P}_d(\mathbb{R})$ for the space of uni-variate polynomials of degree $\leq d$:

$$\mathcal{P}_k := \{t \mapsto \alpha_k t^k + \alpha_{k-1} t^{k-1} + \cdots + \alpha_1 t + \alpha_0 \cdot 1\,,\ \alpha_j \in \mathbb{R}\}\,. \qquad \text{[Lecture → (5.2.1.1)]}$$

(5-13.a) ⚀ (15 min.) What are the dimensions of the following subspaces of $\mathcal{P}_d(\mathbb{R})$, $d \in \mathbb{N}$?

(i) $V_1 := \{p \in \mathcal{P}_d(\mathbb{R}) : \int_{-1}^{1} p(t)\,\mathrm{d}t = 0\}$: $\dim V_1 =$ ☐.

(ii) $V_2 := \{p \in \mathcal{P}_d(\mathbb{R}) : p(1) = p(-1) = 0\}$: $\dim V_2 =$ ☐.

(iii) $V_3 := \{p \in \mathcal{P}_d(\mathbb{R}) : p^{(3)}(t) = 0\,\forall t \in [-1,1]\}$: $\dim V_3 =$ ☐ for ☐, ☐ for ☐.

Here $p^{(3)}$ stands for the third derivative of p.

(iv) $V_4 := \{p \in \mathcal{P}_d(\mathbb{R}) : p(t) = p(-t)\}$: $\dim V_4 =$ ☐ for ☐, ☐ for ☐.

SOLUTION for (5-13.a) → 5-13-1-0:spi1.pdf ▲

(5-13.b) ⚀ (10 min.) Let $\{t_0, t_1, \ldots, t_n\} \subset \mathbb{R}$, $n \in \mathbb{N}$. The following sets of functions provide bases for $\mathcal{P}_d$:

$$\mathfrak{B}_1 := \left\{ t \mapsto \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{t - t_j}{t_i - t_j}\,,\quad i = 0, \ldots, n \right\}\,, \quad ☐$$

$$\mathfrak{B}_2 := \left\{ t \mapsto \prod_{j=0}^{i-1} (t - t_j)\,,\quad i = 0, \ldots, n \right\}\,, \quad ☐$$

$$\mathfrak{B}_3 := \left\{ t \mapsto t^i\,,\quad i = 0, \ldots, n \right\}\,. \quad ☐$$

Note that an empty product is defined to be $\equiv 1$.

Put the right letter in the box indicating the name of the basis:

A ≙ monomial basis , **B** ≙ Newton basis , **C** ≙ Lagrangian basis .

SOLUTION for (5-13.b) → 5-13-2-0:spip2.pdf ▲

(5-13.c) ⚄ (10 min.) Given data points (t_i, y_i), $i = 0, \ldots, n$, $n \in \mathbb{N}$, we write $p_{k,\ell}$, $0 \le k \le \ell \le n$, for the polynomial $p_{k,\ell} \in \mathcal{P}_{\ell-k}$ interpolating through $(t_i, y_i)_{i=k}^{\ell}$: $p_{k,\ell}(t_i) = y_i$, $i = k, \ldots, \ell$. Supplement the missing parts of the following recursion:

$$p_{k,\ell}(t) = \frac{\left(\square\right) p_{k+1,\ell}(t) + \left(\square\right) p_{k,\ell-1}(t)}{\square - \square}, \quad t \in \mathbb{R}, \quad 0 \le k < \ell \le n.$$

SOLUTION for (5-13.c) → 5-13-3-0:pips3.pdf ▲

End Problem 5-13, 35 min.

Problem 5-14: Local Representations of Splines

Locally on knot intervals splines coincide with polynomials of a fixed degree. In this problem we compute those local polynomial representations in special cases.

This is a purely theoretical problem based on [Lecture → Section 5.4.1] and [Lecture → Section 5.4.2].

(5-14.a) ⚄ (20 min.)

Let s denote a *quadratic spline*, $s \in \mathcal{S}_{2,\mathcal{M}}$ with respect to the knot set $\mathcal{M} := \{-3,-2,-1,1,2,3\}$ and satisfying

$$s \equiv 0 \quad \text{on} \quad [-3,-2] \cup [2,3] \,,$$
$$s(-1) = s(1) = 1 \,.$$

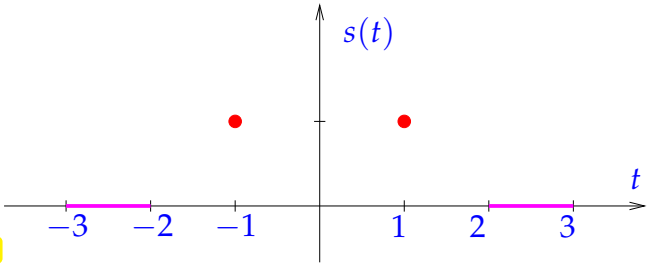


Fig. 48

Determine the unknown real coefficients in the local representations

$$s(t) = \square\, t^2 + \square\, t + \square \quad \text{for} \quad t \in [1,2] \,,$$

$$s(t) = \square\, t^2 + \square\, t + \square \quad \text{for} \quad t \in [-1,1] \,.$$

SOLUTION for (5-14.a) → 5-14-1-0:spls1.pdf ▲

(5-14.b) ⚄ (40 min.) Let the knot set $\mathcal{M} := \{t_0 < t_1 < t_2 < \cdots < t_n\} \subset \mathbb{R}$ be given. The local representation of a cubic spline $s \in \mathcal{S}_{3,\mathcal{M}}$ is given by ($h_j := t_j - t_{j-1}$, $j \in \{1,\ldots,n\}$)

$$s|_{[t_{j-1},t_j]}(t) = \alpha_j H_1(\tau) + \beta_j H_2(\tau) + h_j \gamma_j H_3(\tau) + h_j \delta_j H_4(\tau) \,, \quad \tau := \frac{t - t_{j-1}}{h_j} \,, \quad \alpha_j, \beta_j, \gamma_j, \delta_j \in \mathbb{R} \,,$$

where H_1, H_2, H_3, H_4 are the cardinal basis functions for Hermite interpolation on $[0,1]$,

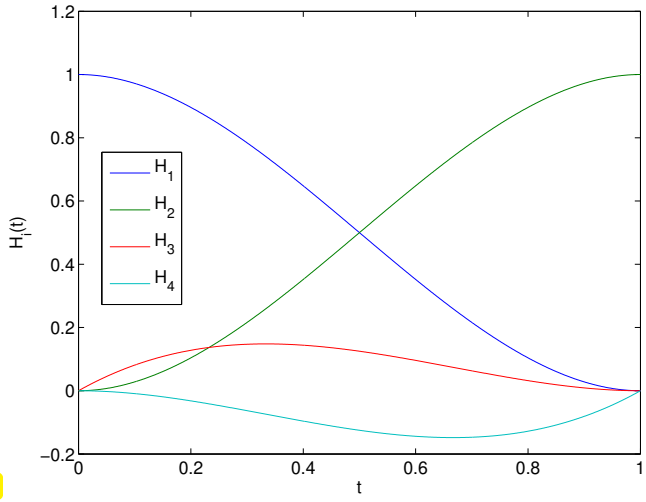


Fig. 49

$$H_1(\tau) = 1 - 3\tau^2 + 2\tau^3 \,,$$
$$H_2(\tau) = 3\tau^2 - 2\tau^3 \,,$$
$$H_3(\tau) = \tau - 2\tau^2 + \tau^3 \,,$$
$$H_4(\tau) = -\tau^2 + \tau^3 \,.$$

Inserting the midpoints of knot intervals gives us the extended knot set

$$\widetilde{\mathcal{M}} := \{\widetilde{t}_0 < \widetilde{t}_1 < \widetilde{t}_2 < \cdots < \widetilde{t}_{2n}\} \,, \quad \begin{aligned} \widetilde{t}_{2j} &:= t_j \,, & j &\in \{0,\ldots,n\} \,, \\ \widetilde{t}_{2j-1} &:= \tfrac{1}{2}(t_j + t_{j-1}) \,, & j &\in \{1,\ldots,n\} \,. \end{aligned}$$

On the knot intervals of $\widetilde{\mathcal{M}}$ the spline s has the local representation ($\widetilde{h}_\ell := \widetilde{t}_\ell - \widetilde{t}_{\ell-1}$, $\ell \in \{1,\ldots,2n\}$)

$$s|_{[\widetilde{t}_{\ell-1},\widetilde{t}_\ell]}(t) = \widetilde{\alpha}_\ell H_1(\tau) + \widetilde{\beta}_\ell H_2(\tau) + \widetilde{h}_\ell\widetilde{\gamma}_\ell H_3(\tau) + \widetilde{h}_\ell\widetilde{\delta}_\ell H_4(\tau)\ ,\ \tau := \frac{t-\widetilde{t}_{\ell-1}}{\widetilde{h}_\ell}\ ,\ \widetilde{\alpha}_\ell,\widetilde{\beta}_\ell,\widetilde{\gamma}_\ell,\widetilde{\delta}_\ell \in \mathbb{R}\ .$$

Express the coefficients $\widetilde{\alpha}_\ell, \widetilde{\beta}_\ell, \widetilde{\gamma}_\ell, \widetilde{\delta}_\ell$ in terms of $\alpha_j, \beta_j, \gamma_j, \delta_j$:

$$\widetilde{\alpha}_\ell = \begin{cases} \square & \text{for even } \ell = 2j\ , \\ \square & \text{for odd } \ell = 2j-1\ , \end{cases}$$

$$\widetilde{\beta}_\ell = \begin{cases} \square & \text{for even } \ell = 2j, \\ \square & \text{for odd } \ell = 2j-1\ , \end{cases}$$

$$\widetilde{\gamma}_\ell = \begin{cases} \square & \text{for even } \ell = 2j, \\ \square & \text{for odd } \ell = 2j-1\ , \end{cases}$$

$$\widetilde{\delta}_\ell = \begin{cases} \square & \text{for even } \ell = 2j, \\ \square & \text{for odd } \ell = 2j-1\ . \end{cases}$$

SOLUTION for (5-14.b) → `5-14-2-0:splss2.pdf` ▲

End Problem 5-14 , 60 min.

Problem 5-15: Closed Curve Interpolation with Quadratic Splines

The reconstruction of curves and surfaces from given points is a central task in computer-aided design (CAD). Here we focus on the problem of finding a closed and C^1-smooth curve running through a sequence of given points. That curve should be modeled by means of a periodic quadratic spline function.

This problem relies on [Lecture → Section 5.4] and has a focus on implementation in C++ using EIGEN.

We are given n, $n \in \mathbb{N}$, mutually different points $\mathbf{p}^1, \ldots, \mathbf{p}^n \in \mathbb{R}^2$ in the plane. Defining a knot sequence $\mathcal{M} := \{0 = t_0 < t_1 < \ldots < t_{n-1} < t_n = 1\}$ as [1]

$$t_j := \frac{\sum_{k=1}^{j} \left\| \mathbf{p}^k - \mathbf{p}^{k-1} \right\|_2}{\sum_{k=1}^{n} \left\| \mathbf{p}^k - \mathbf{p}^{k-1} \right\|_2} \quad (\, \mathbf{p}^0 := \mathbf{p}^n \,) \,, \quad j = 0, \ldots, n \,, \tag{5.15.1}$$

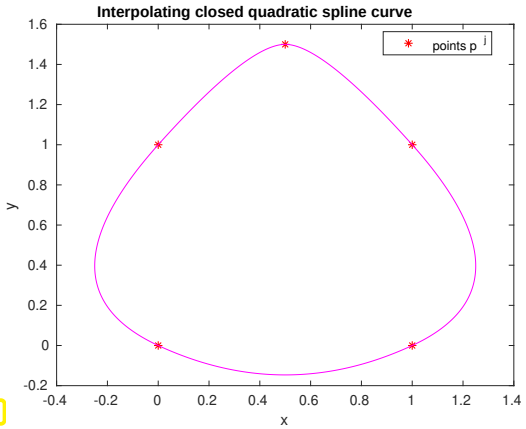


Fig. 50

we want to find a spline curve $\mathbf{s} : [0,1] \to \mathbb{R}^2$ satisfying

$$\mathbf{s} \in (\mathcal{S}_{2,\mathcal{M}})^2 \,, \quad \textit{c.f.} \ [\text{Lecture} \to \text{Def. 5.4.1.1}], \tag{5.15.2}$$

$$\mathbf{s}(t_j) = \mathbf{p}^j \quad \text{for all} \quad j = 1, \ldots, n \,, \tag{5.15.3}$$

$$\mathbf{s}(t_0) = \mathbf{p}^n \quad \text{and} \quad \mathbf{s}'(t_0) = \mathbf{s}'(t_n) \,, \tag{5.15.4}$$

where $'$ indicates differentiation with respect to the curve parameter. The requirement (5.15.4) ensures that $\mathbf{s}$ describes a *closed C^1-smooth* curve.

Throughout, for $\mathbf{S}$ use the following local representation on knot intervals:

$$\mathbf{s}|_{]t_{j-1},t_j[}(t) = \mathbf{y}^{j-1}(1-\tau) + \mathbf{x}^j h_j \tau(1-\tau) + \mathbf{y}^j \tau \,, \quad \tau := \frac{t - t_{j-1}}{h_j} \,, \quad \begin{array}{l} t_{j-1} < t < t_j \,, \\ h_j := t_j - t_{j-1} \,, \end{array} \tag{5.15.5}$$

with unknown vector-valued coefficients $\mathbf{y}^j \in \mathbb{R}^2$, $j = 0, \ldots, n$, and $\mathbf{x}^j \in \mathbb{R}^2$, $j = 1, \ldots, n$.

(5-15.a) ⚀ (6 min.) What are the benefits of using the local representation (5.15.5)?

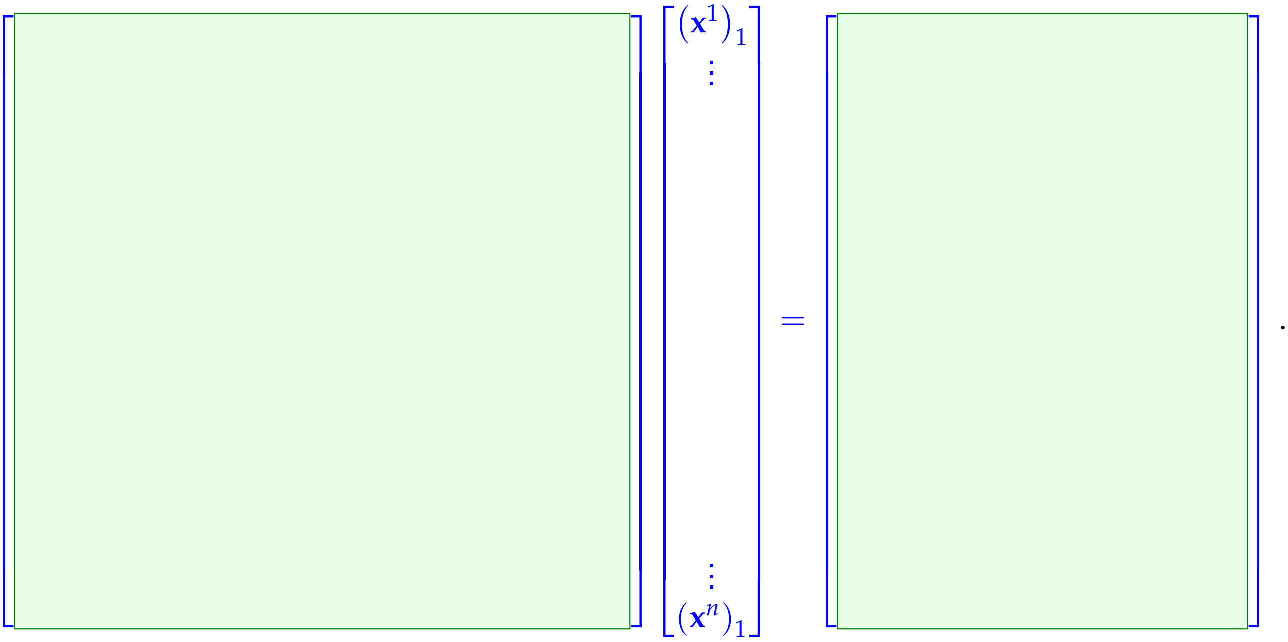
SOLUTION for (5-15.a) → `5-15-1-0:pqss1.pdf` ▲

(5-15.b) ⚁ (30 min.)

In explicit form state the linear system of equations

- whose coefficient matrix and right-hand side vector are to be given in terms of the point locations $\mathbf{p}^j \in \mathbb{R}^2$, $j = 1, \ldots, n$ and the knot positions t_k, $k = 0, \ldots, n$, and
- whose solution provides the vector $\left[(\mathbf{x}^1)_1, (\mathbf{x}^2)_1, \ldots, (\mathbf{x}^n)_1 \right]^\top \in \mathbb{R}^n$ of first components of the vector-valued coefficients $\mathbf{x}^j \in \mathbb{R}^2$, $j = 1, \ldots, n$, in (5.15.5).

[1] Sums whose lower summation bound exceeds the upper are supposed to evaluate to zero.



SOLUTION for (5-15.b) → 5-15-2-0:pqss2.pdf

▲

In an object-oriented C++ code, for handling interpolating closed C^1-smooth curves as specified above we rely on the following data type:

C++ code 5.15.20: Class definition of ClosedQuadraticSplineCurve

```cpp
class ClosedQuadraticSplineCurve {
 public:
  // Constructor
  explicit ClosedQuadraticSplineCurve(const Eigen::Matrix2Xd &p);
  ~ClosedQuadraticSplineCurve() = default;
  ClosedQuadraticSplineCurve() = delete;
  ClosedQuadraticSplineCurve(const ClosedQuadraticSplineCurve &) = delete;
  ClosedQuadraticSplineCurve(const ClosedQuadraticSplineCurve &&) = delete;
  ClosedQuadraticSplineCurve &operator=(const ClosedQuadraticSplineCurve &) =
      delete;
  // Point evaluation operator for sorted parameter arguments
  [[nodiscard]] Eigen::Matrix2Xd curve_points(const Eigen::VectorXd &v) const;
  // Curvature evaluation operator for sorted parameter arguments
  [[nodiscard]] Eigen::VectorXd local_curvatures(
      const Eigen::VectorXd &v) const;
  // Approximate computation of the length of the curve
  [[nodiscard]] double length(double rtol = 1E-6) const;

 private:
  [[nodiscard]] bool checkC1(double tol = 1.0E-4) const;
  // Number of points to be interpolated
  unsigned int n_;
  // Coordinates of interpolated points, duplicate: p^0 = p^n
  Eigen::Matrix2Xd p_;
  // Knot sequence
  Eigen::VectorXd t_;
  // Coefficients in local representation
  Eigen::Matrix2Xd x_;
};
```

Get it on GitLab (periodicquadraticsplines.hpp).

(5-15.c) ⚀ (20 min.) [depends on Sub-problem (5-15.b)]

In the file `periodicquadraticsplines.hpp` provide an *efficient* implementation of the constructor

```
ClosedQuadraticSplineCurve::ClosedQuadraticSplineCurve(
const Eigen::Matrix<double, 2, Eigen::Dynamic> &p);
```

which is supposed to initialize the member variable `t_` with the elements of the knot sequence $\mathcal{M}$ and the columns of `x_` with the vector coefficients $\mathbf{x}^j$, $j = 1,\ldots,n$. The matrix argument `p` passes the coordinate vectors $\mathbf{p}^j$, $j = 1,\ldots,n$, in its columns.

HIDDEN HINT 1 for (5-15.c) → 5-15-3-0:pqsh3.pdf

SOLUTION for (5-15.c) → 5-15-3-1:.pdf ▲

(5-15.d) ⚀ (15 min.) Complete the code in `periodicquadraticsplines.hpp` to achieve an *efficient* implementation, that is, with cost scaling linearly in the number of data and in the number n of points, of the member function

```
Eigen::Matrix<double, 2, Eigen::Dynamic>
ClosedQuadraticSplineCurve::curve_points(const Eigen::VectorXd &v)
   const;
```

that takes a *sorted* sequence $(0 \le v_1 \le v_2 \le \cdots \le v_{N-1} \le v_N \le 1)$, $N \in \mathbb{N}$, of parameter values as `v` and returns the sequence $(\mathbf{s}(v_1), \mathbf{s}(v_2), \ldots, \mathbf{s}(v_N))$ as the columns of a $2 \times N$-matrix.

SOLUTION for (5-15.d) → 5-15-4-0:pqss3a.pdf ▲

(5-15.e) ⚀ (20 min.) Devise an *efficient* implementation of the member function

```
Eigen::VectorXd ClosedQuadraticSplineCurve::local_curvatures(
const Eigen::VectorXd &v) const;
```

that takes a *sorted* sequence $(0 \le v_1 < v_2 < \cdots < v_{N-1} < v_N \le 1)$, $N \in \mathbb{N}$, of parameter values as `v` and returns a vector of local curvatures of the curve $\mathbf{s}$.

The **curvature** of $\mathbf{s} : [0,1] \to \mathbb{R}^2$ is a function $\kappa : [0,1] \to \mathbb{R}$ given by

$$\kappa(t) := \frac{\det\left[\mathbf{s}'(t), \mathbf{s}_j''(t)\right]}{\|\mathbf{s}'(t)\|_2}\,, \quad \mathbf{s}_j := \mathbf{s}|_{]t_{j-1},t_j]}\,, \quad t_{j-1} < t \le t_j\,, \quad j = 1,\ldots,n\,.$$

HIDDEN HINT 1 for (5-15.e) → 5-15-5-0:pqsh4.pdf

SOLUTION for (5-15.e) → 5-15-5-1:.pdf ▲

(5-15.f) ⚀ Based on approximation of the curve $t \mapsto \mathbf{s}(t)$ by means of polygons, implement the following member function of **ClosedQuadraticSplineCurve**:

```
double ClosedQuadraticSplineCurve::length(double rtol) const;
```

It is supposed to return an approximation of the length of the curve with a relative error bounded by `rtol`.

SOLUTION for (5-15.f) → 5-15-6-0:pqss5.pdf ▲

End Problem 5-15 , 91 min.

Problem 5-16: Some Aspects of Interpolation by Global Polynomials

Interpolation by global polynomials is a fundamental numerical technique that forms the basis of many other algorithms. Therefore its various features and related concepts are must-knows for everybody involved in computational science and engineering.

This problem is related to [Lecture → Section 5.2].

(5-16.a) ⚂ (10 min.) The Lagrange polynomials $L_0,\ldots,L_n$ belonging to a node set $\{t_0,\ldots,t_n\} \subset \mathbb{R}$ are polynomials of degree n uniquely defined by the cardinal basis condition

$$L_i(t_j) = \delta_{ij} := \begin{cases} 1 & \text{if } i = j\,, \\ 0 & \text{else.} \end{cases} \tag{5.16.1}$$

Give an explicit formula for L_i ($\prod \hat{=}$ product, equal to 1, when empty):

$$L_i(t) = \prod_{\square}^{\square} \frac{t - t_{\square}}{t_{\square} - t_{\square}}\,, \qquad i = 0,\ldots,n\,.$$

SOLUTION for (5-16.a) → 5-16-1-0:pintps1.pdf

▲

(5-16.b) ⚀ (10 min.) Given a node set $\{t_0,\ldots,t_n\}$ describe the Newton basis $\mathcal{N}_\mathcal{T}$ of the space $\mathcal{P}_n$ of uni-variate polynomials of degree $\le n$, $n \in \mathbb{N}$ belonging to the sequence $\mathcal{T} := (t_0, t_1, \ldots, t_n)$:

$$\mathcal{N}_\mathcal{T} = \left\{ \square \right\}_{i=0}^{n}\,.$$

HIDDEN HINT 1 for (5-16.b) → 5-16-2-0:pipdeth1.pdf

SOLUTION for (5-16.b) → 5-16-2-1:pipds2.pdf ▲

(5-16.c) ⚁ (15 min.) The following classes are meant to evaluate global polynomial interpolants in different settings:

```
class InterpPolyEval_A {
  public:
  // Constructor taking the evaluation point x as argument
  InterpPolyEval_A(double x);
  // Add another data point (t,v) ∈ ℝ² and update internal
     information
  void addPoint(double t, double y);
  // Value of current interpolating polynomial at x
  double eval(void) const;
  // ... private data and member functions
};
```

Which algorithm would you recommend for the efficient implementation of the class **InterpPolyEval_A**, if one has to deal with many calls to the member functions `InterpPolyEval_A::addPoint()` and `InterpPolyEval_A::eval()`? (Pick only one)

Barycentric interpolation formula ○	Aitken-Neville scheme ○	Divided differences ○

```
class InterpPolyEval_B {
  public:
  // Constructors taking node vector (t_0,...,t_n) as argument
  InterpPolyEval_B(const Eigen::VectorXd &t);
  // Evaluation operator for data (y_0,...,y_n); computes
  // p(x_k) for x_ks passed in x
  Eigen::VectorXd eval(const Eigen::VectorXd &y,
  const Eigen::VectorXd &x) const;
  // ... private data and member functions
};
```

Which algorithm is most suitable for the efficient implementation of the class **InterpPolyEval_B** in the case $n \gg 1$ and many invocations of `InterpPolyEval_B::eval()`? (Pick only one)

Barycentric interpolation formula ○	Aitken-Neville scheme ○	Divided differences ○

```
class InterpPolyEval_C {
  public:
  // Idle Constructor
  InterpPolyEval_C();
  // Add another data point (t,y) ∈ ℝ^2 and update internal
     information
  void addPoint(double t, double y);
  // Evaluation of current interpolating polynomial at many points
     passed in x
  Eigen::VectorXd eval(const Eigen::VectorXd &x) const;
  // ... private data and member functions
};
```

Which is the best algorithm for the efficient implementation of the class **InterpPolyEval_C** if one expects many calls to `InterpPolyEval_C::addPoint()` and `InterpPolyEval_C::eval()` in arbitrary order? (Pick only one)

Barycentric interpolation formula ○	Aitken-Neville scheme ○	Divided differences ○

SOLUTION for (5-16.c) → `5-16-3-0:.pdf`

▲

End Problem 5-16, 35 min.

Problem 5-17: Algorithms for Polynomial Interpolation

Global polynomials are the most important set of functions used for interpolation. This problem recalls key algorithms for the evaluating interpolating polynomials.

The problem is connected with [Lecture → Section 5.2.3.4] and [Lecture → Section 5.2.3.3].

(5-17.a) ⚀ (15 min.) In Code 5.17.1 complete the implementation of the C++ function

```
double evalNewtonForm(const std::vector<double> &t,
                      const std::vector<double> &c, double x);
```

which evaluates a polynomial given through the coefficients of its Newton basis expansion at a single point $x \in \mathbb{R}$ (argument `x`). More precisely, setting $n := \texttt{c.size()} - 1$ the function should return

$$p(x) := \sum_{j=0}^{n} \texttt{c[j]}\, N_j(x)\,, \quad N_j(t) := \prod_{i=0}^{j-1} (t - \texttt{t[i]})\,, \quad j = 0,\ldots,n\,,$$

$$\Longrightarrow \quad p(x) = \texttt{c[0]} + \texttt{c[1]}(x - \texttt{t[0]}) + \texttt{c[2]}(x - \texttt{t[0]})(x - \texttt{t[1]}) + \ldots + \texttt{c[n]}(x - \texttt{t[0]}) \cdot \cdots \cdot (x - \texttt{t[n-1]})\,,$$

where an "empty product" is meant to evaluate to 1.

C++ code 5.17.1: Function `evalNewtonForm()`

```
1  double evalNewtonForm(const std::vector<double> &t,
2                        const std::vector<double> &c, double x) {
3    assert((t.size() == c.size() && "Sizes of t and c vectors must agree");
4    const int n = c.size() - 1;
5    double p = c[n];
6    for (int j = ____; j >= ____; --j) {
7      p = ____;
8    }
9    return p;
10 }
```

HIDDEN HINT 1 for (5-17.a) → 5-17-1-0:nheha1.pdf

SOLUTION for (5-17.a) → 5-17-1-1:.pdf ▲

(5-17.b) ⚁ (15 min.) After initialization or `reset()` an object of the type

```
class ConvergentSequence {
  public:
    ConvergentSequence();
    double next();
    void reset();
  private:
    // Some private data
};
```

produces the members of a sequence $(s_n)_{n=1}^{\infty}$ one after another by calls to the `next()` member function. The first value returned is s_1. The sequence is known to converge for $n \to \infty$ with limit $s^* \in \mathbb{R}$: $s^* = \lim_{n\to\infty} s_n$.

In Listing 5.17.3 write valid C++ code in the blanks to complete the code of the function

```
double limitByExtrapolation(
  SEQGEN &&seq, double rtol = 1.0E-6,
  double atol = 1.0E-8, unsigned int maxdeg = 20);
```

which is to compute the limit s^* by extrapolation to zero. The type **SEQGEN** is compatible with **ConvergentSequence** and must offer a member function `next()` fitting the specification given before.

C++ code 5.17.3: Function `limitByExtrapolation()`

```
template <class SEQGEN>
double limitByExtrapolation(SEQGEN &&seq, double rtol = 1.0E-6,
                            double atol = 1.0E-8, unsigned int maxdeg = 20) {
  seq.reset();
  std::vector<double> h{      };
  std::vector<double> s{seq.next()};
  for (unsigned i = 1; i < maxdeg; ++i) {
    h.push_back(      );
    s.push_back(seq.next());
    for (int k =      ; k >=      ; --k)
      s[k] =                                                  ;
    const double errest = std::abs(s[1] - s[0]);
    if (errest < rtol * std::abs(s[0]) || errest < atol)
      return s[0];
  }
  throw std::runtime_error("limitByExtrapolation failed to converge!");
  return std::nan("");
}
```

HIDDEN HINT 1 for (5-17.b) → 5-17-2-0:nheh1.pdf

SOLUTION for (5-17.b) → 5-17-2-1:.pdf ▲

End Problem 5-17, 30 min.

Problem 5-18: Polynomial Interpolation based on Newton basis

This problem revisits polynomial interpolants in Newton form, which is known as an "update-friendly" basis representation with expansion coefficients provided by divided differences.

This problem is linked to [Lecture → Section 5.2.3.3] and [Lecture → Section 5.2.3.4].

This problem centers around the following helper class connected with the Lagrange polynomial interpolation problem:

```
class PolyEval {
 public:
  PolyEval() = default;
  void addPoint(double t, double y, double tol = 1.0E-10);
  double operator()(double x) const;
 private:
  std::vector<double> t_;
  std::vector<double> c_;
};
```

An object of type **PolyEval** represents a polynomial, which is the zero function $p \equiv 0$ immediately after initialization. The ℓ-th invocation of the member function `addPoint()` supplies the data point $(t_{\ell-1}, y_{\ell-1}) \in \mathbb{R}^2$ ($t_{\ell-1} \leftrightarrow$ `t`, $y_{\ell-1} \leftrightarrow$ `y`). Afterwards, assuming that the t_ks, $k = 0, \ldots, \ell-1$, are pairwise distinct, the object represents the unique interpolating polynomial

$$p \in \mathcal{P}_n: \quad p(t_k) = y_k \quad \forall k = 0, \ldots, \ell-1 \,.$$

Whenever called the evaluation operator **operator** `()` returns the number $p(x)$ for the argument $x \leftrightarrow$ `x` given to it. Here p is the polynomial *currently* stored in the **PolyEval** object.

(5-18.a) ⚀ (45 min.) The evaluation operator is implemented as follows and this implementation has to be taken for granted.

C++ code 5.18.1: Implementation of evaluation operator for PolyEval

```
2 double PolyEval::operator()(double x) const {
3   const int n = c_.size();
4   double p = (n == 0) ? 0.0 : c_[0];
5   for (int j = 1; j < n; ++j) {
6     p = (x - t_[j]) * p + c_[j];
7   }
8   return p;
9 }
```

In the file `newtonpolyinterp.hpp` implement the member function `addPoint()` of **PolyEval** so that it modifies the data member in a way fitting the implementation of the evaluation operator. Your implementation should ensure an asymptotic complexity $O(n)$ for $n \to \infty$, where n stands for the number of points added so far.

HIDDEN HINT 1 for (5-18.a) → `5-18-1-0:npoah1.pdf`

SOLUTION for (5-18.a) → `5-18-1-1:.pdf` ▲

(5-18.b) ⚁ (15 min.) After initialization or `reset()` an object of the type

```
class ConvergentSequence {
  public:
```

```
    ConvergentSequence();
    double next();
    void reset();
  private:
    // Some private data
};
```

produces the members of a sequence $(s_n)_{n=1}^{\infty}$ one after another by calls to the `next()` member function. The first value returned is s_1. The sequence is known to converge for $n \to \infty$ with limit $s^* \in \mathbb{R}$: $s^* = \lim_{n\to\infty} s_n$.

In the file `newtonpolyinterp.hpp` finish the implementation of the C++ function

```
template <class SEQGEN>
double extrapolateLimit(
  SEQGEN &&seq, double rtol = 1.0E-6,
  double atol = 1.0E-8, unsigned int maxdeg = 20);
```

which computes the limit s^* by **extrapolation to zero**. The type **SEQGEN** is to be compatible with **ConvergentSequence** and must offer a member function `next()` fitting the specification given before. The depth of extrapolation should be controlled by an a-posteriori error estimate with relative accuracy `rtol` absolute accuracy `atol`, and should be limited to `maxdeg`.

You are expected to make use of the helper class **PolyEval**.

SOLUTION for (5-18.b) → 5-18-2-0:.pdf ▲

End Problem 5-18 , 60 min.

Problem 5-19: Quadratic B-Splines

B-splines are particular spline functions with minimal support [Lecture → § 5.5.3.2]. This problem studies B-splines of degree 2 on an equidistant knot sequence.

This problem merely requires familiarity with [Lecture → Section 5.4.1] and does not involve any coding.

Throughout this problem we consider the knot sequence

$$\mathcal{M} := \{0,1,2,\ldots,n\}\,, \quad n \geq 3\,. \tag{5.19.1}$$

(5-19.a) ⚀ (20 min.)

For the function

$$B_0(t) = \begin{cases} 0 & \text{for } t < 0\,, \\ \frac{1}{2}t^2 & \text{for } 0 < t \leq 1\,, \\ -t^2 + \alpha t + \beta & \text{for } 1 < t \leq 2\,, \\ \frac{1}{2}(3-t)^2 & \text{for } 2 < t \leq 3\,, \\ 0 & \text{for } t \geq 3\,, \end{cases}$$

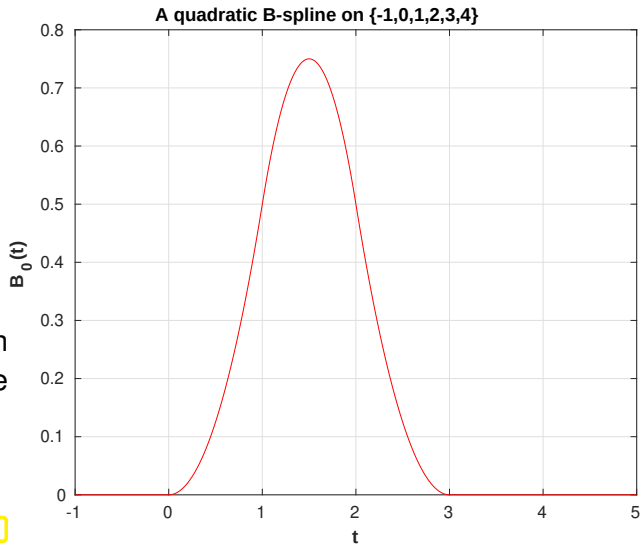


Fig. 51

determine the unknown coefficients $\alpha, \beta \in \mathbb{R}$ such that B_0 is a quadratic spline on the knot sequence $\mathcal{M}$, $B_0 \in \mathcal{S}_{2,\mathcal{M}}$.

$\alpha =$ ☐ , $\beta =$ ☐ .

HIDDEN HINT 1 for (5-19.a) → 5-19-1-0:spls1h.pdf

SOLUTION for (5-19.a) → 5-19-1-1:spls1.pdf

▲

We write

$$B_i(t) := B_0(t-i)\,, \quad i \in \mathbb{Z}\,.$$

Then the ordered set of functions

$$\mathcal{X} := \{B_{-2}, B_{-1}, B_0, \ldots, B_{n-1}\} \tag{5.19.2}$$

is a *ordered basis* of $\mathcal{S}_{2,\mathcal{M}}$.

(5-19.b) ⚁ (5 min.) What is the cardinality (the number of elements) of the set $\mathcal{X}$?

$$\sharp\mathcal{X} = \boxed{}\,.$$

SOLUTION for (5-19.b) → 5-19-2-0:spls2.pdf ▲

(5-19.c) ⚂ (25 min.) [depends on Sub-problem (5-19.a)]

We fix $n = 10$. We can represent every $s \in \mathcal{S}_{2,\mathcal{M}}$ as a unique linear combination of the basis functions in $\mathcal{X}$,

$$s(t) = \sum_{j=-2}^{9} c_j B_j(t)\,, \quad 0 \leq t \leq 10\,. \tag{5.19.3}$$

For given data $y_0, \ldots, y_{10} \in \mathbb{R}$ find a suitable matrix $\mathbf{A} \in \mathbb{R}^{11,12}$ such that the following statement is true:

$$s(i) = y_i \quad \forall i \in \{0, \ldots, 10\} \quad \Longleftrightarrow \quad \mathbf{A} \begin{bmatrix} c_{-2} \\ c_{-1} \\ \vdots \\ c_9 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{10} \end{bmatrix} . \tag{5.19.4}$$

The right matrix is

$$\mathbf{A} = \begin{bmatrix} \quad \end{bmatrix} .$$

Explicitly specify the numerical values of its entries.

SOLUTION for (5-19.c) → 5-19-3-0:.pdf

▲

(5-19.d) ⚄ (25 min.) Again, we fix $n = 10$. For the spline space $\mathcal{S}_{1,\mathcal{M}}$ of piecewise linear and continuous functions we choose the ordered **tent function basis** $\mathcal{T} := \{T_0, T_1, \ldots, T_{10}\}$, with

$$T_j(t) = \widehat{T}(t - j) \quad \text{with} \quad \widehat{T}(t) = \begin{cases} 0 & \text{for} \quad t < -1 , \\ 1 + t & \text{for} \quad -1 \le t < 0 , \\ 1 - t & \text{for} \quad 0 \le t < 1 , \\ 0 & \text{for} \quad t \ge 1 . \end{cases} \tag{5.19.6}$$

What is the matrix representation $\mathbf{D} \in \mathbb{R}^{11,12}$ of the derivative operator

$$\frac{d}{dt} : \mathcal{S}_{2,\mathcal{M}} \to \mathcal{S}_{1,\mathcal{M}}$$

when $\mathcal{X}$ from (5.19.2) is used as a basis of $\mathcal{S}_{2,\mathcal{M}}$ and $\mathcal{T}$ as a basis of $\mathcal{S}_{1,\mathcal{M}}$?

$$\mathbf{D} = \begin{bmatrix} \quad \end{bmatrix} .$$

HIDDEN HINT 1 for (5-19.d) → 5-19-4-0:spl4h.pdf

SOLUTION for (5-19.d) → 5-19-4-1:spls4.pdf ▲

(5-19.e) ⚄ (25 min.) Let $I_C^n : C^0([0,10]) \to \mathcal{P}_n$, $n \in \mathbb{N}$, stand for the $n+1$-point Chebychev interpolation operator on $[0,10]$, that is, the Lagrangian polynomial interpolation operator based on $n+1$ Chebychev nodes in $[0,1]$.

Qualitatively and quantitatively predict the asymptotic behavior of the maximum norm of the interpolation error

$$\epsilon_n(B_0) = \|B_0 - I_C^n B_0\|_{\infty,[0,10]} := \max_{t\in[0,10]} | \quad \text{for} \quad n \to \infty \,.$$

SOLUTION for (5-19.e) → 5-19-5-0:.pdf ▲

End Problem 5-19 , 100 min.

Chapter 6

Approximation of Functions in 1D

Problem 6-1: Piecewise linear approximation on graded meshes

One of the main point made in [Lecture → Section 6.2.3] is that the quality of an interpolant depends heavily on the choice of the interpolation nodes. If the function to be interpolated has a "bad behavior" in a small part of the domain, for instance it has very large derivatives of high order, more interpolation points are required in that area of the domain. Commonly used tools to cope with this task are *graded meshes*, which will be the topic of this problem.

Substantial implementation in C++ is requested.

Given a mesh $\mathcal{M} = \{0 = t_0 < t_1 < \cdots < t_{n-1} < t_n = 1\}$ on the unit interval $I = [0,1]$, $n \in \mathbb{N}$, we define the *piecewise linear interpolant*:

$$\mathsf{I}_{\mathcal{M}} : C^0(I) \to \mathcal{P}_{1,\mathcal{M}} = \{s \in C^0(I),\ s_{|[t_{j-1},t_j]} \in \mathcal{P}_1 \ \forall\, j\}, \quad \text{s.t.} \quad (\mathsf{I}_{\mathcal{M}} f)(t_j) = f(t_j), \quad j = 0,\ldots,n \tag{6.1.1}$$

See also [Lecture → Section 5.3.2].

(6-1.a) ⚀ (10 min.) If we choose the uniform mesh $\mathcal{M} = \{t_j\}_{j=0}^n$ with $t_j = j/n$, given a function $f \in C^2(I)$ what is the asymptotic behavior of the error $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ when $n \to \infty$?

HIDDEN HINT 1 for (6-1.a) → 6-1-1-0:GradedMeshes1h.pdf

SOLUTION for (6-1.a) → 6-1-1-1:GradedMeshes1s.pdf ▲

(6-1.b) ⚀ (5 min.) What is the smoothness class of the function $f : I := [0,1] \to \mathbb{R}$, $f(t) = t^\alpha$, $0 < \alpha < 2$? In other words, for which $k \in \mathbb{N}$ do we have $f \in C^k(I)$?

HIDDEN HINT 1 for (6-1.b) → 6-1-2-0:GradedMeshes2h.pdf

SOLUTION for (6-1.b) → 6-1-2-1:GradedMeshes2s.pdf ▲

(6-1.c) ⚀ (30 min.) Implement a templated C++ function

```
template <typename FUNCTION>
double pwlintpMaxError(FUNCTION &&f, const Eigen::VectorXd &t);
```

that computes $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ when f is passed through the functor `f` (with evaluation operator `double operator () (double) const`) and the mesh $\mathcal{M}$ through its *sorted* vector $[t_0, t_1, \ldots, t_n]^\top \in \mathbb{R}^{n+1}$, $n \in \mathbb{N}$, of node positions, which also defines the interval $I := [t_0, t_n]$.

The maximum norm should be approximated by sampling in 10^5 equidistant points in I, see, e.g., [Lecture → Ex. 6.6.1.7].

SOLUTION for (6-1.c) → 6-1-3-0:sa.pdf ▲

(6-1.d) ⚂ (15 min.) [depends on Sub-problem (6-1.c)]

We consider the root functions $f(t) := t^\alpha$, $\alpha > 0$. Implement a C++ function

```
void cvgplotEquidistantMesh(const Eigen::VectorXd &alpha);
```

that uses MATPLOTLIBCPP to create log-log plots of the the error norms $\|f - I_{\mathcal{M}} f\|_{L^\infty(I)}$ as a function of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *equidistant meshes*. Use meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$, and values for α passed in `alpha`. Do not forget axis labels, plot title, and a meaningful legend. Save the plot in `cvgplotequidistant.png`.

Describe your observations qualitatively.

SOLUTION for (6-1.d) → 6-1-4-0:sa1.pdf ▲

(6-1.e) ⚂ (30 min.) [depends on Sub-problem (6-1.c)]

Create a C++ function

```
Eigen::VectorXd cvgrateEquidistantMesh(const Eigen::VectorXd
  &alpha);
```

that estimates rates of asymptotic algebraic convergence (→ [Lecture → § 6.2.2.9]) of $\|f - I_{\mathcal{M}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$ in terms of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *equidistant meshes* for the passed values of the parameter α and returns the estimates. These regression-based estimates should rely on meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$.

HIDDEN HINT 1 for (6-1.e) → 6-1-5-0:gm3h1.pdf

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

(6-1.f) ⚂ (15 min.) Tabulate and plot the empiric rate of asymptotic n-convergence (in maximum norm) of the piecewise linear interpolant of $f(t) = t^\alpha$, $0 < \alpha < 2$, on *equidistant meshes*. To conduct this experiment implement a C++ function

```
void testcvgEquidistantMesh();
```

that plots (use MATPLOTLIBCPP) and tabulates the measured convergence rates based on $\alpha = 0.05, 0.15, 0.25, \ldots, 2.95$. Save the plot in the file `cvgRateEquidistant.png`.

SOLUTION for (6-1.f) → 6-1-6-0:GradedMeshes3s.pdf ▲

(6-1.g) ⚂ (5 min.) In which mesh interval do you expect $|f - I_{\mathcal{M}} f|$ to attain its maximum?

HIDDEN HINT 1 for (6-1.g) → 6-1-7-0:GradedMeshes4hb.pdf

SOLUTION for (6-1.g) → 6-1-7-1:GradedMeshes4s.pdf ▲

(6-1.h) ⚃ (20 min.) [depends on Sub-problem (6-1.g)]

Compute ("on paper") the exact value of $\|f - I_{\mathcal{M}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$, $0 < \alpha < 2$. Compare the order of convergence obtained with the one observed numerically in (6-1.b).

HIDDEN HINT 1 for (6-1.h) → 6-1-8-0:GradedMeshes5h.pdf

SOLUTION for (6-1.h) → 6-1-8-1:GradedMeshes5s.pdf ▲

(6-1.i) ⚂ (20 min.) Since the interpolation error is concentrated in the left part of the domain, it seems reasonable to use a finer mesh only in this part.

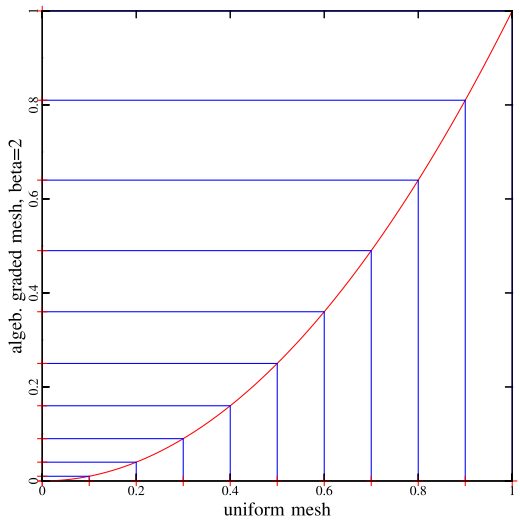


Fig. 54

A suitable choice is an **algebraically graded mesh**, defined as

$$\mathcal{G} = \left\{ t_j = \left(\frac{j}{n}\right)^{\beta}, j = 0, \ldots, n \right\} \tag{6.1.8}$$

for the **grading parameter** $\beta > 1$.

◁ Algebraically graded mesh for $\beta = 2$.

Realize a C++ function

```
Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
                                  const Eigen::VectorXd &beta);
```

that estimates rates of asymptotic algebraic convergence (→ [Lecture → § 6.2.2.9]) of $\|f - \mathsf{I}_{\mathcal{G}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$ in terms of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *graded meshes* for the passed values of the parameter α (argument `alpha`) and of the grading parameter $\beta > 0$ (argument `beta`). The function should return the measured rates as entries of a matrix, a column for each α-value, a row for each β-value. The regression-based estimates should rely on meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$.

SOLUTION for (6-1.i) → `6-1-9-0:.pdf` ▲

(6-1.j) ⚄ (20 min.) [depends on Sub-problem (6-1.i)]

When using algebraically graded meshes for the approximation of $t \mapsto t^\alpha$, $0 < \alpha < 2$, by means of piecewise linear interpolation, how do you have to choose the grading parameter $\beta = \beta(\alpha)$ as a function of α in order to recover an asymptotic algebraic convergence like $O(n^{-2})$ of $\|f - \mathsf{I}_{\mathcal{G}} f\|_{L^\infty(I)}$ for $\sharp\mathcal{G} \sim n \to \infty$?

To arrive at a conjecture rely on the function `cvgrateGradedMesh()` from Sub-problem (6-1.i) and write another function

```
void testcvgGradedMesh();
```

that calls `cvgrateGradedMesh()` for

$$\alpha \in \{0.05, 0.15, 0.25, \ldots, 2.95\}\,,$$
$$\beta \in \{0.1, 0.2, 0.3, \ldots, 2.0\}\,,$$

and creates a useful plot of the data (unsing MATPLOTLIBCPP) and stores it in the file `alphabeta.png`.

SOLUTION for (6-1.j) → `6-1-10-0:c1s.pdf` ▲

End Problem 6-1, 170 min.

Problem 6-2: Piecewise linear interpolation with knots different from nodes

In [Lecture → Ex. 5.1.0.15] we examined piecewise linear interpolation in the case where the interpolation nodes coincide with the abscissas of the corners of the interpolating polygon. However, that one is a very special situation. In this problem we consider a more general setting for piecewise linear interpolation.

Involves moderate implementation in C++.

We are given two ordered sets of real numbers to be read as points on the real line:

(I) the knots $x_0 < x_1 < x_2 < \cdots < x_n$,

(II) the (interpolation) nodes $t_0 < t_1 < \cdots < t_n$, $n \in \mathbb{N}$, satisfying $x_0 \le t_j \le x_n$, $j = 0, \ldots, n$.

The space of piecewise linear continuous functions with respect to the knot set $\mathcal{N} := \{x_j\}_{j=0}^n$ is defined as:

$$\mathcal{S}_{1,\mathcal{N}} := \{s \in C^0([x_0, x_n]) : \ s(t) = \gamma_j t + \beta_j \text{ for } t \in]x_{j-1}, x_j], \ \gamma_j, \beta_j \in \mathbb{R} \ i = 1, \ldots, n\} \ . \tag{6.2.1}$$

Compare [Lecture → Ex. 5.1.0.15] for the special case $x_j := t_j$.

On the function space $\mathcal{S}_{1,\mathcal{N}} \subset C^0(I)$, $I := [x_0, x_n]$, we consider the **interpolation problem**:

Find $s \in \mathcal{S}_{1,\mathcal{N}}$ such that $s(t_i) = y_i$, $i = 0, \ldots, n$, for given values $y_i \in \mathbb{R}$.

This fits the general framework of [Lecture → § 5.1.0.21].

Below we set $I_0 := [x_0, x_1]$, $I_j :=]x_{j-1}, x_{j+1}]$, $j = 1, \ldots, n-1$, $I_n := [x_{n-1}, x_n]$.

(6-2.a) ⊡ Show that the interpolation problem may not have a solution for some values y_j if a single interval $[x_j, x_{j+1}]$, $j = 0, \ldots, n-1$, contains more than 2 nodes t_i.

SOLUTION for (6-2.a) → 6-2-1-0:lips1.pdf ▲

(6-2.b) ⚂ Show that the interpolation problem can have a unique solution for any data values y_j *only if* each interval I_j, $j = 0, \ldots, n$, contains at least one node.

HIDDEN HINT 1 for (6-2.b) → 6-2-2-0:glgh1.pdf

SOLUTION for (6-2.b) → 6-2-2-1:lips2.pdf ▲

(6-2.c) ⚄ Show that the interpolation problem has a unique solution for all data values y_j if each of the closed intervals $]x_0, x_1[,]x_1, x_2[,]x_2, x_3[, \ldots,]x_{n-1}, x_n[$ contains at least one node t_j.

HIDDEN HINT 1 for (6-2.c) → 6-2-3-0:glgh1.pdf

SOLUTION for (6-2.c) → 6-2-3-1:lips3.pdf ▲

(6-2.d) ⚄ Implement a C++ function

```
Eigen::VectorXd tentBasCoeff(const Eigen::VectorXd &x,const
   Eigen::VectorXd &t,
                        const Eigen::VectorXd &y);
```

that returns the vector of values $s(x_j)$ of the interpolant s of the data points (t_i, y_i), $i = 0, \ldots, n$ in $\mathcal{S}_{1,\mathcal{N}}$ in the knots x_j, $j = 0, \ldots, n$. The argument vectors `x`, `t`, and `y` pass the x_j, t_j, and values y_j, respectively.

The function should first check whether the condition formulated in Sub-problem (6-2.c) is satisified. You must not take for granted that knots or nodes are sorted already.

HIDDEN HINT 1 for (6-2.d) → 6-2-4-0:glgh1.pdf

SOLUTION for (6-2.d) → 6-2-4-1:glgs1.pdf ▲

(6-2.e) ⚄ Implement a C++ class

```
class PwLinIP {
public:
  PwLinIP(const Eigen::VectorXd &x, const Eigen::VectorXd &t,
     const Eigen::VectorXd &y);
  double operator()(double arg) const;
private:
  ...
};
```

that realizes an interpolator class in the spirit of [Lecture → Rem. 5.1.0.11]. The meanings of the arguments of the constructor are the same as for the function `tentBasCoeff` from Sub-problem (6-2.d). Pay attention to the efficient implementation of the evaluation operator!

HIDDEN HINT 1 for (6-2.e) → 6-2-5-0:lipx1.pdf

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

SOLUTION for (6-2.e) → 6-2-5-2:dsfg1.pdf ▲

(6-2.f) ⚄ Implement a C++ code that creates plots of the cardinal basis functions for interpolation in $\mathcal{S}_{1,\mathcal{N}}$

- with know set $\mathcal{N} := \{0,1,2,3,\ldots,9,10\}$
- and for interpolation nodes $t_0 := 0$, $t_j = j - \frac{1}{2}$, $j = 1,\ldots,10$.

Recall that the cardinal basis function $b_k \in \mathcal{S}_{1,\mathcal{N}}$ is characterized by the conditions:

$$b_k(t_j) = \delta_{kj}\,, \quad k,j = 0,\ldots,10 \tag{6.2.6}$$

SOLUTION for (6-2.f) → 6-2-6-0:glgs1.pdf ▲

End Problem 6-2

Problem 6-3: Adaptive polynomial interpolation

In [Lecture → Section 6.2.3] we have seen that the *a priori* placement of interpolation nodes is key to a good approximation by a polynomial interpolant. This problem deals with an *a posteriori* adaptive strategy that controls the placement of interpolation nodes depending on the interpolant. It employs a **greedy algorithm** to grow the node set based on an intermediate interpolant.
This strategy has recently gained prominence for a wide range of approximation problems, see [Tem08].

Considerable implementation in C++ is requested in this problem.

A description of the greedy algorithm for adaptive polynomial interpolation is as follows:

Given a function $f : [a,b] \mapsto \mathbb{R}$ one starts $\mathcal{T}_0 := \{\frac{1}{2}(b+a)\}$. Based on a fixed finite set $\mathcal{S} \subset [a,b]$ of *sampling points* one augments the set of nodes according to

$$\mathcal{T}_{n+1} = \mathcal{T}_n \cup \left\{ \operatorname*{argmax}_{t\in\mathcal{S}} |f(t) - \mathrm{I}_{\mathcal{T}_n}(t)| \right\} , \tag{6.3.1}$$

where $\mathrm{I}_{\mathcal{T}_n}$ is the polynomial interpolation operator (→ [Lecture → Cor. 5.2.2.8]) for the node set $\mathcal{T}_n$, until the relative interpolation error (in maximum norm) drops below a prescribed tolerance $\texttt{tol} > 0$:

$$\max_{t\in\mathcal{S}} |f(t) - \mathrm{I}_{\mathcal{T}_n}(t)| \le \texttt{tol} \cdot \max_{t\in\mathcal{S}} |f(t)| . \tag{6.3.2}$$

Throughout this exercise you may rely on a C++ function

```
// IN: t:  vector of nodes t_0,...,t_n
// y:  vector of data y_0,...,y_n
// x:  vector of evaluation points x_1,...,x_N
// OUT: p:  interpolant evaluated at x
Eigen::VectorXd intpolyval(const Eigen::VectorXd& t,
                           const Eigen::VectorXd& y,
                           const Eigen::VectorXd& x);
```

that computes the values $p(x_i)$, $i = 1,\ldots,N$, where $p \in \mathcal{P}_n$ is the global polynomial interpolant through the data points (t_j, y_j), $j = 0,\ldots,n$, see [Lecture → Code 5.2.3.7].

C++ code 6.3.3: Function `intpolyval()`

```
2  Eigen::VectorXd intpolyval(const Eigen::VectorXd& t, const Eigen::VectorXd& y,
3                             const Eigen::VectorXd& x) {
4    const unsigned int
5        n = t.size(), // no. of interpolation nodes = deg. of polynomial
             −1
6        N = x.size(); // no. of evaluation points
7
8    Eigen::VectorXd p = Eigen::VectorXd(N);
9
10   // Precompute the weights λ_i with effort O(n^2)
11   Eigen::VectorXd lambda(n);
12   for (unsigned int k = 0; k < n; ++k) {
13     // little workaround: cannot subtract a vector from a scalar
14     // -> multiply scalar by vector of ones
15     lambda(k) =
```

```
16          1. /
17          ((t(k) * Eigen::VectorXd::Ones(k) - t.head(k)).prod() *
18           (t(k) * Eigen::VectorXd::Ones(n - k - 1) - t.tail(n - k - 1)).prod());
19    }
20    // Compute quotient of weighted sums of λ_i/(t-t_i),
21    // effort O(n)
22    for (unsigned int i = 0; i < N; ++i) {
23      Eigen::VectorXd z = (x(i) * Eigen::VectorXd::Ones(n) - t);
24
25      // check if we want to evaluate at a node <-> avoid division by zero
26      double* ptr = std::find(z.data(), z.data() + n, 0.0);
27      if (ptr != z.data() + n) {  // if ptr = z.data + n = z.end no zero was
            found
28        p(i) = y(ptr - z.data());  // ptr - z.data gives the position of the
              zero
29      } else {
30        Eigen::VectorXd mu = lambda.cwiseQuotient(z);
31        p(i) = (mu.cwiseProduct(y)).sum() / mu.sum();
32      }
33    }
34    return p;
35  }
```

(6-3.a) ⚀ (60 min.) Write a C++ function

```
template <class Function>
Eigen::VectorXd adaptivepolyintp(Function&& f,
                                 double a, double b,
                                 double tol, unsigned int N);
```

that implements the algorithm described above.

The function arguments are: the functor object `f`, the interval bounds `a`, `b`, the relative tolerance `tol`, the number `N` of *equidistant* sampling points, that is

$$\mathcal{S} := \{a + (b-a)j/N : j = 0, \ldots, N\}\,, \tag{6.3.4}$$

and `t` will be used as return parameter for interpolation nodes (i.e. the final set $\mathcal{T}_n$). The type **Function** defines a function and is supposed to provide an operator `double operator() (double) const`. A suitable lambda function can satisfy this requirement. The function should return the final set of interpolation nodes as proposed by the greedy algorithm.

HIDDEN HINT 1 for (6-3.a) → `6-3-1-0:s1h1.pdf`

SOLUTION for (6-3.a) → `6-3-1-1:solfile.pdf` ▲

(6-3.b) ⚀ (15 min.) [depends on Sub-problem (6-3.a)]

Extend the function `adaptivepolyintp()` from Sub-problem (6-3.a) to

```
template <class Function>
  Eigen::VectorXd
    adaptivepolyintp(Function &&f,
      double a, double b,
      double tol, unsigned int N,
      std::vector<double> *errortab = nullptr);
```

so that it also reports (and returns in `errortab`) the quantity

$$\epsilon_n := \max_{t\in\mathcal{S}} |f(t) - \mathsf{T}_{\mathcal{T}_n}(t)| \tag{6.3.5}$$

for each intermediate node set $\mathcal{T}_n$.

SOLUTION for (6-3.b) → 6-3-2-0:solfile.pdf

▲

(6-3.c) ⚀ (30 min.) [depends on Sub-problem (6-3.b)]

For $f_1(t) := \sin(e^{2t})$ and $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$ write a C++ function

```
void plotInterpolationError();
```

that creates (employing MATPLOTLIBCPP) and stores (in `intperrplot.png`) a plot of ϵ_n versus n (the number of interpolation nodes). Choose axis scales that reveal the qualitative decay (types of convergence as given in [Lecture → Def. 6.2.2.7]) of this error norm as the number of interpolation nodes is increased. Use interval $[a,b] = [0,1]$, `N=1000` sampling points, tolerance `tol = 1e-6`.

SOLUTION for (6-3.c) → 6-3-3-0:solfile.pdf

▲

References

[Tem08] V. N. Temlyakov. "Greedy approximation". In: *Acta Numer.* 17 (2008), pp. 235–409. DOI: 10.1017/S0962492906380014 (cit. on p. 205).

End Problem 6-3 , 105 min.

Problem 6-4: Chebyshev polynomials and their properties

Chebyshev polynomials as defined in [Lecture → Def. 6.2.3.3] are a sequence of orthogonal polynomials, which can be defined recursively, see [Lecture → Thm. 6.2.3.4]. In this problem we will examine these polynomials and a few of their many properties.

Involves a little implementation in C++

Let $T_n \in \mathcal{P}_n$ be the n-th Chebyshev polynomial, as defined in [Lecture → Def. 6.2.3.3] and $\xi_0^{(n)}, \ldots, \xi_{n-1}^{(n)}$ be the n zeros of T_n, the Chebychev nodes. According to [Lecture → (6.2.3.10)], these are given by:

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\,\pi\right), \quad j = 0, \ldots, n-1 . \tag{6.4.1}$$

Recall the notion of an "almost" inner product.

Definition [Lecture → Def. 6.3.1.1]. (Semi-)inner product [NS02, Sect. 4.4]

Let V be a vector space over the field $\mathbb{K}$. A mapping $b : V \times V \to \mathbb{K}$ is called an **inner product** on V, if it satisfies

(i) b is linear in the first argument: $b(\alpha v + \beta w, u) = \alpha b(v,u) + \beta b(w,u)$ for all $\alpha, \beta \in \mathbb{K}$, $u, v, w \in V$,
(ii) b is (anti-)symmetric: $b(v,w) = \overline{b(w,v)}$ ($\overline{} \hat{=}$ complex conjugation),
(iii) b is positive definite: $v \neq 0 \;\Leftrightarrow\; b(v,v) > 0$.

b is a semi-inner product, if it still complies with (i) and (ii), but is only positive semi-definite: $b(v,v) \geq 0$ for all $v \in V$.

We define the family of discrete L^2 semi-inner products [Lecture → (6.3.2.4)]:

$$(f,g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)}) g(\xi_j^{(n)}), \quad f,g \in C^0([-1,1]) . \tag{6.4.2}$$

We also define the special weighted L^2 inner product:

$$(f,g)_w := \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t) g(t)\,\mathrm{d}t \quad f,g \in C^0([-1,1]) . \tag{6.4.3}$$

(6-4.a) ⚀ (15 min.) Show that the weight function in (6.4.3) $w(t) := \left(1-t^2\right)^{-\frac{1}{2}}$ satisfies $\int_{-1}^{1} |w(t)|\,\mathrm{d}t < \infty$.

HIDDEN HINT 1 for (6-4.a) → 6-4-1-0:s0h1.pdf

SOLUTION for (6-4.a) → 6-4-1-1:.pdf ▲

(6-4.b) ⚁ (15 min.) [depends on Sub-problem (6-4.a)]

Show that the Chebyshev polynomials are an orthogonal family of polynomials with respect to the inner product defined in (6.4.3) in the sense of [Lecture → Def. 6.3.2.7]:

$$(T_k, T_\ell)_w = 0 \quad \text{for all} \quad k, \ell \in \mathbb{N}_0 , \quad k \neq \ell . \tag{6.4.4}$$

HIDDEN HINT 1 for (6-4.b) → 6-4-2-0:ChebPolyProperties1h.pdf

SOLUTION for (6-4.b) → 6-4-2-1:ChebPolyProperties1s.pdf ▲

Recall the concept of an orthonormal basis:

Definition [Lecture → Def. 6.3.1.14]. Orthonormal basis

A subset $\{b_1,\ldots,b_N\}$ of an N-dimensional vector space V with inner product ($\to$ [Lecture → Def. 6.3.1.1]) $(\cdot,\cdot)_V$ is an **orthonormal basis (ONB)**, if $\left(b_k,b_j\right)_V = \delta_{kj}$.

A basis of $\{b_1,\ldots,b_N\}$ of V is called **orthogonal**, if $\left(b_k,b_j\right)_V = 0$ for $k \neq j$.

The following is a surprising result.

Theorem 6.4.5. Discrete orthogonality of Chebychev polynomials

The family of polynomials $\{T_0,\ldots,T_n\}$ is an orthogonal basis [Lecture → Def. 6.3.1.14] of $\mathcal{P}_n$ with respect to the semi-inner product $(\,,\,)_{n+1}$ defined in (6.4.2).

(6-4.c) ⚀ (10 min.) Show that for every $n \in \mathbb{N}_0$ $(\cdot,\cdot)_{n+1}$ as defined in (6.4.2) is an inner product on the (real) vector space $\mathcal{P}_n$ of polynomials of degree $\leq n$.

SOLUTION for (6-4.c) → 6-4-3-0:sy.pdf ▲

(6-4.d) ⚁ (45 min.) Write a C++ function

```
bool checkDiscreteOrthogonality(unsigned int n);
```

meant to test the assertion of Thm. 6.4.5 numerically.

HIDDEN HINT 1 for (6-4.d) → 6-4-4-0:ChebPolyProperties2h.pdf

HIDDEN HINT 2 for (6-4.d) → 6-4-4-1:2h2.pdf

SOLUTION for (6-4.d) → 6-4-4-2:ChebPolyProperties2s.pdf ▲

(6-4.e) ⚂ (30 min.) Prove Thm. 6.4.5.

HIDDEN HINT 1 for (6-4.e) → 6-4-5-0:ChebPolyProperties3h.pdf

SOLUTION for (6-4.e) → 6-4-5-1:ChebPolyProperties3s.pdf ▲

(6-4.f) ⚃ (30 min.) [depends on Sub-problem (6-4.c) and Thm. 6.4.5]

Given $n,m \in \mathbb{N}$, $m \leq n$, and a function $f \in C^0([-1,1])$, find an expression for the best approximant $q_m \in \mathcal{P}_m$ of f in the discrete L^2-norm:

$$q_m = \operatorname*{argmin}_{p\in\mathcal{P}_m} \|f-p\|_{n+1}\,, \tag{6.4.9}$$

where $\|\cdot\|_{n+1}$ is the norm induced by the scalar product $(\cdot,\cdot)_{n+1}$ from (6.4.2). You should represent q_m through a Chebychev expansion of the form:

$$q_m = \sum_{j=0}^{m} \alpha_j T_j\,, \quad \alpha_j \in \mathbb{R}\,, \tag{6.4.10}$$

see also [Lecture → Section 6.2.3.3].

HIDDEN HINT 1 for (6-4.f) → 6-4-6-0:ChebPolyProperties4h.pdf

SOLUTION for (6-4.f) → 6-4-6-1:ChebPolyProperties4s.pdf ▲

(6-4.g) ⚁ (10 min.) [depends on Sub-problem (6-4.f)]

Predict q_m as defined through (6.4.9) in the case $m = n$.

SOLUTION for (6-4.g) → 6-4-7-0:sw.pdf ▲

(6-4.h) ⚁ (45 min.) [depends on Sub-problem (6-4.f)]

Write a C++ function

```
template <typename Function>
  Eigen::VectorXd
    bestpolchebnodes(Function &&f,
                     unsigned int n, unsigned int m);
```

that returns the vector of coefficients α_j, $j = 0,\ldots,m$, in (6.4.10) given the function f. The degree m of the polynomial and the number $n+1$ of the Chebychev nodes entering the inner product are passed through the arguments `m` and `n`. The input `f` is a functor of signature `std::function<double(double)>`, for instance,

```
auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
```

SOLUTION for (6-4.h) → 6-4-8-0:ChebPolyProperties5s.pdf ▲

(6-4.i) ⚁ (30 min.) [depends on Sub-problem (6-4.h)]

We test the implementation of `bestpolchebnodes()` with the function $f(x) = \frac{1}{(5x)^2+1}$. To that end write a C++ function

```
std::vector<double> testBestPolyChebNodes(unsigned int n);
```

that takes the integer n as argument and both tabulates and returns the sequence of the approximation error norms $\|f - q_m\|_{L^\infty(]-1,1[)}$, $m = 0,\ldots,n$, q_m as in (6.4.9).

Approximate the supremum norm of the approximation error by sampling on an equidistant grid with 10^6 points.

HIDDEN HINT 1 for (6-4.i) → 6-4-9-0:ChebPolyProperties6h.pdf

SOLUTION for (6-4.i) → 6-4-9-1:ChebPolyProperties6s.pdf ▲

(6-4.j) ⚂ (30 min.) [depends on Sub-problem (6-4.f)]

Let $L_j \in \mathcal{P}_n$, $j = 0,\ldots,n$, be the Lagrange polynomials [Lecture → § 5.2.2.3] associated with the Chebychev nodes $t_j := \xi_j^{(n+1)}$ in $[-1,1]$. Show that:

$$L_j(t) = \frac{1}{n+1} + \frac{2}{n+1}\sum_{l=1}^{n} T_l(\xi_j^{(n+1)})T_l(t) \quad , \quad j = 0,\ldots,n\,. \tag{6.4.15}$$

HIDDEN HINT 1 for (6-4.j) → 6-4-10-0:ChebPolyProperties7h.pdf

SOLUTION for (6-4.j) → 6-4-10-1:ChebPolyProperties7s.pdf ▲

References

[NS02] K. Nipp and D. Stoffer. *Lineare Algebra*. 5th ed. Zürich: vdf Hochschulverlag, 2002 (cit. on p. 208).

End Problem 6-4 , 260 min.

Problem 6-5: Chebychev interpolation of analytic functions

This problem concerns Chebychev interpolation as discussed in [Lecture → Section 6.2.3] and takes a close look at the interpolation error estimates derived for analytic interpolands in [Lecture → Section 6.2.2.3].

This problem is closely connected to [Lecture → Rem. 6.2.3.26] and involves implementation in C++ based on EIGEN. You are expected to be able to compute with complex numbers.

Using techniques from complex analysis, notably the residue theorem [Lecture → Thm. 6.2.2.50], in class we derived an expression for the interpolation error [Lecture → (6.2.2.57)] and from it an error bound [Lecture → (6.2.2.58)], as much sharper alternative to [Lecture → Thm. 6.2.2.15] and [Lecture → Lemma 6.2.2.20] for *analytic* interpolands. The bound tells us that for all $t \in [a,b]$

$$|f(t) - \mathsf{L}_{\mathcal{T}} f(t)| \leq \left| \frac{w(x)}{2\pi i} \int_\gamma \frac{f(z)}{(z-t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a\leq\tau\leq b}|w(\tau)|}{\min_{z\in\gamma}|w(z)|} \frac{\max_{z\in\gamma}|f(z)|}{d([a,b],\gamma)} ,$$

where $d([a,b],\gamma)$ is the geometric distance of the integration contour $\gamma \subset \mathbb{C}$ from the interval $[a,b] \subset \mathbb{C}$ in the complex plane. The contour γ must be contractible in the domain D of analyticity of f and must wind around $[a,b]$ exactly once, see [Lecture → Fig. 217].

Now we consider the interval $[-1,1]$. Following [Lecture → Rem. 6.2.3.26], our task is to find an upper bound for this expression, in the case where f possesses an analytical extension to a complex neighbourhood of $[-1,1]$.

For the analysis of the Chebychev interpolation of analytic functions we used the elliptical contours, also known as **Bernstein ellipses**, see [Lecture → Fig. 231],

$$\gamma_\rho(\theta) := \cos(\theta - i\log(\rho)) , \quad \forall 0 \leq \theta \leq 2\pi , \quad \rho > 1 . \tag{6.5.1}$$

(6-5.a) ⚂ (20 min.) Implement a C++ function

```
double lengthEllipse(double rho, unsigned int N);
```

that approximates the length of the ellipse γ_ρ by sampling (6.5.1) for $N \in \mathbb{N}$ (equidistant) arguments θ and returns the result.

SOLUTION for (6-5.a) → 6-5-1-0:sa.pdf ▲

From now on we consider the S-curve function (the logistic function):

$$f(t) := \frac{1}{1+e^{-3t}} , \quad t \in \mathbb{R} . \tag{6.5.3}$$

(6-5.b) ⚂ (20 min.) Determine the maximal domain of analyticity $D \subset \mathbb{C}$ of the extension of f to the complex plane $\mathbb{C}$.

HIDDEN HINT 1 for (6-5.b) → 6-5-2-0:s2h1.pdf

SOLUTION for (6-5.b) → 6-5-2-1:solfile.pdf ▲

(6-5.c) ⚄ (15 min.) [depends on Sub-problem (6-5.b)]

Find the least upper bound for the set of ellipse parameters

$$\{\rho > 1 : \gamma_\rho \subset D\} ,$$

where $D \subset \mathbb{C}$ is the domain of analyticity found in Sub-problem (6-5.b).

HIDDEN HINT 1 for (6-5.c) → 6-5-3-0:s2ah1.pdf

SOLUTION for (6-5.c) → 6-5-3-1:s2a.pdf ▲

(6-5.d) ⚀ (20 min.) [depends on Sub-problem (6-5.c)]

Write a C++ function

```
std::pair<double, double> bestBound();
```

that computes an approximation M of

$$\min_{\rho>1} \frac{|\gamma_\rho| \max_{z\in\gamma_\rho} |f(z)|}{\pi\, d([-1,1],\gamma_\rho)}\,, \tag{6.5.6}$$

by sampling 1000 equidistant ρ-values in the interval $(1,\rho_{\max}]$, where $\rho_{\max}$ is the value obtained in Sub-problem (6-5.c). For sampling along the integration path γ_ρ use 1000 points corresponding to equidistant θ-values, *cf.* Sub-problem (6-5.a). The function should return the pair $(M,\rho) \in \mathbb{R}^2$, where ρ stands for the ρ-value for which the minimum is attained.

The distance of $[a,b]$ from γ_ρ is formally defined as

$$d([a,b],\gamma) := \inf\{|z-t| \mid z \in \gamma, t \in [a,b]\}\,. \tag{6.5.7}$$

HIDDEN HINT 1 for (6-5.d) → prbfile.pdf

SOLUTION for (6-5.d) → 6-5-4-1:solfile.pdf ▲

(6-5.e) ⚀ (15 min.) [depends on Sub-problem (6-5.d)]

Based on the result of Sub-problem (6-5.d) and [Lecture → Rem. 6.2.3.26], give an "optimal" bound for

$$\|f - L_n f\|_{L^\infty([-1,1])},$$

where $L_n : C^0([-1,1]) \to \mathcal{P}_n$ is the operator of Chebychev interpolation, that is, polynomial Lagrange interpolation based on Chebychev nodes [Lecture → (6.2.3.10)], on $[-1,1]$ into the space of polynomials of degree $\le n$.

SOLUTION for (6-5.e) → 6-5-5-0:solfile.pdf ▲

(6-5.f) ⚄ (20 min.) [depends on Sub-problem (6-5.e)]

Implement a C++ function

```
std::pair<std::vector<double>, std::vector<double> >
compareErrorAndBound(unsigned int n_max);
```

that computes and tabulates

1. estimates for $\|f - L_n f\|_{L^\infty([-1,1])}$ obtained by sampling in 10000 equidistant points $\in [-1,1]$,
2. the bounds derived in Sub-problem (6-5.e),

for polynomial degrees $n = 4,\ldots,n_{\max}$, where $n_{\max}$ is passed through the argument `n_max`.

You may rely on the provided function `intpolyval()` for the evaluation of a polynomial interpolant at many points.

HIDDEN HINT 1 for (6-5.f) → 6-5-6-0:s5h1.pdf

SOLUTION for (6-5.f) → 6-5-6-1:solfile.pdf

▲

(6-5.g) ⚄ (20 min.) Rely on pullback to $[-1,1]$ to discuss how the error bounds in [Lecture → (6.2.3.28)] will change qualitiatively when we consider Chebychev interpolation on $[-a,a], a > 0$, instead of $[-1,1]$, whilst keeping the function f fixed.

SOLUTION for (6-5.g) → 6-5-7-0:solfile.pdf ▲

End Problem 6-5 , 130 min.

Problem 6-6: Polynomial Best Approximation

The famous Jackson Theorem [Lecture → Thm. 6.2.1.11] gives detailed information about the best approximation of C^r-functions by means of polynomials. In this exercise we recall that theorem and study best-approximation errors in general.

Purely theoretical problem related to [Lecture → Section 6.2.1]

Throughout this problem we write $\mathcal{P}_n$ for the vector space of uni-variate polynomials of degree $\leq n$ [Lecture → Section 5.2.1].

(6-6.a) ⚀ (15 min.) The following is the assertion of Jackson's theorem of [Lecture → Thm. 6.2.1.11] with some parts missing. Fill either n or r in the green boxes and supplement the right subscript for the norm in the white boxes.

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{\square} \leq \left(1+\frac{\pi^2}{2}\right)^{\square} \frac{(\square-\square)!}{\square!} \left\|f^{(\square)}\right\|_{\square}$$

for all $n, r \in \mathbb{N}$, $n \geq r$, and $f \in C^r([-1,1])$. Here $\mathcal{P}_n$ designates the space of univariate polynomials of degree $\leq n$ and $f^{(k)}$ is a short notation for the k-th derivative of a function.

SOLUTION for (6-6.a) → 6-6-1-0:s1.pdf ▲

(6-6.b) ⚄ (20 min.) For $n \in \mathbb{N}$ we abbreviate

$$E_n(f) := \inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \quad \text{for} \quad f \in C^0([-1,1]) \,. \tag{6.6.2}$$

Which of the following assertions on E_n are correct, which are wrong? Throughout, f and g denote arbitrary continuous functions on $[-1,1]$ and $n \in \mathbb{N}$

		correct	wrong
(i)	$E_n(f+g) \leq E_n(f) + E_n(g)$	○	○
(ii)	$E_n(\lambda f) = \lambda E_n(f)$ for all $\lambda \in \mathbb{R}$	○	○
(iii)	E_n is a norm on $C^0([-1,1])$	○	○
(iv)	$E_n(f) - E_n(g) \leq E_n(f-g)$	○	○

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

SOLUTION for (6-6.b) → 6-6-2-1:s2.pdf ▲

End Problem 6-6, 35 min.

Problem 6-7: Piecewise cubic Hermite Interpolation of Functions

The variant of piecewise cubic Hermite interpolation studied in [Lecture → Section 6.6.2] was a generalized interpolation scheme in the sense that it required point values of the derivative of the interpoland. Conversely, for piecewise cubic Hermite data interpolation we had to use reconstructed slopes. This problem examines the use of reconstructed slopes also for the purpose of function approximation.

Relies on [Lecture → Section 5.3.3] and [Lecture → Section 6.6.2] and involves C++ coding.

Piecewise cubic Hermite interpolation of a function $f \in C^1([a,b])$ with exact slopes on a mesh

$$\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$$

was defined in [Lecture → Def. 6.6.2.1]. For $f \in C^4([a,b])$ it enjoys h-convergence with rate 4 as we have seen in [Lecture → Exp. 6.6.2.2] and which is stated in [Lecture → Thm. 6.6.2.3].

Theorem [Lecture → Thm. 6.6.2.3]. Convergence of approximation by cubic Hermite interpolation

Let s be the cubic Hermite interpolant of $f \in C^4([a,b])$ on a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ according to [Lecture → Def. 6.6.2.1]. Then

$$\|f - s\|_{L^\infty([a,b])} \leq \frac{1}{4!} h_{\mathcal{M}}^4 \left\| f^{(4)} \right\|_{L^\infty([a,b])} ,$$

with the meshwidth $h_{\mathcal{M}} := \max_j |x_j - x_{j-1}|$.

Now we consider cases, where perturbed or reconstructed slopes are used. For instance, this was done in the context of monotonicity preserving piecewise cubic Hermite interpolation as discussed in [Lecture → Section 5.3.3.2].

(6-7.a) ⚄ (30 min.) Assume that piecewise cubic Hermite interpolation is based on perturbed slopes, that is, the piecewise cubic function s on the mesh $\mathcal{M}$ satisfies:

$$s(x_j) = f(x_j) \quad , \quad s'(x_j) = f'(x_j) + \delta_j \, , \tag{6.7.1}$$

where the perturbations $\delta_j \in \mathbb{R}$ may depend on $\mathcal{M}$, too.

Now we consider a sequence of meshes with meshwidth $h_{\mathcal{M}} \to 0$. Which rate of asymptotic h-convergence of the maximum norm $\|f - s\|_{L^\infty(]a,b[)}$ of the approximation error can be expected, if we know that for all j that

$$|\delta_j| = O(h_{\mathcal{M}}^\beta) \quad \text{as} \quad h_{\mathcal{M}} \to 0 \quad \text{for some} \quad \beta \in \mathbb{N}_0 \, , \tag{6.7.2}$$

for mesh-width $h_{\mathcal{M}} \to 0$.

HIDDEN HINT 1 for (6-7.a) → 6-7-1-0:s1h1.pdf

SOLUTION for (6-7.a) → 6-7-1-1:s1.pdf ▲

(6-7.b) ⚄ We define a "strange" piecewise cubic interpolant:

Definition 6.7.3. Flat-point piecewise cubic interpolant

Given $f \in C^0([a,b])$ and a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ the flat-point piecewise cubic interpolant $s : [a,b] \to \mathbb{R}$ is defined as

$$s_{|[x_{j-1},x_j]} \in \mathcal{P}_3 \,, \quad j = 1,\ldots,m \quad , \quad s(x_j) = f(x_j) \,, \quad s'(x_j) = 0 \,, \quad j = 0,\ldots,m \,.$$

Implement a C++ function

```
template <typename Function>
Eigen::VectorXd fppchip(Function &&f,
  const Eigen::VectorXd &x, const Eigen::VectorXd &t);
```

that returns the vector $[s(t_k)]_{k=1}^N$, where

- the argument `x` passes the $m+1$ (ordered) nodes of a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$,
- s is the flat-point piecewise cubic interpolant of f according to Def. 6.7.3,
- the (sorted!) points $t_k \in [x_0, x_m]$ are supplied as components of the vector `t`.

Remark. Please refer to [Lecture → Thm. 5.3.3.16] to understand the motivation for introducing the "flat-point piecewise cubic interpolant". This is the only linear interpolation operator into C^1-functions that is locally monotonicity preserving.

SOLUTION for (6-7.b) → 6-7-2-0:sx2.pdf ▲

(6-7.c) ⚀ (20 min.) [depends on Sub-problem (6-7.b)]

For the function $f(t) = \dfrac{1}{1+t^2}$ and the interval $[-5,5]$ empirically determine (qualitatively and quantitatively) the asymptotic convergence (in terms of meshwidth $h_{\mathcal{M}} \to 0$) of the flat-point piecewise cubic interpolants according to Def. 6.7.3 on sequences of *equidistant meshes*.

To that end study the maximum norm of the interpolation error in numerical experiments. Approximate those maximum norms by sampling in 10000 equidistant points. Implement a C++ function

```
std::vector<double> fppchipConvergence();
```

that tabulates and returns the error norms for meshes with $4, 8, 16, 32, 64, 128, 256, 512$ nodes.

SOLUTION for (6-7.c) → 6-7-3-0:sz3.pdf ▲

(6-7.d) ⚀ (10 min.) [depends on Sub-problem (6-7.a) and Sub-problem (6-7.c)]

Invoke the result of Sub-problem (6-7.a) to give a theoretical explanation for the observations made in Sub-problem (6-7.c).

SOLUTION for (6-7.d) → 6-7-4-0:s3a.pdf ▲

We now define a piecewise cubic Hermite interpolant with slopes reconstructed from sampled function values.

Definition 6.7.6. Piecewise cubic Hermite interpolant with reconstructed slopes

Given $f \in C^0([a,b])$ and a mesh $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_{m-1} < x_m = b\}$ the piecewise cubic Hermite interpolant with reconstructed slopes $s : [a,b] \to \mathbb{R}$ is defined as

$$s_{|[x_{j-1},x_j]} \in \mathcal{P}_3 \,, \quad j = 1,\ldots,m \quad , \quad s(x_j) = f(x_j) \,, \quad j = 0,\ldots,m \,,$$

$$s'(x_j) = \begin{cases} \dfrac{-f(x_2) + 4f(x_1) - 3f(x_0)}{2h} & \text{for } j = 0 \,, \\ \dfrac{f(x_{j+1}) - f(x_{j-1})}{2h} & \text{for } j = 1,\ldots,m-1 \,, \\ \dfrac{3f(x_m) - 4f(x_{m-1}) + f(x_{m-2})}{2h} & \text{for } j = m \,. \end{cases}$$

A class for piecewise cubic Hermite interpolation of continuous functions according to Def. 6.7.6 is defined as follows:

```
class CubicHermiteInterpolant {
public:
  template <typename Function>
  CubicHermiteInterpolant(Function &&f, const Eigen::VectorXd &t);
  virtual ~CubicHermiteInterpolant(void) = default;
  Eigen::VectorXd eval(const Eigen::VectorXd &x) const;
private:
  Eigen::VectorXd t_; // sorted mesh nodes
  Eigen::VectorXd y_; // function values at mesh nodes
  Eigen::VectorXd c_; // slopes at mesh nodes
};
```

The constructor expects a functor `f` with signature `std::function<double(double)>` and a (sorted!) vector `t` of nodes. Its evaluation method `CubicHermiteInterpolant::eval()` returns $s(x_k)$ for the (sorted!) components of the vector passed in `x`. Here $s \in C^1([a,b])$ is the piecewise cubic Hermite interpolant as defined in Def. 6.7.6.

(6-7.e) ⚀ (25 min.) Provide an implementation of the class **CubicHermiteInterpolant**.

SOLUTION for (6-7.e) → 6-7-5-0:s4.pdf ▲

(6-7.f) ⚁ (10 min.) [depends on Sub-problem (6-7.e)]

Based on the class **CubicHermiteInterpolant** realize a C++ function

```
std::vector<double> rspchipConververgence();
```

that answers the questions of Sub-problem (6-7.c) for the interpolant according to Def. 6.7.6.

SOLUTION for (6-7.f) → 6-7-6-0:.pdf ▲

(6-7.g) ⚂ (30 min.) Write $I_H : C^0([a,b]) \to C^1([a,b])$ for the piecewise cubic Hermite interpolation operator with reconstructed slopes introduced in Def. 6.7.6.

Based on the result of Sub-problem (6-7.a) prove that

$$f \in C^3([a,b]) \quad \Rightarrow \quad \exists C > 0: \quad \|f - I_H f\|_{L^\infty(]a,b[)} \leq C h_{\mathcal{M}}^3 \,,$$

with $C > 0$ independent of the mesh and $h_{\mathcal{M}}$ standing for the mesh width of $\mathcal{M}$.

HIDDEN HINT 1 for (6-7.g) → 6-7-7-0:s6h1.pdf

SOLUTION for (6-7.g) → 6-7-7-1:.pdf ▲

End Problem 6-7 , 125 min.

Problem 6-8: Monomial representation of Chebychev polynomials

Chebychev polynomials are important both for theoretical analysis and algorithm design. For instance, as we have seen in [Lecture → Section 6.2.3.3] Chebyshev interpolants should internally be represented through their Chebychev expansion. In this problem we will study an algorithm for converting Chebychev expansions into monomial expansions.

This problem relies about elementary facts about Chebychev polynomials as introduced in [Lecture → Section 6.2.3.1].

On $[-1,1]$ the n-th Chebychev polynomial T_n, $n \in \mathbb{N}_0$ is defined as [Lecture → Def. 6.2.3.3]

$$T_n(t) := \cos(n \arccos t) \quad -1 \le t \le 1 . \tag{6.8.1}$$

That T_n is a polynomial of degree $\le n$ is clear from the 3-term recursion

$$T_{n+1}(t) = 2t\,T_n(t) - T_{n-1}(t) \quad , \quad T_0 \equiv 1 , \quad T_1(t) = t , \quad n \in \mathbb{N} . \qquad [\text{Lecture} \rightarrow (6.2.3.5)]$$

The set $\{T_0, \dots, T_n\}$ forms a basis of $\mathcal{P}_n$.

(6-8.a) ⚄ (30 min.) Below is an incomplete implementation of a C++ function

```
Eigen::VectorXd chebexpToMonom(const Eigen::VectorXd &a);
```

that expects the coefficients $a_j \in \mathbb{R}$, $j = 0, \dots, n$, of the Chebyshev expansion

$$p(t) = \sum_{j=0}^{n} a_j T_j(t) , \quad t \in \mathbb{R} ,$$

of a polynomial $p \in \mathcal{P}_n$ to be passed in the vector `a` and returns the coefficients r_j, $j = 0, \dots, n$ with respect to the monomial basis,

$$p(t) = r_0 + r_1 t + r_2 t^2 + \cdots + r_n t^n ,$$

collected in a vector $[r_0, r_1, \dots, r_n]^\top$.

Fill in the missing pieces of the following code, which relies on [Lecture → (6.2.3.5)].

```
Eigen::VectorXd chebexpToMonom(const Eigen::VectorXd &a) {
  const int n = a.size() -1; // degree of polynomial
  assert(n >= 0);
  Eigen::VectorXd r{ Eigen::VectorXd::Zero(n+1) };
  r[0] = A ;
  if (n > 0) {
    r[1] = B ;
    if (n > 1) {
      // two vectors temporarily storing the monomial coefficients
      // of Chebychev polynomials of two consecutive degrees
      std::array<Eigen::VectorXd, 2> c
       { Eigen::VectorXd::Zero(n+1), Eigen::VectorXd::Zero(n+1) };
      c[0][0] = 1.0; c[1][1] = 1.0;
      for (int j=2; j<= C ; ++j) {
        c[j%2][0] *= -1.0;
```

```
                for (int k= D ; k <= j; ++k)
                  c[j%2][k] = E *c[ F ][ G ] - c[j%2][k];
                r += H *c[j%2];
            }
        }
    }
    return r;
}
```

- A ≙ ☐
- B ≙ ☐
- C ≙ ☐
- D ≙ ☐
- E ≙ ☐
- F ≙ ☐
- G ≙ ☐
- H ≙ ☐

SOLUTION for (6-8.a) → 6-8-1-0:s1.pdf ▲

End Problem 6-8 , 30 min.

Problem 6-9: Finding domains of analyticity

The developments of [Lecture → Lemma 6.2.2.53], [Lecture → Rem. 6.2.3.26], and [Lecture → Section 6.5.3] make clear that knowledge about that maximal subdomain $D \subset \mathbb{C}$ of the complex plane to which an analytic function can be extended from its original real interval of definition is key to predicting the speed of exponential convergence of approximation schemes like Chebychev interpolation and trigonometric interpolation. In this problem we discuss the analytic extension of 1-periodic functions.

Depends on [Lecture → Rem. 6.2.2.67] and is related to [Lecture → § 6.5.3.16]. This problem does not ask for C++ coding.

In order to analyze the trigonometric functions occurring in this problem, the following identities can be useful:

$$\begin{aligned} \sin(x+\imath y) &= \sin(x)\cosh(y) + \imath\cos(x)\sinh(y) \quad \forall x,y \in \mathbb{R}\,, \\ \cos(x+\imath y) &= \cos(x)\cosh(y) - \imath\sin(x)\sinh(y) \quad \forall x,y \in \mathbb{R}\,, \\ \sin^2(z) + \cos^2(z) &= 1 \quad \forall z \in \mathbb{Z}\,. \end{aligned} \tag{6.9.1}$$

The following result gives domains of analyticity of special functions:

Theorem 6.9.2. domains of analyticity of special functions

- *Every polynomial is analytic on* $\mathbb{C}$.
- *Every rational function, that is, a quotient of two polynomials, is analytic in the complement of the set of zeros of its denominator.*
- *The functions* $\exp$, $\sin$, *and* $\cos$ *are analytic on* $\mathbb{C}$.
- *The functions* $\log : \mathbb{R}^+ \to \mathbb{R}$ *and* $\sqrt{}: \mathbb{R}_0^+ \to \mathbb{R}$ *can be extended to analytic functions on* $\mathbb{C} \setminus \mathbb{R}_0^-$.

(6-9.a) ⚄ (15 min.) We consider the 1-periodic function

$$f(t) = b\log(a + \sin(2\pi t))\,, \quad t \in \mathbb{R}\,, \quad b > 0,\ a > 1\,. \tag{6.9.3}$$

Determine the largest possible subset D of $\mathbb{C}$ to which f can be extended analytically.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \boxed{}\,, \\ \mathrm{Im}(z) \in \boxed{} \end{array} \right\}. \tag{6.9.4}$$

SOLUTION for (6-9.a) → 6-9-1-0:s1.pdf ▲

(6-9.b) ⚄ (15 min.) [depends on Sub-problem (6-9.a)]

We consider the 1-periodic function

$$f(t) = a\log(b - \cos(2\pi t))\,, \quad t \in \mathbb{R}\,, \quad b > 1,\ a > 1\,. \tag{6.9.5}$$

Determine the largest possible subset D of $\mathbb{C}$ to which f can be extended analytically.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{l} \mathrm{Re}(z) \in \boxed{}\,, \\ \mathrm{Im}(z) \in \boxed{} \end{array} \right\}.$$

SOLUTION for (6-9.b) → 6-9-2-0:s2.pdf ▲

(6-9.c) ⚄ (15 min.) We consider the 1-periodic function

$$f(t) = a\sqrt{b - \sin(2\pi t)}\,, \quad t \in \mathbb{R}\,, \quad b > 1,\ a > 1\,, \tag{6.9.6}$$

where $\sqrt{\cdot}$ is the main branch of the square root.

Determine the largest possible subset D of $\mathbb{C}$ to which f can be extended analytically.

$$D = \mathbb{C} \setminus \left\{ z \in \mathbb{C} : \begin{array}{ll} \mathrm{Re}(z) \in & \square \\ \mathrm{Im}(z) \in & \square \end{array} \right\}\,.$$

SOLUTION for (6-9.c) → 6-9-3-0:s2.pdf ▲

End Problem 6-9, 45 min.

Problem 6-10: Convergence of Chebychev Interpolants

In [Lecture → Section 6.2.3.2] we saw asymptotic error estimates for Chebychev interpolation, which means Lagrange interpolation in Chebychev nodes. We could establish interpolation error estimates in maximum norm for interpolands of finite smoothness and analytic interpolands. We review these results in this exercise.

You have to be familiar with the contents of [Lecture → Section 6.2.2.1], [Lecture → Section 6.2.2.3] and [Lecture → Section 6.2.3.2].

Throughout we write $I_C^n : C^0[0,1] \to \mathcal{P}_n$, $n \in \mathbb{N}_0$, for the Chebychev interpolation operator on $[0,1]$, which amounts to polynomial Lagrange interpolation in the $n+1$ Chebychev nodes [Lecture → (6.2.3.10)]. For the maximum norm of the **interpolation error** we introduce the notation

$$\epsilon_n(f) = \|f - I_C^n f\|_{\infty,[0,1]} , \quad f \in C^0([0,1]) , \quad n \in \mathbb{N}_0 . \tag{6.10.1}$$

(6-10.a) ⚂ (20 min.) We distinguish two basic types of asymptotic convergence of $\epsilon_n(f) \to 0$ for $n \to \infty$: **algebraic convergence** and **exponential convergence** [Lecture → Def. 6.2.2.7].

Complete the missing parts of the following statements to render them correct, complete, and precise:

❶ For given $f \in C^0([0,1])$ the quantity $\epsilon_n(f)$ enjoys asymptotic **algebraic convergence**, if

$\epsilon_n(f)$ [ ] for [ ] .

❷ For given $f \in C^0([0,1])$ the quantity $\epsilon_n(f)$ enjoys asymptotic **exponential convergence**, if

$\epsilon_n(f)$ [ ] for [ ] .

SOLUTION for (6-10.a) → 6-10-1-0:aecs0.pdf

▲

(6-10.b) ⚂ (25 min.)

We apply Chebychev interpolation to the family of functions

$$f_a(t) := |\log(t+a)| , \quad t \in [0,1] , \quad a > 0 .$$

(i) Predict the type of asymptotic convergence (algebraic or exponential according to [Lecture → Def. 6.2.2.7]) of

$$\epsilon_n(f_a) = \|f_a - I_C^n f_a\|_{\infty,[0,1]}$$

for $n \to \infty$ and for the different values of a given in the following table.

(ii) For those a-values for which you guessed exponential convergence, rank the cases according to the speed of convergence starting with "1" for the fastest convergence.

	exponential convergence	algebraic convergence	rank (exp. cvg. only)
$a = \frac{1}{4}$	◯	◯	
$a = \frac{1}{2}$	◯	◯	
$a = \frac{3}{2}$	◯	◯	
$a = 2$	◯	◯	
$a = 3$	◯	◯	

HIDDEN HINT 1 for (6-10.b) → 6-10-2-0:abslogh.pdf

SOLUTION for (6-10.b) → 6-10-2-1:.pdf

▲

(6-10.c) ⚄ (25 min.) What is the domain of analyticity $D_f \subset \mathbb{C}$ of the function

$$f(t) = \sqrt{t^2+1}\,,$$

where $z \mapsto \sqrt{z}$ stands for the main branch of the square root with domain of analyticity

$$D_{\sqrt{}} = \mathbb{C}\backslash]-\infty, 0] \,. \tag{6.10.4}$$

$D_f =$.

SOLUTION for (6-10.c) → 6-10-3-0:chcs1.pdf

▲

End Problem 6-10 , 70 min.

Chapter 7

Numerical Quadrature

Problem 7-1: Smooth integrand by transformation

In [Lecture → Rem. 7.4.3.10] we saw how knowledge about the structure of a non-smooth integrand can be used to restore its smoothness by transformation. The very same idea can be put to use in this problem.

This problem practises the use of Gauss-Legendre quadrature and of the techniques introduced in [Lecture → Rem. 7.4.3.10].

Given a smooth, odd function $f : [-1,1] \to \mathbb{R}$, consider the integral

$$I := \int_{-1}^{1} \arcsin(t)\, f(t)\, \mathrm{d}t. \tag{7.1.1}$$

We want to approximate this integral using global Gauss-Legendre quadrature as introduced in [Lecture → Section 7.4.2].

The nodes and the weights of the n-point Gauss-Legendre quadrature rule on $[-1,1]$ can be computed using the provided C++ function

```
QuadRule gaussquad(const unsigned n);
```

which initializes a data structure describing a quadrature rule

```
struct QuadRule {
  Eigen::VectorXd nodes_;
  Eigen::VectorXd weights_;
};
```

The names of the fields tell their function. The function `gaussquad()` can be found in `gaussquad.hpp`.

(7-1.a) ⚀ (30 min.) Based on MATPLOTLIBCPP write a C++ function

```
template <class Function>
double gaussConv(const Function& fh, const double I_ex, const
   unsigned N);
```

that produces an *appropriate* plot of the quadrature error of the Gauss-Legendre quadrature rules versus the number $n = 1,\ldots,50$ of quadrature points. The function should output the best approximation of

the integral, i.e. the approximated value at the maximum number of quadrature points. Here `f` is a functor object [Lecture → Section 0.3.3] providing the function f in (7.1.1). `I_ex` is the exact solution of the integral and `N` is the maximum number of quadrature points.

Save your convergence plot for $f(t) = \sinh(t)$ as `GaussConv.png`.

HIDDEN HINT 1 for (7-1.a) → `7-1-1-0:hcv.pdf`

SOLUTION for (7-1.a) → `7-1-1-1:gauq1.pdf` ▲

(7-1.b) ⚀ (15 min.) Describe qualitatively (type of convergence) and quantitatively (speed of convergence) the asymptotic (for $n \to \infty$) convergence of the quadrature error you observe in (7-1.a).

HIDDEN HINT 1 for (7-1.b) → `7-1-2-0:h1.pdf`

SOLUTION for (7-1.b) → `7-1-2-1:gauq2.pdf` ▲

(7-1.c) ⚁ (20 min.) With a suitable change of variable transform the integral of (7.1.1) into an equivalent one, so that Gauss-Legendre quadrature applied to the transformed integral can be expected to *converge exponentially* in the number of quadrature nodes, if f possesses an analytic extension beyond $[-1,1]$.

HIDDEN HINT 1 for (7-1.c) → `7-1-3-0:sp3h0.pdf`

HIDDEN HINT 2 for (7-1.c) → `7-1-3-1:sp3h1.pdf`

SOLUTION for (7-1.c) → `7-1-3-2:gauq3.pdf`

▲

(7-1.d) ⚁ (25 min.) [depends on Sub-problem (7-1.c)]

Now, write a C++ function

```
template <class Function>
double gaussConvCV(const Function& f, const double I_ex, const
   unsigned N);
```

which plots (employing MATPLOTLIBCPP) the quadrature error versus the number $n = 1,\ldots,50$ of quadrature points for the integral obtained in the previous subtask and returns the best approximation of the integral, i.e. the approximated value at the maximum number of quadrature points.

Again, as in Sub-problem (7-1.a), choose $f(t) = \sinh(t)$ and save your convergence plot as `GaussConvCV.png`.

SOLUTION for (7-1.d) → `7-1-4-0:gauq4.pdf`

▲

(7-1.e) ⚀ (10 min.) [depends on Sub-problem (7-1.d)]

Analogous question as in Sub-problem (7-1.b): describe qualitatively the asymptotic (for $n \to \infty$) convergence of the quadrature error you observe in Sub-problem (7-1.d).

SOLUTION for (7-1.e) → `7-1-5-0:gauq2.pdf` ▲

(7-1.f) ⚂ (15 min.) [depends on Sub-problem (7-1.b) and Sub-problem (7-1.d)]

Explain the difference between observations made in Sub-problem (7-1.b) and Sub-problem (7-1.d).

SOLUTION for (7-1.f) → `7-1-6-0:gauq5.pdf` ▲

References

[XB12] Shuhuang Xiang and Folkmar Bornemann. "On the convergence rates of Gauss and Clenshaw-Curtis quadrature for functions of limited regularity". In: *SIAM J. Numer. Anal.* 50.5 (2012), pp. 2581–2587. DOI: 10.1137/120869845.

End Problem 7-1 , 115 min.

Problem 7-2: Generalized "Hermite-type" quadrature formula

In this exercise we will meet a new class of quadrature formulas and then use one of them in an application.

This is a purely theoretical exercise. You may want to have a look at the methods used in [Lecture → Ex. 7.4.2.2], as they will come handy.

(7-2.a) ⚄ Determine $A, B, C, x_1 \in \mathbb{R}$ such that the quadrature formula

$$\int_0^1 f(x)dx \approx Af(0) + Bf'(0) + Cf(x_1) \tag{7.2.1}$$

is exact for polynomials of highest possible degree.

SOLUTION for (7-2.a) → 7-2-1-0:herm1s.pdf ▲

(7-2.b) ⚄ Compute an approximation of $z(2)$ using the quadrature rule of (7.2.1), where the function z is defined as the solution of the initial value problem

$$z'(t) = \frac{t}{1+t^2} \quad , \quad z(1) = 1\,. \tag{7.2.2}$$

HIDDEN HINT 1 for (7-2.b) → 7-2-2-0:herm2h.pdf

SOLUTION for (7-2.b) → 7-2-2-1:herm2s.pdf ▲

End Problem 7-2

Problem 7-3: Numerical Quadrature of Improper Integrals

In analysis you have seen so-called improper integrals over unbounded intervals, which, due to a fast decay of the integrand have a finite value. Occasionally such integrals have to evaluated numerically and this problem will be concerned with corresponding quadrature rules.

This problem makes extensive use of substitution rules for integrals, implementation in C++ is requested.

We want to devise a numerical method for the computation of improper integrals of the form $\int_{-\infty}^{\infty} f(t)dt$ for continuous functions $f : \mathbb{R} \to \mathbb{R}$ that decay sufficiently fast for $|t| \to \infty$ (such that they are integrable on $\mathbb{R}$).

A first option is the truncation of the domain to a bounded interval $[-b, b]$, $b \leq \infty$, that is, we approximate:

$$\int_{-\infty}^{\infty} f(t)dt \approx \int_{-b}^{b} f(t)dt$$

and then use a standard quadrature rule like Gauss-Legendre quadrature [Lecture → Def. 7.4.2.18] on $[-b, b]$.

(7-3.a) ⚀ (10 min.) For the integrand $g(t) := 1/(1+t^2)$ determine b such that the truncation error E_T satisfies:

$$E_T := \left| \int_{-\infty}^{\infty} g(t)dt - \int_{-b}^{b} g(t)dt \right| \leq 10^{-6} . \tag{7.3.1}$$

HIDDEN HINT 1 for (7-3.a) → 7-3-1-0:imp1s1.pdf

SOLUTION for (7-3.a) → 7-3-1-1:op1sol.pdf ▲

(7-3.b) ⚀ (5 min.) What is the algorithmic difficulty faced in the implementation of the truncation approach for a generic integrand?

SOLUTION for (7-3.b) → 7-3-2-0:op1dif.pdf ▲

A second option is the transformation of the improper integral to a bounded domain by substitution. For instance, we may use the map $t = \cot(s) := \frac{\cos s}{\sin s}$, $\cot :]0, \pi[\to]-\infty, \infty[$.

(7-3.c) ⚁ (15 min.) Into which integral does the substitution $t = \cot(s)$ convert $\int_{-\infty}^{\infty} f(t)dt$?

SOLUTION for (7-3.c) → 7-3-3-0:op2sub.pdf ▲

(7-3.d) ⚁ (10 min.) Write down the transformed integral explicitly for the integrand $g(t) := \frac{1}{1+t^2}$. Simplify the integrand.

HIDDEN HINT 1 for (7-3.d) → 7-3-4-0:h1ti.pdf

SOLUTION for (7-3.d) → 7-3-4-1:op2com.pdf ▲

(7-3.e) ⚁ (20 min.) [depends on Sub-problem (7-3.c)]

Write a C++ function that uses the previous transformation together with the n-point Gauss-Legendre quadrature rule to evaluate $\int_{-\infty}^{\infty} f(t)dt$:

```
template <typename Function>
double quadinf(int n, Function && f);
```

f passes an object that provides an evaluation operator of the form:

```
double operator() (double x) const;
```

You can use the function `golubwelsh()` implemented in the file `golubwelsh.hpp` that computes nodes and weights for Gauss quadrature rules using the Golub-Welsch algorithm discussed in [Lecture → Rem. 7.4.2.23], see also [Lecture → Code 7.4.2.24].

HIDDEN HINT 1 for (7-3.e) → 7-3-5-0:hlf1.pdf

SOLUTION for (7-3.e) → 7-3-5-1:op2imp.pdf ▲

(7-3.f) ⚀ (15 min.) Implement a C++ function

```
void cvgQuadInf(void);
```

in order to study the convergence for $n \to \infty$ of the quadrature method implemented in the previous subproblem for the integrand $h(t) := \exp\left(-(t-1)^2\right)$ (shifted Gaussian). To compute the error you can use that $\int_{-\infty}^{\infty} h(t)dt = \sqrt{\pi}$.

Use MATPLOTLIBCPP to create a suitable plot of the quadrature error versus the number of quadrature nodes. What kind of convergence do you observe?

SOLUTION for (7-3.f) → 7-3-6-0:testsol.pdf ▲

References

[Wal11] Jörg Waldvogel. "Towards a general error theory of the trapezoidal rule". In: *Approximation and computation*. Vol. 42. Springer Optim. Appl. Springer, New York, 2011, pp. 267–282. DOI: 10.1007/978-1-4419-6594-3_17.

End Problem 7-3 , 75 min.

Problem 7-4: Nested numerical quadrature

This problem addresses numerical quadrature over a two-dimensional domain. It turns out that this problem can be reduced to two one-dimensional integrals, which are amenable to the techniques covered in [Lecture → Chapter 7].

Implementation in C++ requested connected with [Lecture → Chapter 7].

A laser beam has intensity

$$I(x,y) := \exp(-\alpha((x-p)^2+(y-q)^2)) \,, \quad x,y \in \mathbb{R} \,, \quad \alpha > 0 \,, \tag{7.4.1}$$

on the plane orthogonal to the direction of the beam. The beam is directed orthogonal to the $x-y$-plane.

Note that the radiant power absorbed by a surface is the integral of the intensity over the surface.

(7-4.a) ⚀ (10 min.) Write down a formula involving only 1D integrals and giving the radiant power absorbed by the triangle

$$\Delta := \{(x,y)^T \in \mathbb{R}^2 \mid x \geq 0,\, y \geq 0,\, x+y \leq 1\} \,.$$

as a double integral.

SOLUTION for (7-4.a) → 7-4-1-0:neq1s.pdf

▲

(7-4.b) ⚁ (15 min.) Write a C++ function

```
template <class Function>
double evalquad(const double a, const double b,
                Function &&f, const QuadRule &Q);
```

that evaluates an n-point quadrature formula as introduced in [Lecture → Section 7.2] for an integrand passed in `f` on domain $[a,b]$. It should rely on the quadrature rule on the reference interval $[-1,1]$ that is supplied through an object of type **QuadRule**:

```
struct QuadRule { Eigen::VectorXd nodes_, weights_; };
```

(The vectors `weights` and `nodes` denote the weights and nodes, respectively, of a quadrature rule on the reference interval $[-1,1]$, see [Lecture → Def. 7.2.0.1].)

SOLUTION for (7-4.b) → 7-4-2-0:neq2s.pdf ▲

(7-4.c) ⚂ (30 min.) [depends on Sub-problem (7-4.a)]

Write a C++ function

```
template <class Function>
double gaussquadtriangle(const Function &f, const unsigned N)
```

for the computation of the integral

$$\int_\Delta f(x,y)\,dx\,dy \tag{7.4.3}$$

using nested n-point, 1D **Gauss quadrature** [Lecture → Section 7.4] in combination with the function `evalquad()` of Sub-problem (7-4.b). Use the function `gaussquad()` from `gaussquad.hpp` to obtain weights and nodes of Gauss quadrature formulas on $[-1,1]$.

HIDDEN HINT 1 for (7-4.c) → 7-4-3-0:neq3h1.pdf

HIDDEN HINT 2 for (7-4.c) → 7-4-3-1:neq3h2.pdf

SOLUTION for (7-4.c) → 7-4-3-2:neq3s.pdf ▲

(7-4.d) ⚄ (30 min.) Write a C++ function

```
void convtest2DQuad(unsigned int nmax = 20);
```

that applies the function `gaussquadtriangle()` from Sub-problem (7-4.c) to approximate $\int_\Delta I(x,y)\,\mathrm{d}x\mathrm{d}y$ using the parameters $\alpha = 1$, $p = 0$, $q = 0$. Tabulate and plot (using MATPLOTLIBCPP, storing the figure in `convergence.png`) the quadrature error w.r.t. to the number of nodes $n = 1,2,3,\ldots,20$ using the "exact" value of the integral $I \approx 0.366046550000405$. What kind of convergence do you observe and why?

HIDDEN HINT 1 for (7-4.d) → 7-4-4-0:neq4h.pdf

SOLUTION for (7-4.d) → 7-4-4-1:neq4s.pdf

▲

End Problem 7-4, 85 min.

Problem 7-5: Quadrature Error Plots

In this problem, we will try and deduce the type of quadrature and the integrand from an error plot.

This is a purely theoretical problem discussing convergence of quadrature rules, see [Lecture → Lemma 7.4.3.6], [Lecture → Ex. 7.4.3.9], [Lecture → § 7.5.0.9], [Lecture → Exp. 7.5.0.12].

We consider three different functions on the interval $I = [0,1]$, which have the following properties:

$$\begin{aligned} &\text{function A:} \quad f_A \in C^\infty(I)\,, \quad f_A \notin \mathcal{P}_k \;\forall\, k \in \mathbb{N}\,; \\ &\text{function B:} \quad f_B \in C^0(I)\,, \quad f_B \notin C^1(I)\,; \\ &\text{function C:} \quad f_C \in \mathcal{P}_{12}\,, \end{aligned}$$

where $\mathcal{P}_k$ is the space of the polynomials of degree at most k defined on I. The following quadrature rules are applied to these functions:

- quadrature rule A is a global Gauss quadrature;
- quadrature rule B is a composite trapezoidal rule;
- quadrature rule C is a composite 2-point Gauss quadrature.

The corresponding absolute values of the quadrature errors are plotted against the number of function evaluations in the figure below. Notice that only the quadrature errors obtained with an even number of function evaluations are shown.

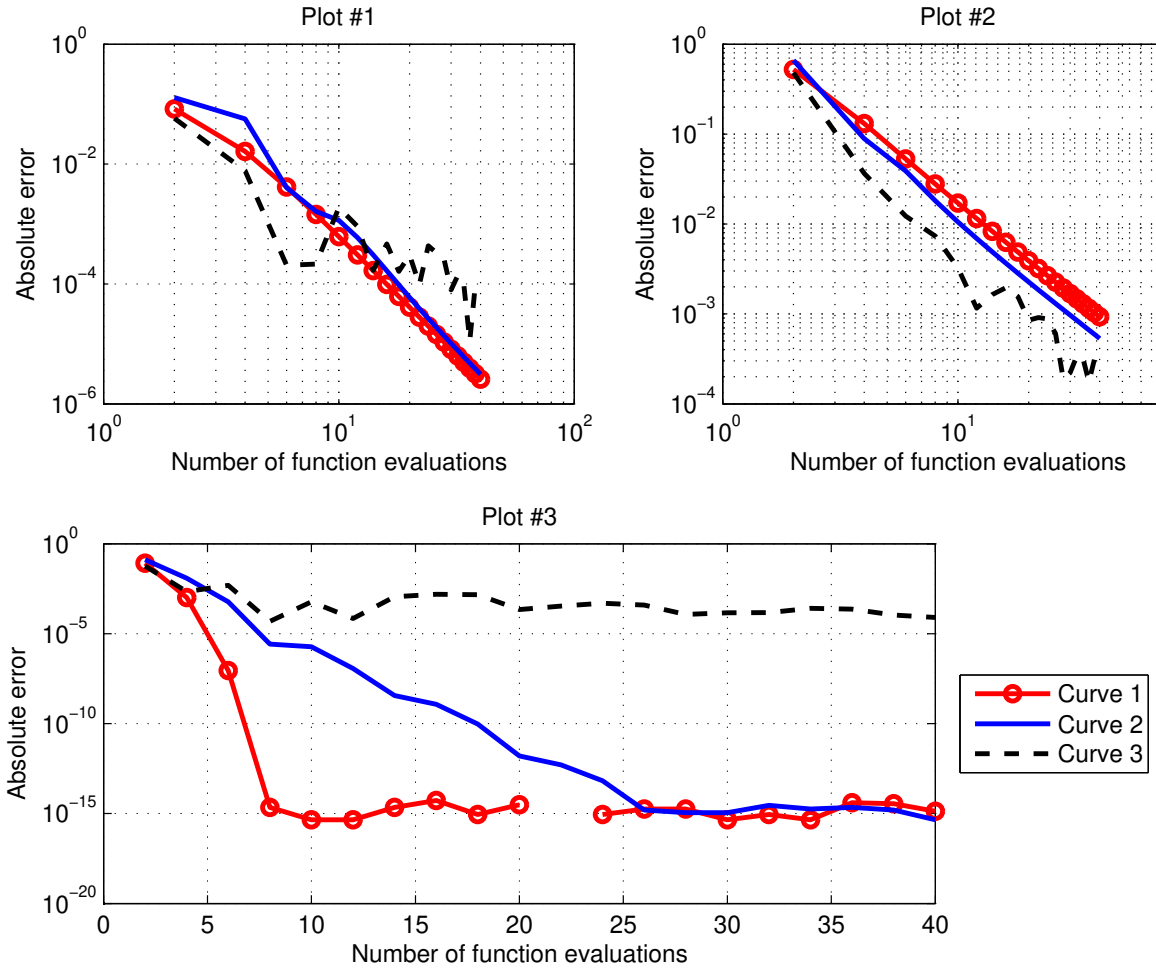


Fig. 70

(7-5.a) ⚂ (15 min.) Match the three plots (plot #1, #2 and #3) with the three quadrature rules (quadrature rule A, B, and C). Justify your answer.

HIDDEN HINT 1 for (7-5.a) → 7-5-1-0:plot1h.pdf

SOLUTION for (7-5.a) → 7-5-1-1:plot1s.pdf ▲

(7-5.b) ⚄ (15 min.) The quadrature error curves for a particular function f_A, f_B and f_C are plotted in the same style (curve 1 as red line with small circles, curve 2 means the blue solid line, curve 3 is the black dashed line). Which curve corresponds to which function (f_A, f_B, f_C)? Justify your answer.

SOLUTION for (7-5.b) → 7-5-2-0:plot2s.pdf ▲

End Problem 7-5 , 30 min.

Problem 7-6: Weighted Gauss quadrature

In [Lecture → Rem. 7.4.3.10] we saw, how, in special cases, integrals with singular integrands can be transformed into integrals with smooth integrands, which allow much faster convergence of Gauss quadrature. This problem examines an alternative method: If the type of the singular behavior of the integrand is known, special quadrature rules, called weighted Gauss rules, can be designed that enjoy fast convergence.

This problem relies on the material of [Lecture → Section 7.4] and requests a little implementation in C++.

The development of an alternative quadrature formula for a particular singular integral relies on the Chebyshev polynomials of the second kind U_n, defined as

$$U_n(t) = \frac{\sin((n+1)\arccos t)}{\sin(\arccos t)}, \qquad n \in \mathbb{N}_0 . \tag{7.6.1}$$

The U_n are **orthogonal polynomials** on $]-1,1[$ with respect to a weighted L^2 inner product (see [Lecture → (6.3.2.3)]) with weight function given by $w(\tau) = \sqrt{1-\tau^2}$:

$$\int_{-1}^{1} \sqrt{1-\tau^2} U_n(\tau) U_m(\tau)\,\mathrm{d}\tau = 0 , \quad \text{if} \quad m \neq n . \tag{7.6.2}$$

Orthogonal polynomials are discussed in [Lecture → Section 6.3.2], but this problem can be solved without studying that section.

(7-6.a) ⚀ (10 min.) Recall the role of the $L^2(]-1,1[)$-orthogonal[1] Legendre polynomials in the derivation and definition of Gauss-Legendre quadrature rules (see [Lecture → § 7.4.2.15]). ▲

(7-6.b) ⚁ (15 min.) Show that the U_n satisfy the **3-term recursion**

$$U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t), \qquad U_0(t) = 1, \qquad U_1(t) = 2t,$$

for every $n \geq 1$.

Remark. According to [Lecture → Thm. 6.3.2.14] all orthogonal polynomials can be constructed through a 3-term recursion. The 3-term recursion for Chebychev polynomials is given in [Lecture → Thm. 6.2.3.4] and that for the Legendre polynomials in [Lecture → (6.3.2.18)].

SOLUTION for (7-6.b) → `7-6-2-0:wgq1s.pdf`

▲

(7-6.c) ⚂ (10 min.) Show that $U_n \in \mathcal{P}_n$ and has leading coefficient 2^n.

SOLUTION for (7-6.c) → `7-6-3-0:wgq2s.pdf`

▲

(7-6.d) ⚃ (15 min.) Show that for every $m,n \in \mathbb{N}_0$, we have

$$\int_{-1}^{1} \sqrt{1-t^2}\, U_m(t) U_n(t)\, dt = \frac{\pi}{2}\delta_{m,n} := \begin{cases} \frac{\pi}{2} & \text{, if} \quad m = n , \\ 0 & \text{else.} \end{cases}$$

[1] We call two square-integrable functions $f,g :]-1,1[\to \mathbb{R}$ $L^2(]-1,1[)$-orthogonal, if $\int_{-1}^{1} f(t)g(t)\,\mathrm{d}t = 0$.

HIDDEN HINT 1 for (7-6.d) → 7-6-4-0:wgqs3h.pdf

SOLUTION for (7-6.d) → 7-6-4-1:wgq3s.pdf

▲

(7-6.e) ⚀ (10 min.) What are the zeros ξ_j^n ($j = 1,\ldots,n$) of U_n, $n \geq 1$? Give an explicit formula similar to the formula for the Chebyshev nodes in $[-1,1]$.

SOLUTION for (7-6.e) → 7-6-5-0:wgq4s.pdf ▲

(7-6.f) ⚁ (30 min.) [depends on Sub-problem (7-6.d)]

Show that the choice of weights

$$w_j := \frac{\pi}{n+1}\sin^2\left(\frac{j}{n+1}\pi\right), \qquad j = 1,\ldots,n, \tag{7.6.4}$$

ensures that the quadrature formula

$$Q_n^U(f) := \sum_{j=1}^{n} w_j f(\xi_j^n)\,, \tag{7.6.5}$$

ξ_j^n the zeros found in Sub-problem (7-6.d), provides the exact value of the integral

$$I := \int_{-1}^{1} \sqrt{1-t^2} f(t)\,\mathrm{d}t \tag{7.6.6}$$

for $f \in \mathcal{P}_{n-1}$ (assuming exact arithmetic).

SOLUTION for (7-6.f) → 7-6-6-0:wgq5s.pdf ▲

(7-6.g) ⚀ (15 min.) Show that the quadrature formula (7.6.5) gives the exact value of (7.6.6) even for every $f \in \mathcal{P}_{2n-1}$.

HIDDEN HINT 1 for (7-6.g) → 7-6-7-0:wgq6h.pdf

SOLUTION for (7-6.g) → 7-6-7-1:wqg6s.pdf ▲

(7-6.h) ⚂ (10 min.) Show that the quadrature error

$$\mathrm{err}_n := |Q_n^U(f) - W(f)|\,, \quad n \in \mathbb{N}\,,$$

decays to 0 exponentially as $n \to \infty$ for every $f \in C^\infty([-1,1])$ that admits an analytic extension to an open subset of the complex plane, which contained $[-1,1]$.

HIDDEN HINT 1 for (7-6.h) → 7-6-8-0:wgq7h.pdf

SOLUTION for (7-6.h) → 7-6-8-1:wqg7s.pdf

▲

(7-6.i) ⚂ (20 min.) Write a C++ function ,

```
template<typename Function>
double quadU(const Function & f, const unsigned n);
```

that gives $Q_n^U(f)$ as output, where `f` is a **functor object** with an evaluation operator, like a lambda function, representing f, e.g.

```
auto f = [] (double & t) { return 1/(2 + exp(3*t)); };
```

SOLUTION for (7-6.i) → 7-6-9-0:wqg8s.pdf

▲

(7-6.j) ⚀ (20 min.) Devise a C++ function

```
void testQuadU(unsigned int nmax = 20);
```

that, for the function $f(t) = 1/(2+e^{3t})$ and $n = 1,\ldots,n_{\max}$ tabulates and plots (employing MATPLOTLIBCPP) the quadrature error $E_n(f) = |W(f) - Q_n^U(f)|$ using the "exact" value $W(f) = 0.483296828976607$. Estimate the parameter $0 \leq q < 1$ in the asymptotic decay law $E_n(f) \approx Cq^n$ characterising (sharp) exponential convergence, see [Lecture → Def. 6.2.2.7].

SOLUTION for (7-6.j) → 7-6-10-0:wgq9s.pdf

▲

End Problem 7-6 , 155 min.

Problem 7-7: Quadrature by transformation

In [Lecture → Rem. 7.4.3.10] a suitable transformation converted a singular integrand into a smooth one. In this problem we see another example. Also Problem 7-9 and Problem 7-1 cover this topic.

This problem does not contain an implementation part. Study [Lecture → Rem. 7.4.3.10] to learn about regularizing transformations.

For $f \in C^0([0,2])$ we consider the definite improper integral with *singular integrand*

$$I(f) := \int_0^2 \sqrt{\frac{2-t}{t}} f(t)\,\mathrm{d}t\,. \tag{7.7.1}$$

(7-7.a) ⚀ Transform $I(f)$ into an integral over $[-1,1]$, whose integrand is $\in C^\infty([-1,1])$ provided that $f \in C^\infty([0,2])$.

HIDDEN HINT 1 for (7-7.a) → 7-7-1-0:hsi1.pdf

HIDDEN HINT 2 for (7-7.a) → 7-7-1-1:hsi5.pdf

SOLUTION for (7-7.a) → 7-7-1-2:s1t.pdf ▲

(7-7.b) ⚁ (20 min.) For $n \in \mathbb{N}^*$, derive a family of $2n$-point quadrature formulas

$$Q_n(f) = \sum_{j=1}^{2n} w_j^n f(c_j^n), \qquad f \in C([0,2])$$

for which $Q_n(f)$ converges to $I(f)$ exponentially as $n \to \infty$, if $f \in C^\infty([0,2])$.

Remark. A more precise statement of the problem would be ", if f has an analytic extension to a complex neighborhood of $[0,2]$". This is, how the statement $f \in C^\infty([0,2])$ above should be read.

SOLUTION for (7-7.b) → 7-7-2-0:qtf1.pdf

▲

(7-7.c) ⚂ (10 min.) Given $n > 0$, determine the maximal $d \in \mathbb{N}$ such that $I(p) = Q_n(p)$ for every polynomial $p \in \mathcal{P}_{d-1}$.

SOLUTION for (7-7.c) → 7-7-3-0:1tf2ZerosLegendre5s.pdf ▲

End Problem 7-7, 30 min.

Problem 7-8: Discretization of an integral operator

In this problem we consider a completely new concept, an **integral operator** of the form

$$(\mathsf{T}f)(x) = \int_I k(x,y)f(y)\,\mathrm{d}y\,, \quad x \in I \subset \mathbb{R}\,, \tag{7.8.1}$$

where the **kernel** $k : I \times I \to \mathbb{R}$ is continuous on $I \times I$, $I \subset \mathbb{R}$ a closed interval. It maps a function on I to another function on I. Such integral operators are very common in models of continuous physics. Of course, their numerical treatment heavily draws on numerical quadrature.

Requires knowledge about Gauss quadrature from [Lecture → Section 7.4] and involves substantial implementation in C++.

Given $f \in C^0([0,1])$, the integral

$$g(x) := \int_0^1 e^{|x-y|} f(y)\,dy$$

defines a function $g \in C^0([0,1])$. We approximate the integral by means of n-point Gauss quadrature, which yields a function $g_n(x)$.

(7-8.a) ⚀ (20 min.) Let $\{\xi_j^n\}_{j=1}^n$ be the nodes for the Gauss quadrature on the interval $[0,1]$. Assume that the nodes are ordered, namely $\xi_j^n < \xi_{j+1}^n$ for every $j = 1,\ldots,n-1$. We can write

$$[g_n(\xi_l^n)]_{l=1}^n = \mathbf{M}\left[f(\xi_j^n)\right]_{j=1}^n,$$

for a suitable matrix $\mathbf{M} \in \mathbb{R}^{n,n}$. Give a formula for the entries of M.

SOLUTION for (7-8.a) → 7-8-1-0:ongp1.pdf ▲

(7-8.b) ⚄ (60 min.) Implement a function

```
template <typename FUNCTION>
Eigen::VectorXd comp_g_gausspts(FUNCTION f, unsigned int n)
```

that computes the n-vector $\left[g_n(\xi_l^n)\right]_{l=1}^n$ with *optimal* asymptotic computational cost $O(n)$ (excluding the computation of the nodes and weights) for $n \to \infty$. Here, `f` is an object with an evaluation operator, a functor [Lecture → Section 0.3.3], e.g. a lambda function, that represents the function f.

You may use the provided function

```
void gaussrule(int n, Eigen::VectorXd &w, Eigen::VectorXd &xi)
```

that computes the weigths `w` and the ordered nodes `xi` relative to the `n`-point Gauss quadrature on the interval $[-1,1]$.

HIDDEN HINT 1 for (7-8.b) → 7-8-2-0:gptsh1.pdf

SOLUTION for (7-8.b) → 7-8-2-1:ongp2.pdf ▲

(7-8.c) ⚂ (15 min.) Test your implementation and write a C++ function

```
double testCompGGaussPts();
```

that computes $g(\xi_{11}^{21})$ for $f(y) = e^{-|0.5-y|}$. What result do you expect?

SOLUTION for (7-8.c) → 7-8-3-0:ongp3.pdf ▲

End Problem 7-8 , 95 min.

Problem 7-9: Efficient quadrature of singular integrands

We know that smoothness of the integrand is essential for fast covergence of high-order numerical quadrature like Gauss quadrature rules. In some cases smart transformation can convert the integral into another with a smooth integrand. This problem uses this trick for the sake of efficient numerical quadrature of non-smooth integrands with a special structure.

Before you tackle this problem, read about regularization of integrands by transformation, cf. [Lecture → Rem. 7.4.3.10]. For other problems on the same topic see Problem 7-7 and Problem 7-1.

Our task is to develop quadrature formulas for integrals of the form:

$$W(f) := \int_{-1}^{1} \sqrt{1-t^2}\, f(t)\, dt, \tag{7.9.1}$$

where f possesses an analytic extension to a complex neighbourhood of $[-1,1]$.

(7-9.a) ⚀ (10 min.) Understand how to use the C++ function

```
QuadRule gauleg(unsigned int n);
```

(contained in `gauleg.hpp`), which returns a structure `QuadRule` containing nodes (x_j) and weights (w_j) of a Gauss-Legendre quadrature [Lecture → Def. 7.4.2.18] on $[-1,1]$ with n nodes. ▲

(7-9.b) ⚁ (15 min.) Study [Lecture → § 7.4.3.2] in order to learn about the convergence of Gauss-Legendre quadrature. What can you say about the least expected convergence for an integrand $f \in C^r([a,b])$? What about convergence in the case $f \in C^\infty([a,b])$?

SOLUTION for (7-9.b) → 7-9-2-0:effs1.pdf

▲

(7-9.c) ⚁ (40 min.) Based on the function `gauleg()`, implement a C++ function

```
template <class Function>
double quadsingint(const Function& f, const unsigned n);
```

that approximately evaluates $W(f)$ using $2n$ evaluations of f. An object of type `Function` must provide an evaluation operator

```
double operator(double t) const;
```

Ensure that, as $n \to \infty$, the error of your implementation has asymptotic exponential convergence to zero.

HIDDEN HINT 1 for (7-9.c) → 7-9-3-0:h2l.pdf

HIDDEN HINT 2 for (7-9.c) → 7-9-3-1:h23.pdf

SOLUTION for (7-9.c) → 7-9-3-2:effs2.pdf ▲

(7-9.d) ⚁ (15 min.) [depends on Sub-problem (7-9.c)]

Give formulas for the nodes c_j and weights $\widetilde{w}_j$ of a $2n$-point quadrature rule on $[-1,1]$, whose application to the integrand f will produce the same results as the function `quadsingint` that you implemented in the previous subproblem, that is,

$$\texttt{quadsingint}(f,n) = \sum_{j=1}^{n} c_j f(\widetilde{x}_j) \quad \text{for any } f \text{ passed through } \texttt{f}.$$

SOLUTION for (7-9.d) → 7-9-4-0:effs3.pdf ▲

(7-9.e) 2 Implement a C++ function

```
void tabAndPlotQuadErr()
```

that tabulates and plots the quadrature error:

$$\epsilon_i := |W(f) - \texttt{quadsingint(f,n)}|$$

for $f(t) := \dfrac{1}{2+\exp(3t)}$ and $n = 1,2,...,25$. Then describe the type of convergence observed, see [Lecture → Def. 6.2.2.7].

SOLUTION for (7-9.e) → 7-9-5-0:effs5.pdf ▲

End Problem 7-9 , 80 min.

Problem 7-10: Zeros of orthogonal polynomials

This problem combines elementary methods for finding zeros from [Lecture → Section 8.4] and 3-term recursions satisfied by orthogonal polynomials, *cf.* [Lecture → Thm. 6.3.2.14]. This will permit us to find the zeros of Legendre polynomials, the so-called Gauss nodes (see [Lecture → Def. 7.4.2.18]).

Connected with [Lecture → Section 7.4] and [Lecture → Section 8.4]

The zeros of the Legendre polynomial P_n (see [Lecture → Def. 7.4.2.16]) are the n Gauss points ζ_j^n, $j = 1, \ldots, n$. In this problem we compute the Gauss points by zero-finding methods applied to P_n. The 3-term recursion [Lecture → (7.4.2.21)] for Legendre polynomials will play an essential role. Moreover, recall that, by definition, the Legendre polynomials are $L^2(]-1,1[)$-orthogonal.

(7-10.a) ⚄ Prove the following interleaving property of the zeros of the Legendre polynomials. For all $n \in \mathbb{N}_0$ we have:

$$-1 < \zeta_j^n < \zeta_j^{n-1} < \zeta_{j+1}^n < 1, \qquad j = 1, \ldots, n-1 \,.$$

HIDDEN HINT 1 for (7-10.a) → 7-10-1-0:ZerosLegendre1h.pdf

SOLUTION for (7-10.a) → 7-10-1-1:ZerosLegendre1s.pdf ▲

(7-10.b) ⚀ By differentiating [Lecture → (7.4.2.21)] derive a combined 3-term recursion for the sequences $(P_n)_n$ and $(P_n')_n$.

SOLUTION for (7-10.b) → 7-10-2-0:ZerosLegendre2s.pdf

▲

(7-10.c) ⚁ Use the recursions obtained in (7-10.b) to write a C++ function

```
void legvals(const Eigen::VectorXd& x,
Eigen::MatrixXd& Lx, Eigen::MatrixXd& DLx);
```

that fills the matrices `Lx` and `DLx` in $\mathbb{R}^{N\times(n+1)}$ with the values $\{P_k(x_j)\}_{jk}$ and $\{P_k'(x_j)\}_{jk}$, $j = 0, \ldots, N-1$ and $k = 0, \ldots, n$, for an input vector $x \in \mathbb{R}^N$ (passed in `x`):

SOLUTION for (7-10.c) → 7-10-3-0:ZerosLegendre3s.pdf

▲

(7-10.d) ⚄ We can compute the zeros of P_k, $k = 1, \ldots, n$, by means of the secant rule (see [Lecture → § 8.4.2.28]) using the endpoints $\{-1, 1\}$ of the interval and the zeros of the previous Legendre polynomial as initial guesses; see (7-10.a). We opt for a correction-based termination criterion (see [Lecture → Section 8.2.3]) based on prescribed relative and absolute tolerance (see [Lecture → Code 8.4.2.31]).

Write a C++ function that computes the Gauss points $\zeta_j^k \in [-1,1]$, $j = 1, \ldots, k$ and $k = 1, \ldots, n$, using the zero finding approach outlined above:

```
Eigen::MatrixXd gaussPts(const int n,
                         const double rtol=1e-10, const double
                           atol=1e-12)
```

The Gauss points should be returned in an upper triangular $n \times n$-matrix.

HIDDEN HINT 1 for (7-10.d) → 7-10-4-0:ZerosLegendre4h.pdf

SOLUTION for (7-10.d) → 7-10-4-1:ZerosLegendre4s.pdf

▲

(7-10.e) ⚀ Validate your implementation of the function `gaussPts` with $n = 8$ by computing the values of the Legendre polynomials in the zeros obtained (use the function `legvals`). Explain the failure of the method.

HIDDEN HINT 1 for (7-10.e) → 7-10-5-0:ZerosLegendre5h.pdf

SOLUTION for (7-10.e) → 7-10-5-1:ZerosLegendre5s.pdf

▲

(7-10.f) ⚀ Fix your function `gaussPts` taking into account the above considerations. You should use the *regula falsi*, that is a variant of the secant method in which, at each step, we choose the old iterate to keep depending on the signs of the function. More precisely, given two approximations $x^{(k)}$, $x^{(k-1)}$ of a zero in which the function f has different signs, compute another approximation $x^{(k+1)}$ as zero of the secant. Use this as the next iterate, but then chose as $x^{(k)}$ the value $z \in \{x^{(k)}, x^{(k-1)}\}$, for which $\operatorname{sign}\left[f\left(x^{(k+1)}\right)\right] \neq \operatorname{sign}[f(z)]$. This ensures that f has always a different sign in the last two iterates.

The regula falsi variation of the secant method can be easily implemented with a little modification of the code.

SOLUTION for (7-10.f) → 7-10-6-0:ZerosLegendre6s.pdf ▲

End Problem 7-10

Problem 7-11: Quadrature rules and quadrature errors

When sequences of quadrature rules are applied to evaluate $\int_a^b f(t)\,\mathrm{d}t$, then the asymptotic convergence of quadrature error in terms of $n \to \infty$, $n \mathrel{\hat{=}}$ the number of quadrature nodes, is determined both by

(i) the smoothness of the integrand f,
(ii) family of quadrature rules used.

This problems asks you to guess the family of quadrature rules from error curves.

To prepare for this problem study [Lecture → Section 7.4], in particular [Lecture → Ex. 7.4.3.9], and also [Lecture → Exp. 7.5.0.12], [Lecture → Exp. 7.5.0.16]

In this problem we consider three different families of quadrature rules, for which we use the following abbreviations:

(TR) $\mathrel{\hat{=}}$ Equidistant trapezoidal rule, see [Lecture → (7.5.0.17)],

(SR) $\mathrel{\hat{=}}$ Equidistant Simpson rule, see [Lecture → (7.5.0.5)],

(GR) $\mathrel{\hat{=}}$ Gauss-Legendre quadrature rules, as defined in [Lecture → Def. 7.4.2.18].

(7-11.a) ⚄ (15 min.) The following C++ function implements the n-point equidistant trapezoidal rule, $n \in \mathbb{N}$, $n \geq 2$, for the approximate evaluation of $\int_a^b f(t)\,\mathrm{d}t$. The integrand is passed through the functor `f`.

```
template <typename Functor>
double equidTrapezoidalRule(Functor &&f, double a, double b, unsigned int n) {
  assert(n>=2);
  const double h = (b - a)/(n-1);
  double t = a + h, s = 0.0;
  for (unsigned int i = 1; i < n-1; t += h, ++i) { s += f(t); }
  return (          * f(a) +          * f(b) +          * s);
}
```

Supplement the missing C++ code in the boxes.

SOLUTION for (7-11.a) → 7-11-1-0:sa.pdf ▲

(7-11.b) ⚄ (15 min.) The following plots display the quadrature errors for the approximation of the integral

$$\int_0^1 f_1(t)\,\mathrm{d}t \quad , \quad f_1(t) := |\sin(5t)| \, ,$$

by the three families of quadrature rules and numbers $n = 4, \ldots, 100$ of quadrature points.

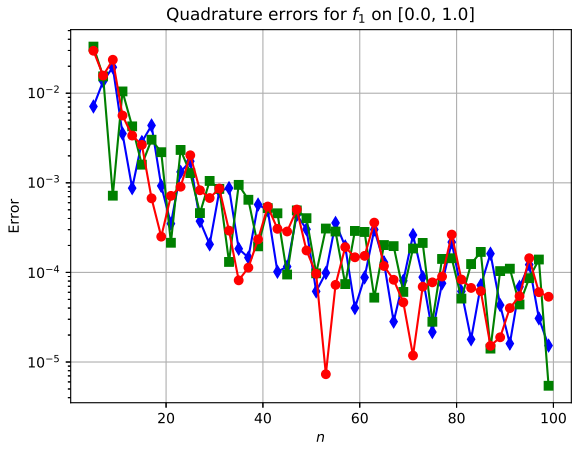


Fig. 74

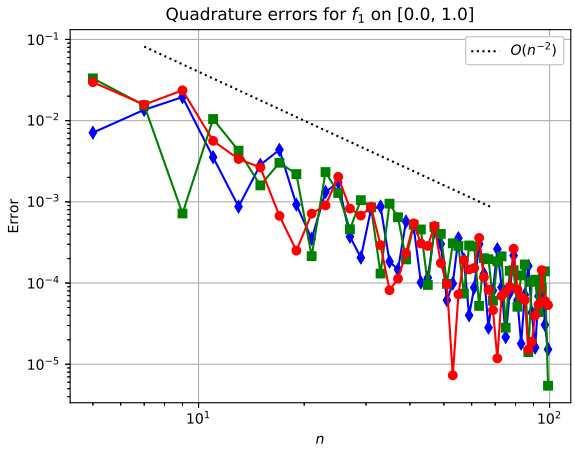
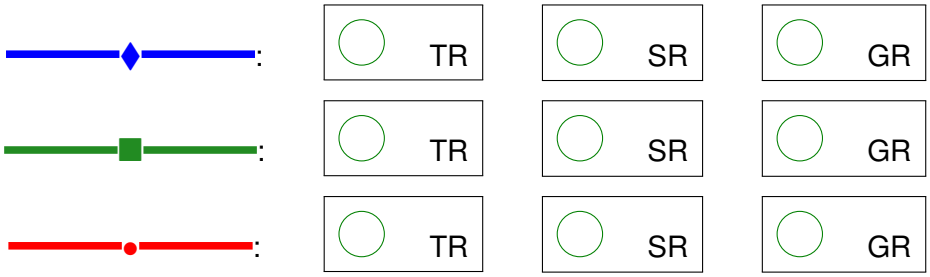


Fig. 75

Tick the families of quadrature rules, if the curve correctly represents the expected behavior of its quadrature error as a function of n.



SOLUTION for (7-11.b) → 7-11-2-0:s1.pdf ▲

(7-11.c) ⚄ (15 min.) Below we have plotted the quadrature errors for the approximation of the integral

$$\int_0^1 f_2(t)\,\mathrm{d}t \quad , \quad f_2(t) := |\sin(5t)|^2 \,,$$

for the three families of quadrature rules and numbers $n = 4, \ldots, 100$ of quadrature points.

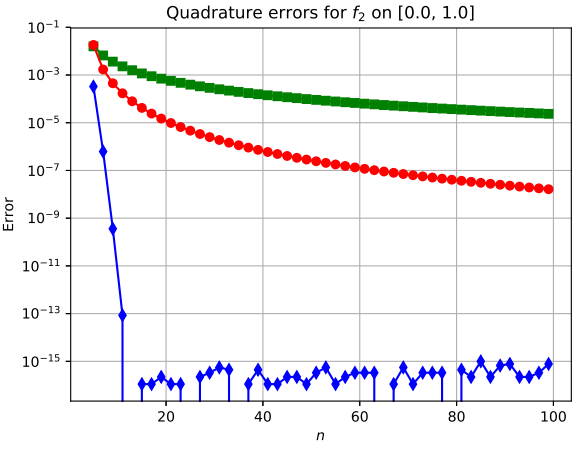


Fig. 78

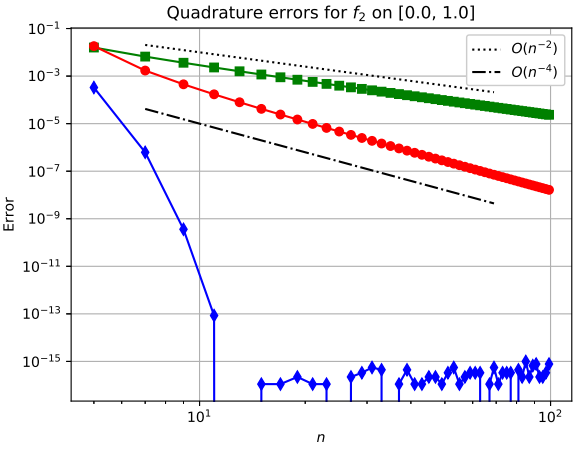
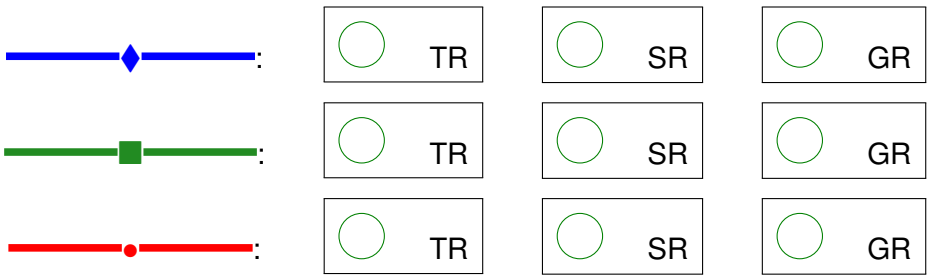


Fig. 79

Select the quadrature rule, if the curve correctly represents the expected behavior of its quadrature error as a function of n.



SOLUTION for (7-11.c) → 7-11-3-0:s2.pdf ▲

(7-11.d) ⚂ (15 min.) The plots below display the quadrature errors for

$$\int_0^1 f_3(t)\,\mathrm{d}t\,, \quad f_3(t) := |\sin(5t)|^5\,,$$

and for equidistant trapezoidal rules (TR), equidistant Simpson rules (SR), and Gauss-Legendre rules (GR) as a function of the number n of quadrature points, $n = 4, \ldots, 100$.

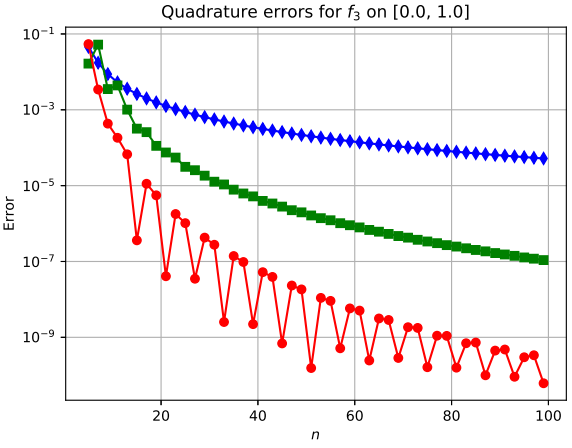


Fig. 82

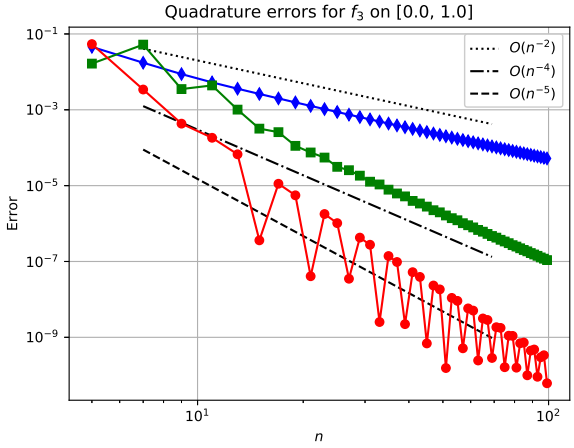
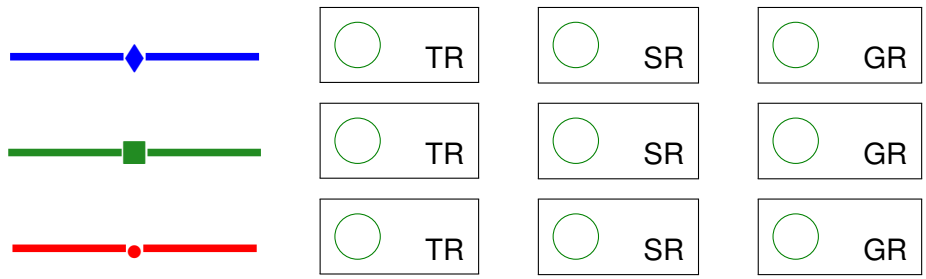


Fig. 83

Select those quadrature rules, for which the curve correctly represents the expected behavior of its quadrature error as a function of n.



SOLUTION for (7-11.d) → 7-11-4-0:s3.pdf ▲

(7-11.e) ⚂ (20 min.) Decide whether the following assertions about quadrature rules are true or false

(i) For any given set of n quadrature nodes $\in [0,1]$ there is a quadrature rule of order n with those nodes.

○ true ○ false

(ii) Let Q_n designate an n-point quadrature rule of order n on $[0,1]$. Then

$$f \geq 0 \quad \Rightarrow \quad Q_n(f) \geq 0\,.$$

○ true ○ false

(iii) An n-point quadrature rule of order $2n-1$ has all positive weights

◯ true ◯ false

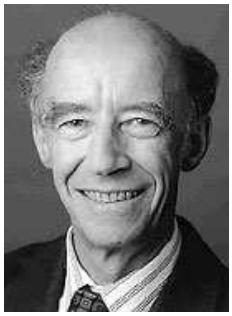
(iv) The family of equidistant trapezoidal rules (TR) enjoys exponential convergence in the number n of quadrature nodes for $\int_0^1 |sin(\pi t)|\,\mathrm{d}t$.

◯ true ◯ false

SOLUTION for (7-11.e) → 7-11-5-0:sx.pdf ▲

End Problem 7-11 , 80 min.

Problem 7-12: Clenshaw-Curtis-Fejer Quadrature Formula



In [J. WALDVOGEL, *Fast construction of the Fejér and Clenshaw-Curtis quadrature rules*, BIT, 46 (2006), pp. 195–202.] Jörg Waldvogel, former professor at the Seminar of Applied Mathematics of ETH Zürich, elaborates formulas for the weights and nodes of so-called Clenshaw-Curtis-Fejer quadrature rules, *cf.* [Lecture → Rem. 7.3.0.8]. This problem is concerned with their efficient implementation.

This problem requires familiarity with the discrete Fourier transform (DFT) from [Lecture → Section 4.2] and the FFT algorithm [Lecture → Section 4.3] and its use in EIGEN. Knowledge about the concept and the order of quadrature formulas is taken for granted.

The following class provides the weights and nodes of a quadrature formula of the Clenshaw-Curtis-Fejer family on the interval $[-1,1]$.

C++ code 7.12.1: Definition of quadrature class

```
2  class CCFQuadRule {
3  public:
4    explicit CCFQuadRule(unsigned int n);
5    ~CCFQuadRule() = default;
6
7    const Eigen::VectorXd &nodes() const { return nodes_; }
8    const Eigen::VectorXd &weights() const { return weights_; }
9
10 private:
11   Eigen::VectorXd nodes_;
12   Eigen::VectorXd weights_;
13 };
```

Get it on GitLab (clenshawcurtisfejer.hpp).

Note that passing an integer n to the constructor builds an $n+1$-point quadrature formula!

The methods `nodes()` and `weights()` return vectors whose entries supply the nodes c_j and weights w_j, $j = 0,\ldots,n$, of the quadrature formula (C++ indexing used!).

This is the listing of the implementation of the constructor:

C++ code 7.12.2: Constructor of CCFQuadRule

```
2  CCFQuadRule::CCFQuadRule(unsigned int n) : weights_(n + 1) {
3    assert(n > 0);
4    nodes_ = (Eigen::ArrayXd::LinSpaced(n + 1, 0, n) * M_PI / n).cos().matrix();
5    const unsigned int m = n / 2; // Integer division!
6    const Eigen::ArrayXd idx = Eigen::ArrayXd::LinSpaced(m, 1.0, m);
7    const Eigen::ArrayXd cos_arg = 2.0 * M_PI * idx / n;
8    Eigen::ArrayXd fac = 2.0 / (4 * idx.pow(2) - 1.0);
9    if (n % 2 == 0)
10     fac[m - 1] /= 2.0;
11   weights_[0] = (1.0 - fac.sum()) / n;
12   for (unsigned int j = 1; j < n; ++j) {
13     weights_[j] = (1.0 - (fac * (j * cos_arg).cos()).sum()) * 2.0 / n;
```

```
14    }
15    weights_[n] = weights_[0];
16  }
```

Get it on GitLab (clenshawcurtisfejer.hpp).

Recall that

- `Eigen::ArrayXd::LinSpaced(n,a,b)` builds a sequence of n values $\left(a + \frac{b-a}{n-1}j\right)_{j=0}^{n-1}$,
- the `cos()` method of **Eigen::ArrayXd** applies the cosine function to all entries of the sequence, and
- the call of `pow(2)` for **Eigen::ArrayXd** squares all members of the sequence,
- that the `sum()` method computes the sum of all entries of a vector/matrix in EIGEN.

(7-12.a) ⚀ (10 min.) Based on Code 7.12.2 give a mathematical expression for the nodes c_j, $j = 0, \ldots, n$, of the $n+1$-point Clenshaw-Curtis-Fejer quadrature formula.

SOLUTION for (7-12.a) → 7-12-1-0:ccfs1.pdf ▲

(7-12.b) ⚁ (15 min.) Based on Code 7.12.2 write down (in mathematical notation) a formula for the quadrature weights w_j, $j = 0, \ldots, n$, of the $n+1$-point Clenshaw-Curtis-Fejer quadrature formula implemented in the class **CCFQuadRule**.

SOLUTION for (7-12.b) → 7-12-2-0:.pdf ▲

(7-12.c) ⚂ (15 min.) The C++ function

```
unsigned int determineOrderCCF(unsigned int n);
```

is supposed to return the **order** [Lecture → Def. 7.4.1.1] of the quadrature formulas provided by an object of the class **CCFQuadRule** when constructed with a parameter `n`.

Fill in the missing parts of the code in the boxes (valid C++ syntax!):

```
unsigned int determineOrderCCF(unsigned int n) {
  const CCFQuadRule ccfqr(n);
  unsigned int d = 0;       // Degree of monomial
  double quaderr;  // Quadrature error
  double I_ex;     // Exact integral value
  double I_qr;     // Value computed with CCF quadrature rule
  const double abstol = std::numeric_limits<double>::epsilon() * n
     * 2;
  const double reltol = std::numeric_limits<double>::epsilon() * n
     * 20;
  do {
    const Eigen::ArrayXd pdvals = ccfqr.[          ]
       .array().pow(d);
    I_qr = [                                        ];
    I_ex = [                                        ];
    quaderr = std::abs(I_qr - I_ex);
    d = d + 1;
  } while ((quaderr <= abstol) || (quaderr <= reltol * I_ex));
```

```
    return                ;
}
```

SOLUTION for (7-12.c) → 7-12-3-0:cfqrs3.pdf ▲

(7-12.d) ⚄ (90 min.) [depends on Sub-problem (7-12.b)]

Your task is to devise a more efficient implementation of (the constructor of) the class **CCFQuadRule**. To that end you have created a copy, renamed it to **CCFQuadRule_Fast** and deleted parts of the constructor. Now you should re-implement the constructor in the file `clenshawcurtisfejer.hpp` so that the asymptotic effort for computing the nodes and weights of the Clenshaw-Curtis-Fejer quadrature rule becomes $O(n \log n)$ for $n \to \infty$.

SOLUTION for (7-12.d) → 7-12-4-0:spx1.pdf ▲

References

[Wal06] Jörg Waldvogel. "Fast construction of the Fejér and Clenshaw-Curtis quadrature rules". In: *BIT* 46.1 (2006), pp. 195–202. DOI: 10.1007/s10543-006-0045-4.

End Problem 7-12 , 130 min.

Problem 7-13: Smooth integrand by transformation (ArcCos)

Integrals with non-smooth integrands can *sometimes* be treated by a suitable transformation, which converts the integrand into a smooth function, see [Lecture → Rem. 7.4.3.10]. In this problem we study an example.

You should be familiar with the transformation of quadrature rules [Lecture → Rem. 7.2.0.4] and Gaussian-Legendre quadrature formulas [Lecture → Section 7.4], [Lecture → Section 7.4.2]. You must be able to perform integration by substitution.

Given a smooth function $f : [-1,1] \to \mathbb{R}$ our task is to come up with a numerical approximation of

$$I(f) := \int_{-1}^{1} \arccos(t)\, f(t)\, \mathrm{d}t\,. \tag{7.13.1}$$

We want to approximate this integral using global Gauss-Legendre quadrature as introduced in [Lecture → Section 7.4.2].

The function $t \mapsto \arccos t$ ▷
($\hat{=}$ the inverse of $\cos$ on $[0,\pi]$)

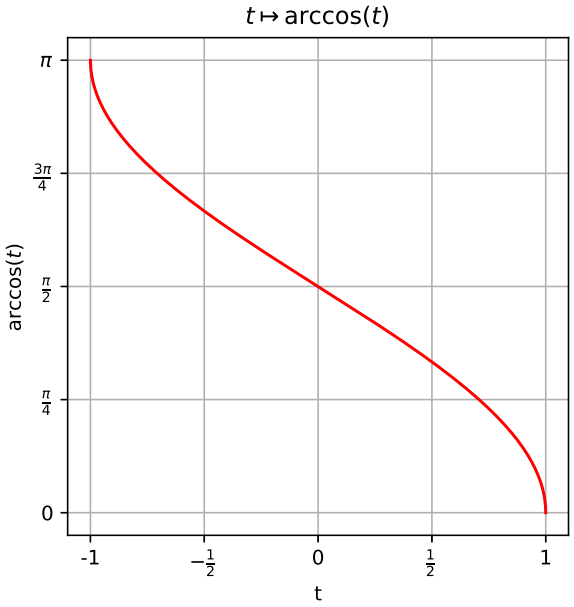


Fig. 86

The nodes and the weights of the n-point Gauss-Legendre quadrature rule on $[-1,1]$ can be computed for $n \leq 256$ using the provided C++ function

```
QuadRule gaussquad(const unsigned int n);
```

which initializes a data structure describing a quadrature rule

```
struct QuadRule {
  QuadRule() = default;
  explicit QuadRule(unsigned int n) : nodes_(n), weights_(n) {}
  Eigen::VectorXd nodes_;
  Eigen::VectorXd weights_;
};
```

The names of the data members tell their function. The function `gaussquad()` is declared in `gaussquad.hpp`.

(7-13.a) ⚀ (15 min.) In the file `arccosquad.hpp` implement a C++ function

```
void testConvGaussQuad();
```

that prints a table that allows you to predict *qualitatively and quantitatively* the *asymptotic behavior* (w.r.t. $n \to \infty$) of the quadrature error of n-point Gauss-Legendre numerical quadrature, when directly applied to (7.13.1) for $f(t) = \dfrac{1}{1+e^t}$.

Based on that table describe the observed empiric convergence of the quadrature error as a function of the number of quadrature nodes qualitatively and quantitatively. Explain, how you arrive at your

conclusions.

HIDDEN HINT 1 for (7-13.a) → 7-13-1-0:s1hacos.pdf

HIDDEN HINT 2 for (7-13.a) → 7-13-1-1:s1ref.pdf

SOLUTION for (7-13.a) → 7-13-1-2:.pdf ▲

(7-13.b) ⚅ (10 min.) With a suitable change of variables transform the integral (7.13.1) into another integral whose integrand will be C^∞-smooth on the *closed* interval of integration, if $f \in C^\infty([-1,1])$.

SOLUTION for (7-13.b) → 7-13-2-0:s2trf.pdf ▲

(7-13.c) ⚄ (10 min.) [depends on Sub-problem (7-13.b)]

In the file `arccosquad.hpp` complete the code of the C++ function

```
template <typename FUNCTION>
double arccosWeightedQuad(FUNCTION &&f,
                          unsigned int n);
```

that

- expects the functor argument f to pass a function $f : [-1,1] \to \mathbb{R}$ and to provide an evaluation operator **operator double** `()(double)const`,
- uses n evaluations of the function f to compute an approximation $I_n(f)$ of $I(f)$ from (7.13.1),
- that achieves exponential asymptotic convergence $I_n(f) \to I(f)$ for $n \to \infty$, if the function f possesses an analytic extension beyond $[-1,1]$.

SOLUTION for (7-13.c) → 7-13-3-0:s3sol.pdf ▲

(7-13.d) ⚄ (15 min.) [depends on Sub-problem (7-13.a) and Sub-problem (7-13.c)]

Analoguous to Sub-problem (7-13.a) in the file `arccosquad.hpp` write a C++ function

```
void testConvTrfGaussQuad();
```

that outputs (in a suitable table) information that allows you to predict *qualitatively and quantitatively* the *asymptotic behavior* (w.r.t. $n \to \infty$) of the approximation error of your implementation of `arccosWeightedQuad()` for $f(t) = \frac{1}{1+e^t}$.

Characterize the observed empiric convergence of the quadrature error as a function of n qualitatively and quantitatively. Justify your conclusions.

SOLUTION for (7-13.d) → 7-13-4-0:.pdf ▲

End Problem 7-13 , 50 min.

Problem 7-14: Aspects of Numerical Quadrature

A quadrature rule approximates the integral of a (continuous) function by means of a weighted sum of function values at so-called quadrature nodes/points. Numerical analysis studies the order of quadrature rules and the asymptotic convergence of the quadrature error as a function of the number of quadrature points.

This problem is linked with [Lecture → Section 7.2], [Lecture → Section 7.4.1], and [Lecture → Section 7.5] and also requires familiarity with [Lecture → § 7.5.0.18].

(7-14.a) ⚀ (10 min.) Complete the definition of the order of a quadrature rule:

Definition *cf.* [Lecture → Def. 7.4.1.1]. Order of a quadrature rule

The **order** of a quadrature rule $Q_n : C^0([a,b]) \to \mathbb{R}$ is defined as

$$\text{order}(Q_n) := \square \left\{ m \in \mathbb{N}_0 : \quad Q_n(\square) = \int_a^b \square \, \mathrm{d}t \quad \forall p \in \square \right\} .$$

SOLUTION for (7-14.a) → 7-14-1-0:bqns1.pdf ▲

(7-14.b) ⚄ (20 min.) The C++ function

```
unsigned int checkQuadOrder(
const Eigen::VectorXd &c, const Eigen::VectorXd &w,
const double tol = 1.0E-10);
```

expects that the input vectors `c` and `w` have the same length and contain the weights and nodes of a quadrature formula on $[-1,1]$. It returns the order of that quadrature formula. Supplement the missing parts of the following listing by writing valid C++ code in the boxes.

```
unsigned int checkQuadOrder(
const Eigen::VectorXd &c, const Eigen::VectorXd &w,
const double tol = 1.0E-10) {
  const unsigned long N = c.size();
  assert(N == w.size());
  for (unsigned int d = 0; d < [          ] ; ++d) {
    double s = 0.0;
    for (int j = 0; j < [          ] ; ++j) {
      s += [          ] * std::pow([          ] , d);
    }
    double val = (d % 2 == 0) ? [          ] : [          ] ;
    if (std::abs(s - val) > tol) {
      return [          ] ;
    }
  }
  return [          ] ;
```

```
}
```

HIDDEN HINT 1 for (7-14.b) → 7-14-2-0:nqms2h1.pdf

SOLUTION for (7-14.b) → 7-14-2-1:nqm2s.pdf ▲

(7-14.c) ⚄ (15 min.) The C++ function

```
template <typename FUNCTOR>
double trapezoidalRule(
FUNCTOR &&f, double a, double b, unsigned int N);
```

evaluates the **equidistant trapezoidal quadrature rule** with `N` nodes on the interval $[a,b]$ for a (continuous) function passed through the functor object `f`.

Write valid C++ code in the boxes of the following listing so that the function meets this specification.

```
template <typename FUNCTOR>
double trapezoidalRule(
FUNCTOR &&f, double a, double b, unsigned int N) {
  assert(a < b);
  assert(N >= 2);
  const double h = ;
  double t = a+h;
  double s = ;
  for (int j = 1; j < ; ++j) {
    s += ;
    t += h;
  }
  s += ;
  return ;
}
```

HIDDEN HINT 1 for (7-14.c) → 7-14-3-0:nqm3h1.pdf

SOLUTION for (7-14.c) → 7-14-3-1:nqms3.pdf ▲

(7-14.d) ⚄ (20 min.) The equidistant trapezoidal quadrature rule as implemented in Subproblem (7-14.c) is used for the approximate evaluation of the integrals

$$\int_a^b f_{\alpha,\beta}(t)\mathrm{d}t\,, \quad a = 0.33,\ b = 1.33\,,$$

where the family of functions $f_{\alpha,\beta} : \mathbb{R} \to \mathbb{R}$ is defined as

$$f_{\alpha,\beta}(t) := \sqrt{|1+\alpha\sin(\beta t)|}\,, \quad \alpha,\beta,t \in \mathbb{R}\,.$$

The following plots display the modulus of the quadrature error for certain members of this family of functions as a function of the number of quadrature nodes.

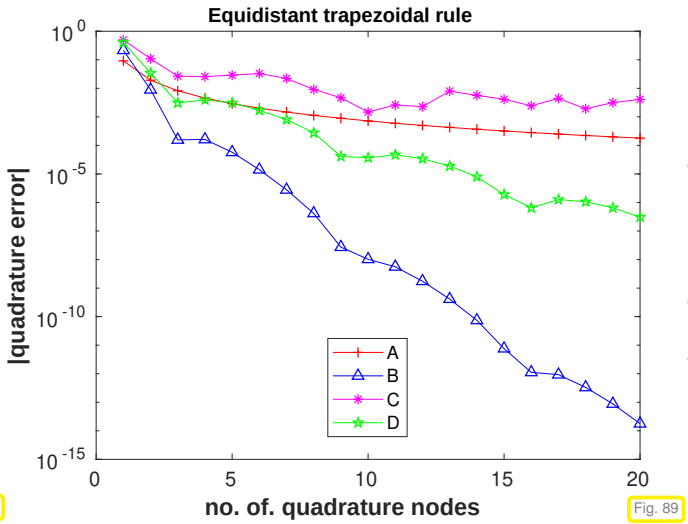


Fig. 88

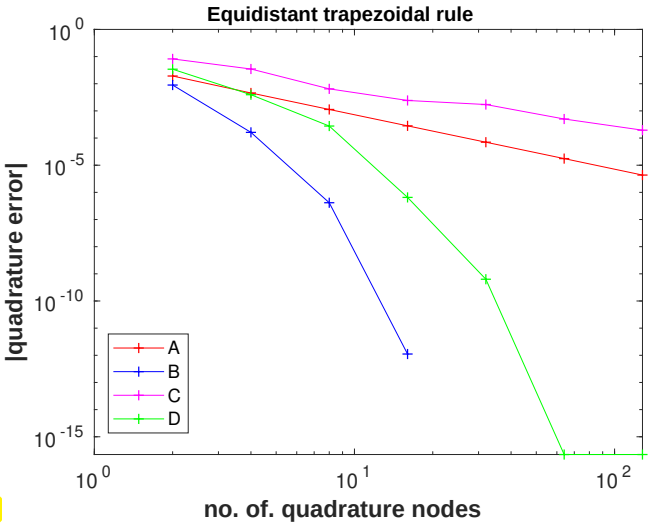


Fig. 89

State, which graph (A,B,C, or D) belongs to which pair of parameters α, β (defining the integrand $f_{\alpha,\beta}$).

Parameters	$\leftrightarrow$	Graph
(1) $\alpha = 0.9, \quad \beta = 2\pi$	$\leftrightarrow$	
(2) $\alpha = 0.5, \quad \beta = \pi$	$\leftrightarrow$	
(3) $\alpha = 1.1, \quad \beta = 2\pi$	$\leftrightarrow$	
(4) $\alpha = 0.5, \quad \beta = 2\pi$	$\leftrightarrow$	

SOLUTION for (7-14.d) → 7-14-4-0:nqms4.pdf ▲

End Problem 7-14 , 65 min.

Problem 7-15: Convolution Quadrature

Convolution quadrature is a special numerical technique for the evaluation of the convolution of two causal functions, when one is given in the form of its Laplace transform. This problem studies formulas for the weights of convolution quadrature and their efficient numerical computation.

This problem requires knowledge of the discrete Fourier transform (DFT) [Lecture → Def. 4.2.1.18], its implementation in EIGEN, some complex analysis, and numerical quadrature for periodic analytic functions [Lecture → § 7.5.0.18].

The **convolution** of two causal integrable functions $f, g : \mathbb{R}^+ := [0, \infty[\to \mathbb{R}$, written as $f * g$, is a function $\mathbb{R}^+ \to \mathbb{R}$ defined as

$$(f * g)(t) := \int_0^t f(t-\tau) g(\tau) \, \mathrm{d}\tau \,, \quad t \geq 0 \,. \tag{7.15.1}$$

Given a fixed "timestep size" $\tau > 0$, we approximate $f * g$ at equidistant times $t_n := \tau n$, $n \in \mathbb{N}_0$, by

$$(f * g)(t_n) \approx \sum_{\ell=0}^{n} w_{n-\ell}^{\tau} g_\ell \,, \quad g_\ell := g(t_\ell) \,, \quad n \in \mathbb{N}_0 \,. \tag{7.15.2}$$

with suitable convolution quadrature weights $w_j^\tau \in \mathbb{C}$, $j \in \mathbb{N}_0$.

In the sequel let the function f be fixed and denote by $F : D \subset \mathbb{C} \to \mathbb{C}$ its **Laplace transform**, which is supposed to be known.

Assumption 7.15.3. Holomorphy of the Laplace transform

The Laplace transform $F : D \subset \mathbb{C} \to \mathbb{C}$ of f is *analytic* (holomorphic) on an open set $D \subset \mathbb{C}$ with

$$\mathbb{C}^+ := \{ z \in \mathbb{C} : \ \operatorname{Re} z > 0 \} \subset D \,,$$

that is, F is analytic in the right half-plane of $\mathbb{C}$.

As explained in [HS16] the convolution quadrature weights w_j^τ in (7.15.2) are given by the following complex contour integrals: for any $0 < r < 1$

$$w_j^\tau = \frac{1}{2\pi \imath} (-1)^j \int_{|z|=r} \frac{1}{z^{j+1}} F\left(\frac{1-z}{\tau} \right) \mathrm{d}z \,, \quad j \in \mathbb{Z} \,. \tag{7.15.4}$$

With the parameterization $z(\varphi) := re^{2\pi\imath\varphi}$, $\varphi \in [0,1[$, we convert (7.15.4) into

$$w_j^\tau = r^{-j} \int_0^1 \psi_j(\varphi) \, \mathrm{d}\varphi \,, \quad \psi_j(\varphi) := e^{-2\pi\imath j\varphi} \cdot F\left(\frac{1 - re^{2\pi\imath\varphi}}{\tau} \right) \,, \quad j \in \mathbb{Z} \,. \tag{7.15.5}$$

(7-15.a) ⚄ (30 min.)

We identify $\psi_j : \mathbb{R} \to \mathbb{C}$ from (7.15.5) with its extension to a neighborhood of the real axis in the complex plane. Merely taking for granted Ass. 7.15.3, what is the *maximal* $\rho > 0$ such that ψ_j is analytic in the strip

$$S_\rho := \{ z \in \mathbb{C} : \ |\operatorname{Im} z| < \rho \} \ ?$$

Give that maximal ρ as a function of r:

$$\rho(r) = \boxed{} \,.$$

SOLUTION for (7-15.a) → 7-15-1-0:cqs1.pdf ▲

(7-15.b) ⚀ (20 min.) The integral in (7.15.5) is approximated by numerical quadrature using the equidistant $(N+1)$-point, $N \in \mathbb{N}_0$, **trapezoidal rule**. Write down the formula for the resulting approximate convolution quadrature weights $w_j^{\tau,N}$, $j \in \mathbb{Z}$.

$$w_j^\tau \approx w_j^{\tau,N} = \quad .$$

SOLUTION for (7-15.b) → 7-15-2-0:cqs2.pdf ▲

(7-15.c) ⚁ (60 min.) [depends on Sub-problem (7-15.b)]

In the file `convolutionquadrature.hpp` implement an *efficient* C++ function

```
template <typename FFUNCTOR>
  Eigen::VectorXcd compute_cq_weights(
    FFUNCTOR&& F, unsigned int N, double tau, double r = 0.0);
```

that computes the approximate convolution quadrature weights $w_j^{\tau,N}$ by means of the $(N+1)$-point equidistant trapezoidal rule and for $j = 0,\ldots,N$, and returns them as the components of a complex column vector. The functor object `F` passes the Laplace transform F. The parameter `r` specifies the radius r of the integration contour. If `r==0.0`, then a default value is chosen, see the code template.

"Efficient" means that the function should enjoy an asymptotic computational cost of $O(N \log N)$ for $N \to \infty$.

SOLUTION for (7-15.c) → 7-15-3-0:cqs3.pdf ▲

(7-15.d) ⚁ (15 min.) [depends on Sub-problem (7-15.a), Sub-problem (7-15.b)]

Assume computations in exact arithmetic. How will the error introduced into the computation of the convolution quadrature weights by numerical quadrature by means of the $(N+1)$-point equidistant trapezoidal rule, that is $|w_j^\tau - w_j^{\tau,N}|$, behave as a function of N asymptotically for $N \to \infty$? Explain your answer.

SOLUTION for (7-15.d) → 7-15-4-0:qc4s4.pdf ▲

References

[HS16] Matthew Hassell and Francisco-Javier Sayas. "Convolution quadrature for wave simulations". In: *Numerical simulation in physics and engineering*. Vol. 9. SEMA SIMAI Springer Ser. Cham: Springer, 2016, pp. 71–159 (cit. on p. 258).

End Problem 7-15 , 125 min.

Problem 7-16: On Quadrature Formulas

This problem addresses some key ideas of numerical quadrature, offering a repetition of considerations already covered in class.

This problem is linked to [Lecture → Section 7.2] and [Lecture → Section 7.4.1].

Descendants of the following abstract base class are supposed to provide weights and nodes for a quadrature formula on $[-1,1]$.

```
struct QuadFormulaRef {
  virtual const std::vector<double>& nodes() const = 0;
  virtual const std::vector<double>& weights() const = 0;
};
```

(7-16.a) ⚂ (10 min.) The following class is a type for a quadrature formula on a general interval $[a,b] \subset \mathbb{R}$.

```
class QuadFormula {
 public:
  QuadFormula(const QuadFormulaRef& qf, double a, double b);
  std::pair<double, double> interval() const { return {a_, b_}; }
  const std::vector<double>& nodes() const { return nodes_; }
  const std::vector<double>& weights() const { return weights_; }

 private:
  double a_;
  double b_;
  std::vector<double> nodes_;
  std::vector<double> weights_;
};
```

The constructor takes a quadrature formula object `qf` defined for the reference interval $[-1,1]$ plus the interval bounds a and b and initializes the class member variables `nodes_` and `weights_` with the nodes and weights of the quadrature formula transformed to $[a,b]$. Fill the blanks in Code 7.16.1 to complete the implementation of the constructor.

C++ code 7.16.1: Constructor of QuadFormula

```
1  QuadFormula::QuadFormula(const QuadFormulaRef& qf, double a, double b)
2      : a_(a), b_(b), nodes_(qf.nodes()), weights_(qf.weights()) {
3    const double len = b - a;
4    const unsigned int n_pts = nodes_.size();
5
6    for (unsigned int j = 0; j < n_pts; ++j) {
7      nodes_[j] =                                    ;
8      weights_[j] *=                                 ;
9    }
10 }
```

SOLUTION for (7-16.a) → 7-16-1-0:.pdf ▲

(7-16.b) ⚄ (20 min.) The C++ function `checkQuadOrder()` whose incomplete listing is given

as Code 7.16.3 determines and returns the order of a quadrature formula on $[-1,1]$ given to it via the `qf` argument.

Complete the listing so that the function can perform its function.

C++ code 7.16.3: Code for function `checkQuadOrder()`

```
unsigned int checkQuadOrder(const QuadFormulaRef& qf,
                            const double tol = 1.0E-10) {
  const std::vector<double>& c = qf.nodes();
  const std::vector<double>& w = qf.weights();
  const unsigned int n_pts = c.size();

  std::vector<double> monom_vals(n_pts, 1.0);
  for (unsigned int d = 0; d < n_pts; ++d) {
    double s = 0.0;
    for (unsigned int j = 0; j < n_pts; ++j) s += w[j] * monom_vals[j];
    double val = 2.0 / (2*d + 1);
    if ( ____________________________________ > tol) return 2 * d;
    for (unsigned int j = 0; j < n_pts; ++j) monom_vals[j] *= ____________________;
    s = 0.0;
    for (unsigned int j = 0; j < n_pts; ++j) s += w[j] * monom_vals[j];
    if ( ____________________________________ > tol) return 2 * d + 1;
    for (unsigned int j = 0; j < n_pts; ++j) monom_vals[j] *= ____________________;
  }
  return 2 * n_pts;
}
```

HIDDEN HINT 1 for (7-16.b) → 7-16-2-0:qfsbh1.pdf

SOLUTION for (7-16.b) → 7-16-2-1:.pdf ▲

End Problem 7-16, 30 min.

Problem 7-17: Quadrature Formulas

A conventional quadrature formula [Lecture → Def. 7.2.0.1] seeks to approximate a definite integral by means of a weighted sum of point values of the integrand:

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \sum_{j=1}^{n} w_j^{(n)} f(c_j^{(n)}) . \qquad \text{[Lecture → (7.2.0.2)]}$$

Here we recall some (classes of) quadrature formulas and the quality of approximation they offer.

This problem revisits many concepts and formulas which play a key role in [Lecture → Chapter 7].

(7-17.a) ⚂ (20 min.) The C++ function

```
template <typename FUNCTOR>
double trapezoidalRule(
FUNCTOR &&f, double a, double b, unsigned int N);
```

evaluates the **equidistant trapezoidal quadrature rule** with `N` nodes on the interval $[a,b]$ for a (continuous) function passed through the functor object `f`.

Write valid C++ code in the boxes of the following listing so that the function meets this specification.

```
template <typename FUNCTOR>
double trapezoidalRule(
FUNCTOR &&f, double a, double b, unsigned int N) {
  const double h = (b - a) / (N - 1);
  double t = a+h;
  double s = 0;
  for (int j = 0; j < N-1; ++j) {
    s += ;
    t += h;
  }
  return s;
}
```

HIDDEN HINT 1 for (7-17.a) → 7-17-1-0:nqm3h1.pdf

SOLUTION for (7-17.a) → 7-17-1-1:nqfs1.pdf

▲

(7-17.b) ⚂ (20 min.) Remember the concept of a polynomial quadrature rule:

Definition 7.17.1. Polynomial Quadrature

An n-point quadrature formula

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \sum_{j=1}^{n} w_j f(c_j) , \quad w_j \in \mathbb{R} , \quad c_j \in [a,b] . \qquad \text{[Lecture → (7.2.0.2)]}$$

is called **polynomial**, if its *order is at least* n.

Complete the following code of a C++ function

```
Eigen::VectorXd compWeightsPolyQR(
  const Eigen::VectorXd& nodes, double a, double b);
```

for the computation of the weights $w_1,\ldots,w_n$ of a polynomial n-point quadrature rule with given nodes c_j, $j = 1,\ldots,n$ passed in the vector `nodes` of length n.

```
Eigen::VectorXd compWeightsPolyQR(
  const Eigen::VectorXd& nodes, double a, double b) {
  // Number of quadrature nodes
  const unsigned int n = nodes.size();
  // Matrix for linear system of equations providing weights
  Eigen::MatrixXd M(n, n);
  for (int i = 0; i < n; ++i) {
    for (int j = 0; j < n; ++j) {
      M(i, j) =                                          ;
    }
  }
  // Set up right-hand side vector
  Eigen::VectorXd rhs(n);
  double b_pow_j = b;
  double a_pow_j = a;
  for (int j = 1; j <= n; ++j, a_pow_j *= a, b_pow_j *= b) {
    rhs[j - 1] = (b_pow_j - a_pow_j) / j;
  }
  return M.lu().solve(rhs);
}
```

HIDDEN HINT 1 for (7-17.b) → 7-17-2-0:nqfh2.pdf

SOLUTION for (7-17.b) → 7-17-2-1:nqfs2.pdf

▲

(7-17.c) ⚄ (20 min.) For a given quadrature formula

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \sum_{j=1}^{n} w_j f(c_j)\,, \quad w_j \in \mathbb{R}\,, \quad c_j \in [a,b]\,. \qquad \text{[Lecture → (7.2.0.2)]}$$

explicitly state a function $f = f(t)$, for which numerical integration with this quadrature formula will result in a *relative quadrature error* exactly equal to 1.

$$f(t) = \qquad .$$

HIDDEN HINT 1 for (7-17.c) → 7-17-3-0:nqfrelerr.pdf

SOLUTION for (7-17.c) → 7-17-3-1:nqfs3.pdf

▲

End Problem 7-17, 60 min.

Problem 7-18: C^∞-Regularizing Transformation

This problem studies numerical quadrature for singular integrands with a logarithmic singularity. In the spirit of [Lecture → Rem. 7.4.3.10] we remove the singularity by a suitable transformation of the integrand, tackle the resulting integral with Gauss quadrature and study convergence.

This problems assumes familiarity with the contents of [Lecture → Section 7.4] and involves some C++ coding based on EIGEN.

The file `cinfinitytrf.hpp` contains the C++ function

```
struct QuadRule { Eigen::VectorXd nodes_, weights_; }
QuadRule gaussquad(const unsigned int n);
```

listed in [Lecture → Code 7.4.2.24], which computes the nodes and weights of the n-point Gauss-Legendre quadrature [Lecture → Def. 7.4.2.18] on the interval $[-1,1]$.

For a continuous function $f : [0,1] \to \mathbb{R}$ we consider the well-defined improper integral

$$L(f) := \int_0^1 \log t \, f(t) \, \mathrm{d}t \, . \tag{7.18.1}$$

Our goal is to develop a quadrature formula for (7.18.1), which delivers fast convergence, if f possesses an analytic extension beyond the interval $[0,1]$. As a tool we use the **mollifier function**

$$\psi(s) := \begin{cases} \exp\left(-\frac{1}{s(1-s)}\right) & \text{for} \quad 0 < s < 1 \, , \\ 0 & \text{for} \quad s \le 0 \ \wedge \ s \ge 1 \, , \end{cases}$$
$$= \begin{cases} \exp\left(-\frac{1}{s}\right) \cdot \exp\left(-\frac{1}{1-s}\right) & \text{for} \quad 0 < s < 1 \, , \\ 0 & \text{for} \quad s \le 0 \ \wedge \ s \ge 1 \, . \end{cases} \tag{7.18.2}$$

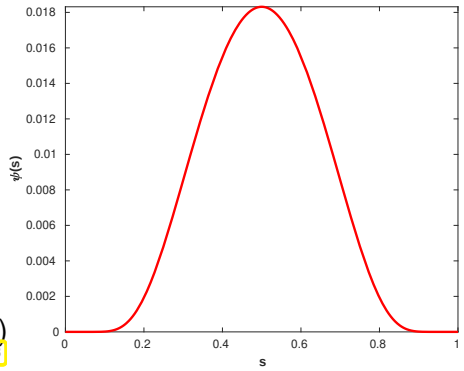


Fig. 93

By direct differentiation we can establish that this function is smooth:

Lemma 7.18.3. Smoothness of $s \mapsto \psi(s)$

The function $s \mapsto \psi(s)$ is well-defined for all $s \in \mathbb{R}$ and belongs to $C^\infty(\mathbb{R})$.

(7-18.a) ⚀ (15 min.) Determine the maximal subset D_ψ of $\mathbb{R}$ on which ψ is real-analytic.

$D_\psi =$ ____ .

HIDDEN HINT 1 for (7-18.a) → 7-18-1-0:cinfth1.pdf

SOLUTION for (7-18.a) → 7-18-1-1:cints1.pdf ▲

Defining the function performing coordinate transformation

$$\Phi(t) = c \int_0^t \psi(s) \, \mathrm{d}s \quad \text{with suitable} \quad c \in \mathbb{R} \tag{7.18.4}$$

and substituting $t = \Phi(\tau)$ ($\Rightarrow \mathrm{d}t = \Phi'(\tau)\mathrm{d}\tau = c\psi(\tau)\mathrm{d}\tau$), we want to transform the integral of (7.18.1):

$$L(f) = \int_0^1 \log t\, f(t)\,\mathrm{d}t = c\int_0^1 \log(\Phi(\tau))f(\Phi(\tau))\,\psi(\tau)\,\mathrm{d}\tau\,. \tag{7.18.5}$$

(7-18.b) ⚂ (20 min.) In the file `cinfinitytrf.hpp` complete the code of the function

```
double findConst(unsigned int n);
```

which computes the value of the constant c in (7.18.4) such that (7.18.5) holds true. The computation uses `n`-point Gauss-Legendre quadrature.

SOLUTION for (7-18.b) → 7-18-2-0:cinfts2.pdf ▲

(7-18.c) ⚂ (20 min.) Applying the n-point Gauss-Legendre quadrature rule to the transformed integral (7.18.5) yields an n-point quadrature formula for (7.18.1),

$$\int_0^1 \log t\, f(t)\,\mathrm{d}t \approx \sum_{j=1}^{n} \omega_j^{(n)} f(\zeta_j^{(n)})\,, \quad \omega_j > 0,\ \zeta_j \in]0,1[\,. \tag{7.18.7}$$

Express $\omega_j^{(n)}$ and $\zeta_j^{(n)}$ in terms of the weights/nodes $\widehat{w}_j^{(n)}$/$\widehat{c}_j^{(n)}$, $j = 1,\ldots,n$, of the n-point Gauss-Legendre quadrature rule on $[-1,1]$, and through the function ψ defined in (7.18.2),

$$\omega_j^{(n)} = \boxed{}\,,$$

$$\zeta_j^{(n)} = \boxed{}\,.$$

SOLUTION for (7-18.c) → 7-18-3-0:cinftrfs3.pdf ▲

(7-18.d) ⚄ (40 min.) [depends on Sub-problem (7-18.c)]

In the file `cinfinitytrf.hpp` complete the code of the function

```
QuadRule compLogQuadRule(unsigned int n);
```

which is supposed to compute *approximations* of the weights $\omega_j^{(n)}$ and nodes $\zeta_j^{(n)}$ from (7.18.7). The parameter `n` specifies the number of quadrature nodes.

SOLUTION for (7-18.d) → 7-18-4-0:cinftrfs4.pdf ▲

End Problem 7-18 , 95 min.

Chapter 8

Iterative Methods for Non-Linear Systems of Equations

Problem 8-1: Convergent Newton iteration

As explained in [Lecture → Section 8.4.2.1], the convergence of Newton's method in 1D may only be local. This problem investigates a particular setting, in which global convergence can be expected.

This is a purely theoretical problem practising "intuitive mathematical reasoning".

We recall the notion of a *convex function* [Lecture → Def. 5.3.1.4] and its geometric meaning [Lecture → Fig. 162]: A differentiable function $f : [a,b] \mapsto \mathbb{R}$ is convex if and only if its graph lies on or above its tangent at any point. Equivalently, differentiable function $f : [a,b] \mapsto \mathbb{R}$ is convex, if and only if its derivative is non-decreasing.

(8-1.a) ⚂ (5 min.) Give an example of a function $F \in C^2(\mathbb{R})$, for which **a** sequence of iterates produced by Newton's method does not converge.

SOLUTION for (8-1.a) → 8-1-1-0:.pdf ▲

(8-1.b) ⚂ (20 min.) Give a "graphical proof" of the following statement:

Theorem 8.1.1. Global convergence of Newton's method in 1D for convex monotone functions

If $F(x)$ belongs to $C^2(\mathbb{R})$, is strictly increasing, is convex, and has a unique zero, then the Newton iteration [Lecture → (8.4.2.1)] for $F(x) = 0$ is well defined and will converge to the zero of $F(x)$ for any initial guess $x^{(0)} \in \mathbb{R}$.

SOLUTION for (8-1.b) → 8-1-2-0:Conv1s.pdf

▲

End Problem 8-1 , (25 min.)

Problem 8-2: Code quiz

A frequently encountered drudgery in scientific computing is the use and modification of poorly documented code. This makes it necessary to understand the ideas behind the code first. Now we practice this in the case of a simple iterative method.

Related to [Lecture → Section 8.4.2.1], involves implementation in C++.

(8-2.a) ⚀ (20 min.) What is the purpose of the following C++ code?

C++ code 8.2.1: Undocumented function

```
2  double myfunction(double x) {
3    constexpr double dy = 0.693147180559945;  // = std::log(2)
4    double y = 0.;
5    while (x > 2. * std::sqrt(2.)) {
6      x /= 2.;
7      y += dy;
8    }  //
9    while (x < std::sqrt(2.)) {
10     x *= 2.;
11     y -= dy;
12   }                    //
13   double z = x - 1.;  //
14   double dz = x * std::exp(-z) - 1.;
15   while (std::abs(dz / z) > std::numeric_limits<double>::epsilon()) {
16     z += dz;
17     dz = x * std::exp(-z) - 1.;
18   }
19   return y + z + dz;  //
20 }
```

▲

HIDDEN HINT 1 for (8-2.a) → 8-2-1-0:Code1ha.pdf

HIDDEN HINT 2 for (8-2.a) → 8-2-1-1:Code1hb.pdf

SOLUTION for (8-2.a) → 8-2-1-2:Code1s.pdf

(8-2.b) ⚁ (10 min.) Explain the rationale behind the first two initial **while** loops in the code Code 8.2.1, Lines 5–12, preceding the main iteration.

SOLUTION for (8-2.b) → 8-2-2-0:Code2s.pdf ▲

(8-2.c) ⚁ (10 min.) Explain what the **while** loop in lines 13–18 of Code 8.2.1 does.

SOLUTION for (8-2.c) → 8-2-3-0:Code3s.pdf ▲

(8-2.d) ⚁ (10 min.) Explain the conditional expression in the last `while` loop.

SOLUTION for (8-2.d) → 8-2-4-0:Code4s.pdf ▲

(8-2.e) ⚄ (60 min.) [depends on Sub-problem (8-2.b)]

Write a C++ function

```
double myfunction_modified(double x);
```

where you replace the last `while`-loop with a fixed number of iterations that, nevertheless, guarantee that the result has a relative accuracy `EPS`. Derive that required minimal number of iterations!

HIDDEN HINT 1 for (8-2.e) → 8-2-5-0:s5h1.pdf

SOLUTION for (8-2.e) → 8-2-5-1:Code5s.pdf ▲

End Problem 8-2 , 110 min.

Problem 8-3: Newton's method for $F(x) := \arctan x = 0$

The merely local convergence of Newton's method is notorious, see [Lecture → Section 8.5.2] and [Lecture → Ex. 8.5.4.1]. The failure of the convergence is often caused by the overshooting of Newton correction. In this problem we try to understand the observations made in [Lecture → Ex. 8.5.4.1].

Moderate implementation in C++ is requested.

(8-3.a) ⚀ (20 min.) Find an equation satisfied by the smallest positive initial guess $x^{(0)}$ for which Newton's method does not converge when it is applied to $F(x) = \arctan x$.

HIDDEN HINT 1 for (8-3.a) → 8-3-1-0:NewtonArctan1ha.pdf

HIDDEN HINT 2 for (8-3.a) → 8-3-1-1:NewtonArctan1hb.pdf

SOLUTION for (8-3.a) → 8-3-1-2:NewtonArctan1s.pdf ▲

(8-3.b) ⚁ (20 min.) Implement a C++ function

```
double newton_arctan(double x0_ = 2.0);
```

that uses Newton's method to find an approximation of such $x^{(0)}$. The argument `x0_` can be used to supply an initial guess for the iteration.

The considerations from Sub-problem (8-3.a) suggest that we use an initial guess in $[1,2]$, we opt for $x^{(0)} = 2$.

SOLUTION for (8-3.b) → 8-3-2-0:NewtonArctan2s.pdf ▲

End Problem 8-3, 40 min.

Problem 8-4: A derivative-free iterative scheme for finding zeros

[Lecture → Rem. 8.2.2.12] shows how to detect the order of convergence of an iterative method from a numerical experiment. In this problem we study the so-called Steffensen's method, which is a derivative-free iterative method for finding zeros of functions in 1D.

Related to [Lecture → Section 8.2.2] and requests simple C++ coding.

Let $f : [a,b] \mapsto \mathbb{R}$ be twice continuously differentiable with $f(x^*) = 0$ and $f'(x^*) \neq 0$. Consider the iteration defined by

$$x^{(n+1)} := x^{(n)} - \frac{f(x^{(n)})}{g(x^{(n)})}, \quad \text{where} \quad g(x) = \frac{f(x + f(x)) - f(x)}{f(x)}. \tag{8.4.1}$$

(8-4.a) ⚀ (30 min.) Write a C++ function for the Steffensen's method:

```
template <class Function>
double steffensen(Function &&f, double x0)
```

`f` is a functor object or lambda function providing f, `x0` is the initial guess $x^{(0)}$. Terminate the iteration when the computed sequence of approximations becomes stationary, see [Lecture → Code 8.2.3.6] for an example.

SOLUTION for (8-4.a) → 8-4-1-0:Quad1s.pdf ▲

(8-4.b) ⚁ (15 min.) [depends on Sub-problem (8-4.a)]

Write a C++ function

```
void testSteffensen();
```

that applies your implementation of `steffensen()` to find the zero of the function $f(x) = xe^x - 1$ (see [Lecture → Exp. 8.3.1.3]). Use $x^{(0)} = 1$ as initial guess.

SOLUTION for (8-4.b) → 8-4-2-0:sq2.pdf ▲

(8-4.c) ⚂ (30 min.) [depends on Sub-problem (8-4.a)]

We now want to investigate the order of Steffensen's method. To do so extend your implementation of `steffensen()` to

```
template <typename Function>
Eigen::VectorXd steffensen_log(Function &&f, double x0, LOGGER
   &&log = [](double) -> void {});
```

such that it logs each iterate using a lambda function `log()` as described in [Lecture → § 0.3.3.4]. Then use this new facility to implement a function

```
void orderSteffensen();
```

that tabulates values from which you can read of the order of Steffensen's method when applied to the test case of Sub-problem (8-4.b).

HIDDEN HINT 1 for (8-4.c) → 8-4-3-0:steffh1.pdf

SOLUTION for (8-4.c) → 8-4-3-1:sq3.pdf ▲

(8-4.d) ⚂ (10 min.) For the test case of Sub-problem (8-4.b) the function $g(x)$ from (8.4.1) contains a term like e^{xe^x}. Therefore it grows very fast in x and the method cannot start for large $x^{(0)}$. How can you modify the function f (keeping the same zero) in order to allow the choice of a larger initial guess?

HIDDEN HINT 1 for (8-4.d) → 8-4-4-0:Quad2h.pdf

SOLUTION for (8-4.d) → 8-4-4-1:Quad2s.pdf ▲

End Problem 8-4 , 85 min.

Problem 8-5: Order-p convergent iterations

In [Lecture → Section 8.2.2] we investigated the speed of convergence of iterative methods for the solution of a general non-linear problem $F(\mathbf{x}) = 0$ and introduced the notion of **convergence of order** $p \geq 1$, see [Lecture → Def. 8.2.2.10]. This problem highlights the fact that for $p > 1$ convergence may not be guaranteed, even if the error norm estimate of [Lecture → Def. 8.2.2.10] may hold for some $\mathbf{x}^* \in \mathbb{R}^n$ and all iterates $\mathbf{x}^{(k)} \in \mathbb{R}^n$.

Problem with a theoretical focus and a little C++ coding. Relies on the concepts introduced in [Lecture → Section 8.2.2]

Given $\mathbf{x}^* \in \mathbb{R}^n$, suppose that a sequence $\mathbf{x}^{(k)}$ satisfies [Lecture → Def. 8.2.2.10]:

$$\exists C > 0: \quad \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C\|\mathbf{x}^{(k)} - \mathbf{x}^*\|^p \qquad \forall k \quad \text{and} \quad p > 1\,. \tag{8.5.1}$$

(8-5.a) ⚀ (20 min.) Determine $\epsilon_0 > 0$ as large as possible such that we have

$$\|\mathbf{x}^{(0)} - \mathbf{x}^*\| \leq \epsilon_0 \quad \Longrightarrow \quad \lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*\,. \tag{8.5.2}$$

In other words, ϵ_0 tells us which distance of the initial guess from $\mathbf{x}^*$ still guarantees local convergence.

HIDDEN HINT 1 for (8-5.a) → 8-5-1-0:ocvh1.pdf

SOLUTION for (8-5.a) → 8-5-1-1:OrdC1s.pdf ▲

(8-5.b) ⚁ (15 min.) Provided that $\|\mathbf{x}^{(0)} - \mathbf{x}^*\| < \epsilon_0$ is satisfied with ϵ_0 from Sub-problem (8-5.a), determine the minimal iteration step $k_{\min} = k_{\min}(\epsilon_0, C, p, \tau)$ such that $\|\mathbf{x}^{(k)} - \mathbf{x}^*\| < \tau$.

SOLUTION for (8-5.b) → 8-5-2-0:OrdC2s.pdf ▲

(8-5.c) ⚂ (30 min.) Implement a C++ function

```
void kminplot();
```

that uses MATPLOTLIBCPP's `plt::imshow()` to create a image of $k_{\min}(\epsilon_0, \tau)$ for the values $p = 1.5$, $C = 2$ and

$$\epsilon_0 \in \left[0, C^{\frac{1}{1-p}}\right]^2 \quad , \quad \tau \in [10^{-5}, 10^{-1}]\,.$$

Use a logarithmic axis scale for τ.

SOLUTION for (8-5.c) → 8-5-3-0:OrdC3s.pdf ▲

End Problem 8-5, 65 min.

Problem 8-6: Order of convergence from error recursion

In [Lecture → Exp. 8.4.2.32] we have observed *fractional* orders of convergence ([Lecture → Def. 8.2.2.10]) for both the secant method and the quadratic inverse interpolation method. This is fairly typical for 2-point methods in 1D and arises from the underlying recursions for error bounds. This problem addresses how to determine the order of convergence from an abstract error recursion.

A similar analysis is elaborated for the secant method in [Lecture → Rem. 8.4.2.33], where a linearised error recursion is given in [Lecture → (8.4.2.37)].

For an iterative scheme producing the sequence $\left(x^{(n)}\right)_{n\in\mathbb{N}}$, $x^{(n)} \in \mathbb{R}$ we suppose a recursive bound for the norms of the iteration errors of the form

$$\|e^{(n+1)}\| \leq \|e^{(n)}\|\sqrt{\|e^{(n-1)}\|}\,, \tag{8.6.1}$$

where $e^{(n)} = x^{(n)} - x^*$ is the error of n-th iterate.

(8-6.a) ⚀ (20 min.) Write C++ function

```
double testOrder(const unsigned int n = 20);
```

that guesses the maximal order of convergence of the method from a numerical experiment with n iterations.

HIDDEN HINT 1 for (8-6.a) → 8-6-1-0:RecursionOrder1h.pdf

SOLUTION for (8-6.a) → 8-6-1-1:RecursionOrder1s.pdf ▲

(8-6.b) ⚄ (30 min.) Find the maximal guaranteed order of convergence of this method by mathematical analysis as in [Lecture → Rem. 8.4.2.33].

HIDDEN HINT 1 for (8-6.b) → 8-6-2-0:RecursionOrder2ha.pdf

HIDDEN HINT 2 for (8-6.b) → 8-6-2-1:RecursionOrder2hb.pdf

HIDDEN HINT 3 for (8-6.b) → 8-6-2-2:RecursionOrder2hc.pdf

SOLUTION for (8-6.b) → 8-6-2-3:RecursionOrder2s.pdf ▲

End Problem 8-6, 50 min.

Problem 8-7: Nonlinear electric circuit

[Lecture → Ex. 2.1.0.3] discusses electric circuits with elements that give rise to linear voltage–current dependence, see [Lecture → Ex. 2.1.0.3] and [Lecture → Ex. 2.8.0.1]. The principles of nodal analysis were explained in these cases.
However, the electrical circuits encountered in practise usually feature elements with a *non-linear* current-voltage characteristic. Then nodal analysis leads to non-linear systems of equations as was elaborated in [Lecture → Ex. 8.1.0.1]. Please note that transformation to frequency domain is not possible for non-linear circuits so that we will always study the direct current (DC) situation.

In this problem we deal with a very simple non-linear circuit element, a diode. The current I through a diode to which a voltage U is applied can be modelled by the relationship

$$I = \alpha\left(e^{\beta\frac{U}{U_T}} - 1\right)$$

with suitable parameters $\alpha, \beta > 0$ and the thermal voltage U_T.
Now we consider the circuit depicted in Fig. 103 and assume that all resistors have non-dimensional resistance $R = 1$.

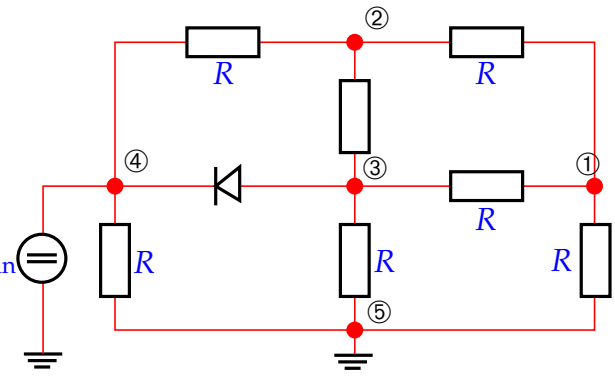


Fig. 103

The circuit is driven by imposing a voltage $U_{\rm in}$ through an ideal voltage source.

(8-7.a) ⚀ (20 min.) Study again [Lecture → Ex. 2.1.0.3] and [Lecture → Ex. 8.1.0.1] to refresh your knowledge about the nodal analysis of electric circuits. ▲

(8-7.b) ⚁ (20 min.) Carry out the nodal analysis of the electric circuit and derive the corresponding non-linear system of equations $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ for the potentials in nodes ①, ② and ③, *cf.* [Lecture → (8.1.0.2)]. Note that the voltages in nodes 4 and 5 are known (given input voltage $U_{\rm in}$ and ground voltage 0).

SOLUTION for (8-7.b) → 8-7-2-0:NonL1s.pdf ▲

(8-7.c) ⚂ (20 min.) [depends on Sub-problem (8-7.b)]

In the file `nonlinear_circuit.hpp` write an EIGEN-based C++ function

```
void circuit(const alpha, double beta,
const Eigen::VectorXd &Uin, Eigen::VectorXd &Uout)
```

that computes the output voltages `Uout`, which is defined to be the voltage at node ① in Fig. 103 for a *sorted* vector of input voltages `Uin` (at node ④) and for a thermal voltage $U_T = 0.5$. The parameters `alpha`, `beta` pass the (non-dimensional) diode parameters α, β.

Use Newton's method to solve $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ with a relative tolerance of $\tau = 10^{-6}$.

SOLUTION for (8-7.c) → 8-7-3-0:NonL2s.pdf ▲

(8-7.d) ⚁ (15 min.) [depends on Sub-problem (8-7.c)]

We are interested in the nonlinear effects introduced by the diode. Using MATPLOTLIBCPP implement a C++ function

```
void plotU();
```

that creates a plot of the output voltage $U_{\text{out}} = U_{\text{out}}(U_{\text{in}})$ as a function of the variable input voltage $U_{\text{in}} \in [0, 20]$ (for non-dimensional parameters $\alpha = 8$, $\beta = 1$ and for a thermal voltage $U_T = 0.5$). How can you discern the non-linearity of the circuit in the plot?

SOLUTION for (8-7.d) → `8-7-4-0:NonL3s.pdf` ▲

End Problem 8-7 , 75 min.

Problem 8-8: Julia Set

Julia sets are famous fractal shapes in the complex plane. They are constructed from the basins of attraction of zeros of complex functions when the Newton method is applied to find them.

This problem treats Newton's method in 2D and is related to [Lecture → Section 8.5]. You may watch this video by 3Blue1Brown to learn some background, see also here.

In the space $\mathbb{C}$ of complex numbers the equation

$$z^3 = 1 \tag{8.8.1}$$

has three solutions,

$$z_1 = 1 \quad , \quad z_2 = -\frac{1}{2} + \frac{1}{2}\sqrt{3}\imath \quad , \quad z_3 = -\frac{1}{2} - \frac{1}{2}\sqrt{3}\imath \, , \tag{8.8.2}$$

the cubic roots of unity.

(8-8.a) ⚀ (15 min.) As you know from the analysis course, the complex plane $\mathbb{C}$ can be identified with $\mathbb{R}^2$ via $(x,y) \in /bbR^2 \mapsto z = x + \imath y \in \mathbb{C}$. Using this identification, convert (8.8.1) into a system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ for a suitable function $\mathbf{F} : \mathbb{R}^2 \mapsto \mathbb{R}^2$.

SOLUTION for (8-8.a) → 8-8-1-0:JuliaSet1s.pdf ▲

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

Formulate the Newton iteration [Lecture → (8.5.1.6)] for the non-linear equation $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ with $\mathbf{x} = [x,y]^T$ and $\mathbf{F}$ from Sub-problem (8-8.a).

SOLUTION for (8-8.b) → 8-8-2-0:JuliaSet2s.pdf ▲

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

In the file `julia.hpp` implement two C++ functions

```
Eigen::Vector2d juliaF(const Eigen::Vector2d& x);
Eigen::Matrix2d juliaDF(const Eigen::Vector2d& x);
```

that return $F(\mathbf{x})$ and the Jacobian $\mathrm{D}F(\mathbf{x})$ for for the function $F : \mathbb{R}^2 \to \mathbb{R}^2$ found in Sub-problem (8-8.a).

SOLUTION for (8-8.c) → 8-8-3-0:sa1.pdf ▲

(8-8.d) ⚂ (60 min.) [depends on Sub-problem (8-8.b)]

Denote by $\mathbf{x}^{(k)}$ the iterates produced by Newton method from the previous subproblem with some initial vector $\mathbf{x}^{(0)} \in \mathbb{R}^2$. Depending on $\mathbf{x}^{(0)}$, the sequence $\mathbf{x}^{(k)}$ will either diverge or converge to one of the three cubic roots of unity.

Analyze the behavior of Newton iterations by means of a C++ code using the following approach:

- Use equally spaced points on the domain $[-2,2]^2 \subset \mathbb{R}^2$ as starting points of Newton iterations.
- Color the starting points differently depending on which of the three roots is the limit of the sequence $\mathbf{x}^{(k)}$.

Concretely complete the implementation of the C++ function

```
void julia();
```

supplied in `julia.hpp`. It is creates and saves a plot in `julia.png`.

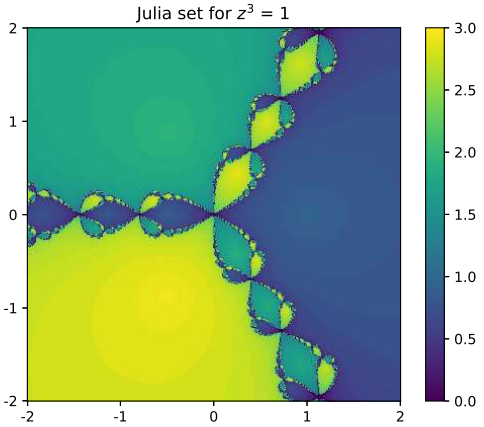


Fig. 105

◁ The Julia set for $z^3 - 1 = 0$ on a mesh containing 780×780 points, `N_it=20`.

This is how your final plot should look like.

Remark. The boundary of the three so-called domain of attraction of the three different zeros is a fractal set: At whatever scale you look at it, it displays the same complex shape.

HIDDEN HINT 1 for (8-8.d) → `8-8-4-0:JuliaSet3h.pdf`

SOLUTION for (8-8.d) → `8-8-4-1:JuliaSet3s.pdf` ▲

End Problem 8-8, 105 min.

Problem 8-9: Modified Newton method

The following problem consists in EIGEN implementation of a modified version of the Newton method (in one dimension [Lecture → Section 8.4.2.1] and many dimensions [Lecture → Section 8.5]) for the solution of a nonlinear system.

Involves coding in C++. Refresh your knowledge of stopping criteria for iterative methods [Lecture → Section 8.2.3].

For the solution of the non-linear system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ (with $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$), the following iterative method has been proposed:

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})\,, \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{y}^{(k)})\,, \end{aligned} \tag{8.9.1}$$

where $\mathrm{D}\,\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$ is the Jacobian matrix of $\mathbf{F}$ evaluated in the point $\mathbf{x}$.

(8-9.a) ⚀ (15 min.) Assume that $\mathbf{F}$ is injective,

$$\mathbf{x}, \mathbf{y} \in \mathbb{R}^n\,, \quad \mathbf{F}(\mathbf{x}) = \mathbf{F}(\mathbf{y}) \quad \Rightarrow \quad \mathbf{x} = \mathbf{y}\,. \tag{8.9.2}$$

Show that under this assumption the iteration (8.9.1) is *consistent* with $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ in the sense of [Lecture → Def. 8.3.1.1], that is, show that, under the assumption that $\mathrm{D}\,\mathbf{F}(\mathbf{x}^*)$ is regular for every fixed point $\mathbf{x}^*$ of the iteration, $\mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every $k \in \mathbb{N}$ if and only if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$.

SOLUTION for (8-9.a) → 8-9-1-0:Modi1s.pdf ▲

(8-9.b) ⚀ (20 min.) Implement a C++ function

```
template <typename Scalar, class Function, class Jacobian>
Scalar mod_newt_step_scalar(const Scalar& x,
                            Function &&f, Jacobian &&df);
```

that computes a step of the modified Newton method for a *scalar* function $\mathbf{F}$, that is, for the case $n = 1$.

Here, `f` is a functor object of type `Function` passing the function $F : \mathbb{R} \mapsto \mathbb{R}$ and `df` a functor object of type `Jacobian` passing the derivative $F' : \mathbb{R} \mapsto \mathbb{R}$. Both require an appropriate evaluation operator **operator**`()` and have to conform with **std**`::function<`**double**`(`**double**`)>`.

SOLUTION for (8-9.b) → 8-9-2-0:Modi2s.pdf ▲

(8-9.c) ⚀ (30 min.) What is the order of convergence of the method described in (8.9.1)?

To investigate it, write a C++ function

```
void mod_newt_ord();
```

that

- uses the function `mod_newt_step` to the following scalar equation: $\arctan(x) - 0.123 = 0$,
- generates a suitable terminal output that makes it possible to empirically determine the order of convergence, in the sense of [Lecture → Rem. 8.2.2.12],
- implements meaningful stopping criteria, see [Lecture → Section 8.2.3].

Use $x_0 = 5$ as initial guess and tabulate data from which you can read off the order.

HIDDEN HINT 1 for (8-9.c) → 8-9-3-0:Modi3ha.pdf

HIDDEN HINT 2 for (8-9.c) → 8-9-3-1:Modi3hb.pdf

SOLUTION for (8-9.c) → 8-9-3-2:Modi3s.pdf ▲

(8-9.d) ⚄ (20 min.) Implement a templated C++ function

```
template <typename Vector, typename Function, typename Jacobian>
Vector mod_newt_step_system(const Vector &x,
                            Function &&f , Jacobian& df);
```

that *efficiently* realizes a step of the iteration (8.9.1) for a function $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ passed through the functor `f`, which complies with `std::function<Vector(const Vector &)>`; The **Vector** type can be assumed to have the capabilities of **Eigen::VectorXd**, whereas the type **Jacobian** must have an evaluation operator that returns an object compatible with **Eigen::MatrixXd**.

HIDDEN HINT 1 for (8-9.d) → 8-9-4-0:3xh1.pdf

SOLUTION for (8-9.d) → 8-9-4-1:s3x.pdf ▲

In the sequel we consider the non-linear system of equations

$$\mathbf{F}(\mathbf{x}) := \mathbf{A}\mathbf{x} + \begin{bmatrix} c_1 e^{x_1} \\ \vdots \\ c_n e^{x_n} \end{bmatrix} = \mathbf{0}\,, \tag{8.9.8}$$

where $\mathbf{A} \in \mathbb{R}^{n,n}$ is symmetric positive definite, $\mathbf{x} = \left[x_j\right]_{j=1}^n \in \mathbb{R}^n$ is the solution vector, and $c_i \geq 0$, $i = 1, \ldots, n$.

(8-9.e) ⚂ Compute the Jacobian matrix $\mathrm{D}\,\mathbf{F}(\mathbf{x})$ for $\mathbf{F}$ as given in (8.9.8).

SOLUTION for (8-9.e) → 8-9-5-0:s4x.pdf ▲

(8-9.f) ⚂ (30 min.) [depends on Sub-problem (8-9.d)]

Based on your implementation of `mod_newt_step_system()`, create a C++ function

```
Eigen::VectorXd mod_newt_sys(const Eigen::Matrix &A,
                             const Eigen::VectorXd& c,
                             double tol 1.0E-6, int maxit = 100);
```

that uses the modified Newton iteration (8.9.1) to solve (8.9.8). The argument `A` passes the matrix $\mathbf{A}$ and `c` supplies the values c_j.

Stop the iteration and return the approximate solution when the Euclidean norm of the increment $\mathbf{x}^{(k+1)} - \mathbf{x}^{(k)}$ relative to the norm of $\mathbf{x}^{(k+1)}$ is smaller than the tolerance passed in `tol` or if `maxit` steps of the iterations have been carried out. Use the zero vector as initial guess.

SOLUTION for (8-9.f) → 8-9-6-0:s4m.pdf ▲

(8-9.g) ⚄ (30 min.) [depends on Sub-problem (8-9.f)]

Write a C++ function

```
void mod_newt_sys_test();
```

that tests your implementation of `mod_newt_sys()` for

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}\,, \quad \mathbf{c} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}\,,$$

and determines the order of convergence from suitable tabulated values.

HIDDEN HINT 1 for (8-9.g) → 8-9-7-0:s4h1.pdf

SOLUTION for (8-9.g) → 8-9-7-1:Modi4s.pdf ▲

End Problem 8-9 , 145 min.

Problem 8-10: Solving a quasi-linear system

In [Lecture → § 8.5.1.25] we studied Newton's method for a so-called quasi-linear system of equations, see [Lecture → (8.5.1.26)]. In [Lecture → Ex. 8.5.1.30] we then dealt with concrete quasi-linear system of equations and in this problem we will supplement the theoretical considerations from class by implementation in EIGEN. We will also learn about a simple fixed point iteration for that system, see [Lecture → Section 8.3].

Refresh yourself about the relevant parts of the lecture. You should also try to recall the Sherman-Morrison-Woodbury formula [Lecture → Lemma 2.6.0.21].

Consider the *nonlinear* (quasi-linear) system:

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}\,, \tag{8.10.1}$$

as in [Lecture → Ex. 8.5.1.30]. Here, $\mathbf{A} : \mathbb{R}^n \to \mathbb{R}^{n,n}$ is a matrix-valued function:

$$\mathbf{A}(\mathbf{x}) := \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \end{bmatrix}, \quad \gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2 \tag{8.10.2}$$

where $\|\cdot\|_2$ is the Euclidean norm.

(8-10.a) ⚀ (10 min.) A fixed point iteration can be obtained from (8.10.1) by the "frozen argument technique"; in a step we take the argument to the matrix valued function from the previous step and just solve a linear system for the next iterate. State the defining recursion and iteration function for the resulting fixed point iteration. ▲

SOLUTION for (8-10.a) → 8-10-1-0:s1.pdf

(8-10.b) ⚁ (30 min.) [depends on Sub-problem (8-10.a)]

We consider the fixed point iteration derived in Sub-problem (8-10.a). Implement an *efficient* EIGEN-based C++ function

```
Eigen::VectorXd fixed_point_step(const Eigen::VectorXd &xk,
                                 const Eigen::VectorXd &b);
```

that computes the iterate $\mathbf{x}^{(k+1)}$ from $\mathbf{x}^{(k)}$.

HIDDEN HINT 1 for (8-10.b) → 8-10-2-0:q2h1.pdf

SOLUTION for (8-10.b) → 8-10-2-1:QuasiLinear2s.pdf ▲

(8-10.c) ⚁ (20 min.) [depends on Sub-problem (8-10.a)]

Write a C++ function

```
Eigen::VectorXd solveQuasiLinSystem(double rtol, double atol
                                    const Eigen::VectorXd &b);
```

that finds the solution $\mathbf{x}^*$ of (8.10.1), (8.10.2) with the fixed point method applied to the previous quasi-linear system. Use $\mathbf{x}^{(0)} = \mathbf{b}$ as initial guess. Supply it with a suitable *correction based stopping criterion*

as discussed in [Lecture → Section 8.2.3] and pass absolute and relative tolerance as arguments `atol` and `rtol`.

SOLUTION for (8-10.c) → 8-10-3-0:QuasiLinear3s.pdf ▲

(8-10.d) ⚀ (15 min.) Let $\mathbf{b} \in \mathbb{R}^n$ be given. Write the recursion formula for the solution of

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \tag{8.10.6}$$

with the Newton method.

SOLUTION for (8-10.d) → 8-10-4-0:QuasiLinear4s.pdf ▲

(8-10.e) ⚁ (15 min.) The matrix $\mathbf{A}(\mathbf{x})$, being symmetric and tri-diagonal, is cheap to invert, see, e.g., [Lecture → Rem. 3.3.4.4]. Rewrite the previous iteration from Sub-problem (8-10.d) exploiting the Sherman-Morrison-Woodbury inversion formula for rank-one modifications from [Lecture → Lemma 2.6.0.21].

SOLUTION for (8-10.e) → 8-10-5-0:QuasiLinear5s.pdf ▲

(8-10.f) ⚂ (30 min.) [depends on Sub-problem (8-10.e)]

Write an EIGEN-based C++ function

```
Eigen::VectorXd newton_step(const Eigen::VectorXd &x
                            const Eigen::VectorXd &b);
```

the realizes a single step of the Newton method as derived in Sub-problem (8-10.e).

HIDDEN HINT 1 for (8-10.f) → 8-10-6-0:QuasiLinear6ha.pdf

SOLUTION for (8-10.f) → 8-10-6-1:QuasiLinear6s.pdf ▲

(8-10.g) ⚁ (10 min.) [depends on Sub-problem (8-10.f)]

Repeat Sub-problem (8-10.c) for the Newton method. To that end implement a C++ function

```
Eigen::VectorXd solveQLSystem_Newton(double rtol, double atol
                                     const Eigen::VectorXd &b);
```

that solves (8.10.1), (8.10.2) with Newton's method. As initial guess use $\mathbf{x}^{(0)} = \mathbf{b}$. The parameters `atol` and `rtol` enter the stopping criterion as discussed in [Lecture → Section 8.5.3].

SOLUTION for (8-10.g) → 8-10-7-0:QuasiLinear7s.pdf ▲

End Problem 8-10, 130 min.

Problem 8-11: Radioactive Decay

Least-squares estimation is still the main mathematical tool for parameter estimation. This exercise studies a particular (non-linear) least-squares parameter estimation problem.

Requires familiarity with [Lecture → Section 8.7] and involves moderate coding in C++.

For radioactive substances A and B consider the decay chain

$$A \xrightarrow{\lambda_1} B \xrightarrow{\lambda_2} C.$$

That is A decays into B with a decay rate λ_1. B itself decays into a third substance C with decay rate λ_2. The evolution of the amounts $\Phi_A(t), \Phi_B(t)$ of the substances A and B can be modelled by the following coupled system of ordinary differential equations, see [Str09, Bsp. 5.6.1 ii)]:

$$\begin{aligned} \frac{d\Phi_A}{dt}(t) &= -\lambda_1 \Phi_A(t) \\ \frac{d\Phi_B}{dt}(t) &= -\lambda_2 \Phi_B(t) + \lambda_1 \Phi_A(t) \,. \end{aligned} \tag{8.11.1}$$

(8-11.a) ⚀ (30 min.) Calculate the analytical solution of the system (8.11.1) using the method of variation of constants, see [Str09, Sect. 5.7].

HIDDEN HINT 1 for (8-11.a) → `8-11-1-0:ra1h1.pdf`

HIDDEN HINT 2 for (8-11.a) → `8-11-1-1:ra1h2.pdf`

SOLUTION for (8-11.a) → `8-11-1-2:rd1.pdf` ▲

(8-11.b) ⚁ (15 min.) [depends on Sub-problem (8-11.a)]

Unfortunately, only the quantity $\Phi_B(t)$ can be measured and the available values are affected by significant measurement errors. Nevertheless, we would like to estimate the initial quantities A_0, B_0 and the decay rates λ_1 and λ_2. This leads to an overdetermined *non-linear* system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, which has to be solved in least squares sense, see [Lecture → Section 8.7]. Here, the vector $\mathbf{x}$ of unknown parameters is $[A_0, B_0, \lambda_1, \lambda_2]^T \in \mathbb{R}^4$.

The measurements (t_i, m_i), $i \in \{1, \ldots, N\}$, are stored in the lines of the file `decay.txt` : t_i are the times of measurements and m_i are the measured values of Φ_B.

Write down the function $\mathbf{F}$ for the current problem.

SOLUTION for (8-11.b) → `8-11-2-0:rd2.pdf` ▲

(8-11.c) ⚂ (20 min.) [depends on Sub-problem (8-11.b)]

We want to use the Gauss-Newton method (see [Lecture → Section 8.7.2]) for solving the non-linear problem described above.

Write down the concrete iteration formulas [Lecture → (8.7.2.1)] for our special problem. In order to do this, compute the Jacobian $\mathbf{DF}$.

Which linear least squares problem is solved in each Gauss-Newton step?

SOLUTION for (8-11.c) → `8-11-3-0:rd3.pdf` ▲

(8-11.d) ⚄ (30 min.) [depends on Sub-problem (8-11.c)]

Write a C++ function

```
template <typename F, typename DF>
std::vector<double> GaussNewton(Eigen::Vector4d& x, F&& f, DF&& df,
const double tol = 1e-14)
```

that implements the Gauss-Newton method for estimating the parameters A_0, B_0, λ_1, and λ_2. The function outputs a vector containing L_∞ norm of the updates at each step of the iteration. The estimated parameters are to be stored in the object `x`.

Describe qualitatively and quantitatively the observed convergence, when choosing as initial guess $\mathbf{x}^{(0)} = [1,1,1,0.1]^T$.

HIDDEN HINT 1 for (8-11.d) → 8-11-4-0:radsh2.pdf

HIDDEN HINT 2 for (8-11.d) → 8-11-4-1:Radioactive4h.pdf

SOLUTION for (8-11.d) → 8-11-4-2:rd4.pdf ▲

References

[Str09] M. Struwe. *Analysis für Informatiker*. Lecture notes, ETH Zürich. 2009 (cit. on p. 283).

End Problem 8-11 , 95 min.

Problem 8-12: Approximation of a circle

In this problem we study how a fitting problem arising in computational geometry can be solved using a least squares approach in several ways, leading to different results.

Relies on [Lecture → Section 8.7] and involves substantial implementation in C++ based on EIGEN.

Let us consider a sequence of N of points $\in \mathbb{R}^2$ approximately located on a circle ($N = 8$):

x_i	0.7	3.3	5.6	7.5	6.4	4.4	0.3	-1.1
y_i	4.0	4.7	4.0	1.3	-1.1	-3.0	-2.5	1.3

A circle with center $[m_1, m_2]^\top \in \mathbb{R}^2$ and radius $r > 0$ is described by

$$(x - m_1)^2 + (y - m_2)^2 = r^2 \,. \tag{8.12.1}$$

8-12.I: linear algebraic fit

(8-12.a) ⚀ (15 min.) Plugging the point coordinates (x_i, y_i) into (8.12.1) we obtain an overdetermined linear system with three unknowns

$$m_1,\ m_2,\ c := r^2 - m_1^2 - m_2^2.$$

Specify the system matrix $\mathbf{A}$ and the right-hand side vector $\mathbf{b}$ of the corresponding *linear* least squares problem [Lecture → (3.1.3.7)].

SOLUTION for (8-12.a) → 8-12-1-0:cala1.pdf ▲

(8-12.b) ⚀ (30 min.) [depends on Sub-problem (8-12.a)]

Write a C++ function

```
Eigen::Vector3d circl_alg_fit(const Eigen::VectorXd& x,
const Eigen::VectorXd& y);
```

that receives point coordinates in the vectors `x` and `y` and solves the overdetermined system in least squares sense and returns a vector $[m_1, m_2, r]^\top \in \mathbb{R}^3$ of estimated circle parameters.

SOLUTION for (8-12.b) → 8-12-2-0:cala2.pdf ▲

8-12.II: geometric fit

The algebraic approach lacks an intuitive geometrical meaning: minimising the equation residual of (8.12.1) in least squares sense does not necessarily yield the best circle fit in aesthetic sense.

A more appealing approach consist in the minimization of the distances between the data points and the (unknown) circle

$$d_i = \left|\sqrt{(m_1 - x_i)^2 + (m_2 - y_i)^2} - r\right| \qquad i = 1, \ldots, N,$$

in the sense of least squares, i.e., determine m_1, m_2 and r such that the sum $\sum_{i=1}^n d_i^2$ is minimal. This is a *non-linear* least squares problem of the form

$$\mathbf{z}^* = \operatorname*{argmin}_{\mathbf{z}} \|\mathbf{F}(\mathbf{z})\|^2,$$

see [Lecture → Section 8.7].

(8-12.c) ⚀ (15 min.) Write down the concrete function $\mathbf{F} : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$ for this non-linear least squares problem.

You can use the auxiliary values

$$R_i = \sqrt{(x_i - m_1)^2 + (y_i - m_2)^2}\,, \quad i = 1, \ldots, N.$$

SOLUTION for (8-12.c) → 8-12-3-0:cage1.pdf ▲

(8-12.d) ⚁ (30 min.) [depends on Sub-problem (8-12.c)]

Define the functional

$$\Phi(\mathbf{z}) := \frac{1}{2}\|\mathbf{F}(\mathbf{z})\|^2.$$

Compute the Jacobian $D\mathbf{F}$ of $\mathbf{F}$, the gradient $\mathbf{grad}\,\Phi(\mathbf{z})$ of Φ and the Hessian $H\Phi(\mathbf{z})$.

SOLUTION for (8-12.d) → 8-12-4-0:cage2.pdf ▲

(8-12.e) ⚁ (30 min.) [depends on Sub-problem (8-12.d)]

Use a C++ code to find the circle that fit best the data given in the table above, according to the distances d_i. In order to do this, implement a C++ function

```
void circl_geo_fit_GN(const Eigen::VectorXd& x, const
   Eigen::VectorXd& y,
                      Eigen::Vector3d& z, std::vector<double>* err
                        = nullptr);
```

that uses the Gauss-Newton method introduced in [Lecture → Section 8.7.2] to minimize the functional Φ. The parameters `x`, `y` and the return values in `z` are the same as for `circl_alg_fit` from Sub-problem (8-12.a). The recorder object linked to the pointer `err` should be passed the current estimate in each step of the Gauss-Newton iteration.

SOLUTION for (8-12.e) → 8-12-5-0:cage3.pdf ▲

(8-12.f) ⚂ (30 min.) [depends on Sub-problem (8-12.d)]

Now solve the same problem using the **Newton method** for least squares equations as described in [Lecture → Section 8.7.1]. To that end, in the file `circle_approx.hpp`, implement a C++ function

```
void circl_geo_fit_N(const Eigen::VectorXd& x,
                     const Eigen::VectorXd& y,
                     Eigen::Vector3d& z, std::vector<double>* err
                       = nullptr)
```

The parameters and return values obey the same specification as those for `circl_geo_fit()` from Sub-problem (8-12.e).

SOLUTION for (8-12.f) → 8-12-6-0:cage4.pdf ▲

(8-12.g) ⚁ Compare the convergence of Gauss-Newton and Newton methods implemented in the previous sub-problems. You can use the parameters determined by the algebraic fit as initial guess.

Rely on the error in the parameters measured in the maximum norm.

To that end, in the file `circle_approx.hpp`, implement a C++ function

```
void compare_convergence(const Eigen::VectorXd& x,
                         const Eigen::VectorXd& y)
```

SOLUTION for (8-12.g) → 8-12-7-0:cage5.pdf ▲

8-12.III: Constrained non-linear fitting

As you will know, for $a \neq 0$ the solution set (for the unknown vector $\mathbf{x}$) of the quadratic equation

$$a\,\mathbf{x}^T\mathbf{x} + \mathbf{b}^T\mathbf{x} + c = 0, \quad \mathbf{b} \in \mathbb{R}^2,\ a, c \in \mathbb{R}\,. \tag{8.12.9}$$

(8-12.h) ⚀ (20 min.) Derive an expression in terms of $a, \mathbf{b}, c$ for the center $\mathbf{m}$ and the radius r of the circle defined by (8.12.9).

SOLUTION for (8-12.h) → 8-12-8-0:casvd1.pdf ▲

(8-12.i) ⚄ (60 min.) According to equation (8.12.9), the same circle can be defined by different parameter vectors $\mathbf{v} = (a, b_1, b_2, c)^T$ and $\mathbf{v}' = (a', b_1', b_2', c')^T$, when $\mathbf{v}' = \lambda\mathbf{v}$, for every $\lambda \in \mathbb{R}$, $\lambda \neq 0$. Thus, this equation, for different data values (x_i, y_i), can be solved in a least squares sense if supplemented by a **non-linear constraint**:

$$a\mathbf{x}_i^T\mathbf{x}_i + \mathbf{b}^T\mathbf{x}_i + c = 0 \quad i = 1, \ldots, N, \qquad \left\|(a, b_1, b_2, c)^T\right\|_2^2 = 1\,.$$

Write a C++ function

```
Eigen::Vector3d circl_svd_fit(const Eigen::VectorXd& x,
                              const Eigen::VectorXd& y)
```

that solves this constrained overdetermined linear system of equations in least squares sense.

To learn how to use SVD to solve this problem, study carefully the hyperplane fitting problem [Lecture → Ex. 3.4.4.5] and [Lecture → (3.4.4.1)].

SOLUTION for (8-12.i) → 8-12-9-0:casvd2.pdf ▲

8-12.IV: comparison of the results

(8-12.j) ⚂ Draw the data points and the fitted circles computed in the previous subtasks. Compare the centers and the radii.

To that end, in the file `circle_approx.hpp`, implement a C++ function

```
void plot(const Eigen::VectorXd& x, const Eigen::VectorXd& y,
const Eigen::Vector3d& z_alg, const Eigen::Vector3d& z_geo_GN,
const Eigen::Vector3d& z_geo_N, const Eigen::Vector3d& z_svd)
```

SOLUTION for (8-12.j) → 8-12-10-0:caco.pdf ▲

End Problem 8-12, 230 min.

Problem 8-13: Computing a Level Set

For a real-valued function $f : D \subset \mathbb{R}^d \to \mathbb{R}$ the level sets $\mathcal{L}_c := \{\boldsymbol{x} \in D : f(\boldsymbol{x}) \leq c\}$, $c \in \mathbb{R}$, are often used to characterize subsets of $\mathbb{R}^d$. This problem examines a numerical method for approximately computing the boundary of level sets of convex functions for $d = 2$.

This problem relies on methods for finding zeros of functions in 1D, see [Lecture → Section 8.4].

Throughout this problem let $f : \mathbb{R}^2 \to \mathbb{R}$ be a continuously differentiable *strictly convex* function which has a *global minimum* in $\boldsymbol{x} = \mathbf{0}$ and satisfies the growth condition $|f(\boldsymbol{x})| \to \infty$ uniformly for $\|\boldsymbol{x}\| \to \infty$

Then the level sets

$$\mathcal{L}_c := \{\boldsymbol{x} \in \mathbb{R}^2 : f(\boldsymbol{x}) \leq c\}\,, \quad c > 0\,, \tag{8.13.1}$$

will be closed and convex subsets of $\mathbb{R}^2$, and every ray

$$R_{\mathbf{d}} := \{\xi \mathbf{d} : \xi \geq 0\}\,, \quad \mathbf{d} \in \mathbb{R}^2 \setminus \{0\}\,, \tag{8.13.2}$$

will intersect the boundary of $\mathcal{L}_c$,

$$\partial\mathcal{L}_c := \{\boldsymbol{x} \in \mathbb{R}^2 : f(\boldsymbol{x}) = c\}\,, \tag{8.13.3}$$

in exactly one point.

(8-13.a) ⚀ (15 min.) Based on point evaluations of f and $\mathbf{grad}\, f$ state the Newton iteration for the *scalar* non-linear equation that has to be solved in order to find the intersection of the ray $R_{\mathbf{d}}$ and $\partial\mathcal{L}_c$. The vector $\mathbf{d} \in \mathbb{R}^2 \setminus \{0\}$ and the number $c > 0$ should be regarded as parameters.

SOLUTION for (8-13.a) → 8-13-1-0:s1.pdf ▲

(8-13.b) ⚁ Now, write down the recursion for the secant method [Lecture → § 8.4.2.28] for solving the *scalar* non-linear equation tackled in Sub-problem (8-13.a). Your formula should involve only f-evaluations and $\mathbf{d} \in \mathbb{R}^2 \setminus \{\mathbf{0}\}$ and $c > 0$ as parameters.

SOLUTION for (8-13.b) → 8-13-2-0:s3.pdf ▲

(8-13.c) ⚄ (15 min.) [depends on Sub-problem (8-13.b)]

Implement an efficient C++ function

```
template <typename Functor>
  Eigen::VectorXd pointLevelSet(
   Functor &&f, const Eigen::Vector2d &d, double c,
   const Eigen::Vector2d &x0,
   double rtol = 1E-10, double atol = 1E-16);
```

that uses the secant method to find the unique point in the intersection of $R_{\mathbf{d}}$ and $\partial\mathcal{L}_c$, $c > 0$. The function f is given in procedural form by a functor object `f`. The coordinates of the intersection point are returned.

The vector argument `x0` passes (a guesses for) the coordinates of a point $\mathbf{x}_0 \in \partial\mathcal{L}_c$. This information may be used to obtain initial guesses for the secant iteration, which should start from the points $\frac{\|\mathbf{x}_0\|_2}{\|\mathbf{d}\|_2}\mathbf{d}$ and $1.1 \cdot \frac{\|\mathbf{x}_0\|_2}{\|\mathbf{d}\|_2}\mathbf{d}$.

Employ a correction-based termination criterion with relative tolerance and absolute tolerance supplied by the arguments `rtol` and `atol`.

SOLUTION for (8-13.c) → 8-13-3-0:s4.pdf ▲

(8-13.d) ⚄ (20 min.) [depends on Sub-problem (8-13.c)]

Based on `pointLevelSet()` write an efficient C++ function

```
template <typename Functor>
  double areaLevelSet(Functor &&f, unsigned int n, double c);
```

that returns the "Archimedean approximation" of the area of $\mathcal{L}_c$, $c > 0$, for the function f passed in `f`.

That Archimedean approximation replaces the set $\mathcal{L}_c$ with a *convex polygonal domain* through the n points ($n > 2$)

$$\mathbf{p}_j \in \mathbb{R}^2\text{:} \quad \{\mathbf{p}_j\} = R_{\mathbf{d}_j} \cap \partial\mathcal{L}_c \,, \quad \mathbf{d}_j = \begin{bmatrix} \cos(2\pi j/n) \\ \sin(2\pi j/n) \end{bmatrix} \,, \quad j = 0, \ldots, n-1 \,.$$

HIDDEN HINT 1 for (8-13.d) → `8-13-4-0:hcp.pdf`

HIDDEN HINT 2 for (8-13.d) → `8-13-4-1:hcp2.pdf`

SOLUTION for (8-13.d) → `8-13-4-2:s4a.pdf` ▲

(8-13.e) ⚄ (10 min.) We consider

$$f(\mathbf{x}) = x_1^2 + 2x_2^4 \,, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2 \,.$$

n	error in area
5	0.7322814883
10	0.2020820848
15	0.0943700007
20	0.0539834484
25	0.0346797600
30	0.0241767180
40	0.0136526713
50	0.0087539878
80	0.0034265044

The table lists the error in the approximate area of $\mathcal{L}_1$ returned by `areaLevelSet()` as $n \to \infty$, using relative tolerance `rtol` $= 10^{-10}$ and absolute tolerance `atol` $= 10^{-16}$.

Describe qualitatively and quantitatively the empiric convergence of the approximation of the area in terms of $\frac{1}{n} \to 0$.

SOLUTION for (8-13.e) → `8-13-5-0:s5.pdf` ▲

End Problem 8-13 , 60 min.

Problem 8-14: Symmetric Rank-1 Best Approximation of a Matrix

The approximation of matrices by data-sparse matrices of low rank has become a fundamental technique in modern computational mathematics and data science, leading to very efficient methods for the compression of non-local operators and multi-dimensional data.

This problems is based on [Lecture → Section 8.7], in particular [Lecture → Section 8.7.2], and also draws on [Lecture → Section 3.4.4.2].

Given a square matrix $\mathbf{M} \in \mathbb{R}^{n,n}$, $\mathbf{M} \neq \mathbf{O}$, we want to find a symmetric rank-1 matrix closest in Frobenius norm $\|\cdot\|_F$ [Lecture → Def. 3.4.4.17]:

$$\mathbf{x} \in \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^n} \left\| \mathbf{M} - \mathbf{z}\mathbf{z}^\top \right\|_F = \operatorname*{argmin}_{\mathbf{z}\in\mathbb{R}^n} \|\boldsymbol{\Phi}(\mathbf{z})\|_2 \,, \tag{8.14.1}$$

$$\text{with} \quad \boldsymbol{\Phi} : \mathbb{R}^n \to \mathbb{R}^{n^2} \,, \quad \boldsymbol{\Phi}(\mathbf{x}) := \mathrm{vec}(\mathbf{M}) - \mathbf{x} \otimes \mathbf{x} \,, \quad \mathbf{x} \in \mathbb{R}^n \,, \tag{8.14.2}$$

where $\|\cdot\|_2$ denotes the Euclidean vector norm, vec the vectorization of matrices

$$\mathrm{vec} : \mathbb{R}^{n,m} \to \mathbb{R}^{n\cdot m} \quad , \quad \mathrm{vec}(\mathbf{A}) := \begin{bmatrix} (\mathbf{A})_{:,1} \\ (\mathbf{A})_{:,2} \\ \vdots \\ (\mathbf{A})_{:,m} \end{bmatrix} \in \mathbb{R}^{n\cdot m} \,, \qquad \text{[Lecture → (1.2.3.5)]}$$

as introduced in [Lecture → Rem. 1.2.3.4], and $\otimes$ the Kronecker product of two matrices/vectors from [Lecture → Def. 1.4.3.7].

(8-14.a) ⚄ (30 min.)

In the file `symrank1.hpp` implement an EIGEN-based C++ function

```
Eigen::VectorXd symRankOneBestApproxSym(const Eigen::MatrixXd &M);
```

that solves (8.14.1) for a *symmetric positive semi-definite* [Lecture → Def. 1.1.2.6] matrix $\mathbf{M}$, that is, provided that $\mathbf{M}^\top = \mathbf{M}$ and $\mathbf{x}^\top\mathbf{M}\mathbf{x} \geq 0$ for all $\mathbf{x} \in \mathbb{R}^n$.

HIDDEN HINT 1 for (8-14.a) → `8-14-1-0:s1h1.pdf`

SOLUTION for (8-14.a) → `8-14-1-1:s1.pdf` ▲

For general $\mathbf{M}$ we intend to solve (8.14.1) by means of the **Gauss-Newton method**, that is, by means of the iteration [Lecture → (8.7.2.1)]

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + \operatorname*{argmin}_{\mathbf{h}\in\mathbb{R}^n} \left\| \boldsymbol{\Phi}(\mathbf{x}^{(k)}) + \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}^{(k)})\mathbf{h} \right\|_2 \,, \tag{8.14.4}$$

with $\boldsymbol{\Phi}$ from (8.14.2).

(8-14.b) ⚅ (30 min.)

Give an expression for the linear mapping $\mathbf{h} \in \mathbb{R}^n \mapsto \mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})\mathbf{h}$, $\mathbf{x} \in \mathbb{R}^n$ fixed.

HIDDEN HINT 1 for (8-14.b) → `8-14-2-0:kpbl.pdf`

SOLUTION for (8-14.b) → `8-14-2-1:s2a.pdf` ▲

(8-14.c) ⚅ (30 min.) [depends on Sub-problem (8-14.b)]

Derive a formula for the Jacobian $\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x}) \in \mathbb{R}^{n^2,n}$ of $\boldsymbol{\Phi}$ as defined in (8.14.2).

Write down $\mathrm{D}\,\boldsymbol{\Phi}(\mathbf{x})$ explicitly for $n = 3$, abbreviating $x_i = (\mathbf{x})_i$, $i = 1,2,3$.

HIDDEN HINT 1 for (8-14.c) → 8-14-3-0:handy2.pdf

SOLUTION for (8-14.c) → 8-14-3-1:s2b.pdf ▲

(8-14.d) ⚄ (30 min.) [depends on Sub-problem (8-14.c)]

The minimization problem in (8.14.4) involves a linear least-squares problem of the form

$$\Delta\mathbf{x} := \operatorname*{argmin}_{\mathbf{h}\in\mathbb{R}^n} \|\mathbf{A}\mathbf{h} - \mathbf{b}\|_2 \,, \tag{8.14.11}$$

for some matrix $\mathbf{A} \in \mathbb{R}^{n^2,n}$ and vector $\mathbf{b} \in \mathbb{R}^{n^2}$.

For general n compute an explicit formula for the system matrix $\mathbf{T} \in \mathbb{R}^{n,n}$ of the **normal equations** for the linear least squares problem (8.14.11).

HIDDEN HINT 1 for (8-14.d) → 8-14-4-0:3hI.pdf

SOLUTION for (8-14.d) → 8-14-4-1:s4.pdf ▲

(8-14.e) ⚂ (20 min.) [depends on Sub-problem (8-14.d)]

Give a simple explicit formula for the inverse $\mathbf{T}^{-1}$ of the system matrix of the normal equations.

HIDDEN HINT 1 for (8-14.e) → 8-14-5-0:smw.pdf

HIDDEN HINT 2 for (8-14.e) → 8-14-5-1:fails.pdf

SOLUTION for (8-14.e) → 8-14-5-2:s5.pdf ▲

(8-14.f) ⚄ (30 min.)

Code an *efficient* C++ function (in the file `symrank1.hpp`)

```
Eigen::VectorXd computeKronProdVecMult(const Eigen::VectorXd &v,
   const Eigen::VectorXd &b);
```

that evaluates the expression

$$(\mathbf{v}^\top \otimes \mathbf{I}_n + \mathbf{I}_n \otimes \mathbf{v}^\top)\mathbf{b} \quad \text{for} \quad \mathbf{v} \in \mathbb{R}^n \,, \quad \mathbf{b} \in \mathbb{R}^{n^2} \,, \tag{8.14.14}$$

for any $n \in \mathbb{N}$.

HIDDEN HINT 1 for (8-14.f) → 8-14-6-0:hrs.pdf

SOLUTION for (8-14.f) → 8-14-6-1:sx.pdf ▲

(8-14.g) ⚄ (45 min.) [depends on Sub-problem (8-14.f) and Sub-problem (8-14.e)]

Based on the normal equation method for the occurring linear least squares problems, in the file `symrank1.hpp` implement an efficient C++ function

```
Eigen::VectorXd symmRankOneApprox(
   const Eigen::MatrixXd &M
   double rtol = 1E-6, atol = 1.0E-8);
```

that implements the **Gauss-Newton iteration** (8.14.4) for solving (8.14.1) using a correction-based termination criterion with relative tolerance `rtol` and absolute tolerance `atol`. The parameter `M` passes the matrix $\mathbf{M} \in \mathbb{R}^{n,n}$.

As initial guess for $\mathbf{z}$ we use that column of the symmetric part $\frac{1}{2}(\mathbf{M}^\top + \mathbf{M})$ of $\mathbf{M}$ with the largest Euclidean norm.

HIDDEN HINT 1 for (8-14.g) → 8-14-7-0:mcgn.pdf

HIDDEN HINT 2 for (8-14.g) → 8-14-7-1:mcgn2.pdf

HIDDEN HINT 3 for (8-14.g) → 8-14-7-2:fail2.pdf

SOLUTION for (8-14.g) → 8-14-7-3:s6.pdf ▲

End Problem 8-14 , 215 min.

Problem 8-15: Non-linear Least-Squares Location Estimation

Least-squares techniques are the main tool for estimating unknown parameters from measurements. In this problem we apply them to an overdetermined non-linear system of equations arising from the problem of estimating the location of an acoustic source. As solution method we use the iterative **Gauss-Newton method**.

This problem requires knowledge about the Gauss-Newton method from [Lecture → Section 8.7.2] and skills in multi-dimensional differentiation. Implementation in C++ based on EIGEN is part of the problem.

A source of acoustic waves at an unknown location $\mathbf{p}^* \in \mathbb{R}^3$ in a room starts emitting a loud and short noise at an unknown time t^*. At times $t_j > t^*$, $j = 1,\ldots,n$, this signal is detected by each of $n > 4$ microphones placed in known locations $\mathbf{q}^j \in \mathbb{R}^3$, $j = 1,\ldots,n$, $\mathbf{q}^i \neq \mathbf{q}^j$ for $i \neq j$. Writing c (physical units $[c] = 1\frac{\mathrm{m}}{\mathrm{s}}$) for the speed of sound in air, it is straightforward that the arrival times t_j of the signal satisfy

$$\left\| \mathbf{q}^j - \mathbf{p}^* \right\|_2 = c(t_j - t^*) \,, \quad j = 1,\ldots,n \,. \tag{8.15.1}$$

From the measured arrival times t_j we want to reconstruct the location $\mathbf{p}^*$ of the noise source and the time t^*, when it went off.

(8-15.a) ⚀ (15 min.)

Rewrite (8.15.1) as an overdetermined non-linear system of equations in the standard form $F(\mathbf{x}) = \mathbf{0}$. Specify $\mathbf{x}$ and F:

$$F : \begin{cases} \mathbb{R}^{\square} & \to \quad \mathbb{R}^{\square} \\ \mathbf{x} := \square & \mapsto \quad F(\mathbf{x}) := \square \end{cases}$$

SOLUTION for (8-15.a) → 8-15-1-0:locs1.pdf ▲

(8-15.b) ⚁ (20 min.) [depends on Sub-problem (8-15.a)]

Compute the Jacobian $\mathrm{D}\,F(\mathbf{x})$ for the function F found in Sub-problem (8-15.a). Determine its maximal domain of definition, that is, the set of $\mathbf{x}$'s for which F is differentiable.

$$\mathrm{D}\,F(\mathbf{x}) = \mathrm{D}\,F\left(\left[\square\right]\right) = \left[\square\right]$$

for $\mathbf{x} \in$.

SOLUTION for (8-15.b) → 8-15-2-0:locs2.pdf ▲

(8-15.c) ⚂ (45 min.) [depends on Sub-problem (8-15.a), Sub-problem (8-15.b)]

Assume a choice of physical units such that $c = 1$. In the file `locationestimation.hpp` write a C++ function

```
std::pair<Eigen::Vector3d, double>
  source_estimation(
    const Eigen::Matrix<double, 3, Eigen::Dynamic> &Q,
    const Eigen::VectorXd &ta);
```

that solves the non-linear system of equations (8.15.1) in least-squares sense using the Gauss-Newton method introduced in [Lecture → Section 8.7.2]. The $3 \times n$-matrix `Q` passes the vectors $\mathbf{q}^j$, $j = 1, \ldots, n$, in its columns, whereas the vector `ta` contains the arrival times t_j. The function should return an estimate for the pair $(\mathbf{p}^*, t^*)$.

As initial guess use

$$\mathbf{p}^{(0)} := \frac{1}{n}\sum_{j=1}^{n} \mathbf{q}^j \quad , \quad t^{(0)} := t_k - \left\| \mathbf{q}^k - \mathbf{p}^{(0)} \right\|_2 , \quad k \text{ such that } t_k = \min\{t_j,\, j = 1, \ldots, n\} .$$

As in [Lecture → Code 8.7.2.2] rely on a correction-based stopping rule in the spirit of [Lecture → (8.5.3.2)] with relative tolerance $\tau_{\text{rel}} = 10^{-6}$ and absolute tolerance $\tau_{\text{abs}} = 10^{-8}$.

SOLUTION for (8-15.c) → 8-15-3-0:locsol3.pdf ▲

(8-15.d) ⚂ (20 min.) The following table displays the history of the Gauss-Newton iteration as used in `source_location()` for $t^* = 0$,

$$\left[\mathbf{q}^1, \ldots, \mathbf{q}^{11}\right] = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0.5 & 0.0 & 1.0 & 0.5 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0.0 & 0.5 & 0.5 & 1.0 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0.5 & 0.5 & 0.5 & 0.5 \end{bmatrix}$$

and the arrival times t_j fitting the source position $\mathbf{p}^* = [0.1, 0.1, 0.9]$.

k	$\left\|\mathbf{x}^{(k)} - \mathbf{x}^*\right\|_2$	$\|\mathbf{s}\|_2$	$\left\|F(\mathbf{x}^{(k)})\right\|$
1	1.021791e+00	1.135326e+00	3.442935e+00
2	2.588111e-01	2.527947e-01	6.041095e-01
3	7.550346e-03	7.504712e-03	1.559289e-02
4	6.192137e-05	6.192055e-05	9.969019e-05
5	9.310988e-10	9.310988e-10	1.862087e-09
6	1.240973e-16	7.295652e-17	3.159274e-16

Characterize the observed convergence and explain your answer.

SOLUTION for (8-15.d) → 8-15-4-0:locsol4.pdf ▲

End Problem 8-15 , 100 min.

Problem 8-16: Periodic Collocation for a Non-Linear Differential Equation

This problem examines the so-called collocation method for the approximate solution of differential equations. Since we look for periodic solutions, it is natural to use linear combinations of trigonometric polynomials as trial space. Equations are obtained by requiring that the equations remains satisfied in finitely many points.

This problem requires the techniques from [Lecture → Section 4.2] and [Lecture → Section 8.5] and involves substantial implementation in C++ using EIGEN.

We look for 1-periodic solutions $u : [0,1] \to \mathbb{C}$, $u = u(t)$, of the differential equation

$$-\frac{d^2u}{dt^2}(t) + u(t)^3 = \sin(\pi t)\,, \quad 0 \le t < 1\,.$$

To approximate them we replace u with

$$u_N(t) := \sum_{j=0}^{N} x_j \cos(2\pi j t) \quad \text{for some} \quad N \in \mathbb{N}\,, \tag{8.16.1}$$

and try to determine the unknown coefficients $x_j \in \mathbb{R}$, $j = 0,\dots,N$, by solving the **collocation equations**

$$-\frac{d^2u_N}{dt^2}(t_k) + u_N(t_k)^3 = \sin(\pi t_k)\,, \quad k = 0,\dots,N\,, \quad t_k := \frac{k}{N+1}\,. \tag{8.16.2}$$

(8-16.a) ⚂ (30 min.) For plotting $t \mapsto u_N(t)$ with u_N as in (8.16.1) we need an *efficient* C++ function

```
Eigen::VectorXd eval_uN(const Eigen::VectorXd &x, unsigned int M);
```

which, given the coefficients $x_j \in \mathbb{R}$, $j = 0,\dots,N$, in (8.16.1) through the vector `x` and a number $M > N$ in `M`, returns the vector

$$\left[u_N(\frac{k}{M})\right]_{k=0}^{M-1} \in \mathbb{R}^M\,.$$

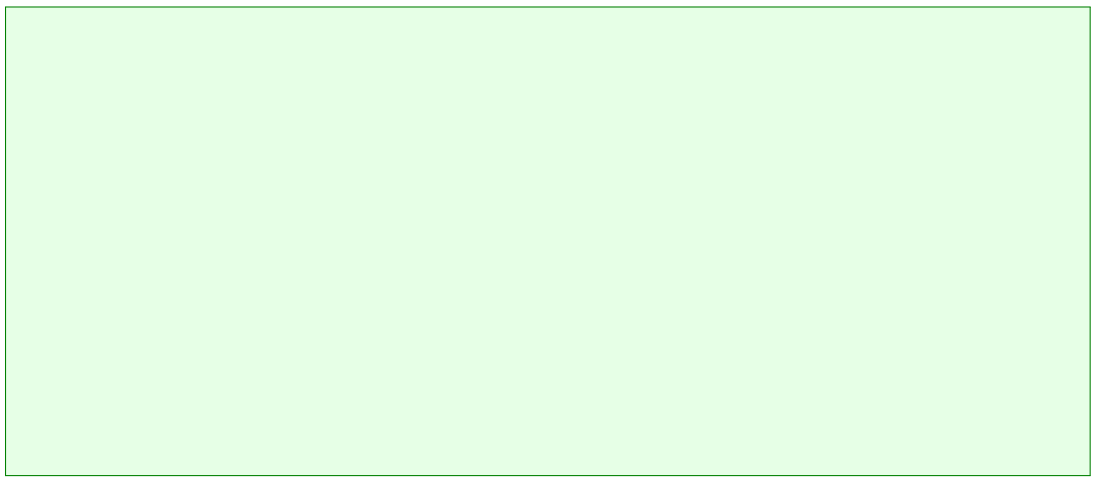
"Efficient" stipulates that the computational cost of `eval_uN()` must scale like $O(M \log M)$ for $M \to \infty$.

HIDDEN HINT 1 for (8-16.a) → 8-16-1-0:pcollh1.pdf

SOLUTION for (8-16.a) → 8-16-1-1:pces1.pdf ▲

(8-16.b) ⚂ (15 min.) Write the collocation equations (8.16.2) in the standard form $F(\mathbf{x}) = \mathbf{0}$ of a non-linear system of equations for an *explicitly given* $F : \mathbb{R}^n \to \mathbb{R}^n$ and suitable $\mathbf{x}$ and $n \in \mathbb{N}$.

$$n = \boxed{}\,, \quad \mathbf{x} = \begin{bmatrix} \boxed{} \end{bmatrix}\,,$$

$F(\mathbf{x}) =$  .

HIDDEN HINT 1 for (8-16.b) → 8-16-2-0:ceh2.pdf

SOLUTION for (8-16.b) → 8-16-2-1:pcolls2.pdf ▲

(8-16.c) ⚄ (20 min.) [depends on Sub-problem (8-16.b)]

Relying on the function `eval_uN()` from Sub-problem (8-16.a) implement another C++ function in `periodiccollocation.hpp`,

```
Eigen::VectorXd eval_F(const Eigen::VectorXd &x);
```

that returns the vector $F(\mathbf{x})$ when given the argument $\mathbf{x}$ in `x`.

SOLUTION for (8-16.c) → 8-16-3-0:pcolls3.pdf ▲

(8-16.d) ⚄ (30 min.) [depends on Sub-problem (8-16.b)]

We want to try Newton's method to solve $F(\mathbf{x}) = \mathbf{0}$ with F from Sub-problem (8-16.b). To that end we need a function

```
Eigen::MatrixXd eval_DF(const Eigen::VectorXd &x);
```

that returns the Jacobian $\mathrm{D}F(\mathbf{x})$, when `x` contains the argument $\mathbf{x}$. Write such a function in the file `periodiccollocation.hpp`

HIDDEN HINT 1 for (8-16.d) → 8-16-4-0:pch4.pdf

SOLUTION for (8-16.d) → 8-16-4-1:pces4.pdf ▲

End Problem 8-16, 95 min.

Problem 8-17: Aspects of Iterative Methods for Non-Linear Equations

This problem revisits central concepts and algorithms connected with iterative methods for non-linear (systems of) equations.

This problem assumes familiarity with [Lecture → Section 8.2.2], [Lecture → Section 8.4.2.1], and [Lecture → Section 8.5.3].

(8-17.a) ⚀ (5 min.) Fill in the blanks in the following definition:

Definition [Lecture → Def. 8.2.2.10]. Order of convergence

A *convergent* sequence $\mathbf{x}^{(k)}$, $k = 0,1,2,\ldots$, in $\mathbb{R}^n$ with limit $\mathbf{x}^* \in \mathbb{R}^n$ converges with **order** p, $p \geq 1$, if

$$\exists C > \boxed{} : \quad \left\| \mathbf{x}^{(k+1)} - \mathbf{x}^* \right\| \leq C \boxed{\phantom{\left\| \mathbf{x}^{(k)} - \mathbf{x}^* \right\|^p}} \quad \forall k \in \mathbb{N}_0 \,,$$

and, in addition, $C <$ $\boxed{}$ in the case $p =$ $\boxed{}$.

SOLUTION for (8-17.a) → 8-17-1-0:sa2.pdf ▲

(8-17.b) ⚀ (10 min.) The templated C++ function

```
template <typename FuncType, typename JacType, typename VecType>
void newton(const FuncType &F, const JacType &DF, VecType &x,
            double rtol, double atol);
```

implements a generic Newton iteration for solving $F(\mathbf{x}) = \mathbf{0}$ with correction-based termination relying on the simplified Newton correction:

STOP, as soon as $\left\| \Delta\bar{\mathbf{x}}^{(k)} \right\| \leq \texttt{rtol} \left\| \mathbf{x}^{(k)} \right\|$ **or** $\left\| \Delta\bar{\mathbf{x}}^{(k)} \right\| \leq \texttt{atol}$,

with simplified Newton correction $\Delta\bar{\mathbf{x}}^{(k)} := \mathrm{D}F(\mathbf{x}^{(k-1)})^{-1} F(\mathbf{x}^{(k)})$.

[Lecture → (8.5.3.5)]

The functor arguments `F` and `DF` pass the function $F : \mathbb{R}^n \to \mathbb{R}^n$ and its derivative (Jacobian) $\mathrm{D}F : \mathbb{R}^n \to \mathbb{R}^{n,n}$, whereas `x` supplies the initial guess and also returns the final results. The user has to specify the absolute tolerance `atol` and relative tolerance `rtol` to control termination.

The type **JacType** must be a *functor type* whose return type has capabilities like a matrix type of EIGEN, whereas **VecType** is a matching vector type, for instance a vector type of EIGEN.

Fill in the blanks in Code 8.17.1 so that the resulting code is a valid C++ implementation of `newton()`

C++ code 8.17.1: Generic driver function for Newton's method

```
template <typename FuncType, typename JacType, typename VecType>
void newton(const FuncType &F, const JacType &DF, VecType &x,
            double rtol, double atol) {
  double sn;
  do {
    auto jacfac = ____________.lu();
    x -= jacfac.solve(_____);

    sn = jacfac.solve(_____).norm();
  }
  while ((sn > ____________) && (sn > ____________));
}
```

SOLUTION for (8-17.b) → 8-17-2-0:imsb.pdf ▲

(8-17.c) ⚄ (10 min.) Let f denote the function

$$f : \begin{cases} [0,1] & \to & [0,1] \\ x & \mapsto & xe^{x-1} \end{cases} .$$

Explicitly state the Newton iteration converging to $g(y)$ for a given $y \in [0,1]$ and sufficiently good initial guess $x^{(0)}$, where $g := f^{-1}$ is the *inverse function* of f.

$$x^{(k+1)} = x^{(k)} - \boxed{} , \quad k \in \mathbb{N}_0 .$$

SOLUTION for (8-17.c) → 8-17-3-0:imsc.pdf ▲

End Problem 8-17 , 25 min.

Problem 8-18: Computing Eigenvalues with Newton's Method

This problem examines an "exotic" method for the solution of algebraic eigenvalue problems in the real numbers: Given $\mathbf{A} \in \mathbb{R}^{n,n}$ seek $\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ and $\lambda \in \mathbb{R}$ such that $\mathbf{A}\mathbf{X} = \lambda\mathbf{x}$. This problem for generic dense matrices is usually solved by special iterative methods implemented in opaque library functions. However, it can also be converted into a non-linear system of equations and solved with Newton's method.

This problem is related to [Lecture → Section 8.5] and [Lecture → Section 3.4.2]. The general topic is treated in [Lecture → Chapter 9].

As we know from linear algebra and also [Lecture → Def. 9.1.0.1], given a matrix $\mathbf{A} \in \mathbb{R}^{n,n}$ a number $\lambda \in \mathbb{C}$ is called an **eigenvalue** of $\mathbf{A}$, if there exists an $\mathbf{x} \in \mathbb{R}^n \setminus \{\mathbf{0}\}$ (**eigenvector**) such that $\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$.

Every matrix has at least one eigenvalue, and for special matrices we know that their eigenvalues will even be real.

Theorem 8.18.1. Real eigenvalues of symmetric matrices [NS02, Satz 7.7]

If $\mathbf{A} \in \mathbb{R}^{n,n}$ is symmetric, $\mathbf{A} = \mathbf{A}^\top$ all eigenvalues of $\mathbf{A}$ will be real.

The eigenvalue problem can be converted into a non-linear system of equations in the following way:

Given a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n,n}$, $\mathbf{A} = \mathbf{A}^\top$, solving

$$\begin{bmatrix}\mathbf{x}\\ \lambda\end{bmatrix} \in \mathbb{R}^{n+1}: \quad F\left(\begin{bmatrix}\mathbf{x}\\ \lambda\end{bmatrix}\right) = \mathbf{0} \quad \text{with} \quad F : \begin{cases} \mathbb{R}^{n+1} & \to & \mathbb{R}^{n+1} \\ \begin{bmatrix}\mathbf{v}\\ \xi\end{bmatrix} & \mapsto & \begin{bmatrix}\mathbf{A}\mathbf{v} - \xi\mathbf{v} \\ 1 - \frac{1}{2}\|\mathbf{v}\|_2^2\end{bmatrix} \end{cases} \tag{8.18.2}$$

amounts to finding an eigenvalue $\lambda \in \mathbb{R}$ and associated eigenvector $\mathbf{x} \in \mathbb{R}^n$.

Thus, a numerical method for computing an eigenvalue-eigenvector pair of $\mathbf{A}$ is the application of Newton's method to find a zero of the function F from (8.18.2).

(8-18.a) ⚀ (10 min.) Compute the Jacobi matrix $\mathrm{D}F\left(\left[\begin{smallmatrix}\mathbf{v}\\ \xi\end{smallmatrix}\right]\right)$ of F at $\left[\begin{smallmatrix}\mathbf{v}\\ \xi\end{smallmatrix}\right] \in \mathbb{R}^{n+1}$.

$$\mathrm{D}F\left(\begin{bmatrix}\mathbf{x}\\ \lambda\end{bmatrix}\right) = \qquad , \quad \begin{bmatrix}\mathbf{x}\\ \lambda\end{bmatrix} \in \mathbb{R}^{n+1} .$$

SOLUTION for (8-18.a) → 8-18-1-0:nlevpas.pdf ▲

(8-18.b) ⚀ (5 min.) [depends on Sub-problem (8-18.a)]

The Newton iteration applied to $F(\mathbf{z}) = \mathbf{0}$, $\mathbf{z} := \left[\begin{smallmatrix}\mathbf{x}\\ \lambda\end{smallmatrix}\right]$, produces a sequence of iterates $\left(\mathbf{z}^{(k)}\right)_{k\in\mathbb{N}_0}$ starting from $\mathbf{z}^{(0)} = \left[\begin{smallmatrix}\mathbf{x}^{(0)}\\ \lambda^{(0)}\end{smallmatrix}\right] \in \mathbb{R}^{n+1}$. State that iteration in detail:

$$\mathbf{z}^{(k+1)} = \boldsymbol{\Phi}_F(\mathbf{z}^{(k)}) , \quad \boldsymbol{\Phi}_F(\mathbf{z}) =$$

for $\mathbf{z} \in \mathbb{R}^{n+1}$.

SOLUTION for (8-18.b) → 8-18-2-0:.pdf ▲

(8-18.c) ⚄ (45 min.) [depends on Sub-problem (8-18.b)]

In the file `eigenvaluesbynewton.hpp` complete the code of the C++ function

```
std::pair<double, Eigen::VectorXd>
  eignewton(const Eigen::MatrixXd& A, double rtol = 1.0E-6,
            double atol = 1.0E-8, unsigned int maxit = 20);
```

that relies on the Newton iteration from Sub-problem (8-18.b) to compute and return an eigenvalue-eigenvector pair of the (square symmetric) matrix passed through `A`.

As initial value use

- for $\lambda^{(0)}$ the value of the largest diagonal entry of the matrix $\mathbf{A}$,
- for $\mathbf{x}^{(0)}$ the vector $\sqrt{2}\mathbf{e}_i$, where $\mathbf{e}_i$ is the i-th Cartesian coordinate vector and i is the index of the largest diagonal entry of $\mathbf{A}$:

$$i := \operatorname*{argmax}_{j=1,\ldots,n} \left| (\mathbf{A})_{j,j} \right| .$$

Implement a correction based termination criterion based on the relative and absolute tolerances passed in `rtol` and `atol`. Also stop the iteration once `maxit` steps have been carried out.

SOLUTION for (8-18.c) → `8-18-3-0:ebnsc.pdf` ▲

References

[NS02] K. Nipp and D. Stoffer. *Lineare Algebra*. 5th ed. Zürich: vdf Hochschulverlag, 2002 (cit. on p. 299).

End Problem 8-18 , 60 min.

Problem 8-19: Energy Minimization for a Discrete System

In physics so-called discrete models with a finite number of state variables are popular, because they often capture key properties of more complicated continuous-state models. A typical task is to determine the so-called ground state of a discrete model characterized as a (local) minimum of some energy functional on state space.

This problem assumes familiarity with [Lecture → Section 8.5].

The state of a discrete physical system is described by $n \in \mathbb{N}$ variables $u_1, \ldots, u_n$, which can be collected into a vector $\mathbf{u} = [u_1, \ldots, u_n]^\top \in \mathbb{R}^n$. For instance, the u_i may be related to a physical quantity at different points in space. A physicist tells us that the free energy of the system is given by the functional

$$J(\mathbf{u}) := \frac{1}{2h}\Big(\sum_{j=1}^{n}\sum_{k=1}^{n} e^{h|j-k|}(u_j+u_k)^2\Big) + h\sum_{j=1}^{n} e^{u_j}\,, \quad \mathbf{u} \in \mathbb{R}^n\,. \tag{8.19.1}$$

where $h := \frac{1}{n+1}$. What he is interested in are (local) minima of $J : \mathbb{R}^n \to \mathbb{R}$, that is, he wants us to find solutions of the non-linear system of equations $\mathbf{grad}\, J(u) = \mathbf{0}$.

(8-19.a) ⚀ (15 min.) For any $\ell \in \{1, \ldots, n\}$ and $\mathbf{u} \in \mathbb{R}^n$ compute the partial derivative of the energy functional

$$\frac{\partial J}{\partial u_\ell}(\mathbf{u}) = \qquad .$$

HIDDEN HINT 1 for (8-19.a) → 8-19-1-0:emah1.pdf

SOLUTION for (8-19.a) → 8-19-1-1:.pdf ▲

(8-19.b) ⚀ (15 min.) [depends on Sub-problem (8-19.a)]

The non-linear system of equations $\mathbf{grad}\, J(\mathbf{u}) = \mathbf{0}$ can be written in the form

$$\mathbf{A}\mathbf{u} + \mathbf{b}(\mathbf{u}) = \mathbf{0} \quad \text{with} \quad \mathbf{A} = \mathbf{A}^\top \in \mathbb{R}^{n,n} \qquad \mathbf{b} : \mathbb{R}^n \to \mathbb{R}^n\,. \tag{8.19.10}$$

Give explicit formulas for the symmetric matrix $\mathbf{A}$ and the vector-valued function $\mathbf{u} \mapsto \mathbf{b}(\mathbf{u})$.

$$\mathbf{b}(\mathbf{u}) = \qquad , \mathbf{u} = [u_1, \ldots, u_n]^\top \in \mathbb{R}^n,$$

$$(\mathbf{A})_{\ell,j} = \qquad , \quad \ell, j \in \{1, \ldots, n\}\,.$$

SOLUTION for (8-19.b) → 8-19-2-0:dukJ.pdf ▲

A popular approach to the iterative solution of the non-linear system of equations $\mathbf{A}\mathbf{u} + \mathbf{b}(\mathbf{u}) = \mathbf{0}$ from (8.19.10) is the **fixed-point iteration** with the recursive definition

$$\mathbf{u}^{(k+1)} = -\mathbf{A}^{-1}\mathbf{b}(\mathbf{u}^{(k)}) , \quad k \in \mathbb{N}_0 . \tag{8.19.13}$$

(8-19.c) ⚂ (30 min.) [depends on Sub-problem (8-19.b)]

Implement an *efficient* C++ function

```
Eigen::VectorXd groundState_fpit(unsigned int n,
  double rtol = 1.0E-8, double atol = 1.0E-10,
  unsigned int maxit = 20);
```

that employs the fixed-point iteration (8.19.13) to solve $\mathbf{grad}\, J(\mathbf{u}) = \mathbf{0}$. The argument `n` passes n. As initial guess use $\mathbf{u}^{(0)} := \mathbf{0}$.

Employ a correction based termination criterion with relative tolerance `rtol` and absolute tolerance `atol`. Stop after at most `maxit` iteration steps.

SOLUTION for (8-19.c) → 8-19-3-0:enmsf.pdf ▲

(8-19.d) ⚄ (15 min.) [depends on Sub-problem (8-19.c)]

We make the following assumption

Assumption 8.19.15. Linear convergence of fixed-point iteration

The fixed point iteration (8.19.13) *converges linearly* [Lecture → Def. 8.2.2.1] with a rate $L \leq 1 - \frac{c}{n} < 1$ with some constant $c > 0$.

For very large $n \gg 1$ what is the worst-case n-asymptotics of the *additional* computational effort your implementation of `groundState_fpit()` requires in order to reduce the iteration error by a factor of $\rho > 0$?

$$\text{Additional computational effort} = O\left(\quad\right) \quad \text{for} \quad n \to \infty .$$

HIDDEN HINT 1 for (8-19.d) → 8-19-4-0:enmgh1.pdf

SOLUTION for (8-19.d) → 8-19-4-1:.pdf ▲

End Problem 8-19 , 75 min.

Problem 8-20: Rational Functions of Type $(m-1,m)$

In this problem we study a particular class of rational functions for which the degree of the numerator polynomial is smaller than the degree of the denominator polynomial. The focus is on algorithms for the converting different representations of these functions into each other.

This problem requires familiarity with [Lecture → Section 8.4].

The following terminology is borrowed from approximation theory [Tre13, Chapter 23]:

Definition 8.20.1. Rational function of type (m,n)

Given $m,n \in \mathbb{N}_0$ and two polynomials $p \in \mathcal{P}_m$ and $q \in \mathcal{P}_n$ with non-vanishing leading coefficient the quotient function

$$x \mapsto \frac{p(x)}{q(x)}\,, \quad x \in \mathbb{R} \setminus \{z \in \mathbb{R} : q(z) = 0\}\,,$$

is a **rational function of type** (m,n).

In this problem we consider a particular class of rational functions. For $m \in \mathbb{N}$ we define

$$\mathcal{X}_m := \left\{ x \mapsto c\frac{(x-z_1)(x-z_2)\cdot\cdots\cdot(x-z_{m-1})}{(x-a_1)(x-a_2)\cdot\cdots\cdot(x-a_m)} : \begin{array}{ll} [z_1,\ldots,z_{m-1}] \in \mathbb{R}^{m-1}, & \\ [a_1,\ldots,a_m] \in \mathbb{R}^m,\ c \in \mathbb{R}, & \\ z_i \neq a_j & \forall\, i = 1,\ldots,m-1, \\ & \forall j = 1,\ldots,m \end{array} \right\}.$$

This set contains special rational functions of type $(m-1,m)$.

The following data type is used to represent rational functions in $\mathcal{X}_m$.

C++ code 8.20.2: Struct describing $r \in \mathcal{X}_m$

```
struct RatFuncX {
  RatFuncX() = delete;
  RatFuncX(const RatFuncX &) = default;
  RatFuncX(RatFuncX &&) = default;
  RatFuncX &operator=(const RatFuncX &) = default;
  ~RatFuncX() = default;

  template <typename VEC1, typename VEC2>
  RatFuncX(const VEC1 &z, const VEC2 &a, double c) :
      a_(a.size()), z_(z.size()), c_(c), m_(a.size()) {
    assertm(z.size() == m_ - 1, "Size misatch for z!");
    for (size_t i = 0; i < z.size(); ++i) z_[i] = z[i];
    for (size_t i = 0; i < a.size(); ++i) a_[i] = a[i];
  }
  std::vector<double> a_;  // zeros of denominator
  std::vector<double> z_;  // zeros of numerator
  double c_;               // coefficient c
  size_t m_;               // rationa function of type (m-1,m)
};
```

(8-20.a) ⚀ (5 min.) Give an example of a rational function r of type $(1,2)$, which does not belong

to $\mathcal{X}_2$.

$$r(x) = \boxed{} .$$

SOLUTION for (8-20.a) → 8-20-1-0:zrfs1.pdf ▲

(8-20.b) ⚄ (50 min.) In the file zerosratfunc.hpp implement the C++ function

```
std::vector<double> partialFractionExpansion(const RatFuncX &r);
```

that computes the weights $w_j \in \mathbb{R}$. $j = 1,\ldots,m$ for the **partial fraction expansion**

$$c\frac{(x-z_1)(x-z_2)\cdot\cdots\cdot(x-z_{m-1})}{(x-a_1)(x-a_2)\cdot\cdots\cdot(x-a_m)} = \sum_{j=1}^{m}\frac{w_j}{x-a_j}\,, \quad x \notin \{a_1,\ldots,a_m\}\,, \tag{8.20.3}$$

and returns $(w_1,\ldots,w_m)$.

The **RatFuncX**-type argument r contains all information on m, z_j and a_j.

We assume that the zeros of the denominator are all distinct: $a_i \neq a_j$, if $i \neq j$.

SOLUTION for (8-20.b) → 8-20-2-0:zrfs2.pdf ▲

(8-20.c) ⚄ (60 min.) In the file zerosratfunc.hpp complete the code of the C++ function

```
std::vector<double>
  zerosPFE(const std::vector<double> &w,
    const std::vector<double> &a,
    double reltol, double abstol)
```

that relies on **Newton's method** to compute the $m-1$ distinct zeros of the rational function

$$r(x) = \sum_{j=1}^{m}\frac{w_j}{x-a_j}\,, \quad x \notin \{a_1,\ldots,a_m\}\,,$$

$$\text{with} \quad w_j > 0 \quad \text{and} \quad a_j \in \mathbb{R}\,, \quad a_1 < a_2 < \cdots < a_m\,.$$

The arguments w and a pass the coefficient sequence $(w_1,\ldots,w_m)$ and the pole position sequence $(a_1,\ldots,a_m)$, respectively, the latter of which is assumed to be sorted.

The parameters reltol and abstol provide the tolerances for the **correction-based termination** of the Newton iterations.

HIDDEN HINT 1 for (8-20.c) → 8-20-3-0:zrfh13.pdf

SOLUTION for (8-20.c) → 8-20-3-1:zrfs4.pdf ▲

References

[Tre13] Lloyd N. Trefethen. *Approximation theory and approximation practice*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2013, viii+305 pp.+back matter (cit. on p. 303).

End Problem 8-20 , 115 min.

Chapter 9

Computation of Eigenvalues and Eigenvectors

Chapter 10

Krylov Methods for Linear Systems of Equations

Chapter 11

Numerical Integration – Single Step Methods

Problem 11-1: Linear ODE in spaces of matrices

In this problem we consider initial value problems (IVPs) for linear ordinary differential equations (ODEs) whose state space is a vector space of $n \times n$ matrices. Such ODEs arise when modelling the dynamics of rigid bodies in classical mechanics. A related problem is Problem 11-5.

Mainly relies on the information contained in [Lecture → Section 11.2].

We consider the *linear* matrix differential equation

$$\dot{\mathbf{Y}} = \mathbf{A}\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad \text{with} \quad \mathbf{A} \in \mathbb{R}^{n,n}. \tag{11.1.1}$$

whose solutions are *matrix-valued functions* $\mathbf{Y} : \mathbb{R} \to \mathbb{R}^{n,n}$ and whose right-hand-side function maps $\mathbb{R}^{n,n} \to \mathbb{R}^{n,n}$.

(11-1.a) ⚄ (15 min.) Show that for *skew-symmetric* $\mathbf{A}$, i.e. $\mathbf{A} = -\mathbf{A}^\top$ we have:

$$\mathbf{Y}(0) \text{ orthogonal} \quad \Longrightarrow \quad \mathbf{Y}(t) \text{ orthogonal} \quad \forall t\,.$$

HIDDEN HINT 1 for (11-1.a) → 11-1-1-0:MatO1h1.pdf

HIDDEN HINT 2 for (11-1.a) → 11-1-1-1:MatO1h2.pdf

SOLUTION for (11-1.a) → 11-1-1-2:MatO1s.pdf ▲

(11-1.b) ⚄ (30 min.) Implement three C++ functions

1. a single step of the explicit Euler method, see [Lecture → Section 11.2.1]:

```
Eigen::MatrixXd eeulstep(const Eigen::MatrixXd & A,
                         const Eigen::MatrixXd & Y0, double h);
```

2. a single step of the implicit Euler method, see [Lecture → Section 11.2.2],

```
Eigen::MatrixXd ieulstep(const Eigen::MatrixXd & A,
                         const Eigen::MatrixXd & Y0, double h);
```

3. a single step of the implicit mid-point method, see [Lecture → Section 11.2.3].

```
Eigen::MatrixXd impstep(const Eigen::MatrixXd & A,
                        const Eigen::MatrixXd & Y0, double h);
```

which compute, for a given initial value $\mathbf{Y}(t_0) = \mathbf{Y}_0$ and for given step size h, approximations for $\mathbf{Y}(t_0 + h)$ using one step of the corresponding method for the approximation of the ODE (11.1.1)

HIDDEN HINT 1 for (11-1.b) → 11-1-2-0:MatO2h.pdf

SOLUTION for (11-1.b) → 11-1-2-1:MatO2s.pdf ▲

(11-1.c) ⚄ (30 min.) [depends on Sub-problem (11-1.b)]

Investigate numerically, which one of the implemented methods preserves orthogonality for the ODE (11.1.1) and which one doesn't. To that end, consider the matrix

$$\mathbf{M} := \begin{bmatrix} 8 & 1 & 6 \\ 3 & 5 & 7 \\ 9 & 9 & 2 \end{bmatrix}$$

and use the matrix $\mathbf{Q}$ arising from the QR-decomposition (→ [Lecture → Section 3.3.3.4]) of $\mathbf{M}$ as initial data $\mathbf{Y}_0$. As matrix $\mathbf{A}$, use the skew-symmetric matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{bmatrix}.$$

Perform $N = 20$ time steps of size $h = 0.01$ with each method and compute and tabulate the Frobenius norm of $\mathbf{Y}_k^\top \mathbf{Y}_k - \mathbf{I}$.

Do all this with a C++ function

```
std::tuple<double, double, double> checkOrthogonality();
```

that also returns the Frobenius norms $\mathbf{Y}_N^\top \mathbf{Y}_N - \mathbf{I}$ for each of the three methods.

SOLUTION for (11-1.c) → 11-1-3-0:MatO3s.pdf ▲

End Problem 11-1, 75 min.

Problem 11-2: Explicit Runge-Kutta methods

The most widely used class of numerical integrators for IVPs is that of *explicit* Runge-Kutta (RK) methods as defined in [Lecture → Def. 11.4.0.11]. They are usually described by giving their coefficients in the form of a Butcher scheme [Lecture → (11.4.0.13)].

Related to [Lecture → Section 11.4], [Lecture → Def. 11.4.0.11], and Butcher schemes as defined in [Lecture → (11.4.0.13)]

(11-2.a) ⚀ (40 min.) In the template file `rkintegrator.hpp` implement a header-only C++ class

```
template <class State>
class RKIntegrator {
public:
    // Constructor
    RKIntegrator(const MatrixXd & A, const VectorXd & b);
    // Explicit Runge-Kutta numerical integrator
    template <class Function>
    std::vector<State>
      solve(Function &&f, double T,
            const State &y0, unsigned int N) const;
    // ....  data (and private methods)
};
```

which provides a generic **explicit Runge-Kutta single-step method** (→ [Lecture → Def. 11.4.0.11]) given by a Butcher scheme [Lecture → (11.4.0.13)] to solve the autonomous initial-value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$. The matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the vector $\mathbf{b} \in \mathbb{R}^s$ from the Butcher scheme are passed to the constructor, whereas the right-hand side vector field $\mathbf{f}$, the final time $T > 0$, the number N of *equidistant timesteps*, and the intial state $\mathbf{y}_0$ are arguments of the `solve()` method, which is supposed to return the sequence $(\mathbf{y}_k)_{k=0}^N$ of states generated by the explicit Runge-Kutta method.

SOLUTION for (11-2.a) → `11-2-1-0:RK3P1s.pdf` ▲

(11-2.b) ⚂ (60 min.) [depends on Sub-problem (11-2.a)]

In the template file `rk3prey.hpp` implement a C++ function

```
double RK3prey();
```

in order to test your implementation of the RK methods with the following test case:

As autonomous initial value problem, consider the predator/prey model (*cf.* [Lecture → Ex. 11.1.2.5]):

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t)) y_1(t)\,, \tag{11.2.2}$$
$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2) y_2(t)\,, \tag{11.2.3}$$
$$\mathbf{y}(0) = [100, 5]^\top\,, \tag{11.2.4}$$

with coefficients $\alpha_1 = 3, \alpha_2 = 2, \beta_1 = \beta_2 = 0.1$.

Use the explicit 3-stage Runge-Kutta single step method described by the following **Butcher scheme** (*cf.* [Lecture → Def. 11.4.0.11]):

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \tag{11.2.5}$$

Compute an approximated solution up to time $T = 10$ for the number of equidistant time steps $N = 2^j$, $j = 7, \ldots, 14$.

As reference solution, use $\mathbf{y}(10) = [0.319465882659820, 9.730809352326228]^\top$.

Tabulate the error at final time and estimate the empiric rate of algebraic convergence of the method by means of linear regression using the supplied function `polyfit` (file `polyfit.hpp`). Your function should return the estimated convergence rate.

SOLUTION for (11-2.b) → `11-2-2-0:RK3P2s.pdf`

▲

End Problem 11-2 , 100 min.

Problem 11-3: Extrapolation of evolution operators

In [Lecture → § 11.3.1.1] we have seen how discrete evolution operators can describe single-step methods for the numerical integration of ODEs. This task will study a way to combine discrete evolution operators of known order in order to build a single-step method with increased order. The method can be generalized to an extrapolation construction of higher-order single-step methods. These can be used for time-local stepsize control following the policy of [Lecture → § 11.5.2.12].

This problem assumes familiarity with the [Lecture → Section 11.3] and, in particular, the concept of discrete evolution, *cf.* [Lecture → Def. 11.3.1.5]. It also addresses the adaptive timestepping strategy presented in [Lecture → Section 11.5].

Let $\boldsymbol{\Psi}^h$ define the discrete evolution of an order p Runge-Kutta single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subseteq \mathbb{R}^d \to \mathbb{R}^d$. We define a new discrete evolution operator:

$$\widetilde{\boldsymbol{\Psi}}^h := \frac{1}{1-2^p}\left(\boldsymbol{\Psi}^h - 2^p \cdot (\boldsymbol{\Psi}^{h/2} \circ \boldsymbol{\Psi}^{h/2})\right), \tag{11.3.1}$$

where $\circ$ denotes the composition of mappings.

(11-3.a) ⚀ (10 min.) If $\boldsymbol{\Psi}^h$ belongs to the explicit/forward Euler method as introduced in [Lecture → Section 11.2.1], give the explicit formula for $\widetilde{\boldsymbol{\Psi}}^h$.

SOLUTION for (11-3.a) → 11-3-1-0:ODESolve1s.pdf ▲

(11-3.b) ⚁ (15 min.) Implement an EIGEN-based C++ function

```
template <class DiscEvlOp>
Eigen::VectorXd psitilde(DiscEvlOp &&Psi, unsigned int p,
          double h, const Eigen::VectorXd & y0);
```

that returns $\widetilde{\boldsymbol{\Psi}}^h \mathbf{y}_0$ when given the underlying $\boldsymbol{\Psi}$ through the functor `Psi`. Objects of type **DiscEvlOp** must provide an evaluation operator:

```
Eigen::VectorXd operator()(double h, const Eigen::VectorXd &y);
```

providing the result of $\boldsymbol{\Psi}^h(\mathbf{y})$. For instance, suitable C++ lambda functions satisfy this requirement, see [Lecture → Section 0.3.3].

SOLUTION for (11-3.b) → 11-3-2-0:ODESolve2s.pdf ▲

(11-3.c) ⚁ (20 min.) Implement a C++ function

```
template <class DiscEvlOp>
std::vector<Eigen::VectorXd>
odeintequi(DiscEvlOp &&Psi, double T,
           const Eigen::VectorXd &y0, unsigned int N);
```

that carries out N equidistant steps of size $\tau := T/N$ of a single step method described by the discrete evolution $\boldsymbol{\Psi}$ passed through the functor object `Psi`. The function should return the sequence $(\mathbf{y}_k)_{k=0}^N$ produced by the single-step method when started with initial value $\mathbf{y}_0$ (given as argument `y0`).

HIDDEN HINT 1 for (11-3.c) → 11-3-3-0:ODESolve3h.pdf

SOLUTION for (11-3.c) → 11-3-3-1:ODESolve3s.pdf ▲

(11-3.d) ⚁ (30 min.) [depends on Sub-problem (11-3.c), Sub-problem (11-3.a)]

For the particular scalar initial-value problem (IVP)

$$\dot{y} = 1 + y^2 \quad , \quad y(0) = 0 , \tag{11.3.4}$$

with exact solution $y(t) = \tan(t)$, determine empirically the order of the single step method induced by $\widetilde{\boldsymbol{\Psi}}^h$ from (11.3.1), when $\boldsymbol{\Psi}$ arises from the explicit Euler method, recall Sub-problem (11-3.a). For that purpose, write a C++ function

```
double testcvpExtrapolatedEuler();
```

that monitors and tabulates the error $|y_N(1) - y_{ex}(1)|$ at final time $T = 1$ for N uniforms steps with $N = 2^q, q = 2, \ldots, 12$, also returns an estimate of the rate of algebraic convergence based on linear regression.

HIDDEN HINT 1 for (11-3.d) → 11-3-4-0:ODESolve4h.pdf

SOLUTION for (11-3.d) → 11-3-4-1:ODESolve4s.pdf ▲

(11-3.e) ⚄ (30 min.) In general, the method defined by $\widetilde{\boldsymbol{\Psi}}^h$ from (11.3.1) has order $p+1$. Thus, it can be used for adaptive timestep control and prediction.

Complete the implementation of a function

```
template <class DiscEvlOp>
  std::pair<std::vector<double>, std::vector<Eigen::VectorXd>>
    odeintssctrl(DiscEvlOp &&Psi, double T,
                 const Eigen::VectorXd &y0, double h0,
                 unsigned int p, double reltol,
                 double abstol, double hmin);
```

for the approximation of the solution of an IVP by means of adaptive timestepping based on $\boldsymbol{\Psi}$ and $\widetilde{\boldsymbol{\Psi}}$, where $\boldsymbol{\Psi}$ is passed through the argument `Psi`.

Step rejection and stepsize correction and prediction as explained in [Lecture → Section 11.5] is to be employed as in [Lecture → Code 11.5.2.19]. The argument `T` supplies the final time, `y0` the initial state, `h0` an initial stepsize, `p` the order of the discrete evolution $\boldsymbol{\Psi}$, `reltol` and `abstol` the respective tolerances, and `hmin` a minimal stepsize that will trigger premature termination.

The function should return a vector of tuples $(t_k, \mathbf{y}_k)$, $k = 0, \ldots, M$, where $M \in \mathbb{N}$ is the total number of timesteps, t_k are the knots of the temporal mesh created by the adaptive integrator, and $(\mathbf{y}_k)_k$ the sequence of states generated by the single-step method.

HIDDEN HINT 1 for (11-3.e) → 11-3-5-0:EPh.pdf

SOLUTION for (11-3.e) → 11-3-5-1:ODESolve5s.pdf ▲

(11-3.f) ⚂ (20 min.) [depends on Sub-problem (11-3.e)]

Implement a C++ function

```
void solveTangentIVP()
```

that relies on `odeintssctrl` from Sub-problem (11-3.e) to compute the solution of the IVP (11.3.4) up to time $T = 1.5$ with the following data: `h0` $= 1/100$, `reltol` $= 10e-4$, `abstol` $= 10e-6$, `hmin` $= 10e-5$. Finally, the function should use MATPLOTLIBCPP to plot the approximated solution and store the plot in the file `tangent.png`. Do not forget to add axis labels.

SOLUTION for (11-3.f) → 11-3-6-0:sx.pdf ▲

End Problem 11-3 , 125 min.

Problem 11-4: System of second-order ODEs

In this problem we practise the conversion of a second-order ODE into a first-order system in the case of a large linear system of ODEs.

This problem assumes familiarity with Runge-Kutta single-step method as introduced in [Lecture → Section 11.4]

Consider the following initial value problem for an (implicit) second-order system of ordinary differential equations in the time interval $[0,T]$:

$$\begin{aligned} 2\ddot{u}_1 - \ddot{u}_2 &= u_1(u_2+u_1)\,, \\ -\ddot{u}_{i-1} + 2\ddot{u}_i - \ddot{u}_{i+1} &= u_i(u_{i-1}+u_{i+1})\,, \qquad i = 2,\ldots,n-1\,, \\ 2\ddot{u}_n - \ddot{u}_{n-1} &= u_n(u_n+u_{n-1})\,, \\ u_i(0) &= u_{0,i} \qquad i = 1,\ldots,n\,, \\ \dot{u}_i(0) &= v_{0,i} \qquad i = 1,\ldots,n\,. \end{aligned} \tag{11.4.1}$$

Here the notation $\ddot{w}$ designates the second derivative of a time-dependent function $t \mapsto w(t)$.

(11-4.a) ⚂ (15 min.) Write (11.4.1) as a first-order IVP of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

HIDDEN HINT 1 for (11-4.a) → 11-4-1-0:Syst1h1.pdf

HIDDEN HINT 2 for (11-4.a) → 11-4-1-1:Syst1h2.pdf

HIDDEN HINT 3 for (11-4.a) → 11-4-1-2:Syst1h3.pdf

SOLUTION for (11-4.a) → 11-4-1-3:Syst1s.pdf ▲

(11-4.b) ⚂ (15 min.) Write down the discrete evolution for the classical Runge-Kutta method of order 4, whose Butcher scheme is given in [Lecture → Ex. 11.4.0.17] for the concrete case of the autonomous linear ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{R}^{d,d}$.

HIDDEN HINT 1 for (11-4.b) → 11-4-2-0:h2.pdf

SOLUTION for (11-4.b) → 11-4-2-1:Syst2s.pdf ▲

(11-4.c) ⚂ (20 min.) [depends on Sub-problem (11-4.b)]

Implement a C++ function

```
template <class Function, class State>
void rk4step(Function &&odefun, double h, const State & y0, State
   & y1);
```

that performs a single step of stepsize h of the classical Runge-Kutta method of order 4 for the first-order ODE obtained in Sub-problem (11-4.a) (associated right-hand side function passed via the functor `odefun`). `y0` contains the state before the step and the function has to return the state after the step in `y1`.

SOLUTION for (11-4.c) → 11-4-3-0:Syst3s.pdf ▲

(11-4.d) ⚂ (40 min.) [depends on Sub-problem (11-4.a), Sub-problem (11-4.c)]

Implement a function

```
double testcvgRK4()
```

that applies the classical Runge-Kutta method of order 4 to solve a particular initial-value problem for the ODE derived in Sub-problem (11-4.a).

Use the parameters and initial values

$$n = 5\,, \qquad u_i(0) = i/n\,, \qquad v_i(0) = -1\,, \qquad T = 1\,,$$

and tabulate the Euclidean norm of the error at final time $T = 1$ for numbers $N = 2, 4, 8, \ldots, 1024$ of uniform timesteps. As reference solution use the value obtained with $N = 2^{12}$ equidistant timesteps. The function should also return an estimate of the rate of algebraic convergence obtained by means of linear regression for which you can use the supplied function `polyfit()` from `polyfit.hpp`.

Note that you should use EIGEN's sparse matrix data type for any sparse matrix encountered. Refer to [Lecture → Section 2.7.2] for details.

SOLUTION for (11-4.d) → `11-4-4-0:Syst4s.pdf` ▲

End Problem 11-4, 90 min.

Problem 11-5: Non-linear Evolutions in Spaces of Matrices

In this problem we consider initial value problems (IVPs) for ordinary differential equations whose state space is a vector space of $n \times n$ matrices. Such IVPs occur in mathematical models of discrete mechanical systems.

Related to this problem is Problem 11-1.

We consider a *non-linear* ODE in the state space of $n \times n$ matrices and study the associated initial value problem

$$\dot{\mathbf{Y}} = -(\mathbf{Y} - \mathbf{Y}^\top)\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad , \quad \mathbf{Y}(0) = \mathbf{Y}_0 \in \mathbb{R}^{n,n}, \tag{11.5.1}$$

whose solution is given by a *matrix-valued function* $t \mapsto \mathbf{Y}(t) \in \mathbb{R}^{n,n}$.

(11-5.a) ⚄ (15 min.) In the file `NLmatode.hpp`, implement the C++ function

```
Eigen::MatrixXd matode(const Eigen::MatrixXd & Y0, double T)
```

which solves (11.5.1) on $[0, T]$ using the C++ header-only class `ode45` (in the file `ode45.hpp`). The initial value should be given by a $n \times n$ EIGEN matrix `Y0`. Set the absolute tolerance to 10^{-10} and the relative tolerance to 10^{-8}. The output should be an approximation of $\mathbf{Y}(T) \in \mathbb{R}^{n,n}$.

Instructions on the use of ode45, see [Lecture → § 11.5.3.3] ➙ GITLAB.

The class **ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` defines the class **ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 11.5.3] and [Lecture → Ex. 11.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 11.5.2.12].

1. Construct an object of **ode45** type: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

```
template <class StateType,
class RhsType = std::function<StateType(const StateType &)>>
class ode45 {
  public:
  ode45(const RhsType &rhs);
  // ......
}
```

   Template parameters are

   - **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
   - **RhsType**: type of rhs function (automatically deduced).

   The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator()(const StateType & vec);
```

   It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is `1e-6`)
- `atol`: absolute tolerance for error control (default is `1e-8`)

e.g.:

```
O.options.rtol = 1e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t),t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

SOLUTION for (11-5.a) → `11-5-1-0:NMatO1s.pdf` ▲

(11-5.b) ⚄ (10 min.) Show that the function $t \mapsto \mathbf{Y}^\top(t)\mathbf{Y}(t)$ is constant for the exact solution $\mathbf{Y}(t)$ of (11.5.1).

HIDDEN HINT 1 for (11-5.b) → `11-5-2-0:NMatO2h1.pdf`

HIDDEN HINT 2 for (11-5.b) → `11-5-2-1:NMatO2h2.pdf`

SOLUTION for (11-5.b) → `11-5-2-2:NMatO2s.pdf` ▲

(11-5.c) ⚄ (15 min.) [depends on Sub-problem (11-5.a)]

In the file `NLmatode.hpp`, implement the C++ function

```
bool checkinvariant(const Eigen::MatrixXd & M, double T);
```

which (numerically) determines if the invariant $t \mapsto \mathbf{Y}(t)^\top\mathbf{Y}(t)$ is preserved for approximate solutions of (11.5.1) as output by `matode()`. You must take into account round-off errors. The function's arguments should be the same as that of `matode()`.

SOLUTION for (11-5.c) → `11-5-3-0:NMatO3s.pdf` ▲

(11-5.d) ⚄ (40 min.) [depends on Sub-problem (11-5.a)]

The so-called **discrete gradient method** for (11.5.1) reads

$$\mathbf{Y}_* = \mathbf{Y}_0 + \tfrac{1}{2}hf(\mathbf{Y}_0) \quad , \quad \mathbf{Y}_1 = (\mathbf{I} + \tfrac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*^\top))^{-1}(\mathbf{I} - \tfrac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*^\top))\mathbf{Y}_0 \,. \tag{11.5.5}$$

Using the solution produced by `matode()` from Sub-problem (11-5.a) as a substitute for an exact solution, determine the order of convergence of the discrete gradient rule in a numerical experiment that uses $M \in \{10, 20, 40, 80, 160, 320, 640, 1280\}$ equidistant integration steps for solving (11.5.1) approximately. As initial value $\mathbf{Y}_0$ use the *orthogonal* "left-shift" matrix

$$\mathbf{Y}_0 := \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix} .$$

Concretely, implement a C++ function

```
double cvgDiscreteGradientMethod();
```

that tabulates the Frobenius norm of the discretization error at final time $T = 1$ and returns an estimate of the rate of algebraic convergence obtained by linear regression using the supplied function `polyfit` (file `polyfit.hpp`).

SOLUTION for (11-5.d) → `11-5-4-0:s5.pdf` ▲

End Problem 11-5, 80 min.

Problem 11-6: Order is not everything

In [Lecture → Section 11.3.2] we have seen that Runge-Kutta single step methods when applied to initial value problems with sufficiently smooth solutions will converge algebraically (with respect to the maximum error in the mesh points) with a rate given by their intrinsic order, see [Lecture → Def. 11.3.2.8].

This problem studies the convergence of some Runge-Kutta single-step methods as discussed in [Lecture → Section 11.3.2]. It relies on a class implemented in Problem 11-2.

In this problem we perform empiric investigations of orders of convergence of several explicit Runge-Kutta single step methods. We rely on two IVPs, one of which has a perfectly smooth solution, whereas the second has a solution that is merely piecewise smooth. Thus in the second case the smoothness assumptions of the convergence theory for RK-SSMs might be violated and it is interesting to study the consequences.

Remark. For this problem you can reuse the class **RKIntegrator** implemented in Problem 11-2. You first need to construct an object of this class, passing as arguments the Butcher tableau matrices `A` and `b`, for instance:

```
RKIntegrator<double> rk(A,b);
```

After that, call the methods `solve`, with parameters: r.h.s. function, final time, initial value and number of steps. For instance:

```
rk.solve(f,T,y0,n);
```

The output of this function will be **std::vector<double>** containing the solution at each equidistant time step.

(11-6.a) ⚄ (30 min.) Consider the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{11.6.1}$$

where $\mathbf{f}: \mathbb{R} \to \mathbb{R}$ and $\mathbf{y}_0 \in \mathbb{R}$. Using the class **RKIntegrate** write a C++ function

```
template <class Function>
double testCvgRKSSM(const Function &f, double T, double y0,
const Eigen::MatrixXd &A, const Eigen::VectorXd &b)
```

in `order_not_all.hpp` that computes an approximated solution $\mathbf{y}_M$ of (11.6.1) up to time T by means of an explicit Runge-Kutta method with $M = 2^k$, $k = 1, \ldots, 12$, uniform timesteps. The method is defined by the Butcher scheme described by the inputs `A` (Butcher matrix) and `b` (weight vector), see [Lecture → (11.4.0.13)]. The input `f` is an object with an evaluation operator (e.g. a lambda function) for arguments of type **double** representing $\mathbf{f}$. The input `y0` passes the initial value $\mathbf{y}_0$.

For each k, the function should print the error at the final point $E_M = |\mathbf{y}_M(T) - \mathbf{y}_{2^{15}}(T)|$, $M = 2^k$, $k = 1, \ldots, 12$, accepting $\mathbf{y}_{2^{15}}(T)$ as exact value. Assuming algebraic convergence for $E_M \approx CM^{-r}$, at each $k = 2, \ldots, 12$ also print an approximation of the order of convergence r_k (recall that $M = 2^k$). This will be an expression involving E_M and $E_{M/2}$.

Finally, compute and return an approximate order of convergence by linear regression. Only take into account results for which the error is larger than 10^{-14} in order not to be misled by the impact of round-off errors.

SOLUTION for (11-6.a) → `11-6-1-0:OrdN1h.pdf` ▲

(11-6.b) ⚄ (30 min.) Calculate the analytical solutions of the logistic ODE (see [Lecture → Ex. 11.1.2.1])

$$\dot{y} = (1-y)y, \quad y(0) = 1/2, \tag{11.6.4}$$

and of the initial value problem

$$\dot{y} = |1.1 - y| + 1, \quad y(0) = 1. \tag{11.6.5}$$

HIDDEN HINT 1 for (11-6.b) → 11-6-2-0:ONA2s.pdf

HIDDEN HINT 2 for (11-6.b) → 11-6-2-1:OMA6s.pdf

HIDDEN HINT 3 for (11-6.b) → 11-6-2-2:OMA8s.pdf

SOLUTION for (11-6.b) → 11-6-2-3:OrdN2h.pdf ▲

(11-6.c) ⚄ (20 min.) [depends on Sub-problem (11-6.a)]

Using `testCvgRKSSM()` write in `order_not_all.hpp` a C++ function

```
void cmpCvgRKSSM();
```

that empirically determines and prints the rates of convergence of

- the explicit Euler method [Lecture → (11.2.1.5)], a RK single step method of order 1,
- the explicit trapezoidal rule [Lecture → (11.4.0.8)], a RK single step method of order 2,
- an RK method of order 3 given by the Butcher tableau [Lecture → (11.4.0.13)]

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 1 & -1 & 2 & \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

- the classical RK method of order 4, see [Lecture → Ex. 11.4.0.17] for details.

when applied for the numerical integration of the initial-value problems (11.6.4) and (11.6.5). Use final time $T = 1$ in each case.

Comment on the calculated order of convergence for the different methods and the two different initial value problems.

SOLUTION for (11-6.c) → 11-6-3-0:x1s.pdf ▲

End Problem 11-6, 80 min.

Problem 11-7: Initial Condition for Lotka-Volterra ODE

In this problem we will face a situation, where we need to compute the derivative of the solution of an initial value problem with respect to the initial state in order to apply Newton's method. So thus exercise covers both numerical integration and the solution of non-linear systems of equations.

You should grasp the abstract view of ODEs from [Lecture → Section 11.1.4] and still remember Newton's method from [Lecture → Section 8.5].

A differential equation for the derivative w.r.t. initial state. We consider IVPs for the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{11.7.1}$$

with smooth right hand side $\mathbf{f}: D \to \mathbb{R}^d$, where $D \subseteq \mathbb{R}^d$ is the state space. We take for granted that for all initial states, solutions exist for all times (global solutions, see [Lecture → Ass. 11.1.4.1]).

By its very definition given in [Lecture → Def. 11.1.4.3], the evolution operator

$$\boldsymbol{\Phi}: \mathbb{R} \times D \to D, \quad (t, \mathbf{y}) \mapsto \boldsymbol{\Phi}(t, \mathbf{y})$$

satisfies

$$\frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y})).$$

Next, we can differentiate this identity with respect to the state variable $\mathbf{y}$. We assume that all derivatives can be interchanged, which can be justified by rigorous arguments (which we won't do here). Thus, by the chain rule, we obtain, after swapping partial derivatives $\frac{\partial}{\partial t}$ and $\mathrm{D}_{\mathbf{y}}$,

$$\frac{\partial \,\mathrm{D}_{\mathbf{y}} \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathrm{D}_{\mathbf{y}} \frac{\partial \boldsymbol{\Phi}}{\partial t}(t, \mathbf{y}) = \mathrm{D}_{\mathbf{y}}(\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))) = \mathrm{D}\,\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))\,\mathrm{D}_{\mathbf{y}}\,\boldsymbol{\Phi}(t, \mathbf{y}).$$

Abbreviating $\mathbf{W}(t, \mathbf{y}) := \mathrm{D}_{\mathbf{y}}\,\boldsymbol{\Phi}(t, \mathbf{y})$ we can rewrite this as the non-autonomous ODE

$$\dot{\mathbf{W}} = \mathrm{D}\,\mathbf{f}(\boldsymbol{\Phi}(t, \mathbf{y}))\mathbf{W} \;. \tag{11.7.2}$$

Here, the state $\mathbf{y}$ can be regarded as a parameter. Since $\boldsymbol{\Phi}(0, \mathbf{y}) = \mathbf{y}$, we also know $\mathbf{W}(0, \mathbf{y}) = \mathbf{I}$ (identity matrix), which supplies an initial condition for (11.7.2). In fact, we can even merge (11.7.1) and (11.7.2) into the ODE

$$\frac{d}{dt}[\mathbf{y}(\cdot)\,,\ \mathbf{W}(\cdot, \mathbf{y}_0)] = [\mathbf{f}(\mathbf{y}(t))\,,\ \mathrm{D}\,\mathbf{f}(\mathbf{y}(t))\mathbf{W}(t, \mathbf{y}_0)]\,, \tag{11.7.3}$$

which is autonomous again.

Now let us apply (11.7.2)/(11.7.3). As in [Lecture → Ex. 11.1.2.5], we consider the following autonomous Lotka-Volterra differential equation of a predator-prey model

$$\begin{aligned} \dot{u} &= (2 - v)u\,, \\ \dot{v} &= (u - 1)v\,, \end{aligned} \tag{11.7.4}$$

on the state space $D = \mathbb{R}^2_+$, $\mathbb{R}_+ = \{\xi \in \mathbb{R} : \xi > 0\}$. All the solutions of (11.7.4) are periodic and their period depends on the initial state $[u(0), v(0)]^T$. In this exercise we want to develop a numerical method which computes a suitable initial condition for a given period.

(11-7.a) ⚀ (10 min.) For fixed state $\mathbf{y} \in D$, (11.7.2) represents an ODE. What is its state space?

SOLUTION for (11-7.a) → 11-7-1-0:Init1h.pdf ▲

(11-7.b) ⚁ (15 min.) What is the right hand side function for the ODE (11.7.2), when the underlying autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ is the Lotka-Volterra ODE (11.7.4)? You may write $u(t), v(t)$ for solutions of (11.7.4).

SOLUTION for (11-7.b) → 11-7-2-0:Init2h.pdf ▲

(11-7.c) ⚂ (15 min.) From now on we write $\boldsymbol{\Phi} : \mathbb{R} \times \mathbb{R}^2_+ \to \mathbb{R}^2_+$ for the evolution operator (→ [Lecture → Def. 11.1.4.3]) associated with (11.7.4). Based on $\boldsymbol{\Phi}$ derive a non-trivial function $\mathbf{F} : \mathbb{R}^2_+ \to \mathbb{R}^2$, $\mathbf{F} \not\equiv \mathbf{0}$, which evaluates to zero for the input $\mathbf{y}_0$ if the period of the solution of system (11.7.4) with initial value

$$\mathbf{y}_0 = \begin{bmatrix} u(0) \\ v(0) \end{bmatrix}$$

is equal to a given value $T_P > 0$.

SOLUTION for (11-7.c) → 11-7-3-0:Init4h.pdf ▲

(11-7.d) ⚁ (10 min.) [depends on Sub-problem (11-7.c)]

We write $\mathbf{W}(T, \mathbf{y}_0)$, $T \geq 0$, $\mathbf{y}_0 \in \mathbb{R}^2_+$ for the solution of (11.7.2) for the underlying ODE (11.7.4). Express the Jacobian of $\mathbf{F}$ by means of $\mathbf{W}$.

SOLUTION for (11-7.d) → 11-7-4-0:Init5h.pdf ▲

(11-7.e) ⚃ (15 min.) [depends on Sub-problem (11-7.c)]

Argue, why the solution of $\mathbf{F}(\mathbf{y}) = 0$ will, in gneneral, not be unique. When will it be unique?

HIDDEN HINT 1 for (11-7.e) → 11-7-5-0:Init1s.pdf

SOLUTION for (11-7.e) → 11-7-5-1:Init6h.pdf ▲

A C++ implementation **ode45** of an adaptive embedded Runge-Kutta method has been presented in [Lecture → **??**].

Instructions on the use of ode45, see [Lecture → § 11.5.3.3] ⇝ GITLAB.

The class **ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` defines the class **ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 11.5.3] and [Lecture → Ex. 11.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 11.5.2.12].

1. Construct an object of **ode45** type: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

```
template <class StateType,
class RhsType = std::function<StateType(const StateType &)>>
class ode45 {
  public:
  ode45(const RhsType &rhs);
  // ......
}
```

   Template parameters are

- **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
- **RhsType**: type of rhs function (automatically deduced).

The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator()(const StateType & vec);
```

It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is `1e-6`)
- `atol`: absolute tolerance for error control (default is `1e-8`)

e.g.:

```
O.options.rtol = 1e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

(11-7.f) ⚀ (30 min.) [depends on Sub-problem (11-7.b)]

Relying on **ode45**, implement a C++ function

```
std::pair<Eigen::Vector2d,Eigen::Matrix2d> PhiAndW(double u0,
  double v0, double T)
```

that computes $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and $\mathbf{W}(T, [u_0, v_0]^T)$. The first component of the output pair should contain $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and the second component the matrix $\mathbf{W}(T, [u_0, v_0]^T)$.

Set relative and absolute tolerances of **ode45** to 10^{-14} and 10^{-12}, respectively.

HIDDEN HINT 1 for (11-7.f) → 11-7-6-0:Init2s.pdf

SOLUTION for (11-7.f) → 11-7-6-1:Init7h.pdf ▲

(11-7.g) ⚂ (30 min.) [depends on Sub-problem (11-7.f) and Sub-problem (11-7.d)]

Relyiong on `PhiAndW()`, write a C++ routine

```
std::pair<double,double> findInitCond();
```

that determines and returns initial conditions $u(0)$ and $v(0)$ such that the solution of the system (11.7.4) has period $T_P = 5$. Use the multi-dimensional **Newton method** for $\mathbf{F}(\mathbf{y}) = 0$ with $\mathbf{F}$. As your initial approximation, use $[3,2]^T$. Terminate the Newton iteration as soon as $|\mathbf{F}(\mathbf{y})| \leq 10^{-5}$. Validate your implementation by comparing the obtained initial data $\mathbf{y}$ with $\mathbf{\Phi}(100,\mathbf{y})$.

Remark. The residual based termination criterion recommended above [Lecture → § 8.2.3.4] is appropriate for this particular application and, in general, should not be used for Newton's method. Better termination criteria are proposed in [Lecture → Section 8.5.3].

HIDDEN HINT 1 for (11-7.g) → 11-7-7-0:lvxh.pdf

SOLUTION for (11-7.g) → 11-7-7-1:Init8h.pdf ▲

End Problem 11-7, 125 min.

Problem 11-8: Integrating ODEs using the Taylor expansion method

In [Lecture → Chapter 11] of the course we studied single step methods for the integration of initial value problems for ordinary differential equations $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, [Lecture → Def. 11.3.1.5]. Explicit single step methods have the advantage that they only rely on *point evaluations* of the right hand side $\mathbf{f}$.
However, if derivatives of $\mathbf{f}$ are also available, one have more options and this problem examines a class of single-step methods that also rely on $\mathrm{D}\,\mathbf{f}$.

The new class of methods is obtained by the following reasoning: if the right hand side $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ of an autonomous initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) , \qquad \mathbf{y}(0) = \mathbf{y}_0 , \tag{11.8.1}$$

with solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is smooth, the solution $\mathbf{y}(t)$ will also be regular and it is possible to expand it into a Taylor sum at $t = 0$, see [Lecture → Thm. 8.3.2.12],

$$\mathbf{y}(t) = \sum_{k=0}^{m} \frac{\mathbf{y}^{(k)}(0)}{k!} t^k + R_m(t) , \tag{11.8.2}$$

with remainder term $R_m(t) = O(t^{m+1})$ for $t \to 0$.

A single step method for the numerical integration of (11.8.1) can be obtained by choosing $m = 3$ in (11.8.2), neglecting the remainder term and taking the remaining sum as an approximation of $\mathbf{y}(h)$, that is

$$\mathbf{y}(h) \approx \mathbf{y}_1 := \mathbf{y}(0) + \frac{d\mathbf{y}(0)}{dt} h + \frac{1}{2} \frac{d^2\mathbf{y}(0)}{dt^2} h^2 + \frac{1}{6} \frac{d^3\mathbf{y}(0)}{dt^3} h^3 .$$

Subsequently, one uses the ODE and the initial condition to replace the temporal derivatives $\frac{d^i\mathbf{y}}{dt^i}$ with expressions in terms of (derivatives of) $\mathbf{f}$. This yields a single step integration method called *Taylor (expansion) method*.

(11-8.a) ⚀ (15 min.) Express $\frac{d\mathbf{y}}{dt}(t)$ and $\frac{d^2\mathbf{y}}{dt^2}(t)$ in terms of $\mathbf{f}$ and its Jacobian $\mathbf{Df}$.

HIDDEN HINT 1 for (11-8.a) → 11-8-1-0:Tayl1h.pdf

SOLUTION for (11-8.a) → 11-8-1-1:Tayl1s.pdf ▲

(11-8.b) ⚄ (30 min.) [depends on Sub-problem (11-8.a)]

Verify the formula

$$\frac{d^3\mathbf{y}}{dt^3}(0) = \mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{f}(\mathbf{y}_0), \mathbf{f}(\mathbf{y}_0)) + \mathbf{Df}(\mathbf{y}_0)^2\mathbf{f}(\mathbf{y}_0) . \tag{11.8.3}$$

HIDDEN HINT 1 for (11-8.b) → 11-8-2-0:Tayl0h.pdf

HIDDEN HINT 2 for (11-8.b) → 11-8-2-1:Tayl2h.pdf

HIDDEN HINT 3 for (11-8.b) → 11-8-2-2:Tayl24h.pdf

SOLUTION for (11-8.b) → 11-8-2-3:Tayl2s.pdf ▲

(11-8.c) ⚄ Now we apply the Taylor expansion method introduced above to the following ODE for the *predator-prey* model, as introduced in [Lecture → Ex. 11.1.2.5]:

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t))y_1(t)\,, \tag{11.8.4a}$$
$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2)y_2(t)\,, \tag{11.8.4b}$$
$$\mathbf{y}(0) = [100,5]^\top\,. \tag{11.8.4c}$$

To this end, in the template file `taylorintegrator.hpp` implement a C++ function

```
std::vector<Eigen::Vector2d> solvePredPreyTaylor(
  double alpha1, double beta1, double alpha2, double beta2,
  double T, const Eigen::Vector2d& y0, unsigned int M);
```

for the numerical integration of initial-value problems for (11.8.4) using the Taylor expansion method with M, $M \in \mathbb{N}$ uniform timesteps on the temporal interval $[0,T]$. The arguments `alpha1`, `beta1`, `alpha2`, `beta2` provide the parameters of the ODE-based model (11.8.4), while `y0` passes the initial value, and `T` the final time. `M` specifies the number of equidistant timesteps to be carried out. The function should return a sequence of approximate states $\mathbf{y}_k$, $k = 0,\ldots,M$.

SOLUTION for (11-8.c) → 11-8-3-0:Tayl3s.pdf ▲

(11-8.d) ⚂ (20 min.) [depends on Sub-problem (11-8.c)]

Based on the function `solvePredPreyTaylor()` implemented in Sub-problem (11-8.c) experimentally determine the order of convergence of the considered Taylor expansion method when it is applied to solve (11.8.4). Study the behaviour of the error at final time $t = 10$ for the initial data $\mathbf{y}(0) = [100,5]^\top$ and model coefficients $\alpha_1 = 3, \alpha_2 = 2, \beta_1 = \beta_2 = 0.1$. As reference solution, use $\mathbf{y}(10) = [0.319465882659820, 9.730809352326228]^\top$. Implement a function

```
double testCvgTaylorMethod();
```

that generates a suitable error table and returns the empiric rate of algebraic convergence in terms of the (uniform) timestep $h \to 0$. This rate should be determined by linear regression.

SOLUTION for (11-8.d) → 11-8-4-0:Tayl4s.pdf ▲

(11-8.e) ⚂ (5 min.) What is the disadvantage of the Taylor's method compared with a Runge-Kutta method?

SOLUTION for (11-8.e) → 11-8-5-0:Tayl5s.pdf ▲

End Problem 11-8, 70 min.

Problem 11-9: Drawing Isolines of a Function

Isoline/countour plots of functions are a popular way to visualize real-valued functions or data defined on a subset of the plane. In this exercise we will use numerical integration for finding isolines.

This problem assumes familiarity with explicit adaptive Runge-Kutta single-step methods as introduced in [Lecture → Section 11.4] and [Lecture → Section 11.5]. It relies on the **ode45** integrator class from [Lecture → **??**].

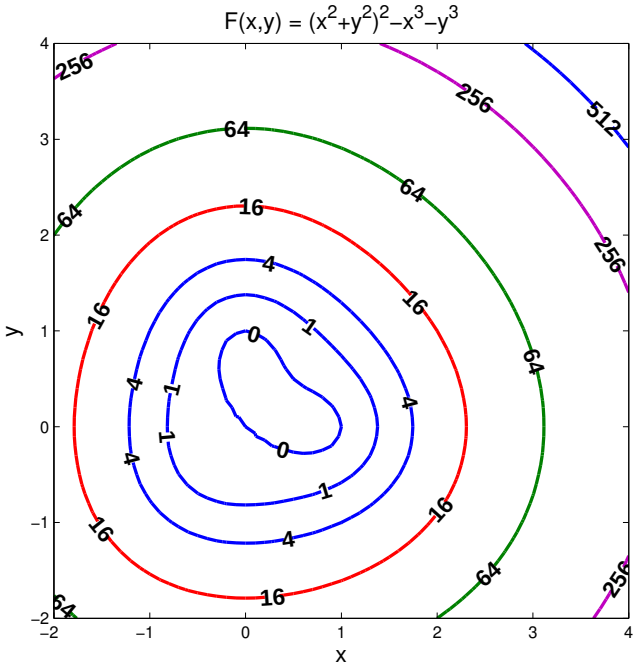


Fig. 115

An isoline is a curve in the plane $\mathbb{R}^2$ that can be defined as the set of zeros of a continuous function $F : \mathbb{R}^2 \mapsto \mathbb{R}$:

$$\mathcal{C} := \{\boldsymbol{x} \in \mathbb{R}^2 : \ F(\boldsymbol{x}) = 0\} \ .$$

In PYTHON (matplotlib) or MATLAB they can be plotted with the function `contour`

◁ Contour plot of the function $F(x,y) = (x^2+y^2)^2 - x^3 - y^3$

If F is continuously differentiable, we can obtain connected components of $\mathcal{C}$ as solution curves of the initial value problems

$$\dot{\mathbf{y}}(t) = \frac{\mathbf{grad}\, F(\mathbf{y}(t))^{\perp}}{\|\mathbf{grad}\, F(\mathbf{y}(t))\|} \ , \quad F(\mathbf{y}(0)) = 0 \ . \tag{11.9.1}$$

Here $^{\perp}$ stands for the counterclockwise rotation of a vector $\in \mathbb{R}^2$ by $90°$, that is,

$$\begin{bmatrix} x \\ y \end{bmatrix}^{\perp} = \begin{bmatrix} -y \\ x \end{bmatrix} \ , \quad x, y \in \mathbb{R} \ . \tag{11.9.2}$$

Then (11.9.1) is a consequence of the fact that an isoline always runs perpendicular to the direction of steepest ascent of F, which is provided by $\mathbf{grad}\, F$.

(11-9.a) ⚂ (5 min.)

State the ordinary differential equation (11.9.1) for the concrete case of the function

$$F : \mathbb{R}^2 \to \mathbb{R} \ , \quad F(x,y) := (x^2+y^2)^2 - x^3 - y^3 \ . \tag{11.9.3}$$

SOLUTION for (11-9.a) → `11-9-1-0:s1.pdf`

▲

(11-9.b) ⚂ (30 min.)

Use (with `rtol = 0.00001, atol = 1e-9`) the provided EIGEN-based integrator class **ode45** that employs an embedded explicit Runge-Kutta pair of methods to implement (in the file `contour.hpp`) a C++ function

```
template <typename GradientFunctor>
  Eigen::Matrix<double,2,Eigen::Dynamic>
computeIsolinePoints(GradientFunctor &&gradF,
                     Eigen::Vector2d y0, double T);
```

that solves an initial value problem for (11.9.1) with initial state $\mathbf{y}_0 \in \mathbb{R}^2$ passed in `y0` from initial time 0 up to final time T (given in `T`). The argument `gradF` supplies a functor object equipped with an evaluation operator

```
Eigen::Vector2d operator(Eigen::Vector2d x) const;
```

that returns $\mathbf{grad}\, F(\boldsymbol{x})$. The function `computeIsolinePoints()` should return the sequence of computed states $\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_N$, $N \in \mathbb{N}$, in the columns of an $2 \times (N+1)$-matrix, where N is the number of steps taken by the adaptive integrator of **ode45**.

Instructions on the use of ode45, see [Lecture → § 11.5.3.3] ↬ GITLAB.

The class **ode45** is header-only, meaning you just include the file and use it right away (no linking required). The file `ode45.hpp` defines the class **ode45** implementing an embedded Runge-Kutta-Fehlberg method of order 4(5), see [Lecture → Section 11.5.3] and [Lecture → Ex. 11.5.3.2], with an adaptive stepsize control as presented in [Lecture → § 11.5.2.12].

1. Construct an object of **ode45** type: create an instance of the class, passing the right-hand-side function $\mathbf{f}$ as a functor object to the constructor

   ```
   template <class StateType,
   class RhsType = std::function<StateType(const StateType &)>>
   class ode45 {
     public:
     ode45(const RhsType &rhs);
     // ......
   }
   ```

   Template parameters are

   - **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
   - **RhsType**: type of rhs function (automatically deduced).

   The argument `rhs` must be of a functor type that provides an evaluation operator

   ```
   StateType operator()(const StateType & vec);
   ```

   It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **ode45.options** to configure the solver:

   ```
   O.options.<option_you_want_to_set> = <value>;
   ```

   Examples:

   - `rtol`: relative tolerance for error control (default is `1e-6`)
   - `atol`: absolute tolerance for error control (default is `1e-8`)

e.g.:

```
O.options.rtol = 1e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

SOLUTION for (11-9.b) → `11-9-2-0:s2.pdf` ▲

(11-9.c) ⚀ (10 min.) [depends on Sub-problem (11-9.a) and Sub-problem (11-9.b)]

The 0-isoline of F from (11.9.3)

$$\mathcal{C}_{\text{egg}} := \{\boldsymbol{x} = (x,y) \in \mathbb{R}^2 \setminus \{\mathbf{0}\} : F(\boldsymbol{x}) := (x^2+y^2)^2 - x^3 - y^3 = 0\} .$$

is known as "crooked egg curve". Based on `computeIsolinePoints()` and with $\mathbf{y}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \in \mathcal{C}_{\text{egg}}$ and $T = 4$ implement a C++ function (in the file `contour.hpp`)

```
Eigen::Matrix<double,2,Eigen::Dynamic> crookedEgg();
```

that returns coordinates of densely spaced points $\in \mathcal{C}_{\text{egg}}$ in the columns of the produced matrix.

HIDDEN HINT 1 for (11-9.c) → `11-9-3-0:s3h1.pdf`

SOLUTION for (11-9.c) → `11-9-3-1:s3.pdf` ▲

(11-9.d) ⚁ (20 min.) [depends on Sub-problem (11-9.b)]

Again rely on the integrator class **ode45** to implement (in the file `contour.hpp`) a function

```
template<typename FFunctor>
  Eigen::Matrix<double,2,Eigen::Dynamic>
    computeIsolinePointsDQ(FFunctor &&F,
                           Eigen::Vector2d y0, double T);
```

that performs the same computation as `computeIsolinePoints()` from Sub-problem (11-9.b). However, this time the argument `F` merely contains a functor object providing the function F itself.

To obtain an approximation of $\mathbf{grad}\, F$ rely on an approximation by **symmetric difference quotients**, e.g.,

$$\frac{\partial F}{\partial x_1}(\boldsymbol{x}) \approx \frac{F(\boldsymbol{x} + \begin{bmatrix} h \\ 0 \end{bmatrix}) - F(\boldsymbol{x} - \begin{bmatrix} h \\ 0 \end{bmatrix})}{2h} \quad \text{for some} \quad h > 0 \,.$$

Choose a suitable fixed $h > 0$ and explain your choice in a comment in your code.

HIDDEN HINT 1 for (11-9.d) → 11-9-4-0:h4.pdf

SOLUTION for (11-9.d) → 11-9-4-1:s4.pdf ▲

End Problem 11-9 , 65 min.

Problem 11-10: Symplectic Timestepping for Equations of Motion

This problem examines a special class of timestepping schemes suitable for solving numerically equations of motions from classical mechanics. These timestepping schemes preserve profound stuctural properties of those equations and have been dubbed **symplectic**.

This problem considers a particular instance of a single-step method as introduced in [Lecture → Section 11.3].

The authors of [RWR04] consider ordinary differential equations on the state space $\mathbb{R}^{2n}$, $n \in \mathbb{N}$, of the form

$$\frac{d}{dt}\begin{bmatrix}\mathbf{p}\\ \mathbf{q}\end{bmatrix} = \begin{bmatrix}\mathbf{f}(\mathbf{q},t)\\ \mathbf{g}(\mathbf{p})\end{bmatrix} \quad \text{with continuous functions} \quad \begin{aligned}&\mathbf{f}:\mathbb{R}^n\times\mathbb{R}\to\mathbb{R}^n\,,\\ &\mathbf{g}:\mathbb{R}^n\to\mathbb{R}^n\,.\end{aligned} \tag{11.10.1}$$

For the temporal discretization of (11.10.1), [RWR04, Sect. IV] proposes the following class of s-stage methods, $s \in \mathbb{N}$, with uniform timestep $h > 0$ for generating sequences $\left(\begin{bmatrix}\mathbf{p}_j\\ \mathbf{q}_j\end{bmatrix}\right)_{j=0,\ldots,M}$ of approximations of $\begin{bmatrix}\mathbf{p}(jh)\\ \mathbf{q}(jh)\end{bmatrix}$, starting from initial values $\mathbf{p}_0 := \mathbf{p}(0)$ and $\mathbf{q}_0 := \mathbf{q}(0)$:

$$\begin{aligned}
&\textbf{for}\quad j := 1 \quad \textbf{to}\quad M \quad \textbf{do}\\
&\quad \mathbf{p}_{\mathrm{in}} := \mathbf{p}_{j-1}\,,\quad \mathbf{q}_{\mathrm{in}} := \mathbf{q}_{j-1}\\
&\quad \textbf{for}\quad k := 1 \quad \textbf{to}\quad s \quad \textbf{do}\\
&\qquad t_k := jh + h\sum_{\ell=1}^{k-1} a_\ell\,;\\
&\qquad \mathbf{p}_{\mathrm{out}} := \mathbf{p}_{\mathrm{in}} + h\,b_k\,\mathbf{f}(\mathbf{q}_{\mathrm{in}}, t_k)\,;\\
&\qquad \mathbf{q}_{\mathrm{out}} := \mathbf{q}_{\mathrm{in}} + h\,a_k\,\mathbf{g}(\mathbf{p}_{\mathrm{out}})\,;\\
&\qquad \mathbf{p}_{\mathrm{in}} := \mathbf{p}_{\mathrm{out}}\,,\quad \mathbf{q}_{\mathrm{in}} := \mathbf{q}_{\mathrm{out}}\,;\\
&\quad \textbf{end}\\
&\quad \mathbf{p}_j := \mathbf{p}_{\mathrm{out}}\,,\quad \mathbf{q}_j := \mathbf{q}_{\mathrm{out}}\,;\\
&\textbf{end}
\end{aligned} \tag{11.10.2}$$

Here $a_1,\ldots,a_s$ and $b_1,\ldots,b_s$ are given coefficients, carefully chosen such that the single-step method achieves a certain order. A possible choice discussed in [RWR04] is, for $s = 3$,

$$a_1 := \tfrac{2}{3}\,,\quad a_2 := -\tfrac{2}{3}\,,\quad a_3 := 1\,,\quad b_1 := \tfrac{7}{24}\quad b_2 := \tfrac{3}{4}\,,\quad b_3 := -\tfrac{1}{24}\,. \tag{11.10.3}$$

(11-10.a) ⚄ (10 min.)

Write a C++ function

```
void sympTimestep(double tau, Eigen::Vector2d& pq_j);
```

that implements a single timestep $\begin{bmatrix}\mathbf{p}_j\\ \mathbf{q}_j\end{bmatrix} \mapsto \begin{bmatrix}\mathbf{p}_{j+1}\\ \mathbf{q}_{j+1}\end{bmatrix}$ of the method (11.10.2) with $s = 3$ and coefficients according to (11.10.3) for the "harmonic oscillator" ODE ($n = 1$)

$$\dot{p} = q \quad,\quad \dot{q} = -p\,. \tag{11.10.4}$$

The components of the argument vector supply p_j and q_j, and will be replaced with p_{j+1} and q_{j+1}.

SOLUTION for (11-10.a) → 11-10-1-0:symp4.pdf ▲

(11-10.b) ⚁ (15 min.) [depends on Sub-problem (11-10.a)]

Experimentally determine the order of convergence of the single-step method implemented in Sub-problem (11-10.a) in terms of the number of timesteps by

- integrating (11.10.4) over the period $[0, 2\pi]$,
- with initial values $p(0) = 0$, $q(0) = 1$,
- and using $M = 10, 20, 40, 80, 160, 320, 640$ timesteps.

Tabulate the following errors at the final time:

$$\mathrm{err}(M) := |p(2\pi) - p^{(M)}| + |q(2\pi) - q^{(M)}| , \quad M = 10, 20, 40, 80, 160, 320, 640 , .$$

where $p^{(M)}$ and $q^{(M)}$ are the approximations of $p(2\pi)$ and $q(2\pi)$ produced by the single-step method. To that end complete the function `sympTimesteppingODETest()` in the file `symplectic.hpp`, which is called from `main()`.

HIDDEN HINT 1 for (11-10.b) → `11-10-2-0:smp5h1.pdf`

SOLUTION for (11-10.b) → `11-10-2-1:symp5.pdf` ▲

The symplectic timestepping method has especially been designed for **Hamiltonian ODEs**

$$\begin{aligned} \dot{\mathbf{p}}(t) &= -\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}(t), \mathbf{q}(t)) , \\ \dot{\mathbf{q}}(t) &= \frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}(t), \mathbf{q}(t)) \end{aligned} \quad \text{with} \quad H : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R} . \tag{11.10.7}$$

The smooth function H is called a **Hamiltonian**.

Notation: The partial derivative vector $\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}_0, \mathbf{q}_0) \in \mathbb{R}^n$ stands for the gradient of the function $\mathbf{q} \mapsto H(\mathbf{p}_0, \mathbf{q})$ in $\mathbf{q} = \mathbf{q}_0$. $\frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}_0, \mathbf{q}_0)$ is defined analogously.

(11-10.c) ⚂ (15 min.) Show that for every solution $t \mapsto (\mathbf{p}(t), \mathbf{q}(t))$, $t \in I \subset \mathbb{R}$, of (11.10.7) the function

$$t \mapsto E(t) := H(\mathbf{p}(t), \mathbf{q}(t)) , \quad t \in I , \tag{11.10.8}$$

is constant.

SOLUTION for (11-10.c) → `11-10-3-0:s6.pdf` ▲

(11-10.d) ⚁ (10 min.) Write down explicitly in the form (11.10.1) the Hamiltonian ODE (11.10.7) for the particular Hamiltonian

$$H(\mathbf{p}, \mathbf{q}) = \tfrac{1}{2}\|\mathbf{p}\|_2^2 + \|\mathbf{q}\|_2^4 , \quad \mathbf{p}, \mathbf{q} \in \mathbb{R}^n , \quad n \in \mathbb{N} . \tag{11.10.9}$$

SOLUTION for (11-10.d) → `11-10-4-0:s7.pdf` ▲

(11-10.e) ⚄ (20 min.) Implement a C++ function

```
Eigen::MatrixXd simulateHamiltonianDynamics(
  const Eigen::VectorXd &p0, const Eigen::VectorXd &q0,
  double T, unsigned int M);
```

that uses M equidistant steps of the sympletic timestepping scheme (11.10.2) + (11.10.3) in order solve the Hamilonian ODE (11.10.7) with Hamiltonian H as in (11.10.9) up to final time $T > 0$ and with initial values $\mathbf{p}_0$ and $\mathbf{q}_0$ passed in the argument vectors `p0` and `q0` of equal length $n \in \mathbb{N}$.

The function should return the numerical solution

$$\begin{bmatrix}\mathbf{p}_0\\ \mathbf{q}_0\end{bmatrix}, \begin{bmatrix}\mathbf{p}_1\\ \mathbf{q}_1\end{bmatrix}, \ldots, \begin{bmatrix}\mathbf{p}_M\\ \mathbf{q}_M\end{bmatrix},$$

in the columns of a $2n \times (M+1)$-matrix.

SOLUTION for (11-10.e) → `11-10-5-0:s8.pdf` ▲

(11-10.f) ⚄ (10 min.) The following table gives the quantities

$$\mathcal{E}(M) := \max\{E(\mathbf{p}_k, \mathbf{q}_k),\ k = 0, \ldots, M\}\,, \tag{11.10.12}$$

where E is the "energy" as defined in (11.10.8) and $\left(\begin{bmatrix}\mathbf{p}_k\\ \mathbf{q}_k\end{bmatrix}\right)_{k=0}^{M}$ is the sequence produced by `simulateHamiltonianDynamics()` for $n = 2$, $\mathbf{p}_0 = \mathbf{0}$, $\mathbf{q}_0 = \begin{bmatrix}1\\ 0\end{bmatrix}$, $T = 5$ and M timesteps.

M	$\mathcal{E}(M)$
10	1.136158638333363
20	1.009854310088357
40	1.001187287267320
80	1.000149549345033
160	1.000018794627444
320	1.000002358738605
640	1.000000295580299
1280	1.000000036994599

Based on the data in table describe qualitatively and quantitatively the expected asymptotic behavior of $\mathcal{E}(M)$ for $M \to \infty$.

SOLUTION for (11-10.f) → `11-10-6-0:s9.pdf` ▲

References

[RWR04] R. Rieben, D. White, and G. Rodrigue. "High Order Symplectic Integration Methods for Finite Element Solutions to Time Dependent Maxwell Equations". In: *IEEE Trans. Antennas and Propagation* 52.8 (2004), pp. 2190–2195 (cit. on p. 331).

End Problem 11-10 , 80 min.

Problem 11-11: RK-SSMs and Discrete Evolutions

This problem takes a look at the discrete evolution operators spawned by explicit Runge-Kutta single-step methods (RK-SSMs).

This problem relies on the material covered in [Lecture → Section 11.3] and [Lecture → Section 11.4].

The formulas for an explicit s-stage, $s \in \mathbb{N}$, Runge-Kutta single-step method described by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & 0 & \cdots & 0 \\ c_2 & a_{21} & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & & \ddots \\ c_s & a_{s1} & \cdots & a_{s,s-1} \quad 0 \\ \hline & b_1 & \cdots & b_{s-1} \quad b_s \end{array} \qquad \text{[Lecture → (11.4.0.13)]}$$

with $c_j, b_j, a_{i,j} \in \mathbb{R}$ are given in the following definition:

Definition [Lecture → Def. 11.4.0.11]. Explicit Runge-Kutta single-step method

For $b_i, a_{i,j} \in \mathbb{R}$, $c_i := \sum_{j=1}^{i-1} a_{ij}$, $i,j = 1,\ldots,s$, $s \in \mathbb{N}$, an **s-stage explicit Runge-Kutta single step method** (RK-SSM) for the ODE $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, $\mathbf{f}: \Omega \to \mathbb{R}^N$, is defined by ($\mathbf{y}_0 \in D$)

$$\mathbf{k}_i := \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + h\sum_{j=1}^{i-1} a_{i,j}\mathbf{k}_j)\,, \quad i = 1,\ldots,s \quad , \quad \mathbf{y}_1 := \mathbf{y}_0 + h\sum_{i=1}^{s} b_i \mathbf{k}_i\,.$$

The vectors $\mathbf{k}_i \in \mathbb{R}^N$, $i = 1,\ldots,s$, are called **increments**, $h > 0$ is the size of the timestep.

(11-11.a) ⚀ (10 min.) What conditions on the coefficients $c_j, b_j, a_{i,j} \in \mathbb{R}$ guarantee that the explicit Runge-Kutta single-step method according to the above definition is consistent with the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with Lipschitz continuous right-hand side function $\mathbf{f}$.

$c_j, b_j, a_{i,j}$ such that ⟹ RK-SSM consistent .

HIDDEN HINT 1 for (11-11.a) → 11-11-1-0:rkssmha1.pdf

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

(11-11.b) ⚀ (20 min.) The templated class **ExplRKSSMEvolOp** implements a functor type realizing the discrete evolution operator $(h,\mathbf{y}) \mapsto \Psi(h,\mathbf{y})$ for an explicit s-stage Runge-Kutta single-step method described by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$, $\mathbf{c},\mathbf{b} \in \mathbb{R}^s$, $\mathfrak{A} \in \mathbb{R}^{s,s}$ strictly lower triangular, applied to an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

```
template <typename RHSFunctor>
class ExplRKSSMEvolOp {
 public:
  ExplRKSSMEvolOp(RHSFunctor& f, const Eigen::MatrixXd& A,
```

```
                    const Eigen::VectorXd& b)
      : f_(f), A_(A), b_(b) {}

  template <typename State>
  State operator()(double h, const State& y) const;

 private:
  RHSFunctor& f_;       // Right-hand-side function f
  Eigen::MatrixXd A_;   // Butcher matrix 𝔄
  Eigen::VectorXd b_;   // Weight vector b
};
```

The constructor takes a functor object encoding the right-hand-side function $\mathbf{f}$ and $\mathfrak{A}$ and $\mathbf{b}$ as input arguments and stores them in class-local member variables.

As stated above, the evaluation operator **operator** `()` is supposed to realize the discrete evolution operator $\Psi(h,\mathbf{y})$ for the RK-SSM applied to $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$. Complete the implementation of Code 11.11.4 so that it meets this specification.

C++ code 11.11.4: Implementation of evaluation operator for ExplRKSSMEvolOp

```
1  template <typename RHSFunctor>
2  template <typename State>
3  State ExplRKSSMEvolOp<RHSFunctor>::operator()(double h, const State& y) const {
4    unsigned int s = A_.cols();
5    State y1{y};
6    std::vector<State> k{s, y};
7    for (int i = 0; i < [      ]; ++i) {
8      for (int j = 0; j < [      ]; ++j) {
9        k[[      ]] += h * A_([      ], [      ]) * k[[      ]];
10     }
11     k[[      ]] = f_([      ]);
12     y1 += h * b_[[      ]] * k[[      ]];
13   }
14   return y1;
15 }
```

HIDDEN HINT 1 for (11-11.b) → 11-11-2-0:rkssmhb1.pdf

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

End Problem 11-11 , 30 min.

Problem 11-12: Leapfrog Timestepping for Oscillators

Oscillating phenomena are often modelled by means of second-order ordinary differential equations, the most famous example being the harmonic oscillation arising as solution of the scalar ODE $\ddot{y} = -y$. In this problem we study a particularly simple method for more general "oscillatory" second-order ODE, which has amazing long-time stability properties.

This problem requires some familiarity with [Lecture → Section 11.3].

For a physical system with state space $\mathbb{R}^{2N}$, $N \in \mathbb{N}$, we know a potential energy function $V : \mathbb{R}^N \to \mathbb{R}$, which is supposed to be smooth and convex. The evolution of the system is governed by the first-order autonomous initial-value problem

$$\left\{ \begin{array}{rcl} \dot{\mathbf{y}} &=& \mathbf{v}\,, \\ \dot{\mathbf{v}} &=& -\mathbf{grad}\, V(\mathbf{y})\,, \end{array} \right. \qquad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^N\,, \quad \mathbf{v}(0) = \mathbf{v}_0 \in \mathbb{R}^N\,. \tag{11.12.1}$$

with given initial state $\left[\begin{smallmatrix} \mathbf{y}_0 \\ \mathbf{v}_0 \end{smallmatrix}\right] \in \mathbb{R}^{2N}$.

(11-12.a) ⚀ (5 min.) Formulate an initial value problem for a *second-order* ODE, whose solution agrees with the $\mathbf{y}$-component of the solution of (11.12.1).

$$\ddot{\mathbf{y}} = \qquad\qquad , \qquad\qquad .$$

SOLUTION for (11-12.a) → 11-12-1-0:lfas.pdf ▲

(11-12.b) ⚁ (10 min.) Show that for the solution $t \mapsto \left[\begin{smallmatrix} \mathbf{y}(t) \\ \mathbf{v}(t) \end{smallmatrix}\right]$ of (11.12.1) the total energy $E(t) := V(\mathbf{y}(t)) + \frac{1}{2}\|\mathbf{v}(t)\|_2^2$ is constant.

SOLUTION for (11-12.b) → 11-12-2-0:lfbs.pdf ▲

The so-called **leapfrog timestepping** for (11.12.1) is defined by the recursions

$$\left\{ \begin{array}{rcl} \dfrac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h} &=& \mathbf{v}_{k+\frac{1}{2}}\,, \\ \dfrac{\mathbf{v}_{k+\frac{1}{2}} - \mathbf{v}_{k-\frac{1}{2}}}{h} &=& -\mathbf{grad}\, V(\mathbf{y}_k)\,, \end{array} \right. \quad \text{and} \quad \tfrac{1}{2}(\mathbf{v}_{\frac{1}{2}} + \mathbf{v}_{-\frac{1}{2}}) = \mathbf{v}_0\,, \tag{11.12.2}$$

where $h > 0$ is a fixed uniform timestep. It generates sequences $(\mathbf{y}_k)_{k\in\mathbb{N}_0}$ and $\left(\mathbf{v}_{k+\frac{1}{2}}\right)_{k\in\mathbb{N}_0}$, whose terms approximate the solution trajectories in the sense that $\mathbf{y}_k \approx \mathbf{y}(kh)$, $\mathbf{v}_{k+\frac{1}{2}} \approx \mathbf{v}((k+\frac{1}{2})h)$.

(11-12.c) ⚂ (21 min.) In the file `leapfrog.hpp` implement an *efficient* (also in terms of memory usage) C++ function

```
template <typename GVFUNCTOR>
std::pair<Eigen::VectorXd, Eigen::VectorXd>
leapFrog(GVFUNCTOR &&gradV, const Eigen::VectorXd &y0,
         const Eigen::VectorXd &v0, double T, unsigned int M);
```

that carries out M uniform timesteps of the leapfrog timestepping method (11.12.2) for (11.12.1) such that $y_M \approx y(T)$, $T > 0$ the final time. The function $\mathbf{y} \mapsto \mathbf{grad}\, V(\mathbf{y})$ is supplied via the argument `gradV` (assumed compatible with `std::function<Eigen::VectorXd(const Eigen::VectorXd &)>`), the initial states through `y0` and `v0`, the final time in `T`, and the number of timesteps in `M`.

The function should return $(\mathbf{y}_M, \mathbf{v}_M)$ with $\mathbf{v}_M := \frac{1}{2}(\mathbf{v}_{M-\frac{1}{2}} + \mathbf{v}_{M+\frac{1}{2}})$.

SOLUTION for (11-12.c) → 11-12-3-0:lfcs.pdf ▲

(11-12.d) ⚄ (21 min.) Complete the code of the C++ function (in the file `leapfrog.hpp`)

```
template <typename GVFUNCTOR>
void testCVGLeapfrog(GVFUNCTOR &&gradV, const Eigen::VectorXd &y0,
                     const Eigen::VectorXd &v0, double T);
```

that prints to `stdout` a *table* of values, from which one can see the rate of algebraic convergence of the error at final time of the leapfrog method at a glance. To compute a reference solution run leapfrog with a significantly larger ($\sim 10\times$) number of timesteps than in all "error-recording" runs.

This function is called from `numExpCvgLeapfrog()` already implemented in `leapfrog.hpp`, which runs it for $N = 1$, $V(y) = \frac{1}{4}y^4$, $y_0 = 1$, $v_0 = 0$, $T = 10$. Running the code will execute `numExpCvgLeapfrog()` and create the table.

What rate of algebraic convergence of the error at final time do your observe?

rate = [ ] .

SOLUTION for (11-12.d) → 11-12-4-0:lfxs.pdf ▲

End Problem 11-12, 57 min.

Problem 11-13: Williamson (2N) Low-Storage Explicit Runge-Kutta Method

Functions that carry out one step of a generic explicit s-stage, $s \in \mathbb{N}$, Runge-Kutta single-step method (RK-SSM) according to [Lecture → Def. 11.4.0.11] for an ODE with state space $\mathbb{R}^N$, $N \in \mathbb{N}$, need to store s vectors of length N, which hold the increments or the stages. If $N \gg 1$, this may exhaust main memory or prevent in-cache execution. Therefore, explicit RK-SSMs whose implementation relies on only two auxiliary vectors have been proposed as "low-storage" Runge-Kutta single-step methods. A particular class will be studied in this problem.

Requires familiarity with explicit RK-SSMs as introduced in [Lecture → Section 11.4]. C++ implementation based on EIGEN is requested.

According to [Ket10] a function realizing a single step (step size $h > 0$, computing $\mathbf{y}_1$ from $\mathbf{y}_0$) of an s-stage **Williamson (2N) explicit single-step method** for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subset \mathbb{R}^N \to \mathbb{R}^N$, can described by the following pseudo-code:

Pseudo-code 11.13.1: Pseudo-code for general Wiliamson (2N) single-step method.

```
VECTOR Williamson2NStep(integer s; VECTOR y_0; real h) {
  VECTOR g := y_0;
  VECTOR p := 0;
  for(i=1; i<= s, ++i) {
    p := α_i p + h f(g);
    g := g + β_i p;
  }
  return g;
}
```

In Code 11.13.1 α_i and β_i are given real coefficients.

(11-13.a) ⚀ (25 min.) All s-stage Williamson (2N) single-step methods are explicit s-stage Runge-Kutta single step methods according to [Lecture → Def. 11.4.0.11].

For $s = 2$, given the coefficients α_2, β_1, and β_2 for a 2-stage Williamson (2N) single-step method, write down the Butcher scheme describing the equivalent 2-stage RK-SSM:

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^\top \end{array} \quad = \quad \begin{array}{c|cc} c_1 & & \\ c_2 & & \\ \hline & & \end{array} \; .$$

HIDDEN HINT 1 for (11-13.a) → 11-13-1-0:w2nh1.pdf

SOLUTION for (11-13.a) → 11-13-1-1:w2ns1.pdf ▲

(11-13.b) ⚁ (15 min.) State sufficient and necessary conditions for the coefficients α_i and β_i, $i = 1, 2$, such that the corresponding *2-stage* Williamson (2N) single-step will exactly solve the scalar "initial value problem" $\dot{y} = 1$, $y(0) = 0$, for any choice of timesteps.

By "solving exactly" we mean that $y_1 = y(h)$, where $t \mapsto y(t)$ is the exact solution of the IVP and y_1 the approximation of $y(h)$ produced by the method defined through Code 11.13.1.

SOLUTION for (11-13.b) → 11-13-2-0:wsns2.pdf ▲

An implementation of this method for solving an initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$ with equidistant timesteps, is given in Code 11.13.6.

C++ code 11.13.6: Williamson (2N) solver for IVP

```
template <typename STATE, typename FUNCTION_F>
std::vector<std::pair<double, STATE>> solveWilliamsonIVP(
    std::vector<double> alpha, std::vector<double> beta, FUNCTION_F &&f,
    double T, STATE y0, unsigned int M) {

  const size_t s = alpha.size(); // Number of stages
  assert((s == beta.size()) && ("size mismath of coefficient vectors"));
  const double h = T / M;  // timestep size
  STATE g{y0};             // Internal state variable
  // Sequence of approximate (extended) states (t_k, y_k)
  std::vector<std::pair<double, STATE>> y_seq{{0.0, y0}};
  // Main timestepping loop:  M timesteps
  for (unsigned int k = 0; k < M; ++k) {
    // Single step of Williamson (2N) method
    STATE p{h * f(g)};
    g += beta[0] * p;
    for (unsigned int i = 1; i < s; ++i) {
      p = alpha[i] * p + h * f(g);
      g += beta[i] * p;
    }
    // Store t_k and approximate state y_k.
    y_seq.emplace_back((k + 1) * h, g);
  }
  return y_seq;
}
```

The **STATE** type must supply elementary vector operations, and a **FUNCTION_F** object must be a valid functor mapping `STATE` → `STATE`. The arguments of `solveWilliamsonIVP` are

- `alpha`, `beta`, passing the coefficients α_i, β_i,
- `f`, a functor incarnating the right-hand side function $\mathbf{f}$,
- `T`, the final time $T > 0$,
- `y0`, the initial value $\mathbf{y}_0$,
- `M`, the number of timesteps to be carried out.

(11-13.c) ⚀ (15 min.) In the file `williamson2nmethod.hpp` complete the implementation of the function

```
void tabulateErrW2N(
  std::vector<unsigned int> &&M_vec,
  std::vector<double> alpha, std::vector<double> beta);
```

that is supposed to call `solveWilliamsonIVP()` from Code 11.13.6 for the scalar initial value problem $\dot{y} = \cos^2 y$, $y(0) = 0$, with exact solution $y(t) = \arctan(t)$ on the time interval $[0, 2]$, and then print a table of the number M of timesteps versus the error

$$\operatorname{err}(M) := \max_{i=0,\ldots,M} |y(ih) - y_i| \,, \quad h := 2/M \,. \tag{11.13.7}$$

The arguments `alpha`, `beta` pass the coefficients characterizing the Williamson (2N) method, while `M_vec` is a sequence of numbers of timesteps to be used.

HIDDEN HINT 1 for (11-13.c) → `11-13-3-0:stdmax.pdf`

SOLUTION for (11-13.c) → `11-13-3-1:w2nss3.pdf` ▲

(11-13.d) ⚀ (10 min.) In [Car94] we find the following coefficients for a 4-stage Williamson (2N) method

i	1	2	3	4
α_i	0	$-\frac{756391}{934407}$	$-\frac{36441873}{15625000}$	$-\frac{1953125}{1085297}$
β_i	$\frac{8}{141}$	$\frac{6627}{2000}$	$\frac{609375}{1085297}$	$\frac{198961}{526383}$

M	error
10	2.447815e-04
20	2.944395e-05
40	3.627954e-06
80	4.506308e-07
160	5.616633e-08
320	7.010145e-09
640	8.756145e-10

Examining this method by means of `tabulateErrW2N()` from Sub-problem (11-13.c) produces the output on the left.

Describe qualitatively and quantitatively the asymptotic behavior of the error (11.13.7):

$\mathrm{err}(M)$ converges ☐ in rate $O($☐$)$

in terms of the timestep size $h := T/M$ tending to zero.

SOLUTION for (11-13.d) → `11-13-4-0:w2ns4.pdf` ▲

(11-13.e) ⚄ (40 min.) In `williamson2nmethod.hpp` you find the C++ function

```
Eigen::VectorXd integrateAPsi(
  std::vector<double> alpha,
  std::vector<double> beta,
  const Eigen::SparseMatrix<double> &A,
  std::function<double(double)> &psi, double T,
  const Eigen::VectorXd &y0, unsigned int M);
```

which applies an Williamson (2N) single-step method for the numerical solution of an initial-value problem for the ODE

$$\dot{\mathbf{y}} = \mathbf{A}\mathbf{y} + [\psi((\mathbf{y})_i)]_{i=1}^N, \quad \mathbf{A} \in \mathbb{R}^{N,N} \text{ large, sparse}, \quad \psi : \mathbb{R} \to \mathbb{R}.$$

The function ψ (which applies to state vector $\mathbf{y}$ entrywise) is passed through the functor argument `psi`, the matrix $\mathbf{A}$ through `A` and the initial value through `y0`. All other arguments are the same as those with the same name for `solveWilliamsonIVP()` listed in Code 11.13.6.

C++ code 11.13.9: Code for function `integrateAPsi()`

```
2 Eigen::VectorXd integrateAPsi(std::vector<double> alpha,
3                               std::vector<double> beta,
4                               const Eigen::SparseMatrix<double> &A,
5                               std::function<double(double)> &psi, double T,
6                               const Eigen::VectorXd &y0, unsigned int M) {
7   const Eigen::Index N = A.cols();
8   assert((N == A.rows()) && ("A must be square"));
9   assert((N == y0.size()) && ("Wrong size of y0"));
```

```
10
11   const size_t s = alpha.size(); // Number of stages
12   assert((s == beta.size()) && ("size mismath of coefficient vectors"));
13   const double h = T / M;  // timestep size
14   Eigen::VectorXd g{y0};   // Internal state variable
15   Eigen::VectorXd p(N);    // Auxiliary vector
16
17   // Right-hand side function for the ODE
18   auto rhs_f = [&A, &psi, N](const Eigen::VectorXd &y) -> Eigen::VectorXd {
19     Eigen::VectorXd tmp{A * y};
20     for (Eigen::Index i = 0; i < N; ++i) {
21       tmp[i] += psi(y[i]);
22     }
23     return tmp;
24   };
25   // Main timestepping loop:  M timesteps
26   for (unsigned int k = 0; k < M; ++k) {
27     // Single step of Williamson (2N) method
28     p = h * rhs_f(g);
29     g += beta[0] * p;
30     for (unsigned int i = 1; i < s; ++i) {
31       p = alpha[i] * p + h * rhs_f(g);
32       g += beta[i] * p;
33     }
34   }
35   return g;
36 }
```

Now, we want to avoid memory allocation for temporary EIGEN vector when calling `rhs_f` (highlighted in green color in Code 11.13.9).

Task: You should complete the implementation of the lambda function `rhs_f` in the C++ function `integrateAPsi_lowmem()` in the file `williamson2nmethod.hpp` such that it

1. behaves exactly like `integrateAPsi()`,
2. but *is guaranteed not to allocate a single vector* (even not temporary ones "behind the scene" through EIGEN's expression template mechanism) in the part of the code that you add!

Please notice that in `integrateAPsi_lowmem()` the signature of `rhs_f` has changed. The timestepping has also changed accordingly.

HIDDEN HINT 1 for (11-13.e) → `11-13-5-0:w2nhy.pdf`

SOLUTION for (11-13.e) → `11-13-5-1:w2nss5.pdf` ▲

(11-13.f) ⚀ (25 min.) A single step of a Williamson (2N) single-step method as defined by Code 11.13.1 amounts to the application of a discrete evolution operator $\Psi : \mathbb{R}^+ \times D \to \mathbb{R}^N$ with $\mathbf{y}_1 = \Psi(h, \mathbf{y}_0)$.

When applying a Williamson (2N) single step method to the scalar linear ODE $\dot{y} = \lambda y$, $\lambda \in \mathbb{C}$, then we find for all $h > 0$ and $y \in \mathbb{C}$ that

$$\Psi(h, y) = S(h\lambda)y \quad \text{with some } \textbf{stability function} \quad S : \mathbb{C} \to \mathbb{C} .$$

In the file `williamson2nmethod.hpp` complete the code of the C++ function

```
std::complex<double> stabfnWilliamson(
  std::vector<double> alpha,
```

```
std::vector<double> beta,
std::complex<double> z);
```

that evaluates the stability function $z \mapsto S(z)$ for the Williamson (2N) single-step method defined by the coefficient sequences `alpha` and `beta`.

SOLUTION for (11-13.f) → `11-13-6-0:w2nss6.pdf` ▲

References

[Car94] M. Carpenter. *Third-order 2N-storage Runge-Kutta schemes with error control*. Technical Report 19940029698. NASA, June 1994 (cit. on p. 340).

[Ket10] David I. Ketcheson. "Runge-Kutta methods with minimum storage implementations". In: *J. Comput. Phys.* 229.5 (2010), pp. 1763–1773. DOI: 10.1016/j.jcp.2009.11.006 (cit. on p. 338).

End Problem 11-13 , 130 min.

Chapter 12

Single Step Methods for Stiff Initial Value Problems

Problem 12-1: Implicit Runge-Kutta method

This problem addresses the implementation of general implicit Runge-Kutta methods [Lecture → Def. 12.3.3.1]. We will adapt an integrator class so far available for explicit Runge-Kutta single-step methods to the implicit case.

Related to Problem 11-2. This problem assumes familiarity with [Lecture → Section 12.3], and, especially, [Lecture → Section 12.3.3] and [Lecture → Rem. 12.3.3.9]. It practices the implementation of a full implicit RK-SSM in C++ based on EIGEN.

Problem 11-2 introduced the class **RKIntegrator** that implemented the timestepping for a general explicit Runge-Kutta method according to [Lecture → Def. 11.4.0.11]. Keeping the interface we now extend this class so that it realizes a general Runge-Kutta timestepping method as given in [Lecture → Def. 12.3.3.1] for solving an autonomous initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

(12-1.a) ⚀ (15 min.) Rederive the **stage form**

$$\begin{aligned} \mathbf{g}_i &= h \sum_{j=1}^{s} a_{ij} \mathbf{f}(t_0 + c_j h, \mathbf{y}_0 + \mathbf{g}_j) \,, \\ \mathbf{y}_1 &= \mathbf{y}_0 + h \sum_{i=1}^{s} b_i \mathbf{f}(t_0 + c_i h, \mathbf{y}_0 + \mathbf{g}_i) \,, \end{aligned} \qquad \text{[Lecture → (12.3.3.8)]}$$

of a general (implicit) Runge-Kutta method from the increment equations as given in [Lecture → Def. 12.3.3.1].

SOLUTION for (12-1.a) → `12-1-1-0:0s.pdf` ▲

(12-1.b) ⚁ (30 min.) Let $\mathbf{g}_i$, $i = 1, \ldots, s$, be the so-called stages of a general Runge-Kutta method as defined in [Lecture → (12.3.3.7)]. In [Lecture → Rem. 12.3.3.9] the stages are recovered by solving a non-linear system of equations $F(\mathfrak{g}) = \mathbf{0}$ with a suitable function $F : \mathbb{R}^{sN} \to \mathbb{R}^{sN}$. Formulate the Newton iteration for it when only autonomous ODEs $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with differentiable right-hand-side functions $\mathbf{f} : D \subset \mathbb{R}^N \to \mathbb{R}^N$ are considered.

SOLUTION for (12-1.b) → `12-1-2-0:0as.pdf` ▲

(12-1.c) ⚂ (60 min.) [depends on Sub-problem (12-1.b)]

By modifying the class **RKIntegrator** for the implementation of explicit Runge-Kutta methods, design in `implicit_rkintegrator.hpp` a similar header-only C++ class **implicitRKIntegrator** which implements a general implicit RK method given through a Butcher scheme [Lecture → (12.3.3.3)] to solve the autonomous initial value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

```
class implicitRKIntegrator {
 public:
  implicitRKIntegrator(const Eigen::MatrixXd &A, const
     Eigen::VectorXd &b);
  template <class Function, class Jacobian>
  std::vector<Eigen::VectorXd> solve(Function &&f, Jacobian &&Jf,
                                     double T,
                                    const Eigen::VectorXd &y0,
                                    unsigned int N) const;
 private:
  .......
};
```

This class should implement a generic implicit Runge-Kutta single-step method for an autonomous ODE according to [Lecture → Def. 12.3.3.1]. The method is specified through its Butcher matrix $\mathcal{A} \in \mathbb{R}^{s,s}$ and the weight vector $\mathbf{b} \in \mathbb{R}^s$ that are passed to the constructor as arguments `A` and `b`.

The `solve()` method carries out N equidistant timesteps of the method for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on the time interval $[0, T]$, $T > 0$, and with initial value $\mathbf{y}_0$. The right-hand side function $\mathbf{f}$ and its Jacobian $D\mathbf{f}$ are passed through the functor objects `f` and `Jf` equipped with suitable evaluation operators.

In the `solve()` method the stages $\mathbf{g}_i$ as introduced in ((12-1.a)) are to be computed with the **damped Newton method** (see [Lecture → Section 8.5.4]) applied to the nonlinear system of equations satisfied by the stages (see [Lecture → Rem. 12.3.3.6] and [Lecture → Rem. 12.3.3.9]). Use the provided function

```
template <typename FuncType, typename JacType>
void dampnewton(const FuncType &F, const JacType &DF,
                Eigen::VectorXd &x,
                double rtol = 1e-4,
                double atol = 1e-6);
```

from the file `dampnewton.hpp`, that is a simplified version of [Lecture → Code 8.5.4.5]. Note that we do not use the simplified Newton method as discussed in [Lecture → Rem. 12.3.3.9].

In the code template you will find large parts of **implicitRKIntegrator** already implemented. In fact, you only have to write the method `step()` for the actual implicit RK-SSM timestepping.

SOLUTION for (12-1.c) → `12-1-3-0:Impl1h.pdf` ▲

(12-1.d) ⚀ (15 min.) Examine the following code

C++11-code 12.1.5: Initialization of implicitRKIntegrator object

```
2  // Definition of coefficients in Butcher scheme
3  constexpr unsigned int s = 2;
4  Eigen::MatrixXd A(s, s);
5  Eigen::VectorXd b(s);
6  // What method is this?
7  A << 5. / 12., -1. / 12., 3. / 4., 1. / 4.;
8  b << 3. / 4., 1. / 4.;
9  // Initialize implicit RK with Butcher scheme
```

```
10 implicitRKIntegrator RK(A, b);
```

Get it on GitLab (main.cpp).

Write down the complete Butcher scheme according to [Lecture → (12.3.3.3)] for the implicit Runge-Kutta method now available through `RK`. Which method is it? Is it **A-stable** [Lecture → Def. 12.3.4.9], **L-stable** [Lecture → Def. 12.3.4.15]?

HIDDEN HINT 1 for (12-1.d) → 12-1-4-0:Impl1s.pdf ▲

SOLUTION for (12-1.d) → 12-1-4-1:Impl2h.pdf

(12-1.e) ⚄ (15 min.) [depends on Sub-problem (12-1.e)]

The file `main.cpp` uses your implementation in **implicitRKIntegrator** of general implicit RK-SSMs to solve an initial value problem for the Lotka-Volterra ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_1(\alpha_1 - \beta y_2) \\ y_2(-\alpha_2 + \beta y_2) \end{bmatrix}. \tag{12.1.6}$$

with the particular paramters $\alpha_1 = 3$, $\alpha_2 = 2$, $\beta = 0.1$. The right-hand-side function and its Jacobian are implemented as suitable lambda functions. Initial state is $\mathbf{y}_0 = [100\ 5]^\top$ and final time $T = 10$. The norm of the error at final time is tabulated.

Run the code. What information can be gleaned from the table?

SOLUTION for (12-1.e) → 12-1-5-0:Impl3h.pdf ▲

End Problem 12-1, 135 min.

Problem 12-2: Damped precession of a magnetic needle

This problem deals with a dynamical system from mechanics describing the movement of a rod-like magnet in a strong magnetic field. This can be modelled by an ODE with a particular invariant that can become stiff in the case of large friction.

Assumes an idea about the notion of stiffness [Lecture → Notion 12.2.0.7] and knowledge of heuristic criteria for predicting stiffness of an IVP [Lecture → § 12.2.0.18]. Also looks at simple implicit and semi-implicit [Lecture → Section 12.4] single-step methods for a concrete ODE.

We consider the initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c\mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1,1,1]^\top , \tag{12.2.1}$$

where $c > 0$ and $\mathbf{a} \in \mathbb{R}^3$, $\|\mathbf{a}\|_2 = 1$.

Note: $\mathbf{x} \times \mathbf{y}$ denotes the **cross product** of the two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^3$. It is defined by

$$\mathbf{x} \times \mathbf{y} = \begin{bmatrix} x_2 y_3 - x_3 y_2 \\ x_3 y_1 - x_1 y_3 \\ x_1 y_2 - x_2 y_1 \end{bmatrix} .$$

It satisfies $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$. In Eigen, it is available as `x.cross(y)` → EIGEN documentation, **#include** `<Eigen/Geometry>` is required.

(12-2.a) ⚀ (10 min.) Show that $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}_0\|_2$ for every solution $\mathbf{y}$ of the ODE.

HIDDEN HINT 1 for (12-2.a) → `12-2-1-0:Cros1s.pdf`

SOLUTION for (12-2.a) → `12-2-1-1:Cros1h.pdf` ▲

(12-2.b) ⚁ (20 min.) Compute the Jacobian $D\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{3,3}$ and its spectrum (= set of eigenvalues) $\sigma(D\mathbf{f}(\mathbf{y}))$ in the stationary state $\mathbf{y} = \mathbf{a}$, for which $\mathbf{f}(\mathbf{y}) = 0$. For simplicity, you may consider only the case $\mathbf{a} = [1,0,0]^\top$.

SOLUTION for (12-2.b) → `12-2-2-0:Cros2h.pdf` ▲

(12-2.c) ⚀ (10 min.) [depends on Sub-problem (12-2.b)]

Discuss whether and when IVP for (12.2.1) will be stiff in the case $\mathbf{y}_0 \approx \mathbf{a}$.

HIDDEN HINT 1 for (12-2.c) → `12-2-3-0:stiffcrit.pdf`

SOLUTION for (12-2.c) → `12-2-3-1:s22.pdf` ▲

(12-2.d) ⚀ (10 min.) [depends on Sub-problem (12-2.b)]

For $\mathbf{a} = [1,0,0]^\top$, (12.2.1) was solved with the standard explicit adaptive Runge-Kutta numerical integrator **Ode45** and another A-stable semi-implicit adaptive numerical integrator up to the point $T = 10$ (same tolerances for the stepsize control). Explain the different dependence of the total number of steps from the parameter c observed in the figure.

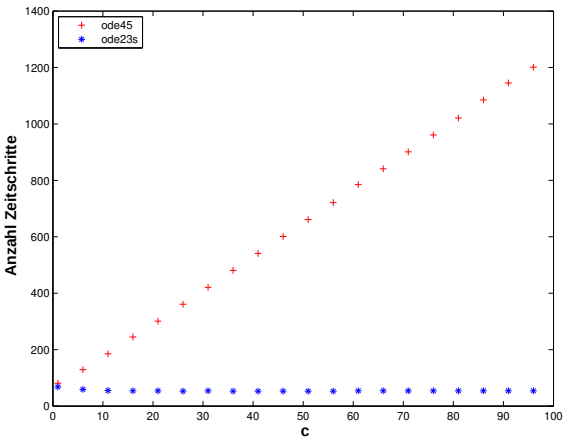


Fig. 119

SOLUTION for (12-2.d) → 12-2-4-0:Cros3h.pdf ▲

(12-2.e) ⚀ (10 min.) Formulate the non-linear equation resulting from the implicit mid-point method from [Lecture → Section 11.2.3] for the initial value problem (12.2.1).

SOLUTION for (12-2.e) → 12-2-5-0:Cros4h.pdf ▲

(12-2.f) ⚁ (20 min.) In `cross.hpp`, implement the C++ function

```
void tab_crossprod();
```

which solves (12.2.1) with $\mathbf{a} = [1,0,0]^\top$, $c = 1$, up to $T = 10$, using $M = 128$ uniform time steps of the implicit mid-point method, and tabulates $\|\mathbf{y}_k\|_2$ for the generated sequence of approximate states. What do you observe?

For this task rely on the helper class **implicitRKIntegrator**:

```
class implicitRKIntegrator {
public:
  implicitRKIntegrator(
    const Eigen::MatrixXd &A,
    const Eigen::VectorXd &b);
  template <class Function, class Jacobian>
  std::vector<Eigen::VectorXd> solve(
    Function &&f, Jacobian &&Jf, double T,
    const Eigen::VectorXd &y0,
    unsigned int M) const;
private:
  .......
};
```

It implements a generic implicit Runge-Kutta single-step method for an autonomous ODE according to [Lecture → Def. 12.3.3.1]. The method is specified through its Butcher matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the weight vector $\mathbf{b} \in \mathbb{R}^s$ that are passed to the constructor as arguments `A` and `b`.

The `solve()` method carries out M equidistant timesteps of the method for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on the time interval $[0,T]$, $T > 0$, and with initial value $\mathbf{y}_0$. The right-hand side function $\mathbf{f}$ and its Jacobian $D\mathbf{f}$ are passed through the functor objects `f` and `J` equipped with suitable evaluation operators. The `solve()` method relies on the damped Newton iteration to solve the stage equations, see [Lecture → Rem. 12.3.3.6] and [Lecture → Rem. 12.3.3.9].

SOLUTION for (12-2.f) → 12-2-6-0:Cros5h.pdf ▲

(12-2.g) ⚁ (20 min.) The so-called **linear-implicit mid-point method** can be obtained by a simple *linearization* of the (single) increment equation of the implicit mid-point method around the current state.

Equivalently, we can obtain it from the standard implicit midpoint method by carrying out a single step of the Newton iteration for the increment equations with initial guess $\mathbf{0}$.

Give the defining equation of the linear-implicit mid-point method for the general autonomous differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$$

with smooth right-hand-side function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$, $N \in \mathbb{N}$.

HIDDEN HINT 1 for (12-2.g) → 12-2-7-0:linhi.pdf

SOLUTION for (12-2.g) → 12-2-7-1:Cros6h.pdf ▲

(12-2.h) ⚂ (20 min.) [depends on Sub-problem (12-2.g)]

Implement the linear-implicit midpoint method in the C++ function

```
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solve_lin_mid(
  Function &&f, Jacobian &&Jf,
  double T, const Eigen::VectorXd &y0,
  unsigned int M);
```

The signature of this function is the same as that of the `solve()` method of **implicitRKIntegrator**.

Extend your implementation of `tab_crossprod()` so that it uses `solve_lin_mid()` to solve (12.2.1) with $\mathbf{a} = [1,0,0]^\top$, $c = 1$ up to $T = 10$ and $M = 128$. Tabulate $\|\mathbf{y}_k\|_2$ for the sequence of approximate states generated by the linear implicit midpoint method. What do you observe?

SOLUTION for (12-2.h) → `12-2-8-0:Cros7h.pdf` ▲

End Problem 12-2 , 120 min.

Problem 12-3: Singly Diagonally Implicit Runge-Kutta Method

SDIRK methods (**S**ingly **D**iagonally **I**mplicit **R**unge-**K**utta methods) are distinguished by Butcher schemes with a lower triangular coefficient matrix $\mathfrak{A}$ whose diagonal entries are all the same:

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} = \begin{array}{c|ccccc} c_1 & \gamma & & \cdots & & 0 \\ c_2 & a_{21} & \ddots & & & \vdots \\ \vdots & \vdots & & & & \vdots \\ \vdots & \vdots & & \ddots & & \vdots \\ \vdots & \vdots & & & \ddots & \vdots \\ c_s & a_{s1} & & \cdots & a_{s,s-1} & \gamma \\ \hline & b_1 & & \cdots & b_{s-1} & b_s \end{array} , \tag{12.3.1}$$

with $\gamma \neq 0$. Their advantage is that systems of equations of the same structure have to be solved in order to compute the s increments sequentially.

Familiarity with [Lecture → Section 12.3.3] is required. This problem has a focus on stability theory.

In this problem, the scalar linear initial value problem of second order

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{12.3.2}$$

should be solved numerically using a family of 2-stage SDIRK method described by the Butcher scheme

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array} , \tag{12.3.3}$$

where $\gamma > 0$ is a parameter.

(12-3.a) ⚀ (5 min.) Assume you want to solve an initial-value problem for an ordinary differential equation $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$, $\mathbf{f} : I \times D \subset \mathbb{R} \times \mathbb{R}^N \to \mathbb{R} \times \mathbb{R}^N$, numerically. Explain the benefit of using SDIRK-SSMs compared to using Gauss-Radau RK-SSMs as introduced in [Lecture → Ex. 12.3.4.21]. In what situations will this benefit matter much?

HIDDEN HINT 1 for (12-3.a) → 12-3-1-0:SDIR0h.pdf

SOLUTION for (12-3.a) → 12-3-1-1:SDIR0s.pdf ▲

(12-3.b) ⚁ (10 min.) State the equations for the increments $\mathbf{k}_1$ and $\mathbf{k}_2$ of the Runge-Kutta method (12.3.3) applied to the initial value problem corresponding to the differential equation $\dot{\boldsymbol{y}} = \mathbf{f}(t, \boldsymbol{y})$.

SOLUTION for (12-3.b) → 12-3-2-0:SDIR1s.pdf ▲

(12-3.c) ⚁ (20 min.) Show that the stability function $S(z)$ of the SDIRK method (12.3.3) is given by

$$S(z) = \frac{1 + z(1-2\gamma) + z^2(1/2 - 2\gamma + \gamma^2)}{(1-\gamma z)^2}$$

and plot the stability domain $\mathcal{S}_\Psi$ for the parameter value $\gamma = 1$ using the supplied PYTHON script `stabdomSDIRK.py`

SOLUTION for (12-3.c) → 12-3-3-0:SDIR2s.pdf ▲

(12-3.d) ⚂ (25 min.) Find out whether for $\gamma = 1$ the SDIRK RK-SSM (12.3.3) is

- A–stable,

- L–stable.

Argue, based on the following version of the maximum principle for analytic functions:

Lemma 12.3.4. Maximum principle for rational functions

*If a rational function $z \in \mathbb{C} \mapsto R(z)$ has no pole in the left half plane $\mathbb{C}^- := \{z \in \mathbb{C} : \operatorname{Re} z \leq 0\}$, then **either***

$$\max\{|R(z)| : z \in \mathbb{C}^-\} = \max\{|R(z)| : \operatorname{Re} z = 0\}$$

or

$$\min\{|R(z)| : z \in \mathbb{C}^-\} = \min\{|R(z)| : \operatorname{Re} z = 0\} ,$$

that is, the function attains on the imaginary axis either its maximum or its minimum over the whole left half plane.

SOLUTION for (12-3.d) → 12-3-4-0:SDIRxs.pdf ▲

(12-3.e) ⚀ (10 min.) Formulate (12.3.2) as an initial value problem for a linear first order system for the function $\boldsymbol{z}(t) = (y(t), \dot{y}(t))^\top$.

SOLUTION for (12-3.e) → 12-3-5-0:SDIR3s.pdf ▲

(12-3.f) ⚁ (20 min.) [depends on Sub-problem (12-3.e)]

Implement a C++ function

```
Eigen::Vector2d SdirkStep(const Eigen::Vector2d &z0, double h,
   double gamma);
```

that realizes one step of the method (12.3.3) for the linear differential equation (12.3.2), starting from the initial state `z0` and returning the state after a single step of size `h`.

SOLUTION for (12-3.f) → 12-3-6-0:SDIR4s.pdf ▲

(12-3.g) ⚂ (25 min.) [depends on Sub-problem (12-3.f)]

Realize a C++ function

```
double cvgSDIRK();
```

to conduct a numerical experiment, which gives an indication of the order of the method (with $\gamma = \frac{3+\sqrt{3}}{6}$) for the initial value problem from (12.3.3). Choose $\mathbf{z}_0 = [1,0]^\top$ as initial value, `T=10` as end time and `M=20,40,80,...,10240` as numbers of equidistant timesteps. Tabulate the error $|y(T) - y_M|$ ($y := (\mathbf{z})_1$) at final time and use it as a basis for the estimation of the rate of algebraic convergence by means of the supplied function `polyfit()`.

HIDDEN HINT 1 for (12-3.g) → 12-3-7-0:sd5h1.pdf

SOLUTION for (12-3.g) → 12-3-7-1:SDIR5s.pdf ▲

End Problem 12-3 , 115 min.

Problem 12-4: Semi-implicit Runge-Kutta SSM

General implicit Runge-Kutta methods as introduced in [Lecture → Section 12.3.3] entail solving systems of non-linear equations for the increments, see [Lecture → Rem. 12.3.3.9]. Semi-implicit Runge-Kutta single step methods, also known as Rosenbrock-Wanner (ROW) methods [Lecture → (12.4.0.9)] just require the solution of linear systems of equations. This problem deals with a concrete ROW method, its stability and aspects of its implementation.

Relies on the contents of [Lecture → Section 12.1] and also depends on [Lecture → Section 12.3.4].

We consider the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \tag{12.4.1}$$

and discretize it with a *semi-implicit* Runge-Kutta SSM: a so-called **Rosenbrock method** given by

$$\begin{cases} \mathbf{W}\mathbf{k}_1 = \mathbf{f}(\mathbf{y}_0)\,, \\ \mathbf{W}\mathbf{k}_2 = \mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\,\mathbf{k}_1) - ah\,\mathbf{J}\,\mathbf{k}_1\,, \\ \mathbf{y}_1 = \mathbf{y}_0 + h\mathbf{k}_2\,, \end{cases} \tag{12.4.2}$$

where

$$\mathbf{J} = D\mathbf{f}(\mathbf{y}_0) \quad , \quad \mathbf{W} = \mathbf{I} - ah\,\mathbf{J} \quad , \quad a := \frac{1}{2+\sqrt{2}}\,.$$

(12-4.a) ⚀ (10 min.) In [Lecture → § 12.4.0.8] the general form of s-stage Rosenbrock-Wanner single-step methods for the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ was given as

$$(\mathbf{I} - ha_{ii}\mathbf{J})\mathbf{k}_i = \mathbf{f}(\mathbf{y}_0 + h\sum_{j=1}^{i-1}(a_{ij}+d_{ij})\mathbf{k}_j) - h\mathbf{J}\sum_{j=1}^{i-1} d_{ij}\mathbf{k}_j\,, \quad \mathbf{J} = \mathrm{D}\,\mathbf{f}(\mathbf{y}_0)\,,$$
$$\mathbf{y}_1 := \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{k}_j\,. \qquad \text{[Lecture → (12.4.0.9)]}$$

What is s and what are the coefficients a_{ij}, d_{ij}, and b_j for the method (12.4.2)?

SOLUTION for (12-4.a) → 12-4-1-0:simplA.pdf ▲

(12-4.b) ⚀ (15 min.) Refresh your knowledge about how you can compute the stability function of a Runge-Kutta single-step function.

SOLUTION for (12-4.b) → 12-4-2-0:.pdf ▲

(12-4.c) ⚀ (15 min.) [depends on Sub-problem (12-4.b)]

Compute the stability function S of the Rosenbrock method (12.4.2) for the linear scalar model ODE $\dot{y} = \lambda y, \lambda \in \mathbb{C}$. In other words, compute the (rational) function $S(z)$, such that

$$y_1 = S(z)y_0, \quad z := h\lambda,$$

when the method is applied to perform a single setp of size h starting from y_0.

SOLUTION for (12-4.c) → 12-4-3-0:SemI1s.pdf ▲

(12-4.d) ⚂ (15 min.) [depends on Sub-problem (12-4.c)]

Compute the first 4 terms of the Taylor expansion of $S(z)$ around $z = 0$. What is the maximal $q \in \mathbb{N}$ such that

$$|S(z) - \exp(z)| = O(|z|^q)$$

for $|z| \to 0$? Deduce the maximal possible order of the method (12.4.2).

HIDDEN HINT 1 for (12-4.d) → `12-4-4-0:SemI2h.pdf`

SOLUTION for (12-4.d) → `12-4-4-1:SemI2s.pdf` ▲

(12-4.e) ⚂ (30 min.) In `rosenbrock.hpp`, implement a C++ function:

```
template <class Function, class Jacobian>
std::vector<Eigen::VectorXd>
 solveRosenbrock(Function &&f, Jacobian &&df,
                 const Eigen::VectorXd &y0,
                 unsigned int M, double T) ;
```

that applies the Rosenbrock method (12.4.2) for solving an initial-value problem for an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$.

It takes as input functor objects for $\mathbf{f}$ and $D\mathbf{f}$ (e.g., as lambda functions), an initial data (vector or scalar) `y0` $= \mathbf{y}(0)$, a number of steps M and a final time T. The function returns the sequence of states generated by the single step method up to $t = T$, using M equidistant steps of the Rosenbrock method.

SOLUTION for (12-4.e) → `12-4-5-0:SemI3s.pdf` ▲

(12-4.f) ⚂ (15 min.) [depends on Sub-problem (12-4.e)]

Explore the order of the method (12.4.2) empirically by applying it to the IVP for the limit cycle [Lecture → Ex. 12.2.0.4]:

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\mathbf{y}\,, \tag{12.4.5}$$

with $\lambda = 1$ and initial state $\mathbf{y}_0 = [1,1]^\top$ on $[0,10]$. Use uniform timesteps of size $h = 2^{-k}, k = 4,\ldots,10$ and compute a reference solution $\mathbf{y}_{\mathrm{ref}}$ with timestep size $h = 2^{-12}$. Monitor the maximal error on the temporal mesh

$$\max_j \|\mathbf{y}_j - \mathbf{y}_{\mathrm{ref}}(t_j)\|_2\,,$$

and use it to estimate a rate of algebraic convergence by means of linear regression.

To do this write in `rosenbrock.hpp` a C++ function

```
double cvgRosenbrock();
```

that tabulates the errors and returns the estimated rate of convergence.

SOLUTION for (12-4.f) → `12-4-6-0:SemI4s.pdf` ▲

In complex analysis you might have hear about the maximum principle for holomorphic/analytic functions. The following is a special version of it.

Theorem 12.4.7. Maximum principle for holomorphic functions

Let

$$\mathbb{C}^- := \{z \in \mathbb{C} \mid Re(z) < 0\} .$$

Let $f : D \subset \mathbb{C} \to \mathbb{C}$ be non-constant, defined on $\overline{\mathbb{C}^-}$, and analytic in $\mathbb{C}^-$. Furthermore, assume that $w := \lim_{|z|\to\infty} f(z)$ exists and $w \in \mathbb{C}$, then:

$$\forall z \in \mathbb{C}^- : \quad |f(z)| < \sup_{\tau\in\mathbb{R}} |f(i\tau)| .$$

(12-4.g) ⚄ (20 min.) Appealing to Thm. 12.4.7 show that the method (12.4.2) is **L-stable** (*cf.* [Lecture → § 12.3.4.14]).

HIDDEN HINT 1 for (12-4.g) → 12-4-7-0:SemI5h.pdf

SOLUTION for (12-4.g) → 12-4-7-1:SemI5s.pdf ▲

End Problem 12-4 , 120 min.

Problem 12-5: Exponential integrator

The exponential integrators are a modern class of single step methods developed for special initial value problems (problems that can be regarded as perturbed linear ODEs), see

M. HOCHBRUCK AND A. OSTERMANN, *Exponential integrators*, Acta Numerica, 19 (2010), pp. 209–286.

These methods fit the concept of single step methods as introduced in [Lecture → Def. 11.3.1.5] and, usually, converge algebraically according to [Lecture → (11.3.2.7)].

You have to be familiar with fundamental concepts for single-step methods as introduced in [Lecture → Section 11.3.1], their asymptic convergence properties from [Lecture → Section 11.3.2] and the notion of a stability domain [Lecture → Def. 12.1.0.51].

A step with size h of the so-called **exponential Euler** single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with continuously differentiable $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$ reads:

$$\mathbf{y}_1 = \mathbf{y}_0 + h\, \varphi\big(h\, \mathrm{D}\,\mathbf{f}(\mathbf{y}_0)\big)\, \mathbf{f}(\mathbf{y}_0), \tag{12.5.1}$$

where $\mathrm{D}\,\mathbf{f}(\mathbf{y}) \in \mathbb{R}^{N,N}$ is the Jacobian of $\mathbf{f}$ at $\mathbf{y} \in \mathbb{R}^N$, and the matrix function $\varphi : \mathbb{R}^{N,N} \to \mathbb{R}^{N,N}$ is defined as

$$\varphi(\mathbf{Z}) = (\exp(\mathbf{Z}) - \mathrm{Id})\, \mathbf{Z}^{-1}\,, \quad \mathbf{Z} \in \mathbb{R}^{N,N}\,. \tag{12.5.2}$$

Here, $\exp(\mathbf{Z})$ is the matrix exponential of $\mathbf{Z}$, a special function $\exp : \mathbb{R}^{N,N} \to \mathbb{R}^{N,N}$, see [Lecture → (12.1.0.34)].

The function φ is implemented in the file `exponentialintegrator.hpp` as the function

```
Eigen::MatrixXd phim(const Eigen::MatrixXd &Z);
```

When plugging in the exponential series, it is clear that the function $z \mapsto \varphi(z) := \frac{\exp(z)-1}{z}$ is analytic on $\mathbb{C}$ with $\varphi(0) = 1$. Thus, $\varphi(\mathbf{Z})$ is well defined for all matrices $\mathbf{Z} \in \mathbb{R}^{N,N}$.

(12-5.a) ⚀ (20 min.) Is the exponential Euler single step method defined in (12.5.1) consistent with the ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ (see [Lecture → Def. 11.3.1.12])? Explain your answer.

SOLUTION for (12-5.a) → `12-5-1-0:Expo1h.pdf` ▲

(12-5.b) ⚁ (15 min.) Show that the exponential Euler single step method defined in (12.5.1) solves the linear initial value problem

$$\dot{\mathbf{y}} = \mathbf{A}\,\mathbf{y}\,, \quad \mathbf{y}(0) = \mathbf{y}_0 \in \mathbb{R}^d\,, \qquad \mathbf{A} \in \mathbb{R}^{d,d}\,, \tag{12.5.3}$$

exactly.

HIDDEN HINT 1 for (12-5.b) → `12-5-2-0:Expo1s.pdf`

SOLUTION for (12-5.b) → `12-5-2-1:Expo2h.pdf` ▲

(12-5.c) ⚂ (15 min.) [depends on Sub-problem (12-5.b)]

Determine the region of stability of the exponential Euler single step method defined in (12.5.1) (see [Lecture → Def. 12.1.0.51]).

SOLUTION for (12-5.c) → `12-5-3-0:Expo3h.pdf` ▲

(12-5.d) ⚃ (20 min.) In `exponentialintegrator.hpp` , write a C++ function

```
template <class Function, class Jacobian>
Eigen::VectorXd exponentialEulerStep(const Eigen::VectorXd &y0,
   Function &&f, Jacobian &&df, double h)
```

that implements (12.5.1). Here `f` and `df` are objects with evaluation operators representing the ODE right-hand side function $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$ and its Jacobian, respectively.

SOLUTION for (12-5.d) → 12-5-4-0:Expo4h.pdf ▲

(12-5.e) ⚄ (30 min.) What is the order of the single step method (12.5.1)?

To investigate it empirically, in the file `exponentialintegrator.hpp`, implement a C++ function

```
void testExpEulerLogODE();
```

that applies the method to the scalar logistic ODE

$$\dot{y} = y\,(1-y)\,, \quad y(0) = 0.1\,,$$

in the time interval $[0,1]$. Tabulate the error at the final time against the stepsize $h = T/M$, $M = 2^k$ for $k = 1,\ldots,15$. Discuss qualitatively and quantitative the empiric asymptotic convergence of the single-step method.

HIDDEN HINT 1 for (12-5.e) → 12-5-5-0:Expo3s.pdf

SOLUTION for (12-5.e) → 12-5-5-1:Expo5h.pdf ▲

End Problem 12-5, 100 min.

Problem 12-6: Mono-implicit Runge-Kutta single step method

As explained in [Lecture → Section 12.3.4] we must resort to implicit single-step methods as solvers for stiff initial-value problems for ODEs. The big challenge about *implicit* Runge-Kutta single-step methods (RK-SSMs) is the need to solve possibly big non-linear systems of equations. Many attempts have been made to alleviate this drawback and this problem presents one of them.

This problem needs the linear model problem analysis as introduced in [Lecture → Section 12.1] and [Lecture → Section 12.3.4]. It practices the implementation of implicit single step methods with a focus on solving the non-linear systems of equations, *cf.* [Lecture → Rem. 12.3.3.9]. A little C++ implementation based on EIGEN is requested.

A so-called s-stage **mono-implicit Runge-Kutta single step method (MIRK)** for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subset \mathbb{R}^N \to \mathbb{R}^N$, is defined as:

$$\begin{aligned} \mathbf{g}_i &:= (1 - v_i)\mathbf{y}_0 + v_i\mathbf{y}_1 + h\sum_{j=1}^{i-1} d_{i,j}\mathbf{f}(\mathbf{g}_j), \quad i = 1, \dots s\,, \\ \mathbf{y}_1 &:= \mathbf{y}_0 + h\sum_{j=1}^{s} b_j\mathbf{f}(\mathbf{g}_j) \end{aligned} \tag{12.6.1}$$

for suitable coefficients $v_i, b_i, d_{i,j} \in \mathbb{R}$, fixed to achieve a desired order.

(12-6.a) ⚁ (20 min.) Single step methods defined as in (12.6.1) belong to the class of implicit Runge-Kutta methods [Lecture → Def. 12.3.3.1]. Write down the corresponding Butcher scheme [Lecture → (12.3.3.3)] in terms of the coefficients $v_i, b_i, d_{i,j}$.

HIDDEN HINT 1 for (12-6.a) → 12-6-1-0:mirk0.pdf

SOLUTION for (12-6.a) → 12-6-1-1:MIRK1h.pdf

▲

(12-6.b) ⚄ (20 min.) Compute the stability function of a MIRK scheme defined as in (12.6.1) for $s = 2$.

HIDDEN HINT 1 for (12-6.b) → 12-6-2-0:MIRK1.pdf

SOLUTION for (12-6.b) → 12-6-2-1:MIRK2h.pdf

▲

(12-6.c) ⚁ (10 min.) Now, we consider the special case of a scalar ODE ($N = 1$) and $s = 2$. Abbreviating $\mathbf{z} := [g_1, g_2, y_1]^\top$, rewrite (12.6.1) as a non-linear system of equations in the form $\mathbf{F}(\mathbf{z}) = \mathbf{0}$ for an explicitly specified suitable function $\mathbf{F} : \mathbb{R}^3 \to \mathbb{R}^3$.

SOLUTION for (12-6.c) → 12-6-3-0:MIRK3h.pdf

▲

(12-6.d) ⚁ (10 min.) [depends on Sub-problem (12-6.c)]

Find the Jacobian $D\mathbf{F}(\mathbf{z})$ of the function $\mathbf{F}$ (in terms of $d_{i,j}, v_i, b_i$ and the derivative of f) from the previous sub-problem.

SOLUTION for (12-6.d) → 12-6-4-0:MIRK4h.pdf ▲

In the following sub-problems, we will implement the 2-stage MIRK scheme for the scalar case: $N = 1, s = 2$.

(12-6.e) ⚄ (30 min.) [depends on Sub-problem (12-6.c) and Sub-problem (12-6.c)]

In `mirk.hpp`, implement a function

```
template <class Func, class Jac>
Eigen::VectorXd Newton2Steps(Func &&F, Jac &&DF,
               Eigen::VectorXd z);
```

that approximates the solution of $\mathbf{F}(\mathbf{z}) = \mathbf{0}$ by performing two steps of the Newton method applied to $\mathbf{F}(\mathbf{z}) = \mathbf{0}$. Use z to pass the initial guess. Objects of type `Func` and `Jac` have to supply suitable evaluation operators **operator** `()`, which, for the type `Jac` must return an EIGEN-compatible matrix.

SOLUTION for (12-6.e) → `12-6-5-0:MIRK5h.pdf`

▲

(12-6.f) (20 min.) [depends on Sub-problem (12-6.e)]

Consider the particular MIRK scheme given by the coefficients:

$$v_1 = 1, \quad v_2 = \frac{344}{2025}, \quad d_{21} = -\frac{164}{2025}, \quad b_1 = \frac{37}{82}, \quad b_2 = \frac{45}{82}. \tag{12.6.6}$$

Using the function `Newton2Steps`, implement in `mirk.hpp` a function

```
template <class Func, class Jac>
double MIRKStep(Func &&f, Jac &&df, double y0, double h);
```

that realizes one step of the MIRK scheme defined by (12.6.1) for $s = 2$ and a scalar ODE, that is, $d = 1$. The right hand side function $\mathbf{f}$ and its Jacobian are passed through f and df. The solution of the nonlinear system arising from (12.6.1) is approximated using two Newton steps, namely by using the function `Newton2Steps()`. The initial guess has to be chosen appropriately!

SOLUTION for (12-6.f) → `12-6-6-0:MIRK6h.pdf` ▲

(12-6.g) ⚄ (20 min.) [depends on Sub-problem (12-6.f)]

Implement in `mirk.hpp` a function

```
template <class Func, class Jac>
double MIRKSolve(Func &&f, Jac &&df, double y0,
    double T, unsigned int M);
```

for the solution of a scalar ODE up to time T, using M equidistant steps of the mono-implicit Runge-Kutta single step method defined by (12.6.6). The initial value is passed in `y0`.

SOLUTION for (12-6.g) → `12-6-7-0:MIRK7h.pdf` ▲

(12-6.h) ⚄ (30 min.) In the file `mirk.hpp` implement a C++ function

```
void cvgMIRK();
```

that applies your implementation of the MIRK method to the IVP

$$\dot{y} = 1 + y^2 \quad , \quad y(0) = 0\,, \tag{12.6.9}$$

on $[0,1]$. The exact solution of (12.6.9) is $y_{ex}(t) := \tan t$. Compute the solution y_M at $T = 1$ with a sequence of uniform temporal meshes with $M = 4, \ldots, 512$ intervals. Compute the error $|y_M(1) - y_{ex}(1)|$ and output an error table for $M = 4, \ldots, 512$.

Determine the empiric rate of convergence of the MIRK method for (12.6.9).

SOLUTION for (12-6.h) → `12-6-8-0:MIRK8h.pdf` ▲

End Problem 12-6 , 160 min.

Problem 12-7: Stability of a Runge-Kutta method

This problem is devoted to an empirical study of stability problems haunting explicit Runge-Kutta single-step methods, whose stability functions invariably are polynomials.

This problem is meant to supplement the discussion in [Lecture → Section 12.1].

We focus on a 3-stage Runge-Kutta single step method described by the following Butcher-Tableau [Lecture → (11.4.0.13)].

$$\begin{array}{c|ccc} 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1/2 & 1/4 & 1/4 & 0 \\ \hline & 1/6 & 1/6 & 2/3 \end{array} \tag{12.7.1}$$

We also consider the following concrete case of the prey/predator model as introduced in [Lecture → Ex. 11.1.2.5]:

$$\begin{aligned} \dot{y}_1(t) &= (1 - y_2(t))y_1(t)\,, \\ \dot{y}_2(t) &= (y_1(t) - 1)y_2(t)\,. \end{aligned} \tag{12.7.2}$$

(12-7.a) ⚀ (30 min.) Create a C++ function

```
Eigen::Vector2d PredPrey(Eigen::Vector2d y0, double T, unsigned
    int M);
```

that uses the RK-SSM (12.7.1) to solve an initial value problem for (12.7.2) with initial value $\mathbf{y}_0$ and M equidistant timestep up to final time $T > 0$. It should return $\mathbf{y}_N \approx \mathbf{y}(T)$.

SOLUTION for (12-7.a) → 12-7-1-0:Stab0s.pdf ▲

(12-7.b) ⚀ (20 min.) [depends on Sub-problem (12-7.a)]

Write a C++ function

```
void SimulatePredPrey();
```

to approximate the solution at time $T = 1$ of the IVP for (12.7.2) with initial value $\mathbf{y}_0 = \mathbf{y}(0) = [100, 1]^\top$. Use the RK-SSM (12.7.1) with uniform timestep $h > 0$ and the function from Sub-problem (12-7.a).

Qualitatively and quantitatively determine the type of asymptotic convergence of the method for uniform steps of size 2^{-j}, $j = 2, \ldots, 13$. As a reference solution, use an approximation obtained with 2^{14} steps. Tabulate the norm of the error at final time.

What do you notice for big step sizes? Try to find the maximum step size for which blow-up of the numerical solution can still be avoided.

HIDDEN HINT 1 for (12-7.b) → 12-7-2-0:Stab1h2.pdf

SOLUTION for (12-7.b) → 12-7-2-1:Stab1s.pdf ▲

(12-7.c) ⚀ (15 min.) Calculate the stability function $S = S(z)$, $z \in \mathbb{C}$, of the method given by the Butcher scheme (12.7.1).

SOLUTION for (12-7.c) → 12-7-3-0:Stab2s.pdf ▲

End Problem 12-7, 65 min.

Problem 12-8: Implicit Runge-Kutta Method for Gradient-Flow ODEs

In this problem we apply a special class of *implicit* Runge-Kutta single-step methods to a special class of ODEs, many of which give rise to *stiff* initial-value problems.

You need to master [Lecture → Section 12.3.3] and [Lecture → Section 12.3.4] and should also be familiar with the notion of a "stiff" IVP, see [Lecture → Section 12.2].

We consider the autonomous **gradient-flow ODE**

$$\dot{\mathbf{y}}(t) = \mathbf{f}(\mathbf{y}) := -\mathbf{grad}\, V(\mathbf{y}(t)) \,, \tag{12.8.1}$$

where $V : D \subset \mathbb{R}^N \to \mathbb{R}$ is a continuously differentiable function.

(12-8.a) ⚀ (15 min.)

Show that for a solution $t \mapsto \mathbf{y}(t)$ of (12.8.1) defined on $I \subset \mathbb{R}$ the function $t \in I \mapsto V(\mathbf{y}(t))$ is non-increasing.

SOLUTION for (12-8.a) → 12-8-1-0:s1.pdf ▲

(12-8.b) ⚁ (10 min.)

The scalar ($N = 1$) linear ODE $\dot{y} = -\lambda y$, $\lambda \in \mathbb{R}$, belongs to the class of gradient-flow ODEs of the form (12.8.1). What is V in this case?

SOLUTION for (12-8.b) → 12-8-2-0:s2.pdf ▲

For the numerical integration of gradient-flow ODEs we opt for a so-called **SDIRK** (singly diagonally implicit Runge-Kutta) method, an implicit 5-stage Runge-Kutta method described by the Butcher scheme

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^\top \end{array} \quad \widehat{=} \quad \begin{array}{c|ccccc} \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\ \frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & 0 & 0 \\ \frac{1}{2} & \frac{371}{1360} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & 0 \\ 1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\ \hline & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \end{array} \tag{12.8.2}$$

In the file `gradientflow.hpp` you find a function

```
Eigen::MatrixXd ButcherMatrix();
```

that gives you the 6×5-matrix $\begin{bmatrix} \mathfrak{A} \\ \mathbf{b}^\top \end{bmatrix}$ of the Butcher scheme (12.8.2). Its bottom row is the vector $\mathbf{b}^\top$.

(12-8.c) ⚁ (20 min.)

State the non-linear systems of equations of size $N \times N$ that have to be solved when applying the implicit Runge-Kutta single-step method (with timestep $h > 0$) described by the Butcher scheme (12.8.2) to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$.

SOLUTION for (12-8.c) → 12-8-3-0:s3.pdf ▲

(12-8.d) ⚂ (20 min.) [depends on Sub-problem (12-8.c)]

Give the **stage form** [Lecture → (12.3.3.8)] of the increment equations for the implicit Runge-Kutta single-step method (timestep $h > 0$) described by the Butcher scheme (12.8.2) and applied to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$.

SOLUTION for (12-8.d) → 12-8-4-0:s4.pdf ▲

(12-8.e) ⚂ (45 min.) [depends on Sub-problem (12-8.d)]

In the file `gradientflow.hpp` implement a C++ function

```
template <typename Functor, typename Jacobian>
std::array<Eigen::VectorXd, 5> computeStages(
Func &&f, Jac &&df,
const Eigen::VectorXd &y, double h,
double rtol = 1E-6, double atol = 1E-8);
```

that uses a standard (undamped) *Newton iteration* (correction-based termination with relative tolerance `rtol` and absolute tolerance `atol`) to solve for the *stages* $\mathbf{g}_i$ (→ [Lecture → (12.3.3.8)])required for one step of size $h > 0$ of the implicit Runge-Kutta single-step method described by the Butcher scheme (12.8.2) and applied to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$.

The right-hand side function $\mathbf{f}$ and its Jacobian are passed through suitable functor objects `f` and `J` respectively, whereas `y` supplies the initial state for the step.

HIDDEN HINT 1 for (12-8.e) → 12-8-5-0:newt.pdf

SOLUTION for (12-8.e) → 12-8-5-1:s5.pdf ▲

(12-8.f) ⚂ (30 min.) [depends on Sub-problem (12-8.e)]

Based on `computeStages()` in `gradientflow.hpp` realize a a C++ function

```
template <typename Func, typename Jac>
Eigen::VectorXd discEvolSDIRK(
Func &&f, Jac &&df, const Eigen::VectorXd &y,
double h, double rtol = 1E-6, double atol = 1E-8);
```

that evaluates the discrete evolution operator $\Psi^h\mathbf{y}$ for the implicit Runge-Kutta single-step method described by the Butcher scheme (12.8.2) and applied to a generic autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ with $\mathbf{f} : D \subset \mathbb{R}^N \mapsto \mathbb{R}^N$, $N \in \mathbb{N}$. Of course, the parameters `y` and `h` supply the arguments $\mathbf{y}$ and h for Ψ. The meaning of `f`, `J`, `rtol` and `atol` has been explained before

SOLUTION for (12-8.f) → 12-8-6-0:s6.pdf ▲

Finally we consider the concrete gradient-flow ODE (12.8.1) spawned by the potential function

$$V(\mathbf{y}) = \sin(\|\mathbf{y}\|^2) + \lambda(\mathbf{d}^\top\mathbf{y})^2\,, \quad \lambda \in \mathbb{R}\,, \quad \mathbf{y},\mathbf{d} \in \mathbb{R}^N\,, \quad \|\mathbf{d}\| = 1\,. \tag{12.8.8}$$

We tackle associated initial value-problems with initial state vector $\mathbf{y}_0$, $\|\mathbf{y}_0\| = 1$, and want to solve them on the time interval $[0, 0.1]$.

(12-8.g) ⚂ (15 min.)

Write down the autonomous gradient-flow ODE induced by $\mathbf{y} \mapsto V(\mathbf{y})$ from (12.8.8).

SOLUTION for (12-8.g) → 12-8-7-0:s6a.pdf ▲

(12-8.h) ⚄ (30 min.) [depends on Sub-problem (12-8.g)]

For what values of the parameter $\lambda \in \mathbb{R}$ do we face a *stiff* ODE for states $\mathbf{y}$ close to zero: $\|\mathbf{y}\| \ll 1$?

HIDDEN HINT 1 for (12-8.h) → 12-8-8-0:h7s.pdf

SOLUTION for (12-8.h) → 12-8-8-1:s7.pdf ▲

(12-8.i) ⚅ (30 min.) [depends on Sub-problem (12-8.f) and Sub-problem (12-8.g)]

Based on your implementation of `discEvolSDIRK()` in `gradientflow.hpp` write a C++ function

```
std::vector<Eigen::VectorXd>
solveGradientFlow(const Eigen::VectorXd &d,
double lambda,
const Eigen::VectorXd &y0,
double T, unsigned int M);
```

that uses M equidistant steps of the SDIRK RK-SSM with the Butcher scheme (12.8.2) to solve the gradient-flow ODE (12.8.1) spawned by V from (12.8.8) up to final time $T > 0$. The arguments `d` and `lambda` supply the parameters, and `y0` the initial state $\mathbf{y}^{(0)}$. Use the default tolerance parameters for Newton's method as specified in the signature of `computeStages()`.

HIDDEN HINT 1 for (12-8.i) → 12-8-9-0:h8lf.pdf

SOLUTION for (12-8.i) → 12-8-9-1:s8.pdf ▲

(12-8.j) ⚂ (10 min.)

We have solved the gradient-flow ODE (12.8.1) with V from (12.8.8) and $N = 2$, $\mathbf{d} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $\lambda = 10$, $\mathbf{y}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$, $T = 0.1$.

N	error norm
10	5.99348e-07
20	3.65190e-08
40	2.25261e-09
80	1.39860e-10
160	8.71007e-12
320	5.41081e-13

The table lists the norms of the error $\mathbf{y}_N - \mathbf{y}(T)$ at final time for different numbers N of equidistant timesteps.

Which empiric order of convergence of the SDIRK single-step method do the data suggest?

SOLUTION for (12-8.j) → 12-8-10-0:s9.pdf ▲

End Problem 12-8 , 225 min.

Problem 12-9: ROW Method for Simulating the van der Pol Oscillator

The so-called van der Pol oscillator is a well-known model for a stable oscillator with non-linear damping, see Wikipedia page. In this problem we study how to apply a special semi-implicit single-step methods to simulate it.

This problem is related to [Lecture → Section 12.4] and [Lecture → Section 12.1].

The van der Pol equation is the scalar second-order ODE

$$\ddot{y} - \mu(1-y^2)\dot{y} + y = 0\,, \quad \mu > 0\,. \tag{12.9.1}$$

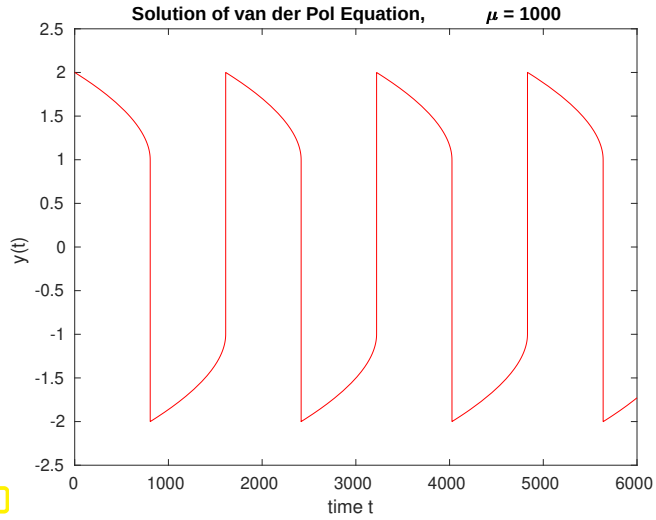


Fig. 121

◁ Plot of the solution for the special initial-value problem for (12.9.1):
- $\mu = 1000$,
- $t_0 = 0$,
- $y(0) = 2$, $\dot{y}(0) = 0$

For $t \to \infty$ the solution will be periodic.

(12-9.a) ⚄ (30 min.) Give rigorous arguments, why the initial-value problem

$$\ddot{y} - \mu(1-y^2)\dot{y} + y = 0\,, \quad \mu = 10^4\,, \quad y(0) = 2\,,\ \dot{y}(0) = 0\,, \tag{12.9.2}$$

is stiff, at least in the beginning of a simulation.

SOLUTION for (12-9.a) → 12-9-1-0:vdpsa.pdf ▲

In the research article [Ran15] new so-called **Rosenbrock-Wanner (ROW)** semi-implicit single-step methods for the numerical integration of general first-order ODEs $\dot{\mathbf{y}} = \mathbf{f}(t,\mathbf{y})$, $\mathbf{f} : I \times D \to \mathbb{R}^N$, $I \subset \mathbb{R}$, $D \subset \mathbb{R}^N$, are presented. The author introduces a single step of an s-stage ROW method with timestep size $h > 0$ as follows:

$$\mathbf{k}_i = \mathbf{f}(t_0 + \alpha_i h, \mathbf{z}_i) + h\mathbf{J}\sum_{j=1}^{i}\gamma_{i,j}\mathbf{k}_j + h\gamma_i\frac{\partial \mathbf{f}}{\partial t}(t_0,\mathbf{y}_0)\,, \quad i = 1,\ldots,s\,, \tag{12.9.11a}$$

$$\mathbf{z}_i = \mathbf{y}_0 + h\sum_{j=1}^{i-1}\alpha_{i,j}\mathbf{k}_j\,, \quad i = 1,\ldots,s \tag{12.9.11b}$$

$$\mathbf{y}_1 = \mathbf{y}_0 + h\sum_{i=1}^{s} b_i\mathbf{k}_i\,, \tag{12.9.11c}$$

where $\mathbf{J} := \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(t_0,\mathbf{y}_0) \in \mathbb{R}^{N,N}$ is a Jacobian, and $\alpha_i, \gamma_i, \alpha_{i,j}, \gamma_{i,j}, b_i \in \mathbb{R}$ are suitably chosen parameters, which must satisfy

$$\alpha_i = \sum_{j=1}^{i-1}\alpha_{i,j} \quad , \quad \gamma_i = \sum_{j=1}^{i-1}\gamma_{i,j}\,, \quad i = 1,\ldots,s\,.$$

Moreover, all $\gamma_{i,i}$, $i = 1,\ldots,s$, are equal and positive: $\gamma_{i,i} = \gamma > 0$. The coefficients are stored in a special data structure based in EIGEN's datatype, explanations in the comments.

```
struct ROW {
  explicit ROW(unsigned int s)
  : s_(s),
  alpha_(Eigen::MatrixXd::Zero(s, s)),
  gamma_(Eigen::MatrixXd::Zero(s, s)),
  b_(Eigen::VectorXd::Zero(s)) {}
  unsigned int s_;          // Number of stages
  Eigen::MatrixXd alpha_;   // Coefficients α_{i,j}, i,j = 1,...,s
  double d_gamma_;          // Value γ > 0 for γ_{i,i}, i = 1,...,s
  Eigen::MatrixXd gamma_;   // Coefficients γ_{i,j}, i,j = 1,...,s, j < i
  Eigen::VectorXd b_;       // Weights b_i, i = 1,...,s
};
```

From [Ran15, Table 1] we get appropriate values for the coefficients for the 3-stage ROW method called ROSxPR. Note that the $\alpha_{i,j}$ and $\gamma_{i,j}$ are needed only for $i > j$.

C++ code 12.9.12: Initialization of coefficient for 3-stage ROW method ROSxPR

```
ROW init_ROSxPR(void) {
  ROW row(3);                                  // s = 3
  row.alpha_(1, 0) = 2.3660254037844388;       // α_{2,1}
  row.alpha_(2, 0) = 0.0;                      // α_{3,1}
  row.alpha_(2, 1) = 1.0;                      // α_{3,2}
  row.gamma_(1, 0) = -2.3660254037844388;      // γ_{2,1}
  row.gamma_(2, 0) = -0.28468642516567449;     // γ_{3,1}
  row.gamma_(2, 1) = -1.0813389786187642;      // γ_{3,2}
  row.d_gamma_ = 0.78867513459481287;          // γ_{1,1} = γ_{2,2} = γ_{3,3}
  row.b_[0] = 0.29266384402395124;             // b_1
  row.b_[1] = -0.081338978618764143;           // b_2
  row.b_[2] = 0.78867513459481287;             // b_3
  return row;
}
```

(12-9.b) ⚂ (15 min.) The templated C++ function

```
template <typename STATETYPE, typename RHSTYPE, typename JACTYPE>
std::vector<STATETYPE>
odeROW(RHSTYPE &&rhs, JACTYPE &&Jacobian, const STATETYPE &y0,
double T, unsigned int M, const ROW &row);
```

is supposed to realize `M` uniform steps of the ROW method specified in `row` for the *autonomous* IVP

$$\dot{\mathbf{y}} = \mathtt{rhs}(\mathbf{y}) \quad , \quad \mathbf{y}(0) = \mathtt{y0}\,.$$

until final time `T`. The argument `Jacobian` should contain a functor providing the Jacobi matrix $\mathbf{y} \mapsto \mathrm{D}\,\mathtt{rhs}(\mathbf{y})$. The following requirements are imposed on the template argument types:

- **STATETYPE** should be compatible with a vector type of EIGEN,
- **RHSTYPE** must be `std::function<STATETYPE(const STATETYPE &)>`,
- **JACTYPE** must be a functor taking a **STATETYPE** argument and returning an object with the capabilities of an EIGEN matrix.

Fill in the blank in the following listing so that it properly implements `odeROW()`.

```
template <typename STATETYPE, typename RHSTYPE, typename JACTYPE>
std::vector<STATETYPE> odeROW(RHSTYPE &&rhs, JACTYPE &&Jacobian,
const STATETYPE &y0, double T, unsigned int M,
const ROW &row) {
  using MATTYPE = decltype(Jacobian(y0)); // Type for Jacobi matrix
  unsigned int N = y0.size();
  std::vector<STATETYPE> y(M + 1); // vector of computed states
  y[0] = y0;
  std::vector<STATETYPE> k(row.s_); // increments k_i
  std::vector<STATETYPE> z(row.s_); // Auxiliary states z_i
  const double h = T / M; // timestep size h
  for (unsigned int m = 0; m < M; ++m) { // main timestepping loop
    const MATTYPE J = Jacobian(y[m]);
    [illegible]
    auto Ig0_lu = Ig0.lu();
    z[0] = y[m];
    [illegible]
    [illegible]
    for (unsigned int i = 1; i < row.s_; ++i) {
      z[i] = y[m];
      for (unsigned int j = 0; j < i; ++j) {
        [illegible]
        [illegible]
      for (unsigned int j = 1; j < i; ++j) {
        [illegible]
        [illegible]
        [illegible]
    }
  }
  return y;
}
```

HIDDEN HINT 1 for (12-9.b) → 12-9-2-0:vpdxh1.pdf

SOLUTION for (12-9.b) → 12-9-2-1:vpdsx.pdf ▲

(12-9.c) ⚂ (15 min.) In the file `vanderpol.hpp` implement a C++ function

```
std::complex<double> stabFnROW(std::complex<double> z, const ROW
  &row);
```

that evaluates the **stability function** $z \in \mathbb{C} \mapsto S(z)$ [Lecture → Thm. 12.1.0.17] for the ROW method specified through `row`.

HIDDEN HINT 1 for (12-9.c) → 12-9-3-0:vdpyh1.pdf

SOLUTION for (12-9.c) → 12-9-3-1:vdpsts.pdf ▲

(12-9.d) ⚂ (15 min.) [depends on Sub-problem (12-9.b)]

In the file `vanderpol.hpp` complete the code of the C++ function

```
std::vector<double> solveROWVanDerPol(double mu, double T,
   unsigned int M);
```

that solves an IVP for the van der Pol equation (12.9.1) with initial data $y(0) = 2$, $\dot{y}(0) = 0$, using the ROSxPR method provided by `init_ROSxPR()` in Code 12.9.12. The arguments `mu`, `T`, and `M` pass the model parameter $\mu > 0$, the final time $T > 0$, and the number of uniform timesteps. Of course, your implementation can call the templated function `odeROW()`.

SOLUTION for (12-9.d) → 12-9-4-0:vdpzs.pdf ▲

(12-9.e) ⚂ (15 min.) Use the C++ function `solveROWVanDerPol()` implemented in Sub-problem (12-9.d) to create (in the file `vanderpol.hpp`) a C++ function

```
void testCvgROWVanDerPol(void);
```

that prints to `stdout` a *table* of values, from which one can immediately see the rate of algebraic convergence of the error at final time of the ROW method applied to solve the IVP for the van der Pol equation detailed in Sub-problem (12-9.d). Use the parameter value $\mu = 1$ and final time $T = 6$.

The function `testCvgROWVanDerPol()` is called from the `main()` program. Run the code and conclude the order of the ROW method ROSxPR from the output.

$$\text{order of ROSxPR} = \boxed{} .$$

HIDDEN HINT 1 for (12-9.e) → 12-9-5-0:vdpxhx.pdf

SOLUTION for (12-9.e) → 12-9-5-1:.pdf ▲

References

[Ran15] Joachim Rang. "Improved traditional Rosenbrock-Wanner methods for stiff ODEs and DAEs". In: *J. Comput. Appl. Math.* 286 (2015), pp. 128–144. DOI: 10.1016/j.cam.2015.03.010 (cit. on pp. 363, 364).

End Problem 12-9 , 90 min.

Problem 12-10: A Runge-Kutta Single-Step Method

In this problem we review some basic facts about Runge-Kutta single-steps method and solve a second-order ODE.

This problem requires familiarity with Runge-Kutta single-step method as introduced in [Lecture → Section 11.4]

Consider the Runge-Kutta method with the Butcher tableau

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^\top \end{array} := \begin{array}{c|cc} 0 & 0 & 0 \\ 2/3 & 2/3 & 0 \\ \hline & 1/4 & 3/4 \end{array} \tag{12.10.1}$$

(12-10.a) ⚀ (5 min.) Is the Runge-Kutta scheme (12.10.1) explicit or implicit?

◯ Explicit ◯ Implicit

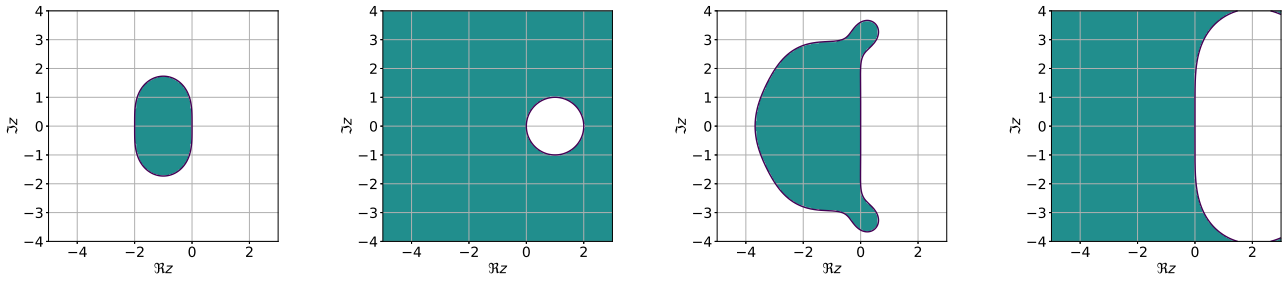
SOLUTION for (12-10.a) → 12-10-1-0:odeexas.pdf ▲

(12-10.b) ⚁ (5 min.) Write down the stability function $z \mapsto S(z)$ of scheme (12.10.1).

SOLUTION for (12-10.b) → 12-10-2-0:odeexbs.pdf ▲

(12-10.c) ⚂ (10 min.) Which of the colored areas shown below is the (absolute) stability domain of the RK-SSM (12.10.1)?

(I) (II) (III) (IV)



◯ ◯ ◯ ◯

SOLUTION for (12-10.c) → 12-10-3-0:odeexcs.pdf ▲

(12-10.d) ⚃ (20 min.) For *the first and second* of the stability domains in the picture above, give the range of step size h such that the corresponding RK-SSM method converges for any initial-value problem for the linear scalar ODE $\dot{y} = \lambda y$ with $\lambda \in \mathbb{R}$, $\lambda < 0$.

SOLUTION for (12-10.d) → 12-10-4-0:odeexds.pdf ▲

The file `rkintegrator.hpp` implements a header-only C++ class

```
template <class State, class RHSFunctor>
class RKIntegrator {
public:
    // Constructor
    RKIntegrator(RHSFunctor& f, const Eigen::MatrixXd & A, const
       Eigen::VectorXd & b)
```

```
        : f_(f), A_(A), b_(b) {};
    // Explicit Runge-Kutta numerical integrator
    State step(double h, const State& y) const;
    std::vector<State> solve(double T, const State &y0, int N) const;

private:
    RHSFunctor& f_;
    Eigen::MatrixXd A_; // Butcher matrix 𝔄
    Eigen::VectorXd b_; // Weight vector b
};
```

which provides a *generic* Runge-Kutta single-step method (→ [Lecture → Def. 11.4.0.11]) given by a Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^\top \end{array}$, $\mathbf{c},\mathbf{b} \in \mathbb{R}^s$, $\mathfrak{A} \in \mathbb{R}^{s,s}$ *strictly lower triangular*, to solve the autonomous initial-value problem $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(t_0) = \mathbf{y}_0$. The matrix $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the vector $\mathbf{b} \in \mathbb{R}^s$ from the Butcher scheme and the right-hand side vector field $\mathbf{f}$ are passed to the constructor. Objects of **RHSFunction** class should provide an evaluation operator of the form

```
State operator()(State);
```

(12-10.e) ⚀⚀ (30 min.) Finish the implementation of the member function `step()` which takes arguments `h` time-step size and `y` current state, and returns the state at the next time step.

SOLUTION for (12-10.e) → `12-10-5-0:odeexes.pdf` ▲

(12-10.f) ⚀ (20 min.) Complete the implementation of the member function `solve()` which takes arguments `T` final time, `y0` initial state, and `N` the number of *equidistant timesteps*, and returns the sequence $(\mathbf{y}_k)_{k=0}^N$ of states generated by the explicit Runge-Kutta method. Function `step()` implemented in the previous subproblem can be invoked.

SOLUTION for (12-10.f) → `12-10-6-0:odeexfs.pdf` ▲

(12-10.g) ⚀ (15 min.) Consider the following initial value problem for a second-order system of ordinary differential equations in the time interval $[0,T]$:

$$\ddot{\mathbf{u}} + \mathbf{M}_1\dot{\mathbf{u}} + \mathbf{M}_2\mathbf{u} = 0, \quad \mathbf{u}(0) = \mathbf{u}_0, \quad \dot{\mathbf{u}}(0) = \mathbf{v}_0, \tag{12.10.4}$$

where $\mathbf{u} = \mathbf{u}(t) \in \mathbb{R}^d$ is a time-dependent vector-valued variable and $\mathbf{M}_1, \mathbf{M}_2 \in \mathbb{R}^{d,d}$ constant matrices. Here the notation $\dot{\mathbf{u}}$ and $\ddot{\mathbf{u}}$ designates the first and the second derivative of a time-dependent function $t \mapsto \mathbf{u}(t)$.

Write (12.10.4) as a first-order IVP of the form $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{y}(0) = \mathbf{y}_0$.

HIDDEN HINT 1 for (12-10.g) → `12-10-7-0:auxvar.pdf`

SOLUTION for (12-10.g) → `12-10-7-1:odeexgs.pdf`

▲

(12-10.h) ⚀ (10 min.) [depends on Sub-problem (12-10.f), Sub-problem (12-10.g)]

Implement a function

```
void testcvgRK()
```

that tests the convergence rate of the Runge-Kutta scheme (12.10.1) applied to the second-order initial-value problem (12.10.4) with the parameters and initial values

$$\mathbf{M}_1 = \begin{bmatrix} 1 & 2 \\ 1 & 0 \end{bmatrix}, \quad \mathbf{M}_2 = \begin{bmatrix} -1 & 0 \\ 2 & 2 \end{bmatrix}, \quad \mathbf{u}(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \dot{\mathbf{u}}(0) = \begin{bmatrix} 2 \\ 3 \end{bmatrix}.$$

Set the final time $T = 1.0$. What is the *rate of algebraic convergence* you observe?

rate $=$ ☐ .

SOLUTION for (12-10.h) → 12-10-8-0:odeexhs.pdf ▲

End Problem 12-10 , 115 min.

Problem 12-11: Implicit Runge-Kutta Single-Step Method

In [Lecture → Chapter 12] we learned that the use of *implicit* Runge-Kutta single step methods (IRK-SSMs) is mandatory for the efficient numerical integration of *stiff* initial value problems for ordinary differential equations. In this problem we examine the details of an implementation of a generic IRK-SSM.

This problem assumes familiarity with [Lecture → Section 12.3.3] and [Lecture → Section 8.5.4]. Small-scale coding in C++ is part of it.

Recall that a general s-stage, $s \in \mathbb{N}$, Runge-Kutta single-step method according to [Lecture → Def. 12.3.3.1] is fully characterized by its **Butcher scheme**,

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} := \begin{array}{c|ccc} c_1 & a_{11} & \cdots & a_{1s} \\ \vdots & \vdots & & \vdots \\ c_s & a_{s1} & \cdots & a_{ss} \\ \hline & b_1 & \cdots & b_s \end{array} , \qquad [\text{Lecture} \rightarrow (12.3.3.3)]$$

with the Butcher matrix $\mathfrak{A} = \left[a_{i,j}\right]_{i,j=1}^{s}$, $\mathbf{c} \in \mathbb{R}^s$, and weight vector $\mathbf{b} = [b_i]_{i=1}^{s} \in \mathbb{R}^s$.

(12-11.a) ⚂ (25 min.) We apply the general s-stage RK-SSM given by the Butcher scheme $\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array}$, to the *non-autonomous* linear ODE $\mathbf{M}\dot{\mathbf{y}} + \mathbf{T}\mathbf{y} = \mathbf{q}(t)$ with state space $\mathbb{R}^N$, $N \in \mathbb{N}$, and a function $\mathbf{q} : \mathbb{R} \to \mathbb{R}^N$. Here $\mathbf{M}, \mathbf{T} \in \mathbb{R}^{N,N}$ are symmetric positive definite (s.p.d.) matrices.

Complete the following formulas describing one step $t_0 \to t_1$, $\mathbf{y}_0 \to \mathbf{y}_1$ of the RK-SSM for that ODE:

$$\mathbf{k}_i \in \mathbb{R}^N: \quad \begin{bmatrix} \mathbf{X}_{1,1} & \mathbf{X}_{1,2} & \dots & \dots & \mathbf{X}_{1,s} \\ \mathbf{X}_{2,1} & \mathbf{X}_{2,2} & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ \mathbf{X}_{s,1} & \mathbf{X}_{s,2} & \dots & \dots & \mathbf{X}_{s,s} \end{bmatrix} \begin{bmatrix} \mathbf{k}_1 \\ \mathbf{k}_2 \\ \vdots \\ \vdots \\ \mathbf{k}_s \end{bmatrix} = \begin{bmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \\ \vdots \\ \vdots \\ \mathbf{b}_s \end{bmatrix} ,$$

$$\mathbf{y}_1 := \mathbf{y}_0 + h \sum_{i=1}^{s} b_i \mathbf{k}_i, \quad h := t_1 - t_0 ,$$

with $\mathbf{X}_{i,j} =$ ☐ $\in \mathbb{R}^{N,N}$, $i,j \in \{1,\dots,s\}$,

$\mathbf{b}_i =$ ☐ $\in \mathbb{R}^N$, $i \in \{1,\dots,s\}$.

No inverses of matrices must occur in the formulas!

SOLUTION for (12-11.a) → 12-11-1-0:irkes1.pdf ▲

The following data type has been introduced to store the coefficients of any Runge-Kutta single step method according to [Lecture → Def. 12.3.3.1].

C++ code 12.11.1: Data type meant for storing coefficients of a Runge-Kutta method

```
2 struct RKData {
```

```
  RKData() = delete;
  RKData(const RKData &) = default;
  RKData &operator=(const RKData &) = default;
  virtual ~RKData() = default;

  RKData(Eigen::MatrixXd A, Eigen::VectorXd b) : A_(A), b_(b) {
    s_ = A_.rows();
    assertm(A_.cols() == s_, "Butcher matrix must be square");
    assertm(b.size() == s_, "Size mismatch for weight vector");
  }
  unsigned int s_;     // Number of stages
  Eigen::MatrixXd A_;  // Butcher matrix
  Eigen::VectorXd b_;  // Weight vector
};

/** Make RKData object for implicit Euler scheme */
RKData make_implicit_Euler(void) {
  Eigen::MatrixXd A(1, 1);
  A << 1.0;
  Eigen::VectorXd b(1);
  b[0] = 1.0;
  return RKData(A, b);
}
```

The coefficients c_i are not stored, but assumed to be computed by the formula $c_i = \sum_{j=1}^{s} a_{i,j}$, $i \in \{1,\ldots,s\}$.

(12-11.b) ⚄ (75 min.) The C++ function `IRKSSMSolver()` in the file `irkssm.hpp`,

```
template <typename RECORDER = std::function<
              void(double, const Eigen::VectorXd &, unsigned int)>>
Eigen::VectorXd IRKSSMSolver(
    const RKData &RKSSM_BS,
    std::function<Eigen::VectorXd(const Eigen::VectorXd &)> f_rhs,
    std::function<Eigen::MatrixXd(const Eigen::VectorXd &)> f_Jac,
    const Eigen::VectorXd &y0, double T_final, unsigned int M,
    double newt_reltol, double newt_abstol,
    RECORDER rec = [](double, const Eigen::VectorXd &, unsigned
       int) {});
```

is supposed to implement a numerical integrator for an initial-value problem for the *autonomous* ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ on a finite time interval $[0,T]$. It relies on an implicit Runge-Kutta single step method whose coefficients are passed through the `RKSSM_BS` argument. The other parameters are

- `f_rhs`: a functor object providing $\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^N$,
- `f_Jac`: a functor encoding the Jacobian $D\mathbf{f} : \mathbb{R}^N \to \mathbb{R}^{N,N}$,
- `y0`: the initial state vector $\mathbf{y}_0 \in \mathbb{R}^N$,
- `T_final`: the final time $T > 0$,
- `M`: the number $M \in \mathbb{N}$ of *uniform* timesteps,
- `newt_reltol`, `newt_abstol`: tolerances for the termination of Newton's method,
- `rec`: an object for recording the sequence of states generated by the numerical integrator.

The function returns an approximation of the final state at time $t = T$.

In each time step, a nonlinear system of equations $F(x) = 0$ needs to be solved. The associated lambda function `F()` and `DF()` have not be implemented yet and it is your task to do this.

SOLUTION for (12-11.b) → `12-11-2-0:rsmmembs2.pdf` ▲

(12-11.c) ⚂ (25 min.) The following C++ function (not implemented!) prints a single line

```
Error ratio = ...
```

to the terminal.

C++ code 12.11.5: Output ratio or error norms

```
void test_Output(const RKData &RKSSM) {
  // Smooth right-hand side for autonomous ODE
  auto f_rhs = [](Eigen::VectorXd y) -> Eigen::VectorXd {
    assertm(y.size() == 3, "3D state space!");
    return ((Eigen::VectorXd(3) << (-y[0] - y[1] + std::cos(y[2])),
             (y[0] - y[1] + std::sin(y[2])), 1)
                .finished());
  };
  // Jacobian of right-hand side function
  auto f_Jac = [](Eigen::VectorXd y) -> Eigen::MatrixXd {
    assertm(y.size() == 3, "3D state space!");
    return ((Eigen::MatrixXd(3, 3) << -1.0, -1.0, -std::sin(y[2]), 1.0, -1.0,
             std::cos(y[2]), 0.0, 0.0, 0.0)
                .finished());
  };
  // Initial value
  Eigen::VectorXd y0 = (Eigen::VectorXd(3) << 1.0, 0.0, 0.0).finished();
  // Known exact solution for given initial value
  auto y_exact = [](double t) -> Eigen::Vector3d {
    return Eigen::Vector3d(std::cos(t), std::sin(t), t);
  };
  // Tolerances for Newton's method
  const double reltol = 1.0E-10;
  const double abstol = reltol / 100.0;
  Eigen::VectorXd y_H =
      IRKSSMSolver(RKSSM, f_rhs, f_Jac, y0, 1.0, 100, reltol, abstol);
  Eigen::VectorXd y_h =
      IRKSSMSolver(RKSSM, f_rhs, f_Jac, y0, 1.0, 200, reltol, abstol);
  std::cout << "Error ratio = "
            << (y_H - y_exact(1.0)).norm() / (y_h - y_exact(1.0)).norm()
            << std::endl;
}
```

With *two-digit* precision predict the output for different implicit Runge-Kutta single step methods:

- **Implicit Euler method**:

  `Error ratio =` ☐ ,

- 2-stage **Radau RK-SSM** [Lecture → Ex. 12.3.4.21]:

  `Error ratio =` ☐ ,

- 2-stage **Gauss collocation RK-SSM** [Lecture → Section 12.3.2]:

  `Error ratio =` ☐ .

SOLUTION for (12-11.c) → 12-11-3-0:rkssmes3.pdf ▲

(12-11.d) ⚂ (35 min.) The C++ function

```
double quadByIntegration(double T, unsigned int M);
```

that you can find in the file `irkssm.hpp` calls `IRKSSMSolver()` to approximately evaluate the integral

$$\int_0^T e^{\sin(t)}\,\mathrm{d}t\,.$$

The parameter `T` supplies the integration bound $T > 0$, whereas `M` is the number of uniform timesteps.

You are asked to supply the missing function bodies for the two lambda functions supplying the `f_rhs` and `f_Jac` parameters of `IRKSSMSolver()` and to specify the initial value.

HIDDEN HINT 1 for (12-11.d) → 12-11-4-0:irkssmh7.pdf

HIDDEN HINT 2 for (12-11.d) → 12-11-4-1:irkssmh8.pdf

SOLUTION for (12-11.d) → 12-11-4-2:.pdf ▲

End Problem 12-11 , 160 min.