

ETH Lecture 401-0663-00L Numerical Methods for CSE

# Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2020  
(C) Seminar für Angewandte Mathematik, ETH Zürich

[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.

### 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 1

## Introduction

### Problem 1-1: 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.

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

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

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

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

that returns the  $2 \times 2$ -matrix  $\begin{bmatrix} a & b \\ 0 & c \end{bmatrix}$ .

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

SOLUTION for (1-1.b) → [1-1-2-1:s1.pdf](#) ▲

- (1-1.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 (1-1.c) → [1-1-3-0:ctr.pdf](#)

SOLUTION for (1-1.c) → [1-1-3-1:s2.pdf](#) ▲

**(1-1.d)** ☐ 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  $2(1 - i) + 2(5 + i)$ . To that end use EIGEN's matrix multiplication.

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

**(1-1.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

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

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

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

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

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

SOLUTION for (1-1.e) → [1-1-5-1:s3.pdf](#) ▲

**End Problem 1-1 , 60 min.**

### Problem 1-2: 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.

(1-2.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 (1-2.a) → [1-2-1-0:t1.pdf](#)

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

(1-2.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!)

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

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

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

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

(1-2.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

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

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 (1-2.c) → [1-2-3-0:t3.pdf](#)

SOLUTION for (1-2.c) → [1-2-3-1:s3.pdf](#)



**End Problem 1-2 , 55 min.**

### Problem 1-3: EIGEN's reduction operations on matrices

Eigen 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 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`.

(1-3.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 (1-3.a) → [1-3-1-0:s1h1.pdf](#)

HIDDEN HINT 2 for (1-3.a) → [1-3-1-1:tavg.pdf](#)

SOLUTION for (1-3.a) → [1-3-1-2:s1.pdf](#) ▲

(1-3.b)  (20 min.) Supply the missing parts of the C++ function

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

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

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

HIDDEN HINT 2 for (1-3.b) → [1-3-2-1:pzt.pdf](#)

SOLUTION for (1-3.b) → [1-3-2-2:s2.pdf](#) ▲

(1-3.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 (1-3.c) → [1-3-3-0:s3h1.pdf](#)

HIDDEN HINT 2 for (1-3.c) → [1-3-3-1:hzct.pdf](#)

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

(1-3.d)  (25 min.) Finish the implementation of the C++ function

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

that returns a matrix that agrees with the *square* matrix passed in `A` except for the diagonal. Its diagonal entries are set in way that makes all row sums vanish.

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

SOLUTION for (1-3.d) → [1-3-4-1:s4.pdf](#)



**End Problem 1-3 , 80 min.**

# Chapter 2

## Computing with Matrices and Vectors

### Problem 2-1: Arrow matrix $\times$ vector multiplication

Innocent looking linear algebra operation can burn considerable CPU power when implemented carelessly, see [Lecture  $\rightarrow$  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  $\rightarrow$  Rem. 1.3.1.5].

Related to [Lecture  $\rightarrow$  Section 1.4.3]

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 2.1.1: Computing $\mathbf{A}^2\mathbf{y}$ for an arrow matrix $\mathbf{A}$

```
2 void arrow_matrix_2_times_x(const VectorXd &d, const VectorXd &a,
3                             const VectorXd &x, VectorXd &y) {
4     assert(d.size() == a.size() && a.size() == x.size() &&
5            "Vector size must be the same!");
6     int n = d.size();
7     VectorXd d_head = d.head(n - 1);
8     VectorXd a_head = a.head(n - 1);
9     MatrixXd d_diag = d_head.asDiagonal();
10    MatrixXd A(n, n);
11    A << d_diag, a_head, a_head.transpose(), d(n - 1);
12    y = A * A * x;
13 }
```

Get it on  GitLab (arrowmatvec.cpp).

(2-1.a)  (10 min.) For general vectors  $\mathbf{d} = [d_1, \dots, d_n]^T$  and  $\mathbf{a} = [a_1, \dots, a_n]^T$  sketch the pattern of nonzero entries of the matrix  $\mathbf{A}$  created in the function `arrow_matrix_2_times_x` in Code 2.1.1. [Lecture  $\rightarrow$  § 1.1.2.2] shows a way to visualize that pattern.

HIDDEN HINT 1 for (2-1.a)  $\rightarrow$  2-1-1-0:arrmatha.pdf

SOLUTION for (2-1.a)  $\rightarrow$  2-1-1-1:arrws.pdf

(2-1.b)  (15 min.) [ depends on Sub-problem (2-1.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].

## Timings of arrow\_matrix\_2\_times\_x

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 (`arrowmatvec.cpp`)).

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.

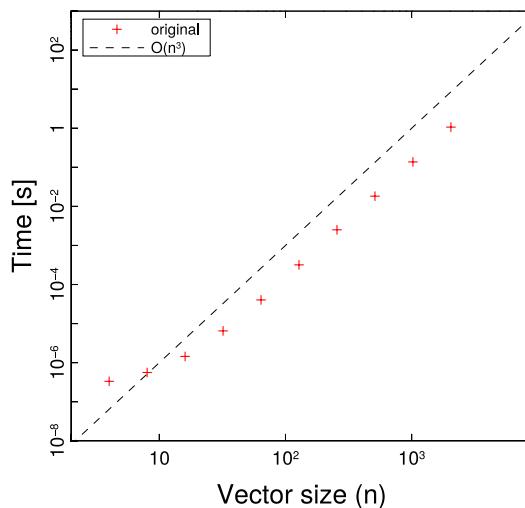


Fig. 1

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

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

Write an *efficient* C++ function

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

that computes the same product as in Code 2.1.1, 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  $[d_1, \dots, d_n]^T$  and  $a$  passes the vector  $[a_1, \dots, a_n]^T$ .

SOLUTION for (2-1.c) → [2-1-3-0:arrwi.pdf](#) ▲

(2-1.d)  (5 min.) [ depends on Sub-problem (2-1.c) ]

What is the asymptotic complexity of your algorithm from sub-problem (2-1.c) (with respect to matrix size  $n$ )?

SOLUTION for (2-1.d) → [2-1-4-0:arrwce.pdf](#) ▲

(2-1.e)  (20 min.) [ depends on Sub-problem (2-1.c) ]

Compare the runtime of your implementation and the implementation given in Code 2.1.1 for  $n = 2^5, \dots, 2^{12}$ . 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:

```
1 std::cout << std::setw(8) << 1./3e4
2     << std::scientific << std::setprecision(3)
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.

**§2.1.6 (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++11-code 2.1.7: Usage of Timer

```

1 Timer t ;
2 t.start() ;
3 // HERE CODE TO TIME
4 t.stop() ;
5
6 // Now you can get the time passed between start and stop using
7 t.duration() ;
8 // You can start() and stop() again, a number of times
9 // Ideally: repeat experiment many times and use min() to obtain the
10 // fastest run
11 t.min() ;
12 // You can also obtain the mean():
13 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++11). The `chrono` header provides the function:

#### C++11-code 2.1.8: `now()` function

```

1 std::chrono::high_resolution_clock::time_point
2     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++11-code 2.1.9: `now()` function

```

1 using time_point_t = std::chrono::high_resolution_clock::time_point;
2 // Declare a starting point and a termination time
3 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`:

```

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

```

To obtain the actual number (as integer) used to represent `elapsed`, 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 these units. Make sure you use a sufficiently “refined” unit of measure (e.g. `std::chrono::nanoseconds`). □

SOLUTION for (2-1.e) → [2-1-5-0:arrwc.pdf](#) ▲

**End Problem 2-1 , 65 min.**

### Problem 2-2: Gram-Schmidt orthonormalization with EIGEN

In this exercise we rely on EIGEN to implement a fundamental algorithm from linear algebra, Gram-Schmidt orthonormalization, see [NIS02], in order to practice the use of EIGEN's basic vector and matrix operations.

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.

**(2-2.a)** (10 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 (2-2.a) → [2-2-1-0:grsi.pdf](#)

**(2-2.b)** (20 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]:

```
1 MatrixXd gram_schmidt (const MatrixXd &A);
```

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

Use EIGEN's block operations see [EIGEN documentation](#).

SOLUTION for (2-2.b) → [2-2-2-0:grsi.pdf](#)

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

Implement a function

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

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

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

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

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

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

HIDDEN HINT 3 for (2-2.c) → [2-2-3-2:sp2QQ.pdf](#)

SOLUTION for (2-2.c) → [2-2-3-3:grst.pdf](#)

**End Problem 2-2 , 45 min.**

### Problem 2-3: 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].

#### 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

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

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

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

and

$$\mathbf{B} = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix}.$$

SOLUTION for (2-3.a) → [2-3-1-0:cmpm.pdf](#)

**(2-3.b)**  (20 min.) Implement a C++ function

```
void kron(const MatrixXd & A, const MatrixXd & B, MatrixXd & C);
```

that computes the Kronecker product of the argument matrices A and B and stores the result in the matrix C. You can use Eigen “block” operations, see the  **EIGEN documentation**.

SOLUTION for (2-3.b) → [2-3-2-0:cmpk.pdf](#)

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

What is the **asymptotic complexity** ([Lecture → Def. 1.4.1.1]) in terms of the problem size parameter  $n \rightarrow \infty$  of your implementation of **kron** from (2-3.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 (2-3.c) → [2-3-3-0:cmpk1.pdf](#)

**(2-3.d)**  (30 min.) In general, computing the entire matrix is unnecessary. Devise an *efficient* implementation of an EIGEN-based C++ function

```
void kron_mult(const MatrixXd & A, const MatrixXd & B, 1
const VectorXd & x, VectorXd & y);
```

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 (2-3.d) → [2-3-4-0:h1.pdf](#)

SOLUTION for (2-3.d) → [2-3-4-1:cmpk.pdf](#) ▲

**(2-3.e)** (30 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 “reshaping” functions. The function to implement is:

```
void kron_reshape(const MatrixXd &A, const MatrixXd &B,
                  const VectorXd &x, VectorXd &y);
```

Study [Lecture → Rem. 1.2.3.6] about “reshaping” matrices in EIGEN. See also the [EIGEN documentation](#).

HIDDEN HINT 1 for (2-3.e) → [2-3-5-0:ss3rs.pdf](#)

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

**(2-3.f)** Compare the runtimes of your implementations in sub-problems (2-3.b), (2-3.d) and (2-3.e). To this end, measure the runtime of the functions `kron`, `kron_mult`, `kron_reshape`, with  $n = 2^1, \dots, 2^8$ . 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 (2-3.f) → [2-3-6-0:cmpk2.pdf](#) ▲

**End Problem 2-3 , 100 min.**

### Problem 2-4: 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.

**Template:** Get it on GitLab., **Solution:** Get it on GitLab.

- (2-4.a) Using EIGEN , implement a **recursive** function

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

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

You can use the template we provide. Get it on GitLab (strassen.hpp).

SOLUTION for (2-4.a) → 2-4-1-0:fastmi.pdf

- (2-4.b) Validate the correctness of your code by comparing the result with EIGEN's built-in matrix multiplication.

You can use the template. Get it on GitLab (test.cpp).

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

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

- (2-4.c) Measure the runtime of your function strassenMatMult for random matrices of sizes  $2^k$ ,  $k = 4, \dots, 9$ , 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 2-1, § 2.1.6. Base your code on the supplied template. Get it on GitLab (test.cpp).

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 (2-4.c) → 2-4-3-0:fastmt.pdf

**End Problem 2-4**

### Problem 2-5: 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 [NIS02], 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.

- (2-5.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 2.5.1: Householder reflection

```
w ← v / ||v||₂
u ← w
u₁ ← u₁ + 1
q ← u / ||u||₂
X ← I - 2qqᵀ
Z ← ((X)[:,2:n]
```

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

```
void houseref1(const VectorXd &v, MatrixXd &Z);
```

that implements the algorithm from Pseudocode 2.5.1.

HIDDEN HINT 1 for (2-5.a) → [2-5-1-0:hrfH1.pdf](#)

SOLUTION for (2-5.a) → [2-5-1-1:impl.pdf](#)

- (2-5.b)** (15 min.) Show that the matrix  $\mathbf{X}$ , defined in Code 2.5.1, 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 (2-5.b) → [2-5-2-0:norm.pdf](#)

SOLUTION for (2-5.b) → [2-5-2-1:orth.pdf](#)

- (2-5.c)** (15 min.) Show that the first column of  $\mathbf{X}$ , in Code 2.5.1, is a multiple of the vector  $\mathbf{v}$ .

HIDDEN HINT 1 for (2-5.c) → [2-5-3-0:mulh.pdf](#)

SOLUTION for (2-5.c) → [2-5-3-1:muls.pdf](#)

- (2-5.d)** (10 min.) [ depends on Sub-problem (2-5.b) ]

What property does the set of columns of the matrix  $\mathbf{Z}$  have? What is the purpose of the function `houseref1`?

SOLUTION for (2-5.d) → [2-5-4-0:propertyimpl.pdf](#)

- (2-5.e)** (10 min.) [ depends on Sub-problem (2-5.a) ]

What is the asymptotic complexity of the function `houseref1` as the length  $n$  of the input vector  $\mathbf{v}$  tends to  $\infty$ ?

Specify it in leading order using the Landau symbol  $\mathcal{O}$ .

SOLUTION for (2-5.e) → [2-5-5-0:impl.pdf](#)



**End Problem 2-5 , 70 min.**

### Problem 2-6: Matrix powers

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

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

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

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

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

**(2-6.a)** (30 min.) Implement a C++ function

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

that, using only basic linear algebra operations (including matrix-vector or matrix-matrix multiplications), computes the  $k^{\text{th}}$  power of the  $n \times n$  matrix  $\mathbf{A}$  as efficiently as possible. You can use the provided template. [Get it on GitLab \(matPow.cpp\)](#).

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

HIDDEN HINT 1 for (2-6.a) → [2-6-1-0:.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 validation the EIGEN implementation of Matrix power (`MatrixXcd::pow(unsigned int)`) can be used writing:

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

SOLUTION for (2-6.a) → [2-6-1-1:powi.pdf](#) ▲

**(2-6.b)** (20 min.) Find (in leading order and state using the Landau symbol  $O$ ) the asymptotic complexity in terms of  $k \rightarrow \infty$  (and  $n \rightarrow \infty$ ) of your implementation of `matPow()` taking into account that in EIGEN a matrix-matrix multiplication requires a  $O(n^3)$  computational effort for  $n \rightarrow \infty$ .

SOLUTION for (2-6.b) → [2-6-2-0:powc.pdf](#) ▲

**(2-6.c)**  (30 min.) Compute the runtime of the EIGEN matrix power function (`Eigen::MatrixXcd::pow(int)`) ( $\rightarrow$   [EIGEN documentation](#)) and find out the complexity w.r.t  $k$ . Compare this with the function `matPow()` from Sub-problem (2-6.a). For  $n > 0$ , use the complex  $n \times n$  matrix  $\mathbf{A}$  defined by

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

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

You should use the class **Timer** or the STL library **std::chrono** as explained in Problem 2-1.

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  $k = 2, \dots, 2^{31}$ . Output the time in seconds, using 3 decimal digits in scientific notation.

SOLUTION for (2-6.c)  $\rightarrow$  [2-6-3-0:powr.pdf](#)



**End Problem 2-6**, 80 min.

### Problem 2-7: 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

$$(\mathbf{A})_{i,j} = a_{i,j} = \min\{i, j\}, i, j = 1, \dots, n. \quad (2.7.1)$$

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

#### C++11-code 2.7.2: Computing $\mathbf{y} = \mathbf{Ax}$ for $\mathbf{A}$ from (2.7.1)

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

Get it on  GitLab (multAmin.cpp).

**(2-7.a)**  (10 min.) What is the asymptotic complexity (for  $n \rightarrow \infty$ ) of the evaluation of the C++ code displayed above, with respect to the problem size parameter  $n$ ?

SOLUTION for (2-7.a) → [2-7-1-0:strc.pdf](#)

**(2-7.b)**  (30 min.) Write an *efficient* C++ function

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

that computes the same multiplication as Code 2.7.2 but with a better asymptotic complexity with respect to  $n$  (you can use the template). Get it on  GitLab (multAmin.cpp).

You can test your implementation by comparing the returned values with the ones obtained with Code 2.7.2.

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

SOLUTION for (2-7.b) → [2-7-2-1:stri.pdf](#)

**(2-7.c)**  (10 min.) What is the asymptotic complexity (in terms of problem size parameter  $n$ ) of your function `multAmin`?

SOLUTION for (2-7.c) → [2-7-3-0:cmpk.pdf](#)

**(2-7.d)**  (20 min.) Compare the runtimes of your implementation and the implementation given in Code 2.7.2 for  $n = 2^5, \dots, 2^{12}$ .

You can use the class `Timer` or the STL class `std::chrono` as explained in Problem 2-1, § 2.1.6.

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

SOLUTION for (2-7.d) → [2-7-4-0:cmpk.pdf](#)

**(2-7.e)**  (10 min.) Consider the following C++ snippet:

#### C++11-code 2.7.6: Initializing $\mathbf{B}$

```
2   MatrixXd B = 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;
6     if(i > 0) B(i-1,i) = -1;
7   }
8   B(n-1,n-1) = 1;

```

Get it on ❤️ GitLab (multAmin.cpp).

Sketch the matrix **B** created by these lines.

SOLUTION for (2-7.e) → 2-7-5-0:cmpk.pdf ▲

(2-7.f) ☐ (15 min.) With **A** from (2.7.1) and **B** from (2-7.e) write a short EIGEN-based C++ function

`Eigen::MatrixXd multABunitv();`

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

Formulate a conjecture based on your observations.

SOLUTION for (2-7.f) → 2-7-6-0:cmpkinv.pdf ▲

**End Problem 2-7 , 95 min.**

### Problem 2-8: 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++.

**(2-8.a)** (10 min.)

We consider the function

$$f_1(x_0, h) := \sin(x_0 + h) - \sin(x_0). \quad (2.8.1)$$

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 (2-8.a) → [2-8-1-0:A1h1.pdf](#)

SOLUTION for (2-8.a) → [2-8-1-1:sA1.pdf](#) ▲

**(2-8.b)** (20 min.) [ depends on Sub-problem (2-8.a) ]

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

$$f'(x) \approx \frac{f(x_0 + h) - f(x_0)}{h} \quad (2.8.3)$$

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

Write a C++ function

```
void sinederv();
```

that implements

1. the difference quotient (2.8.3) directly, and
2. your new formula

and computes the resulting approximation of  $f'(1.2)$ , for  $h = 1 \cdot 10^{-20}, 1 \cdot 10^{-19}, \dots, 1$ . 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 (2-8.b) → [2-8-2-0:A2h1.pdf](#)

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

**(2-8.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 (2-8.c) → [2-8-3-0:logcnch.pdf](#)

SOLUTION for (2-8.c) → [2-8-3-1:canclog.pdf](#) ▲

**(2-8.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 (2-8.d) → 2-8-4-0:cancsqr.pdf



**End Problem 2-8 , 60 min.**

### Problem 2-9: 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 2-6.

**Template:** Get it on GitLab. **Solution:** Get it on GitLab.

We consider the C++ function defined in `getit.cpp` (cf. Code 2.9.1). [Get it on GitLab \(getit.cpp\)](#).

#### C++11-code 2.9.1: Code for `getit`.

```

2  VectorXd getit(const MatrixXd &A, const VectorXd &x, unsigned int k) {
3
4      EigenSolver<MatrixXd> eig = EigenSolver<MatrixXd>(A);
5      const VectorXcd & V = eig.eigenvalues();
6      const MatrixXcd & W = eig.eigenvectors();
7
8      VectorXcd cx = x.cast<std::complex<double>>();
9
10     VectorXcd ret = W *
11         (
12             V.array().pow(k) *
13             (W.partialPivLu().solve(cx)).array()
14         ).matrix();
15
16     return ret.real();
17 }
```

[Get it on GitLab \(getit.cpp\)](#).

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  $W \cdot x = cx$  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](#).

**(2-9.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 (2-9.a) → [2-9-1-0:eigpoww.pdf](#)

**(2-9.b)** Fix  $k \in \mathbb{N}$ . Discuss (in detail) the asymptotic complexity of `getit` for  $n \rightarrow \infty$ .

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

SOLUTION for (2-9.b) → [2-9-2-0:eigpowc.pdf](#)

**End Problem 2-9**

### Problem 2-10: 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].

This problem discusses a case, in which Taylor approximation has to be employed in order to curb the impact of cancellation, as discussed in [Lecture → Ex. 1.5.4.26].

Consider the C++ code given in Code 2.10.1.

#### C++11-code 2.10.1: Implementation of sinh

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

Get it on  GitLab ([sinh.cpp](#)).

**(2-10.a)**  (20 min.) Explain why the function given in Listing 2.10.1 may not give a good approximation of the hyperbolic sine for small values of  $x$ , and compute the relative error

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

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

```
void siinhError();
```

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

SOLUTION for (2-10.a) → [2-10-1-0:sinhex.pdf](#) ▲

**(2-10.b)**  (15 min.) Write the Taylor expansion with  $m$  terms around  $x = 0$  of the function  $e^x$ . Specify the remainder, *cf.* [Lecture → Ex. 1.5.4.26].

SOLUTION for (2-10.b) → [2-10-2-0:sinhwrt.pdf](#) ▲

**(2-10.c)**  (10 min.) Prove that, for every  $x \geq 0$  the following inequality holds true:

$$\sinh x \geq x. \quad (2.10.3)$$

SOLUTION for (2-10.c) → [2-10-3-0:sinhleq.pdf](#) ▲

**(2-10.d)**  (30 min.) Based on the Taylor expansion, find an approximation for  $\sinh(x)$ , for every  $0 \leq x \leq 10^{-3}$ , so that the relative approximation error  $\epsilon_{\text{rel}}$  is smaller than  $10^{-15}$ . Follow the considerations of [Lecture → Ex. 1.5.4.26].

SOLUTION for (2-10.d) → [2-10-4-0:sinhty.pdf](#) ▲

**End Problem 2-10 , 75 min.**

### Problem 2-11: Complex roots

This problem deals with complex numbers and cancellation errors.

Related to [Lecture → Section 1.5.4].

Given a complex number  $w = u + iv$ ,  $u, v \in \mathbb{R}$  with  $v \geq 0$ , its root  $\sqrt{w} = x + iy$  can be defined by

$$x := \sqrt{(\sqrt{u^2 + v^2} + u)/2}, \quad (2.11.1)$$

$$y := \sqrt{(\sqrt{u^2 + v^2} - u)/2}. \quad (2.11.2)$$

Here,  $\sqrt{-1} =: i$ .

**(2-11.a)** (15 min.) For what  $w \in \mathbb{C}$  will the direct implementation of (2.11.1) and (2.11.2) be vulnerable to cancellation?

SOLUTION for (2-11.a) → [2-11-1-0:root1.pdf](#)

**(2-11.b)** (5 min.) Compute  $xy$  as an expression of  $u$  and  $v$ .

SOLUTION for (2-11.b) → [2-11-2-0:root2.pdf](#)

**(2-11.c)** (15 min.) Implement a function

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

that computes the root of  $w$  as given by (2.11.1) and (2.11.2) 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 (2-11.c) → [2-11-3-0:root3.pdf](#)

**End Problem 2-11 , 35 min.**

### Problem 2-12: 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  $\mathbf{D}_\mathbf{A}, \mathbf{L}_\mathbf{A}, \mathbf{U}_\mathbf{A} \in \mathbb{R}^{n,n}$  by:

$$(\mathbf{D}_\mathbf{A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i = j, \\ 0, & i \neq j, \end{cases}, (\mathbf{L}_\mathbf{A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i > j, \\ 0, & i \leq j, \end{cases}, (\mathbf{U}_\mathbf{A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i < j, \\ 0, & i \geq j. \end{cases} \quad (2.12.1)$$

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

$$\mathbf{x}^{(k+1)} = (\mathbf{U}_\mathbf{A} + \mathbf{D}_\mathbf{A})^{-1}\mathbf{b} - (\mathbf{U}_\mathbf{A} + \mathbf{D}_\mathbf{A})^{-1}\mathbf{L}_\mathbf{A}(\mathbf{L}_\mathbf{A} + \mathbf{D}_\mathbf{A})^{-1}(\mathbf{b} - \mathbf{U}_\mathbf{A}\mathbf{x}^{(k)}). \quad (2.12.2)$$

**(2-12.a)** (15 min.) Give a necessary and sufficient condition on  $\mathbf{A}$ , such that the iteration (2.12.2) is well-defined.

SOLUTION for (2-12.a) → [2-12-1-0:gsitw.pdf](#)

**(2-12.b)** (20 min.) Assume that (2.12.2) is well-defined. Show that a fixed point  $\mathbf{x}$  of the iteration (2.12.2) is a solution of the linear system of equations  $\mathbf{Ax} = \mathbf{b}$ .

SOLUTION for (2-12.b) → [2-12-2-0:sinhex.pdf](#)

**(2-12.c)** (20 min.) Implement a C++ function

```
void GSIT(const MatrixXd & A, const VectorXd & b,
          VectorXd & x, double rtol);
```

solving the linear system  $\mathbf{Ax} = \mathbf{b}$  using the iterative scheme (2.12.2). To that end, apply the iterative scheme to an initial guess  $\mathbf{x}^{(0)}$  passed through  $\mathbf{x}$ . The approximated solution given by the final iterate is then stored in  $\mathbf{x}$  as an output.

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

SOLUTION for (2-12.c) → [2-12-3-0:gsitf.pdf](#)

**(2-12.d)** (15 min.) Test your implementation ( $n = 9$ ,  $rtol = 10e-8$ ) with the linear system given by

$$\mathbf{A} = \begin{bmatrix} 3 & 1 & 0 & \dots & 0 \\ 2 & \ddots & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 0 & \dots & 0 & 2 & 3 \end{bmatrix} \in \mathbb{R}^{n,n}, \mathbf{b} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \in \mathbb{R}^n \quad (2.12.11)$$

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

SOLUTION for (2-12.d) → [2-12-4-0:gsitim.pdf](#)

**(2-12.e)** (15 min.) Using the same matrix  $\mathbf{A}$  and the same r.h.s. vector  $\mathbf{b}$  as above in Subproblem (2-12.d), we have tabulated the quantity  $\|\mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\|_2$  for  $k = 1, \dots, 20$ .

$k$	$\ \mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\ _2$	$k$	$\ \mathbf{x}^{(k)} - \mathbf{A}^{-1}\mathbf{b}\ _2$
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 (2-12.e) → [2-12-5-0:gsitc.pdf](#)



**End Problem 2-12 , 85 min.**

### Problem 2-13: 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:

$$\mathbf{x} = \mathbf{A}\mathbf{y} \quad (2.13.1)$$

with

$$\mathbf{A} = \begin{bmatrix} a_1 & & & & a_n \\ & \ddots & & & \ddots \\ & & \ddots & & \ddots \\ & & & \ddots & \ddots \\ a_1 & & & & a_n \end{bmatrix} \in \mathbb{R}^{n,n}, \quad (2.13.2)$$

that is

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

(2-13.a) (15 min.) Implement a C++ function

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

that efficiently evaluates (2.13.1) for  $\mathbf{A}$  as described above.

HIDDEN HINT 1 for (2-13.a) → 2-13-1-0:template.pdf

SOLUTION for (2-13.a) → 2-13-1-1:xmatmulta.pdf

(2-13.b) (10 min.) What is the complexity of your function with respect to the problem size  $n$ ?

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

(2-13.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  $n = 2^k$ ,  $k \in \{1, \dots, 14\}$ .

HIDDEN HINT 1 for (2-13.c) → 2-13-3-0:xm3h1.pdf

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

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

**End Problem 2-13 , 45 min.**

# Chapter 3

## Direct Methods for Linear Systems of Equations

### Problem 3-1: 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.

(3-1.a)  Study [Lecture → § 2.5.0.8] and  EIGEN documentation to learn how to solve linear systems using direct elimination solvers of EIGEN. ▲

(3-1.b) Write a C++ function

```
VectorXd fruitPrice();
```

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

Mango – Kiwi – Lychee – Banana – Pomegranate – Pineapple

SOLUTION for (3-1.b) → [3-1-2-0:prb:sola.pdf](#) ▲

**End Problem 3-1**

### Problem 3-2: 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].

Connected with [Lecture → Section 2.3]

We consider a block partitioned linear system of equations

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

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  $\mathbf{d}_1$ ,  $\mathbf{c}$ , and  $\mathbf{d}_2$  to the C++ codes below.

(3-2.a)  Which permutation of rows and columns converts the matrix into a tridiagonal matrix?

SOLUTION for (3-2.a) → [3-2-1-0:blockrnt.pdf](#)

(3-2.b)  Write an efficient C++ function

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

that returns  $\mathbf{y} := \mathbf{Ax}$ . The argument  $\mathbf{x}$  passes a column vector  $\mathbf{x} \in \mathbb{R}^{2n}$ .

SOLUTION for (3-2.b) → [3-2-2-0:blockim.pdf](#)

(3-2.c)

Compute the LU-factors of  $\mathbf{A}$ , where you may assume that they exist.

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

SOLUTION for (3-2.c) → [3-2-3-1:blocklu.pdf](#)

(3-2.d)  Write an efficient C++ function

```
VectorXd solveA(const VectorXd & d1, const VectorXd & d2,
                const VectorXd & c, const VectorXd & x);
```

that solves  $\mathbf{Ax} = \mathbf{b}$  with Gaussian elimination.

Do not use EIGEN built-in linear solvers. Test for “near singularity” of the matrix.

Test your code with the arguments given in sub-problem Sub-problem (3-2.f). Compare with reference solution obtained in C++ using EIGEN linear solvers.

HIDDEN HINT 1 for (3-2.d) → [3-2-4-0:hsmal1se.pdf](#)

SOLUTION for (3-2.d) → [3-2-4-1:blockim.pdf](#)

(3-2.e)  Analyse the asymptotic complexity of your implementation of `solveA` in term of the problem size parameter  $n \rightarrow \infty$ .

SOLUTION for (3-2.e) → [3-2-5-0:blockco.pdf](#)

- (3-2.f)  Determine the asymptotic complexity of `solveA` in a numerical experiment. As test case use  $\mathbf{d}_1 = [1, \dots, n]^\top = -\mathbf{d}_2$ ,  $\mathbf{c} = \mathbf{1}$  and  $\mathbf{b} = [d_1; d_1]$ . Tabulate the runtimes (in seconds, 3 decimal digit, scientific notation) for meaningful values of  $n$ .

SOLUTION for (3-2.f) → 3-2-6-0:blockrnt.pdf



**End Problem 3-2**

### Problem 3-3: Block-wise LU-decompositon

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.

Connected with [Lecture → Section 2.6]

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

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

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

**(3-3.a)** Find the LU-decomposition of  $\mathbf{A}$ .

SOLUTION for (3-3.a) → [3-3-1-0:b1LUa.pdf](#)

**(3-3.b)** Show that  $\mathbf{A}$  is regular if and only if  $\mathbf{u}^T \mathbf{R}^{-1} \mathbf{v} \neq 0$ .

SOLUTION for (3-3.b) → [3-3-2-0:b1LUb.pdf](#)

**(3-3.c)** [ depends on Sub-problem (3-3.a) ]

Write a C++ function

```
VectorXd solve_LSE(const MatrixXd& R, const VectorXd& u, const
VectorXd& v, const 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 (3-3.a) and only use elementary Operations (no solvers from `Eigen` vor example).

HIDDEN HINT 1 for (3-3.c) → [3-3-3-0:<tag>.pdf](#)

SOLUTION for (3-3.c) → [3-3-3-1:b1LUc.pdf](#)

**End Problem 3-3**

### Problem 3-4: 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 → § 2.8.0.13]. 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 → § 2.8.0.13] and [Lecture → Section 3.3.3]

- (3-4.a) ☐ Show that, for every matrix  $\mathbf{A} \in \mathbb{R}^{m,n}$  such that  $\text{rank}(\mathbf{A}) = n$ , the product matrix  $\mathbf{A}^\top \mathbf{A}$  admits a Cholesky decomposition.

HIDDEN HINT 1 for (3-4.a) → 3-4-1-0:CholeskyQR1h.pdf

SOLUTION for (3-4.a) → 3-4-1-1:CholeskyQR1s.pdf ▲

- (3-4.b) ☒ The following C++ functions with EIGEN are given:

#### C++11-code 3.4.1: QR-decomposition via Cholesky decomposition

```

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

Get it on  GitLab (choleskyQR.cpp).

#### C++11-code 3.4.2: Standard way to do QR-decomposition in EIGEN

```

2 void DirectQR(const MatrixXd & A, MatrixXd & R, MatrixXd & Q) {
3
4     size_t m = A.rows();
5     size_t n = A.cols();
6
7     HouseholderQR<MatrixXd> QR = A.householderQr();
8     Q = QR.householderQ() * MatrixXd::Identity(m, std::min(m, n));
9     R = MatrixXd::Identity(std::min(m, n), m) *
10        QR.matrixQR().triangularView<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
13     // are not needed.
14     // Matlab returns this "economy-size" format calling "qr(A, 0)",
15     // which does not compute these extra entries.
16     // With the code above, Eigen is smart enough to not compute the
17     // discarded vectors.
18 }
```

Get it on  GitLab (choleskyQR.cpp).

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 (3-4.b) → 3-4-2-0:CholeskyQR2h.pdf

SOLUTION for (3-4.b) → 3-4-2-1:CholeskyQR2s.pdf ▲

(3-4.c) ☒ Let  $\text{EPS}$  denote the machine precision. Why does the function CholeskyQR from Subproblem (3-4.b) fail to return the correct result for  $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ \frac{1}{2}\text{EPS} & 0 \\ 0 & \frac{1}{2}\text{EPS} \end{bmatrix}$ ?

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

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

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



**End Problem 3-4**

### Problem 3-5: Resistance to impedance map

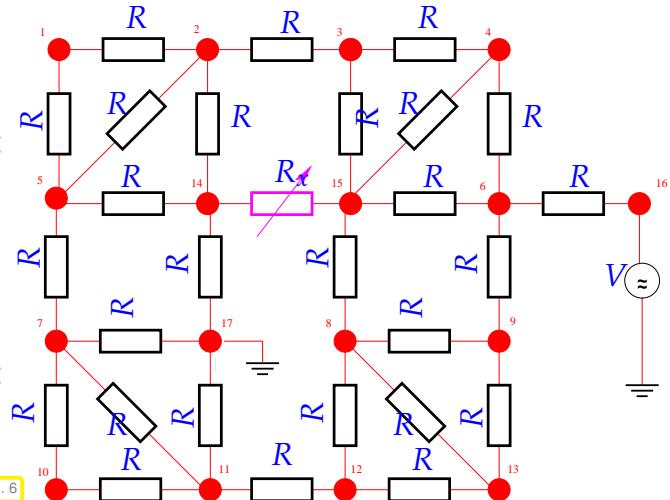
In [Lecture → § 2.6.0.12], we learned about the Sherman-Morrison-Woodbury update formula [Lecture → Lemma 2.6.0.21], which allows the efficient solution of a linear system of equations after a low-rank update according to [Lecture → Eq. (2.6.0.16)], provided that the setup phase of an elimination (→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

Connected with [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.

This circuit contains only identical linear resistors with the resistance  $R$ , that is, the relationship between branch currents and voltages is  $I = RU$  throughout. Excitation is provided by a voltage  $V$  imposed at node 16.

Here we consider DC operation (stationary setting), that is, all currents and voltages are real-valued.



**(3-5.a)**  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. The fundamental laws of circuit analysis should be known from physics as well as the principles of nodal analysis. ▲

**(3-5.b)**  Use nodal analysis to derive the linear system of equations  $\mathbf{A}_{R_x} \mathbf{x} = \mathbf{b}$  satisfied by the nodal potentials of the circuit from Figure 6. Here,  $\mathbf{x}$  denotes the unknown voltages at the nodes. The voltage  $V$  is applied to node #16. Node #17 is grounded (set voltage to 0V). All resistors except for the controlled one ( $R_x$ , colored magenta) have the same resistance  $R$ . Use the numbering of nodes indicated in Figure 6.

Optionally, you can make the computer work for you and find a fast way to build a matrix providing only the essential data. This is less tedious, less error prone and more flexible than specifying each entry individually. For this you can use auxiliary data structures.

SOLUTION for (3-5.b) → [3-5-2-0:circmat.pdf](#)

**(3-5.c)**  Characterise the change in the circuit matrix  $\mathbf{A}_{R_x}$  derived in sub-problem (3-5.b) induced by a change in the value of  $R_x$  as a low-rank modification of the circuit matrix  $\mathbf{A}_0$ . Use the matrix  $\mathbf{A}_0$  (with  $R_x = 0$ ) as your “base state”.

HIDDEN HINT 1 for (3-5.c) → [3-5-3-0:circh.pdf](#)

SOLUTION for (3-5.c) → [3-5-3-1:circsmw.pdf](#)

**(3-5.d)**  Based on the EIGEN library, implement a C++ class

#### C++11-code 3.5.2: ImpedanceMap Class signature

```

2  class ImpedanceMap {
3      std::size_t nnodes; //< Number of nodes in the circuit
4  public:
5      /* \brief Build system matrix and r.h.s. and perform a LU decomposition

```

```

6   * The LU decomposition is stored in 'lu' and can be
7   * reused in the SMW formula
8   * to avoid expensive matrix solves
9   * for repeated usages of the operator()
10  * \param R Resistance (in Ohm) value of R
11  * \param V Source voltage V at node 16 (in Volt),
12  *         ground is set to 0V at node 17
13  */
14 ImpedanceMap(double R, double V) : R(R), V(V) {
15     // TODO: build the matrix A_0.
16     // Compute lu factorization of A_0.
17     // Store LU factorization in 'lu'.
18     // Compute the right hand side and store it in 'b'.
19 }
20
21 /* \brief Compute the impedance given the resistance R_x.
22  * Use SMW formula for low rank perturbations to reuse LU
23  * factorization.
24  * \param Rx Resistance R_x > 0 between node 14 and 15
25  * \return Impedance V/I of the system A_{R_x}
26 */
27 double operator()(double Rx) {
28     // TODO: use SMW formula to compute the solution of A_{R_x}x = b
29     // Compute and return the impedance of the system.
30 }
31 private:
32     MatrixXd lu //< Store LU decomp. of matrix A.
33     double R //< Resistance R and source voltage W.
34     VectorXd b //< R.h.s vector prescribing sink/source voltages.
35 };

```

Get it on  [GitLab](#) (`impedancemap.cpp`).

whose `operator()` returns the impedance of the circuit from Figure 6, when supplied with a concrete value for  $R_x$ . This function should be implemented efficiently using [Lecture → Lemma 2.6.0.21]. The setup phase of Gaussian elimination should be carried out in the constructor.

Test your class using  $R = 1, V = 1$  and  $R_x = 1, 2, 4, \dots, 1024$ .

For  $R_x = 1024$ , we obtain an impedance of 2.65744.

See the file `impedancemap.cpp`.

The impedance of the circuit is the quotient of the voltage at the input node #16 and the current through the voltage source.

SOLUTION for (3-5.d) → [3-5-4-0:circimp.pdf](#)



**End Problem 3-5**

### Problem 3-6: 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 ( $a_i, b_i \in \mathbb{R}$ ):

$$\mathbf{A} := \begin{bmatrix} 2 & a_1 & 0 & \dots & \dots & \dots & 0 \\ 0 & 2 & a_2 & 0 & \dots & \dots & 0 \\ b_1 & 0 & \ddots & \ddots & \ddots & & \vdots \\ 0 & b_2 & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & 0 & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & a_{n-1} \\ 0 & 0 & \dots & 0 & b_{n-2} & 0 & 2 \end{bmatrix} \in \mathbb{R}^{n \times n} \quad (3.6.1)$$

This matrix is an instance of a banded matrix.

(3-6.a)  Implement an efficient C++ function `multAx` for the computation of  $\mathbf{y} = \mathbf{Ax}$ :

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

`Vector a` and `Vector b` contain the nonzero entries along the subdiagonals of matrix  $\mathbf{A}$  (Eq. (3.6.1)). `Vector y` and `Vector x` are the vectors of equation  $\mathbf{y} = \mathbf{Ax}$  (`Vector y` is the output).

SOLUTION for (3-6.a) → 3-6-1-0:effbandimpl.pdf ▲

(3-6.b)  Show that  $\mathbf{A}$  is invertible if  $a_i, b_i \in [0, 1]$ .

HIDDEN HINT 1 for (3-6.b) → 3-6-2-0:effbandinvh.pdf

SOLUTION for (3-6.b) → 3-6-2-1:effbandinv.pdf ▲

(3-6.c)  Assume that  $b_i = 0, \forall i = 1, \dots, n - 2$ . Implement an efficient C++ function solving  $\mathbf{Ax} = \mathbf{r}$ :

```
template <class Vector>
void solvelseAupper(const Vector & a, const Vector & r, Vector &
                     x);
```

`Vector a` contains the nonzero entries along the upper subdiagonal of matrix  $\mathbf{A}$  (Eq. (3.6.1)). `Vector r` and `Vector x` are the vectors of the equation  $\mathbf{r} = \mathbf{Ax}$  (`Vector x` is the output).

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

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

(3-6.d)  For general  $a_i, b_i \in [0, 1]$ , implement an efficient C++ function that computes the solution of  $\mathbf{Ax} = \mathbf{r}$  by Gaussian elimination:

```
template <class Vector>
void solvelseA(const Vector & a, const Vector & b, const Vector &
                r, Vector & x);
```

You must not use any high-level solver routines of EIGEN.

HIDDEN HINT 1 for (3-6.d) → [3-6-4-0:effbandgaussh.pdf](#)

SOLUTION for (3-6.d) → [3-6-4-1:effbandhaussol.pdf](#) ▲

(3-6.e)  What is the asymptotic complexity of your implementation of `solvalseA` for  $n \rightarrow \infty$ ?

HIDDEN HINT 1 for (3-6.e) → [3-6-5-0:effbandcompl.pdf](#)

SOLUTION for (3-6.e) → [3-6-5-1:effbandcompls.pdf](#) ▲

(3-6.f)  Implement `solvalseAEigen` as in (3-6.d), but this time using EIGEN's sparse elimination solver.

HIDDEN HINT 1 for (3-6.f) → [3-6-6-0:effbandsprs.pdf](#)

SOLUTION for (3-6.f) → [3-6-6-1:effbandsprssol.pdf](#) ▲

**End Problem 3-6**

### Problem 3-7: 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 → Eq. (3.3.3.11)]. 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

$$\mathbf{H} = \mathbf{I} - 2\mathbf{v}\mathbf{v}^H \quad \text{with} \quad \mathbf{v}^H\mathbf{v} = 1, \quad \mathbf{v} \in \mathbb{C}^n. \quad (3.7.1)$$

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

(3-7.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 (3-7.a) → [3-7-1-0:householder1.pdf](#)

(3-7.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 (3-7.b) → [3-7-2-0:householder2.pdf](#)

(3-7.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 ( $\mathbf{HC} = \mathbf{R}$ ).

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 (3-7.c) → [3-7-3-0:householder3.pdf](#)

(3-7.d)  Implement a function

**void** applyHouseholder(**VectorXd** &x, **const VectorXd** &v)

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

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

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

SOLUTION for (3-7.d) → [3-7-4-0:householder2.pdf](#)



(3-7.e)  As we learned in [Lecture → Rem. 3.3.3.20], the sequence of Householder transformations  $\mathbf{H}_1, \dots, \mathbf{H}_{n-1}$  effecting the conversion of an  $n \times n$  matrix into upper triangular form (→ [Lecture → § 3.3.3.10]) 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

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

Write a C++/EIGEN function

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

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

HIDDEN HINT 1 for (3-7.e) → [3-7-5-0:house:mb.pdf](#)

HIDDEN HINT 2 for (3-7.e) → [3-7-5-1:house:h1.pdf](#)

SOLUTION for (3-7.e) → [3-7-5-2:householder2.pdf](#)



**End Problem 3-7**

### Problem 3-8: 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].

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

$$\mathbf{AX} + \mathbf{XA}^\top = \mathbf{I}, \quad (3.8.1)$$

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

**(3-8.a)** (10 min.) Show that for a fixed matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  the following mapping is linear:

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

HIDDEN HINT 1 for (3-8.a) → [3-8-1-0:ly1h.pdf](#)

SOLUTION for (3-8.a) → [3-8-1-1:ly1s.pdf](#) ▲

In the following let  $\text{vec}(\mathbf{M}) \in \mathbb{R}^{n^2}$  denote the column vector obtained by storing the internal coefficient array of a matrix  $\mathbf{M} \in \mathbb{R}^{n,n}$  in column major format, i.e. a data array with  $n^2$  components ([Lecture → Rem. 1.2.3.4]). In PYTHON, MATLAB,  $\text{vec}(\mathbf{M})$  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 (3-8.a), (3.8.1) is equivalent to the following linear system of equations:

$$\mathbf{C} \text{vec}(\mathbf{X}) = \mathbf{b} \quad (3.8.2)$$

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

**(3-8.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]. ▲

**(3-8.c)** (15 min.) Determine  $\mathbf{C}$  and  $\mathbf{b}$  from Eq. (3.8.2) for  $n = 2$  and

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

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

SOLUTION for (3-8.c) → [3-8-3-1:ly3s.pdf](#) ▲

**(3-8.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 (3-8.d) → [3-8-4-0:ly4s.pdf](#) ▲

**(3-8.e)** (30 min.) [ depends on Sub-problem (3-8.d) ]

Implement a C++ function that builds the EIGEN matrix  $\mathbf{C}$  from  $\mathbf{A}$ :

```
Eigen::SparseMatrix<double> buildC(const 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 (3-8.e) → [3-8-5-0:arrmatha.pdf](#)

SOLUTION for (3-8.e) → [3-8-5-1:arrwc.pdf](#)

(3-8.f)  $\square$  (20 min.) Implement a C++ function that returns the solution of (3.8.1), i.e. the  $n \times n$ -matrix  $X$  if  $A \in \mathbb{R}^{n,n}$  is passed in the argument A.

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

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

HIDDEN HINT 1 for (3-8.f) → [3-8-6-0:ly6h.pdf](#)

SOLUTION for (3-8.f) → [3-8-6-1:ly6s.pdf](#)

(3-8.g)  $\square$  (10 min.) Write a C++ function

```
bool testLyapunov();
```

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

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

SOLUTION for (3-8.g) → [3-8-7-0:ly7s.pdf](#)

(3-8.h)  $\square$  (15 min.) Give an upper bound (as sharp as possible) for  $\text{nnz}(C)$  in terms of  $\text{nnz}(A)$  ( $\text{nnz}$  is the number of nonzero elements). Can C be legitimately regarded as a sparse matrix for large  $n$  even if A is dense?

SOLUTION for (3-8.h) → [3-8-8-0:ly8s.pdf](#)

**End Problem 3-8 , 135 min.**

### Problem 3-9: 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.

Connected with [Lecture → Section 2.6]

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

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix}, \quad (3.9.1)$$

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*.

(3-9.a) □ (10 min.) Give a necessary and sufficient condition for the *triangular* matrix  $\mathbf{R}$  to be invertible.

HIDDEN HINT 1 for (3-9.a) → 3-9-1-0:s1.pdf

SOLUTION for (3-9.a) → 3-9-1-1:partinv.pdf ▲

(3-9.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

$$\begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{b} \\ \beta \end{bmatrix} \quad (3.9.2)$$

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

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

SOLUTION for (3-9.b) → 3-9-2-0:partgauss.pdf ▲

(3-9.c) □ (10 min.) [ depends on Sub-problem (3-9.b) ]

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

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

(3-9.d) □ (25 min.) [ depends on Sub-problem (3-9.b) ]

Implement the C++ function

```
void solvelse(const MatrixXd & R,
               const VectorXd & v, const VectorXd & u,
               const VectorXd & b, VectorXd & x);
```

for the efficient computation of the solution of  $\mathbf{Ax} = \mathbf{b}$  (with  $\mathbf{A}$  as in (3.9.1)). Perform size checks on input matrices and vectors.

Use the decomposition from (3-9.b).

You can rely on the `triangularView()` function to instruct EIGEN of the triangular structure of  $\mathbf{R}$ , see [Lecture → Code 1.2.1.6].

SOLUTION for (3-9.d) → 3-9-4-0:partimpl.pdf ▲

(3-9.e) □ (20 min.) [ depends on Sub-problem (3-9.d) ]

Code a C++ function

```
bool testSolveLSE(const MatrixXd & R,  
                  const VectorXd & v, const VectorXd & u,  
                  const VectorXd & b, 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 (3-9.e) → [3-9-5-0:s3h1.pdf](#)

HIDDEN HINT 2 for (3-9.e) → [3-9-5-1:s3h2.pdf](#)

SOLUTION for (3-9.e) → [3-9-5-2:parttest.pdf](#)

(3-9.f)  (10 min.) [ depends on Sub-problem (3-9.d) ]

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

SOLUTION for (3-9.f) → [3-9-6-0:partcmp.pdf](#)

End Problem 3-9 , 90 min.

### Problem 3-10: 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 3-5

Consider the following pseudocode:

#### Algorithm 3.10.1: 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
     $l_{\min} \leftarrow l_{\text{new}};$ 
     $\mathbf{M} \leftarrow \text{diag}(\mathbf{d}) + \mathbf{ev} \cdot \mathbf{ev}^\top;$ 
     $\mathbf{ev} \leftarrow \mathbf{M}^{-1} \mathbf{ev};$ 
     $\mathbf{ev} \leftarrow \mathbf{ev} / |\mathbf{ev}|;$ 
     $l_{\text{new}} \leftarrow \mathbf{ev}^\top \mathbf{M} \mathbf{ev};$ 
end while
return  $l_{\text{new}}$ 
```

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.

(3-10.a) (15 min.) Directly port the function `rankoneinvit` (Algorithm 3.10.1) to C++ using EIGEN. To that end create a function

```
double rankoneinvit(const 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 (3-10.a) → 3-10-1-0:s1h1.pdf

SOLUTION for (3-10.a) → 3-10-1-1:smwps.pdf

(3-10.b) (5 min.) What is the asymptotic complexity of the body of the loop in function `rankoneinvit`?

HIDDEN HINT 1 for (3-10.b) → 3-10-2-0:smwch.pdf

SOLUTION for (3-10.b) → 3-10-2-1:smwcs.pdf

(3-10.c) (30 min.) [ depends on Sub-problem (3-10.a) ]

Implement an EIGEN-based C++ function

```
double rankoneinvit_fast(const 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 (3-10.c) → 3-10-3-0:smweh.pdf

SOLUTION for (3-10.c) → [3-10-3-1:smwes.pdf](#) ▲

(3-10.d)  (5 min.) [ depends on Sub-problem (3-10.c) ]

What is the asymptotic complexity of the execution of the loop body in your implementation of `rankoneinvit_fast()` from Sub-problem (3-10.c)?

SOLUTION for (3-10.d) → [3-10-4-0:smwecls.pdf](#) ▲

(3-10.e)  (15 min.) Tabulate the runtimes of the two C++ implementations with different vector sizes  $n = 2^p$ , with  $p = 2, 3, \dots, 8$ . 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 (3-10.e) → [3-10-5-0:smwclds.pdf](#) ▲

**End Problem 3-10 , 70 min.**

### Problem 3-11: Sequential linear systems

Consider a sequence of linear systems when all the linear systems share the same matrix  $\mathbf{A}$ :  $\mathbf{Ax} = \mathbf{b}_i$ . 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 3.11.1: Code `solveperm`

**Require:** Matrix  $\mathbf{A} \in \mathbb{R}^{n \times n}$ , vector  $\mathbf{b} \in \mathbb{R}^n$

**Ensure:** Matrix  $\mathbf{X} = [\mathbf{X}_1, \dots, \mathbf{X}_n] \in \mathbb{R}^{n \times n}$ ,  $\mathbf{X}_i \in \mathbb{R}^n$

**while** not all cyclic permutations of  $\mathbf{b}$  tested yet **do**

$\mathbf{b} \leftarrow [b_n, b_1, b_2, \dots, b_{n-1}]^\top$

$\mathbf{X}_i = \mathbf{A}^{-1}\mathbf{b}$

**end while**

(3-11.a) ☐ What is the worst case asymptotic complexity of the function `solveperm` for  $n \rightarrow \infty$ ?

HIDDEN HINT 1 for (3-11.a) → 3-11-1-0:permch.pdf

SOLUTION for (3-11.a) → 3-11-1-1:permcs.pdf

(3-11.b) ☐ Port the function `solveperm` (Code 3.11.1) to C++ using EIGEN.

HIDDEN HINT 1 for (3-11.b) → 3-11-2-0:permph.pdf

SOLUTION for (3-11.b) → 3-11-2-1:permph.pdf

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

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

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

**End Problem 3-11**

### Problem 3-12: 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  $\mathbf{A}\mathbf{x} = \mathbf{b}$ , where the  $n \times n$  matrix  $\mathbf{A}$  has the following structure:

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

- (3-12.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 (3-12.a) → 3-12-1-0:structuredLSE1h.pdf

SOLUTION for (3-12.a) → 3-12-1-1:structuredLSE1s.pdf

- (3-12.b)**  Write a C++ function that builds the matrix  $\mathbf{A}$  given the vector  $\mathbf{a}$ :

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

You can use EIGEN classes for your code.

HIDDEN HINT 1 for (3-12.b) → 3-12-2-0:structuredLSE2h.pdf

SOLUTION for (3-12.b) → 3-12-2-1:structuredLSE2s.pdf

- (3-12.c)**  Implement a function in C++ which solves the linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$  by means of “structure oblivious” Gaussian elimination, for which EIGEN’s dedicated classes and methods should be used.

```
void solveA(const VectorXd & a, const VectorXd & b, 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 (3-12.c) → 3-12-3-0:structuredLSE3h.pdf

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

- (3-12.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  $\mathbf{A}\mathbf{x} = \mathbf{b}$ .

HIDDEN HINT 1 for (3-12.d) → 3-12-4-0:structuredLSE4h.pdf

SOLUTION for (3-12.d) → 3-12-4-1:structuredLSE4s.pdf

- (3-12.e)**  Implement a function in C++ which efficiently solves the linear system  $\mathbf{A}\mathbf{x} = \mathbf{b}$ , using EIGEN classes. The input consists of vectors  $\mathbf{a}$  and  $\mathbf{b}$ .

HIDDEN HINT 1 for (3-12.e) → 3-12-5-0:structuredLSE5h.pdf

SOLUTION for (3-12.e) → 3-12-5-1:structuredLSE5s.pdf

- (3-12.f)  Compare the timing of the naive implementation of (3-12.c) with the efficient solution of (3-12.e).

HIDDEN HINT 1 for (3-12.f) → [3-12-6-0:structuredLSE6h.pdf](#)

SOLUTION for (3-12.f) → [3-12-6-1:structuredLSE6s.pdf](#)



**End Problem 3-12**

### Problem 3-13: Triplet 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]).

Related to [Lecture → Section 2.7.1]

**Remark.** The ultimate goal is to devise a function that converts a matrix given in *triplet* (or *COOrdinate list format*) (COO, see [Lecture → § 2.7.1.1]) to the *compressed row storage format* (CRS, see [Lecture → § 2.7.1.4]). You do not have to follow the subproblems, if you devise a suitable conversion function and data structures on your own. You do not need to rely on EIGEN to solve this problem.

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 together when fully expressing the matrix.

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 only made by zero elements require special consideration. The usual 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, we always consider *matrices without rows only made by zeros*.

**(3-13.a)**  Define a C++ structure that stores a matrix of type `scalar` (template parameter) in COO format:

```
template <class scalar>
struct TripletMatrix;
```

You should store sizes and indices as `std::size_t`.

HIDDEN HINT 1 for (3-13.a) → [3-13-1-0:triplettoCRS1h.pdf](#)

SOLUTION for (3-13.a) → [3-13-1-1:triplettoCRS1s.pdf](#) ▲

**(3-13.b)**  Define a C++ structure that stores a matrix of type `scalar` (template parameter) in CRS format:

```
template <class scalar>
struct CRSMatrix;
```

You can store sizes and indices as `std::size_t`.

HIDDEN HINT 1 for (3-13.b) → [3-13-2-0:triplettoCRS2h.pdf](#)

SOLUTION for (3-13.b) → [3-13-2-1:triplettoCRS2s.pdf](#) ▲

**(3-13.c)**  *This subproblem is optional.* Implement C++ member functions for `TripletMatrix` ((3-13.a)) and `CRSMatrix` ((3-13.b)) that convert your structures to EIGEN dense matrices types:

```

Eigen::Matrix<scalar, Eigen::Dynamic, Eigen::Dynamic>
TripletMatrix<scalar>::densify();
Eigen::Matrix<scalar, Eigen::Dynamic, Eigen::Dynamic>
CRSMatrix<scalar>::densify();

```

HIDDEN HINT 1 for (3-13.c) → 3-13-3-0:triplettoCRS3h.pdf

SOLUTION for (3-13.c) → 3-13-3-1:triplettoCRS3s.pdf ▲

**(3-13.d)** ☒ Write a C++ function that converts a matrix **T** in COO format to a matrix **C** in CRS format:

```

template <class scalar>
void tripletToCRS(const TripletMatrix<scalar>& T,
CRSMatrix<scalar>& C);

```

Try to be as efficient as possible.

HIDDEN HINT 1 for (3-13.d) → 3-13-4-0:triplettoCRS4h.pdf

SOLUTION for (3-13.d) → 3-13-4-1:triplettoCRS4s.pdf ▲

**(3-13.e)** ☐ What is the worst-case complexity of your function `tripletToCRS` ((3-13.d)) in the number of triplets?

HIDDEN HINT 1 for (3-13.e) → 3-13-5-0:triplettoCRS5h.pdf

SOLUTION for (3-13.e) → 3-13-5-1:triplettoCRS5s.pdf ▲

**(3-13.f)** ☐ Test the correctness and runtime of your function `tripletToCRS`.

HIDDEN HINT 1 for (3-13.f) → 3-13-6-0:triplettoCRS6h.pdf

SOLUTION for (3-13.f) → 3-13-6-1:triplettoCRS6s.pdf ▲

**End Problem 3-13**

### Problem 3-14: 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`.

**(3-14.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:

```
void CCS(const MatrixXd & A, VectorXd & val,
        VectorXd & row_ind, VectorXd & col_ptr);
```

The CCS data are returned through the vectors `val`, `row_ind`, and `col_ptr`.

**Prohibited.** While you can use EIGEN classes such as `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 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 (3-14.a) → [3-14-1-0:smwps.pdf](#)

**(3-14.b)** (5 min.) [ depends on Sub-problem (3-14.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 (3-14.b) → [3-14-2-0:smwps.pdf](#)

**End Problem 3-14 , 35 min.**

### Problem 3-15: 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]

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

#### C++11-code 3.15.1: Declaration of `EllpackMat` class

```

2  class EllpackMat {
3  public:
4      EllpackMat(const Triplets &triplets, index_t m, index_t n);
5      double operator()(index_t i, index_t j) const;
6      void mvmult(const VectorXd &x, VectorXd &y) const;
7      void mtvmult(const VectorXd &x, VectorXd &y) const;
8  private:
9      std::vector<double> val; //< Vector containing values
10     // corresponding to entries in 'col'
11     std::vector<index_t> col; //< Vector containing column
12     // indices of the entries in 'val'.
13     // The position of a cell in 'val' and 'col'
14     // is determined by its row number and original position in 'triplets'.
15     index_t maxcols; //< Number of non-empty columns
16     index_t m, n;    //< Number of rows, number of columns
17 };

```

Get it on  [GitLab \(ellpack.cpp\)](#).

In the code above, `maxcols` is defined as:

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

The entry-read-access `operator()` method is implemented as follows:

#### C++11-code 3.15.2: Definition of `operator()`

```

2  double EllpackMat::operator()(index_t i, index_t j) const {
3      assert(0 <= i && i < m && 0 <= j && j < n && "Index out of bounds!");
4
5      for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
6          if (col.at(l) == j)
7              return val.at(l);
8      }
9      return 0;
10 }

```

Get it on  [GitLab \(ellpack.cpp\)](#).

It is well defined for  $i = 0, \dots, m - 1$  and for  $j = 0, \dots, n - 1$ .

**(3-15.a)** (30 min.) Implement an efficient constructor 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. However, in this subproblem you may assume that duplicate index pairs do not occur in the triplets vector.

HIDDEN HINT 1 for (3-15.a) → [3-15-1-0:ellpack1h.pdf](#)

SOLUTION for (3-15.a) → [3-15-1-1:ellpack1s.pdf](#) ▲

**(3-15.b)** (20 min.) Implement an efficient method of class `EllpackMat` that, given an input `n`-vector `x`, returns the `m`-vector `y` from the matrix-vector product  $\mathbf{Ax}$ :

```
void mvmult(const Vector &x, Vector &y) const;
```

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

HIDDEN HINT 1 for (3-15.b) → [3-15-2-0:ellpack2h.pdf](#)

SOLUTION for (3-15.b) → [3-15-2-1:ellpack2s.pdf](#) ▲

**(3-15.c)** (15 min.) Similarly to (3-15.b), implement now the method `mtvmult` that computes the matrix-vector product  $\mathbf{A}^T \mathbf{x}$ .

HIDDEN HINT 1 for (3-15.c) → [3-15-3-0:ellpack3h.pdf](#)

SOLUTION for (3-15.c) → [3-15-3-1:ellpack3s.pdf](#) ▲

**End Problem 3-15 , 65 min.**

**Problem 3-16: 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  $\mathbb{R}^{n,m}$ :

$$L : \begin{cases} \mathbb{R}^{n,m} & \rightarrow \mathbb{R}^{n,m} \\ \mathbf{X} & \mapsto L(\mathbf{X}) : (L(\mathbf{X}))_{i,j} := \sum_{k,l=1}^3 s_{k,l}(\mathbf{X})_{i+k-2,j+l-2}, \quad i \in \{1, \dots, n\}, \quad m \in \{1, \dots, m\} \end{cases}$$

where we adopt the convention that in the sum non-existent matrix entries are set to zero.

**(3-16.a)** □ (10 min.) Show that  $L$  is a linear operator.

SOLUTION for (3-16.a) → 3-16-1-0:grf1.pdf

**(3-16.b)** □ The space  $\mathbb{R}^{n,m}$ , as a vector-space, is isomorphic to  $\mathbb{R}^{n \cdot m}$ , via the mapping  $\mathbf{X} \mapsto \text{vec}(\mathbf{X})$ , see [Lecture → Eq. (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.

$$\mathbf{A} \cdot \text{vec}(\mathbf{X}) = \text{vec}(L(\mathbf{X})).$$

Write down the matrix  $\mathbf{A}$  for  $n = m = 3$ .

SOLUTION for (3-16.b) → 3-16-2-0:gfrwa.pdf

**(3-16.c)** □ (15 min.) Write a C++ function

```
void eval(MatrixXd & X, std::function<double(Eigen::Index,
Eigen::Index)> f);
```

which, given a matrix  $\mathbf{X} \in \mathbb{R}^{n,m}$  and a function  $f : \mathbb{N}^2 \rightarrow \mathbb{R}$ , fills the matrix  $\mathbf{X}$  according to the rule  $(\mathbf{X})_{i,j} := f(i,j)$ ,  $i \in \{1, \dots, n\}$ ,  $j \in \{1, \dots, m\}$ .

Additionally, inside the auxiliary function `define_f()` define a lambda function  $f$ , with signature `double(Eigen::Index, Eigen::Index)`, implementing:

$$f : \begin{cases} \mathbb{N}^2 & \rightarrow \mathbb{R}, \\ (i,j) & \mapsto \begin{cases} 1 & i > n/4 \wedge i < 3n/4 \wedge j > m/4 \wedge j < 3m/4, \\ 0 & \text{otherwise.} \end{cases} \end{cases}$$

SOLUTION for (3-16.c) → 3-16-3-0:gfrwa.pdf

(3-16.d) (30 min.) [ depends on Sub-problem (3-16.c) ]

Implement a function

```
SparseMatrix<double> build_matrix(const 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  $X_n, X_m$ ), 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 (3-16.d) → 3-16-4-0:2h1.pdf

SOLUTION for (3-16.d) → 3-16-4-1:gfrwb.pdf ▲

(3-16.e) (20 min.) Implement a function

```
void mult(const SparseMatrix<double> & A,
    const MatrixXd & X, MatrixXd & Y);
```

which, given a matrix  $\mathbf{X}$  and the sparse matrix  $\mathbf{A}$ , returns the matrix  $\mathbf{Y}$  s.t.  $\text{vec}(\mathbf{Y}) := \mathbf{A} \text{vec}(\mathbf{X})$ . You can use objects of type `Eigen::Map` to “reshape” a matrix into a column vector.

SOLUTION for (3-16.e) → 3-16-5-0:gfrwm.pdf ▲

(3-16.f) (20 min.)

Implement a function

```
void solve(const SparseMatrix<double> & A,
    const MatrixXd & Y, MatrixXd & X);
```

which, given a matrix  $\mathbf{Y}$  and the sparse matrix  $\mathbf{A}$ , returns the matrix  $\mathbf{X}$  s.t.  $\text{vec}(\mathbf{Y}) := \mathbf{A} \text{vec}(\mathbf{X})$ . 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 (3-16.f) → 3-16-6-0:gfrwi.pdf ▲

**End Problem 3-16 , 95 min.**

### Problem 3-17: 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>`.

- (3-17.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 (3-17.a) → [3-17-1-0:matmatCOO1h.pdf](#)

SOLUTION for (3-17.a) → [3-17-1-1:matmatCOO1s.pdf](#) ▲

- (3-17.b)** ☐ Implement a C++ function which returns the input matrix  $\mathbf{A} \in \mathbb{R}^{m \times n}$  in COO format:

```
TripVec Mat2COO(const MatrixXd &A);
```

You can use EIGEN classes.

SOLUTION for (3-17.b) → [3-17-2-0:matmatCOO2s.pdf](#) ▲

- (3-17.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 (3-17.c) → [3-17-3-0:matmatCOO3h.pdf](#)

SOLUTION for (3-17.c) → [3-17-3-1:matmatCOO3s.pdf](#) ▲

- (3-17.d)** ☐ What is the asymptotic complexity of your naive implementation in (3-17.c)?

HIDDEN HINT 1 for (3-17.d) → [3-17-4-0:matmatCOO4h.pdf](#)

SOLUTION for (3-17.d) → [3-17-4-1:matmatCOO4s.pdf](#) ▲

- (3-17.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 (3-17.c)?

HIDDEN HINT 1 for (3-17.e) → 3-17-5-0:matmatCOO5h.pdf

SOLUTION for (3-17.e) → 3-17-5-1:matmatCOO5s.pdf ▲

(3-17.f) ☐ What is the asymptotic complexity of your efficient implementation in (3-17.e)?

HIDDEN HINT 1 for (3-17.f) → 3-17-6-0:matmatCOO6h.pdf

SOLUTION for (3-17.f) → 3-17-6-1:matmatCOO6s.pdf ▲

(3-17.g) ☐ Compare the timing of COOprod\_naive (3-17.c) and COOprod\_effic (3-17.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 (3-17.g) → 3-17-7-0:matmatCOO7h.pdf

SOLUTION for (3-17.g) → 3-17-7-1:matmatCOO7s.pdf ▲

**End Problem 3-17**

### Problem 3-18: A special Sylvester Equation

A **Sylvester equations** is a *linear matrix equation* of the form

$$\mathbf{AX} + \mathbf{XB} = \mathbf{C}, \quad \mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{R}^{n,n},$$

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 3-8.

**(3-18.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]. ▲

**(3-18.b)** (5 min.) Recall the “vectorization”  $\text{vec}(\mathbf{X}) \in \mathbb{R}^{n^2}$  of a matrix  $\mathbf{X} \in \mathbb{R}^{n,n}$  from [Lecture → Rem. 1.2.3.4] and jot down  $\text{vec}(\mathbf{X})$  for

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \in \mathbb{R}^{2,2}.$$

SOLUTION for (3-18.b) → [3-18-2-0:s1a.pdf](#) ▲

**(3-18.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

$$\text{vec}(\mathbf{SXT}) = \mathbf{C} \text{vec}(\mathbf{X}) \quad \forall \mathbf{X} \in \mathbb{R}^{2,2},$$

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 (3-18.c) → [3-18-3-0:s1b.pdf](#) ▲

**(3-18.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

$$\text{vec}(\mathbf{SXT}) = \mathbf{C} \text{vec}(\mathbf{X}).$$

HIDDEN HINT 1 for (3-18.d) → [3-18-4-0:s1h1.pdf](#)

SOLUTION for (3-18.d) → [3-18-4-1: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:

$$\text{seek } \mathbf{X} \in \mathbb{R}^{n,n} : \quad \mathbf{XA}^{-1} + \mathbf{AX} = \mathbf{I}, \tag{3.18.4}$$

and the equivalent matrix equation:

$$\text{seek } \mathbf{X} \in \mathbb{R}^{n,n} : \quad \mathbf{X} + \mathbf{AXA} = \mathbf{A}. \tag{3.18.5}$$

**(3-18.e)** (5 min.) Both (3.18.4) and (3.18.5) can be recast as an  $n^2 \times n^2$  linear system of equations for the unknown vector  $\text{vec}(\mathbf{X})$ . Based on the result of Sub-problem (3-18.d) express their coefficient matrices by means of the Kronecker product [Lecture → Def. 1.4.3.7].

SOLUTION for (3-18.e) → [3-18-5-0:s3a.pdf](#) ▲

**(3-18.f)** (10 min.) Based on the result of Sub-problem (3-18.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 (3.18.4) and (3.18.5).

HIDDEN HINT 1 for (3-18.f) → [3-18-6-0:s3h1.pdf](#)

SOLUTION for (3-18.f) → [3-18-6-1:s4.pdf](#) ▲

**(3-18.g)** (15 min.) Show that both (3.18.4) and (3.18.5) 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 3.18.9. 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

$$\sigma(\mathbf{S} \otimes \mathbf{T}) = \{\lambda\mu : \lambda \in \sigma(\mathbf{S}), \mu \in \sigma(\mathbf{T})\}.$$

HIDDEN HINT 1 for (3-18.g) → [3-18-7-0:s4h1.pdf](#)

HIDDEN HINT 2 for (3-18.g) → [3-18-7-1:s4h2.pdf](#)

SOLUTION for (3-18.g) → [3-18-7-2:s4.pdf](#) ▲

**(3-18.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 (3.18.4) 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 (3-18.h) → [3-18-8-0:s5.pdf](#) ▲

**(3-18.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 points 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 (3-18.i) → [3-18-9-0:s6.pdf](#) ▲

(3-18.j)  (20 min.) [ depends on Sub-problem (3-18.i), Sub-problem (3-18.e) ]

Write a C++ function

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

that solves (3.18.4) and returns the solution  $\mathbf{X}$ .

SOLUTION for (3-18.j) → [3-18-10-0:s7.pdf](#) ▲

**End Problem 3-18 , 155 min.**

# Chapter 4

## Direct Methods for Linear Least Squares Problems

### Problem 4-1: 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  $(t_i, y_i), i = 1, \dots, n, 2 < n \in \mathbb{N}$ . 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):

$$f(t_i) = y_i, \quad \text{with } y_i > 0, i = 1, \dots, n. \quad (4.1.1)$$

(4-1.a)  (5 min.) Which overdetermined system of *nonlinear* equations directly stems from (4.1.1)?

SOLUTION for (4-1.a) → [4-1-1-0:AdaptedLinReg1s.pdf](#) ▲

(4-1.b)  (10 min.) In which overdetermined system of *linear* equations can you transform the nonlinear system derived in (4-1.a)?

HIDDEN HINT 1 for (4-1.b) → [4-1-2-0:AdaptedLinReg2h.pdf](#)

SOLUTION for (4-1.b) → [4-1-2-1:AdaptedLinReg2s.pdf](#) ▲

(4-1.c)  (10 min.) Determine the normal equations for the overdetermined linear system of (4-1.b).

HIDDEN HINT 1 for (4-1.c) → [4-1-3-0:AdaptedLinReg3h.pdf](#)

SOLUTION for (4-1.c) → [4-1-3-1:AdaptedLinReg3s.pdf](#) ▲

(4-1.d)  (20 min.) Consider the data  $(t_i, y_i) \in \mathbb{R}^2, i = 1, \dots, m$ , stored in two vectors `t` and `y` of equal length. Implement a C++ function

```
VectorXd linReg(const VectorXd &t, const 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  $\alpha$  and  $\beta$  in a vector of length 2.

SOLUTION for (4-1.d) → [4-1-4-0:AdaptedLinReg4s.pdf](#) ▲

(4-1.e) (15 min.) [ depends on Sub-problem (4-1.d), Sub-problem (4-1.a) ]

Now consider the data  $(t_i, y_i)$  stored in two vectors  $t$  and  $f$  of equal length. Implement a C++ function

```
VectorXd expFit(const VectorXd &t, const VectorXd &f);
```

that returns a least squares estimate of  $\alpha$  and  $\beta$ , given the nonlinear relationship (4.1.1) between  $t$  and  $f$ :

You can use the function `linReg()` implemented in (4-1.d).

HIDDEN HINT 1 for (4-1.e) → [4-1-5-0:AdaptedLinReg5h.pdf](#)

SOLUTION for (4-1.e) → [4-1-5-1:AdaptedLinReg5s.pdf](#) ▲

**End Problem 4-1 , 60 min.**

### Problem 4-2: Estimating a Tridiagonal Matrix

In [Lecture → Section 3.1] we learned that to determine the least squares solution of an over-determined 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].

Let two vectors  $\mathbf{z}, \mathbf{c} \in \mathbb{R}^n$ ,  $n > 2 \in \mathbb{N}$  of measurements be given. The two real numbers  $\alpha^*$  and  $\beta^*$  are defined as:

$$(\alpha^*, \beta^*) = \underset{\alpha, \beta \in \mathbb{R}}{\operatorname{argmin}} \|\mathbf{T}_{\alpha, \beta} \mathbf{z} - \mathbf{c}\|_2, \quad (4.2.1)$$

where  $\mathbf{T}_{\alpha, \beta} \in \mathbb{R}^{n \times n}$  is the following tridiagonal matrix

$$\mathbf{T}_{\alpha, \beta} = \begin{bmatrix} \alpha & \beta & 0 & \dots & 0 \\ \beta & \alpha & \beta & \ddots & \vdots \\ 0 & \beta & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \alpha & \beta \\ 0 & \dots & 0 & \beta & \alpha \end{bmatrix}. \quad (4.2.2)$$

**(4-2.a)** (15 min.) Reformulate Eq. (4.2.1) as a linear least squares problem in the usual form:

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^k}{\operatorname{argmin}} \|\mathbf{Ax} - \mathbf{b}\|_2 \quad (4.2.3)$$

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 (4-2.a) → 4-2-1-0:TridiagLeastSquares1h.pdf

SOLUTION for (4-2.a) → 4-2-1-1:TridiagLeastSquares1s.pdf ▲

**(4-2.b)** (20 min.) [ depends on Sub-problem (4-2.a) ]

Write a C++ function

```
VectorXd lsqEst(const VectorXd &z, const VectorXd &c);
```

that computes the optimal parameters  $\alpha^*$  and  $\beta^*$  according to Eq. (4.2.1) from data vectors  $\mathbf{z}$  and  $\mathbf{c}$  (i.e.  $\mathbf{z}$  and  $\mathbf{c}$  from Eq. (4.2.1)). Use the *method of normal equations*.

HIDDEN HINT 1 for (4-2.b) → 4-2-2-0:TridiagLeastSquares2h.pdf

After the matrix  $\mathbf{A}$  has been initialized, the solution just copies [Lecture → Code 3.2.0.1].

SOLUTION for (4-2.b) → 4-2-2-1:TridiagLeastSquares2s.pdf ▲

**End Problem 4-2**, 35 min.

### Problem 4-3: Linear Data Fitting

Abstract linear data fitting and its connection to the linear least squares problem [Lecture → Eq. (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

$$f(t) = \sum_{j=1}^4 \gamma_j \phi_j(t) \quad (4.3.1)$$

of the functions

$$\phi_1(t) = \frac{1}{t}, \quad \phi_2(t) = \frac{1}{t^2}, \quad \phi_3(t) = e^{-(t-1)}, \quad \phi_4(t) = e^{-2(t-1)}. \quad (4.3.2)$$

We aim for calculating the coefficients  $\gamma_j$ ,  $j = 1, 2, 3, 4$ , such that

$$\sum_{i=1}^{10} (f(t_i) - f_i)^2 \rightarrow \min. \quad (4.3.3)$$

(4-3.a)  Write a C++ function

```
VectorXd data_fit_normal(const VectorXd &t, const VectorXd &f);
```

that computes a least squares estimate for the coefficients  $\gamma_i$  by related to the data (4.3.2) and (4.3.3) by means of solving the *normal equations*.

SOLUTION for (4-3.a) → 4-3-1-0:ldf1.pdf



(4-3.b)  (20 min.) Write a C++ function

```
VectorXd data_fit_qr(const VectorXd &t, const 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 (4-3.b) → 4-3-2-0:ldf2.pdf



(4-3.c)  (10 min.)

Write a C++ function

```
VectorXd fit_plot(const VectorXd &gamma, const VectorXd &t)
```

that takes the estimated parameters  $\gamma_i$ ,  $i = 1, 2, 3, 4$ , as argument vector gamma, a vector t of times  $\tau_j$ , and returns the vector of values  $f(\tau_j)$ .

SOLUTION for (4-3.c) → 4-3-3-0:ldf3.pdf



(4-3.d)  (10 min.) Write a function

`VectorXd error_plot(const 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 (4.3.1) using the values for the  $\gamma_j$ -coefficients passed in the `gamma` argument.

SOLUTION for (4-3.d) → [4-3-4-0:s5.pdf](#)



**End Problem 4-3 , 40 min.**

### Problem 4-4: QR-Factorization of Tridiagonal Matrices

In [Lecture → Rem. 3.3.3.21] 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 → § 3.3.3.14] and their compressed storage [Lecture → Rem. 3.3.20].

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);
};
```

Let's 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  $\mathbf{d}$ ,  $\mathbf{l}$ ,  $\mathbf{u}$ . Then the matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  stored in an **TriDiagonalMatrix** object is defined by

$$(A)_{i,j} := \begin{cases} (d)_i & \text{for } i = j, \\ (u)_i & \text{for } j = i + 1, \\ (l)_j & \text{for } i = j + 1, \\ 0 & \text{else,} \end{cases} \quad i, j \in \{1, \dots, n\}.$$

$$\Leftrightarrow A = \begin{bmatrix} d_1 & u_1 & & & \\ l_1 & d_2 & \ddots & & \\ & \ddots & \ddots & u_{n-1} & \\ & & l_{n-1} & d_n & \end{bmatrix}, \quad \mathbf{d} = [d_1, \dots, d_n]^\top \in \mathbb{R}^n, \\ \mathbf{u} = [u_1, \dots, u_{n-1}]^\top \in \mathbb{R}^{n-1}, \\ \mathbf{l} = [l_1, \dots, l_{n-1}]^\top \in \mathbb{R}^{n-1}.$$

(4-4.a) (15 min.) Prove that for a *tridiagonal* invertible matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  a QR-factorization  $\mathbf{A} = \mathbf{Q}\mathbf{R}$ ,  $\mathbf{Q} \in \mathbb{R}^{n,n}$  orthogonal,  $\mathbf{R} \in \mathbb{R}^{n,n}$  upper triangular with positive diagonal entries, satisfies

$$(R)_{i,j} = 0 \quad \text{for } j > i + 2 \quad \text{and} \quad (Q)_{i,j} = 0 \quad \text{for } i > j + 1.$$

HIDDEN HINT 1 for (4-4.a) → [4-4-1-0:s1h1.pdf](#)

SOLUTION for (4-4.a) → [4-4-1-1:s1.pdf](#)

To store a QR-factorization  $\mathbf{A} = \mathbf{Q}\mathbf{R}$  of a real-valued tridiagonal matrix  $\mathbf{A} \in \mathbb{R}^{n,n}$  we use the data structure

```
class TriDiagonalQR {
public:
```

```

explicit TriDiagonalQR(const TriDiagonalMatrix &A);

template <typename VectType>
Eigen::VectorXd applyQT(const VectType &x) const;
template <typename VectType>
Eigen::VectorXd solve(const VectType &b) 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
public:
// For debugging purposes: extract factors as dense matrices
std::pair<Eigen::MatrixXd, Eigen::MatrixXd> getQRFactors(void) const;
};

```

Here the non-zero entries of  $\mathbf{R}$  are stored in an  $n \times 3$ -matrix  $\mathbf{B}$  adopting the convention

$$(\mathbf{R})_{i,j} := \begin{cases} (\mathbf{B})_{i,1} & , \text{ if } i = j , \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,2} & , \text{ if } j = i + 1 , \\ (\mathbf{R})_{i,j} = (\mathbf{B})_{i,3} & , \text{ if } j = i + 2 , \\ 0 & \text{else,} \end{cases} \quad i, j \in \{1, \dots, n\}, \quad (4.4.1)$$

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. 3.3.3.21]. Every Givens rotation is stored through a single number  $\rho$  following the convention described in [Lecture → Rem. 3.3.3.20]. These  $n - 1$  numbers are collected in the data member `rho`. Its  $j$ -th entry corresponds to the Givens rotation acting on rows  $j1$  &  $j + 1$ .

**(4-4.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 3.3.3.16]. The function is to return the triplet  $(\rho, \gamma, \sigma)$ , where  $\mathbf{G} = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}$  is the transformation matrix, and  $\rho \in \mathbb{R}$  encodes it according to the rule set

$$\begin{cases} \rho = \pm 1 & \Rightarrow \gamma = 0, \sigma = \pm 1 \\ |\rho| < 1 & \Rightarrow \sigma = 2\rho, \gamma = \sqrt{1 - \sigma^2} \\ |\rho| > 1 & \Rightarrow \gamma = 2/\rho, \sigma = \sqrt{1 - \gamma^2}. \end{cases} \quad (4.4.2)$$

SOLUTION for (4-4.b) → [4-4-2-0:s2.pdf](#)

(4-4.c) (45 min.) [ depends on [Lecture → Rem. 3.3.3.21] ]

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  $\mathbf{A}$ .

HIDDEN HINT 1 for (4-4.c) → [4-4-3-0:h2a.pdf](#)

HIDDEN HINT 2 for (4-4.c) → [4-4-3-1:h2b.pdf](#)

SOLUTION for (4-4.c) → [4-4-3-2:s2a.pdf](#)



(4-4.d) (30 min.) Provide an *efficient* implementation of the method `applyQT()` of **TriDiagonalQR** that computes  $\mathbf{Q}^T \mathbf{x}$  for a supplied vector  $\mathbf{x} \in \mathbb{R}^n$  (passed in  $\mathbf{x}$ ).

The type **VecType** must describe a vector with real-valued entries supplying component access through operator `[] const` and a `size()` method.

SOLUTION for (4-4.d) → [4-4-4-0:s3.pdf](#)



(4-4.e) (30 min.) [ depends on [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 (4-4.e) → [4-4-5-0:h41.pdf](#)

SOLUTION for (4-4.e) → [4-4-5-1:s4.pdf](#)



(4-4.f) (10 min.) [ depends on Sub-problem (4-4.b), Sub-problem (4-4.d), Sub-problem (4-4.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  $n \rightarrow \infty$ ?

SOLUTION for (4-4.f) → [4-4-6-0:s5.pdf](#)



(4-4.g) (20 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 (4-4.g) → [4-4-7-0:s6h1.pdf](#)

SOLUTION for (4-4.g) → [4-4-7-1:.pdf](#)



**End Problem 4-4 , 170 min.**

### Problem 4-5: 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  $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^\top$  be the singular value decomposition of  $\mathbf{A} \in \mathbb{C}^{m,n}$ ,  $m \geq n$ . Consider the following problem:

$$\mathbf{x}^* = \underset{\|\mathbf{x}\|_2=1}{\operatorname{argmin}} \{ \|\mathbf{Ax}\|_2 \} \quad (4.5.1)$$

(4-5.a) (45 min.) Show that the solution of (4.5.1) is:

$$\mathbf{x}^* = \mathbf{v}_n := \mathbf{V}_{:,n} \quad (4.5.2)$$

Show that the following holds:

$$\|\mathbf{Av}_n\|_2 = \sigma_n := \Sigma_{n,n}. \quad (4.5.3)$$

HIDDEN HINT 1 for (4-5.a) → [4-5-1-0:MinimumSVD1h.pdf](#)

SOLUTION for (4-5.a) → [4-5-1-1:MinimumSVD1s.pdf](#) ▲

**End Problem 4-5 , 45 min.**

### Problem 4-6: Sparse Approximate Inverse (SPAI [GRH97])

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]. The last subproblem is connected with [Lecture → Section 10.3].

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 as  $\mathbf{A}$ :

$$\mathcal{P}(\mathbf{A}) := \{\mathbf{X} \in \mathbb{R}^{N,N} : (\mathbf{A})_{ij} = 0 \Rightarrow (\mathbf{X})_{ij} = 0\}. \quad (4.6.1)$$

The “primitive” SPAI (sparse approximate inverse)  $\mathbf{B}$  of  $\mathbf{A}$  is defined as

$$\mathbf{B} := \underset{\mathbf{X} \in \mathcal{P}(\mathbf{A})}{\operatorname{argmin}} \|\mathbf{I} - \mathbf{AX}\|_F, \quad (4.6.2)$$

where  $\|\cdot\|_F$  stands for the Frobenius norm. The solution of (4.6.1) 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 [GRH97].

- (4-6.a)** ☐ 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 (4-6.a) → 4-6-1-0:spaih1.pdf

SOLUTION for (4-6.a) → 4-6-1-1:spai1.pdf ▲

- (4-6.b)** ☐ Implement an efficient C++ function

```
SparseMatrix<double> spai(const Eigen::SparseMatrix<double> & A);
```

for the computation of  $\mathbf{B}$  according to (4.6.2). 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.

HIDDEN HINT 1 for (4-6.b) → 4-6-2-0:spai2h.pdf

SOLUTION for (4-6.b) → 4-6-2-1:spai2s.pdf ▲

- (4-6.c)** ☐ What is the total asymptotic computational effort of `spai` in terms of the problem size parameters  $N$  and  $n$ .

SOLUTION for (4-6.c) → 4-6-3-0:SPAI3.pdf ▲

- (4-6.d)** ☐ 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 4.6.6: Initialization of matrix $\mathbf{A}$

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 (4.6.2). Use  $\mathbf{b} = [1, 1, \dots, 1]^T$ . 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  $n = 2^{1, \dots, L}$ .

SOLUTION for (4-6.d) → [4-6-4-0:spai4.pdf](#)



### End Problem 4-6

### Problem 4-7: 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  $\mathbf{p}_i \in \mathbb{R}^2, i = 1, \dots, n$ , we have to decide whether those point represent a stop sign, or a “priority road sign”. In this exercise we consider  $n = 8$ .

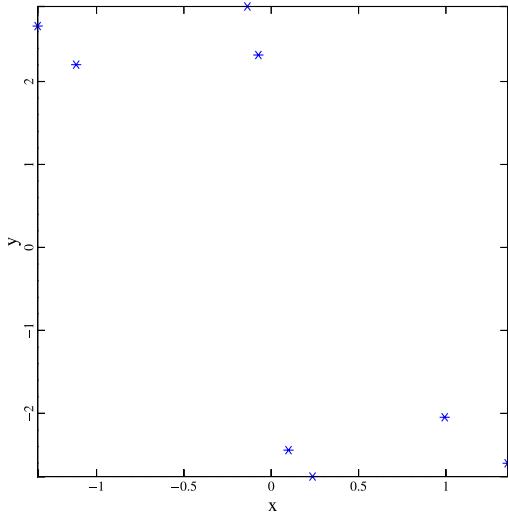


Fig. 11

Example

of input points  $\mathbf{p}^i$ .

The shape of the sign can be represented by a  $2 \times 8$  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)  $\mathbf{x}_{stop}^i \in \mathbb{R}^2$  resp.  $\mathbf{x}_{priority}^i \in \mathbb{R}^2$  for  $i = 2, \dots, 15$ .

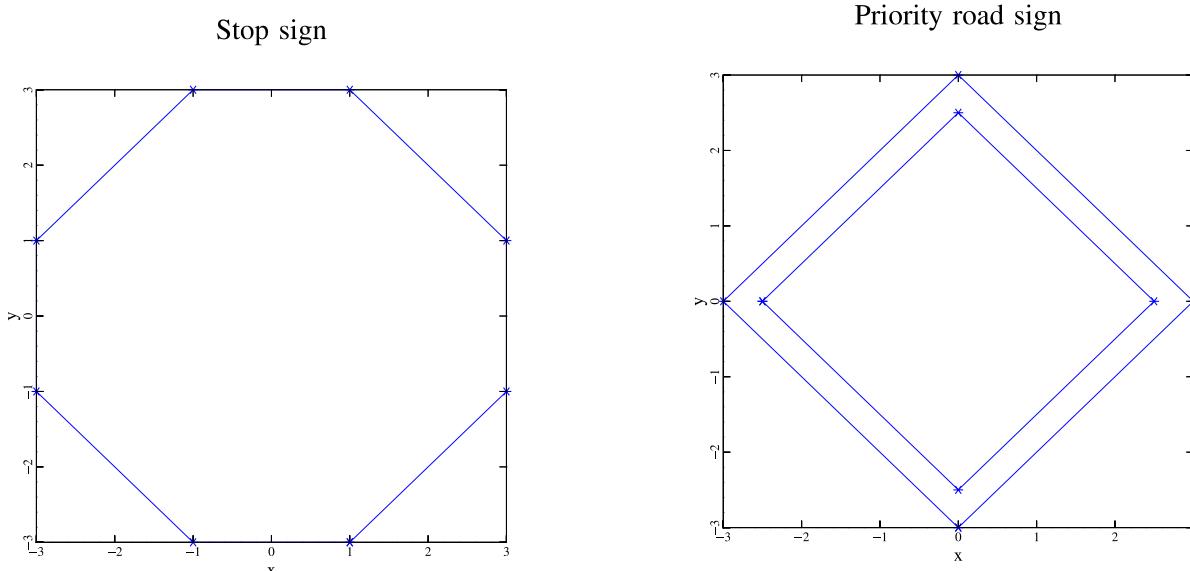


Fig. 12

8-points  $\mathbf{x}_{stop}^i$  defining the model of a stop sign.

Fig. 13

The 8-points  $\mathbf{x}_{priority}^i$  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  $i = 1, \dots, 15$ ; i.e. we can assume that there exists a matrix  $\mathbf{A} \in \mathbb{R}^{2,2}$ , s.t.

$$\mathbf{p}^i = \mathbf{Ax}^i, i = 1, \dots, n. \quad (4.7.1)$$

We do not know whether  $\mathbf{x}^i = \mathbf{x}_{stop}^i$  or  $\mathbf{x}^i = \mathbf{x}_{priority}^i$ .

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. (4.7.1) 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.

**(4-7.a)**  For the moment, assume we know the points  $\mathbf{x}^i$  (i.e. we know the shape of our sign). Explicitly write (4.7.1) as an overdetermined linear system:

$$\mathbf{w} = \mathbf{Bv},$$

whose least square solution will allow to determine the “best” linear transformation  $\mathbf{A}$  (in the least square sense). What is the size and what are the unknown of the system?

SOLUTION for (4-7.a) → [4-7-1-0:ShapeIdent1.pdf](#)

**(4-7.b)**  When does the matrix  $\mathbf{B}$  have full rank? Give a geometric interpretation of this condition.

SOLUTION for (4-7.b) → [4-7-2-0:si2s.pdf](#)

**(4-7.c)**  Implement a C++ function

```
MatrixXd shape_ident_matrix(const MatrixXd & X);
```

that returns the matrix  $\mathbf{B}$ . Pass the vectors  $\mathbf{x}^i$  as a  $2 \times n$  EIGEN matrix  $\mathbf{X}$ .

SOLUTION for (4-7.c) → [4-7-3-0:si3s.pdf](#)

(4-7.d)  Implement a C++ function

```
double solve_lsq(const MatrixXd & X,
                  const MatrixXd & P,
                  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  $\mathbf{X}$  resp.  $\mathbf{P}$ .

SOLUTION for (4-7.d) → [4-7-4-0:si4s.pdf](#)

(4-7.e)  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 MatrixXd Xstop,
               const MatrixXd Xpriority,
               const MatrixXd & P,
               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  $\mathbf{P}$ . Return the linear transformation in the matrix  $\mathbf{A}$ . The “model points”  $\mathbf{x}_{stop}^i$  resp.  $\mathbf{x}_{priority}^i$  are passed through  $\mathbf{X}_{stop}$  resp.  $\mathbf{X}_{priority}$ .

SOLUTION for (4-7.e) → [4-7-5-0:si5s.pdf](#)

(4-7.f)  Use the points provided in the variables  $P1$ ,  $P2$ , and  $P3$  in the `main()` function to identify the shape (or: which shape is best suited) of the objects defined by those points.

SOLUTION for (4-7.f) → [4-7-6-0:si6s.pdf](#)

**End Problem 4-7**

### Problem 4-8: 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 → Eq. (3.4.1.8)]. Thus, low rank matrices are of considerable interest for *matrix compression* (see [Lecture → Ex. 3.4.4.23]). 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 SVD [Lecture → Section 3.4.2] and QR factorization [Lecture → Section 3.3.3.4].

**(4-8.a)** (10 min.) Show that for a matrix  $\mathbf{X} \in \mathbb{R}^{m,n}$  the following statements are equivalent:

- (i)  $\text{rank}(\mathbf{X}) = k$
- (ii)  $\mathbf{X} = \mathbf{AB}^\top$  for matrices  $\mathbf{A} \in \mathbb{R}^{m,k}$ ,  $\mathbf{B} \in \mathbb{R}^{n,k}$ ,  $k \leq \min\{m, n\}$ , both of full rank.

HIDDEN HINT 1 for (4-8.a) → 4-8-1-0:LowRankRep1h.pdf

SOLUTION for (4-8.a) → 4-8-1-1:LowRankRep1s.pdf ▲

**(4-8.b)** (20 min.) [ depends on Sub-problem (4-8.a) ]

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  $\text{rank}(\mathbf{X}) = k$  into  $\mathbf{AB}^\top$ , as in Sub-problem (4-8.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  $\text{rank}(\mathbf{X}) = k$  is in doubt.

HIDDEN HINT 1 for (4-8.b) → 4-8-2-0:LowRankRep2h.pdf

SOLUTION for (4-8.b) → 4-8-2-1:LowRankRep2s.pdf ▲

**(4-8.c)** (15 min.) Let  $\mathbf{A} \in \mathbb{R}^{m,k}$ ,  $\mathbf{B} \in \mathbb{R}^{n,k}$ , with  $k \leq \min\{m, n\}$ . Based on the **economical QR-factorization** [Lecture → Thm. 3.3.3.4] of both  $\mathbf{A}$  and  $\mathbf{B}$

$$\begin{aligned} \mathbf{A} &= \mathbf{Q}_A \mathbf{R}_A , \quad \mathbf{Q}_A \in \mathbb{R}^{m,k} \quad \text{with} \quad \mathbf{Q}_A^\top \mathbf{Q}_A = \mathbf{I}, \quad \mathbf{R}_A \in \mathbb{R}^{k,k} \quad \text{upper triangular ,} \\ \mathbf{B} &= \mathbf{Q}_B \mathbf{R}_B , \quad \mathbf{Q}_B \in \mathbb{R}^{n,k} \quad \text{with} \quad \mathbf{Q}_B^\top \mathbf{Q}_B = \mathbf{I}, \quad \mathbf{R}_B \in \mathbb{R}^{k,k} \quad \text{upper triangular ,} \end{aligned}$$

outline, how the **economical singular value decomposition**  $\mathbf{AB}^\top = \mathbf{U}\Sigma\mathbf{V}^\top$  [Lecture → Eq. (3.4.1.4)] of  $\mathbf{AB}^\top$  can be computed using only the singular value decomposition of a  $k \times k$ -matrix.

SOLUTION for (4-8.c) → 4-8-3-0:s4.pdf ▲

**(4-8.d)** (30 min.) 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 MatrixXd & A, const MatrixXd & B);
```

that calculates a singular value decomposition (SVD) [Lecture → Thm. 3.4.1.1] of the product  $\mathbf{AB}^\top = \mathbf{U}\Sigma\mathbf{V}^\top$ , where  $\mathbf{U}, \mathbf{V} \in \mathbb{R}^{n,k}$  have orthogonal columns and  $\Sigma \in \mathbb{R}^{k,k}$  is a diagonal matrix. The SVD-factors are returned as a triple  $(\mathbf{U}, \Sigma, \mathbf{V})$ .

HIDDEN HINT 1 for (4-8.d) → 4-8-4-0:LowRankRep3h.pdf

SOLUTION for (4-8.d) → [4-8-4-1:LowRankRep3s.pdf](#) ▲

(4-8.e) ☐ (10 min.) [ depends on Sub-problem (4-8.d) ]

What is the asymptotic computational cost of the function `svd_AB` for (fixed) small  $k$  and  $m = n \rightarrow \infty$ ? Discuss the effort required by the different steps of your algorithm.

SOLUTION for (4-8.e) → [4-8-5-0:LowRankRep4s.pdf](#) ▲

(4-8.f) ☐ (10 min.) If  $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{m,n}$  satisfy  $\text{rank}(\mathbf{Y}) = \text{rank}(\mathbf{X}) = k$ , show that  $\text{rank}(\mathbf{Y} + \mathbf{X}) \leq 2k$ .

SOLUTION for (4-8.f) → [4-8-6-0:LowRankRep5s.pdf](#) ▲

(4-8.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 sum  $\mathbf{X} + \mathbf{Y}$  as  $\mathbf{X} + \mathbf{Y} = \mathbf{AB}^\top$ , with  $\mathbf{A} \in \mathbb{R}^{m,2k}$  and  $\mathbf{B} \in \mathbb{R}^{n,2k}$ .

HIDDEN HINT 1 for (4-8.g) → [4-8-7-0:LowRankRep6h.pdf](#)

SOLUTION for (4-8.g) → [4-8-7-1:LowRankRep6s.pdf](#) ▲

(4-8.h) ☐ (25 min.) [ depends on Sub-problem (4-8.d), Sub-problem (4-8.g) ]

Rely on the previous subproblems to write an *efficient* C++ function

```
std::pair<Eigen::MatrixXd, Eigen::MatrixXd>
rank_k_approx(const MatrixXd & Ax, const MatrixXd & Ay, const
               MatrixXd & Bx, const MatrixXd & By);
```

that return 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  $\mathbf{X} + \mathbf{Y} = \mathbf{A}_X \mathbf{B}_X^\top + \mathbf{A}_Y \mathbf{B}_Y^\top$ :

$$\mathbf{Z} := \underset{\substack{\mathbf{M} \in \mathbb{R}^{m,n} \\ \text{rank}(\mathbf{M}) \leq k}}{\operatorname{argmin}} \left\| \mathbf{A}_X \mathbf{B}_X^\top + \mathbf{A}_Y \mathbf{B}_Y^\top - \mathbf{M} \right\|_F$$

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  $By$ .

HIDDEN HINT 1 for (4-8.h) → [4-8-8-0:LowRankRep7h.pdf](#)

SOLUTION for (4-8.h) → [4-8-8-1:LowRankRep7s.pdf](#) ▲

(4-8.i) ☐ (10 min.) [ depends on Sub-problem (4-8.h) ]

What is the asymptotic computational cost of the function `rank_k_approx` for a small  $k$  and  $m = n \rightarrow \infty$ ?

SOLUTION for (4-8.i) → [4-8-9-0:LowRankRep8s.pdf](#) ▲

**End Problem 4-8 , 140 min.**

### Problem 4-9: 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 smallest matrix fitting a linear system of equations with given solution and right-hand side vector:

$$\text{Given } \mathbf{n} \in \mathbb{N}, \mathbf{z} \in \mathbb{R}^n, \mathbf{g} \in \mathbb{R}^n, \text{ find } \mathbf{M}^* = \underset{\mathbf{M} \in \mathbb{R}^{n,n}, \mathbf{M}\mathbf{z}=\mathbf{g}}{\operatorname{argmin}} \|\mathbf{M}\|_F, \quad (4.9.1)$$

where  $\|\cdot\|_F$  denotes the Frobenius norm of a matrix as introduced in [Lecture → Def. 3.4.4.16].

**(4-9.a)**  (15 min.) Reformulate the problem as an equivalent standard linearly constrained least squares problem [Lecture → Eq. (3.6.0.1)]

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^N, \mathbf{Cx}=\mathbf{d}}{\operatorname{argmin}} \|\mathbf{Ax} - \mathbf{b}\|_2, \quad (4.9.2)$$

for suitable matrices  $\mathbf{A}$ ,  $\mathbf{C}$  and vectors  $\mathbf{b}$  and  $\mathbf{d}$ , see [Lecture → Eq. (3.6.0.1)]. These matrices and vectors have to be specified based on  $\mathbf{z}$  and  $\mathbf{g}$ .

SOLUTION for (4-9.a) → [4-9-1-0:lsqfrobo.pdf](#)

**(4-9.b)**  (5 min.) [ depends on Sub-problem (4-9.a) ]

State a necessary and sufficient condition for the matrix  $\mathbf{C}$  found in Sub-problem (4-9.a) to possess full rank.

SOLUTION for (4-9.b) → [4-9-2-0:lsqf1.pdf](#)

**(4-9.c)**  (15 min.) [ depends on Sub-problem (4-9.a) ]

State the **augmented normal equations** [Lecture → Eq. (3.6.1.6)] corresponding to the constrained linear least squares problem found in Sub-problem (4-9.a) and give necessary and sufficient conditions on  $\mathbf{g}$  and  $\mathbf{z}$  that ensure existence and uniqueness of solutions.

HIDDEN HINT 1 for (4-9.c) → [4-9-3-0:lsqfh1.pdf](#)

SOLUTION for (4-9.c) → [4-9-3-1:.pdf](#)

**(4-9.d)**  (20 min.) [ depends on Sub-problem (4-9.c) ]

Write a C++ function

```
MatrixXd min_frob(const VectorXd & z, const VectorXd & g);
```

that computes the solution  $\mathbf{M}^*$  of the minimization problem (4.9.1) given the vectors  $\mathbf{z}, \mathbf{g} \in \mathbb{R}^n$ . Use the augmented normal equations [Lecture → Eq. (3.6.1.6)] derived in ((4-9.c)).

It is convenient to use EIGEN's built-in Kronecker product class **Eigen::KroneckerProduct** (#include <unsupported/Eigen/KroneckerProduct> required).

SOLUTION for (4-9.d) → [4-9-4-0:lsqfrobi.pdf](#)

(4-9.e)  (15 min.) [ depends on Sub-problem (4-9.d) ]

Using the test vectors  $\mathbf{z} = [1, 2, \dots, 100]^\top \in \mathbb{R}^{100}$  and  $\mathbf{g} = [1, -1, 1, \dots, 1, -1]^\top \in \mathbb{R}^{100}$ , implement a C++ function

```
bool testMformula(void);
```

that checks *in a numerical experiment* whether

$$\mathbf{M} = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2} \quad (4.9.6)$$

is a solution to (4.9.1). To that end call the function `min_frob` implemented in Sub-problem (4-9.d), conducts the test and return `true` if it succeeds `stdout`.

SOLUTION for (4-9.e) → 4-9-5-0:lsqfrobc.pdf



(4-9.f)  (15 min.) [ depends on Sub-problem (4-9.c) ]

Now, give a rigorous proof that (4.9.6) gives a solution of (4.9.1), provided that  $\mathbf{z} \neq \mathbf{0}$ .

SOLUTION for (4-9.f) → 4-9-6-0:lsqfrx.pdf



So far, we have simply applied the formulas from [Lecture → Section 3.6.1] to (4.9.1) 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.

(4-9.g)  (10 min.) As in [Lecture → Eq. (3.6.1.2)] and [Lecture → Eq. (3.6.1.3)] from the lecture notes, find a **Lagrange functional**  $L : \mathbb{R}^{n,n} \times \mathbb{R}^n \rightarrow \mathbb{R}$  such that:

$$\mathbf{M}^* = \underset{\mathbf{M} \in \mathbb{R}^{n,n}}{\operatorname{argmin}} \left\{ \sup_{\mathbf{m} \in \mathbb{R}^n} [L(\mathbf{M}, \mathbf{m})] \right\}. \quad (4.9.9)$$

The matrix  $\mathbf{M}^*$  should provide the solution of (4.9.1).

HIDDEN HINT 1 for (4-9.g) → 4-9-7-0:LSQFrobLagr1h.pdf

SOLUTION for (4-9.g) → 4-9-7-1:LSQFrobLagr1s.pdf



(4-9.h)  (10 min.) Find the derivative  $\operatorname{grad} \Phi \in \mathbb{R}^{n,n}$  of the function  $\Phi$  defined as:

$$\Phi : \mathbb{R}^{n,n} \rightarrow \mathbb{R}, \quad \Phi(\mathbf{X}) = \|\mathbf{X}\|_F^2 \quad (4.9.10)$$

HIDDEN HINT 1 for (4-9.h) → 4-9-8-0:LSQFrobLagr2h.pdf

SOLUTION for (4-9.h) → 4-9-8-1:LSQFrobLagr2s.pdf



(4-9.i)  (15 min.) [ depends on Sub-problem (4-9.h) ]

Use the result of Sub-problem (4-9.h) to derive **saddle point equations** for  $\operatorname{grad} L(\mathbf{M}, \mathbf{m}) = \mathbf{0}$ , for the  $\mathbf{L}$  obtained in Sub-problem (4-9.g).

HIDDEN HINT 1 for (4-9.i) → 4-9-9-0:LSQFrobLagr3h.pdf

SOLUTION for (4-9.i) → 4-9-9-1:LSQFrobLagr3s.pdf



**(4-9.j)**  (10 min.) [ depends on Sub-problem (4-9.i) ]

Solve the saddle point equations obtained in Sub-problem (4-9.i) in symbolic form.

HIDDEN HINT 1 for (4-9.j) → [4-9-10-0:LSQFrobLagr4h.pdf](#)

SOLUTION for (4-9.j) → [4-9-10-1:LSQFrobLagr4s.pdf](#)



**End Problem 4-9 , 130 min.**

### Problem 4-10: 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 over-determined 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  $x_1, \dots, x_n \in \mathbb{R}$  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  $d_{i,j} := x_j - x_i \in \mathbb{R}$ ,  $1 \leq i < j \leq n$ , 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

$$\begin{cases} x_j - x_i = d_{i,j}, & 1 \leq i < j \leq n - 1, \\ -x_i = d_{i,n}, & i \in \{1, \dots, n - 1\}. \end{cases} \quad (4.10.1)$$

Collecting the unknown positions in the vector  $\mathbf{x} := [x_1, \dots, x_{n-1}]^\top \in \mathbb{R}^{n-1}$ , the equations (4.10.1) can be recast as an  $m \times (n - 1)$  linear system of equations  $\mathbf{Ax} = \mathbf{b}$ , where  $\mathbf{b} \in \mathbb{R}^m$  contains each of the distances  $d_{i,j}$ ,  $1 \leq i < j \leq n$ , exactly once.

**(4-10.a)** (10 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 (4-10.a) → [4-10-1-0:s1.pdf](#)

**(4-10.b)** (18 min.) Based on EIGEN implement 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 (4-10.b) → [4-10-2-0:s4.pdf](#)

**(4-10.c)** (25 min.) [ depends on Sub-problem (4-10.b) ]

According to [Lecture → § 3.2.0.7] the **extended normal equations** for the generic overdetermined linear system  $\mathbf{Mx} = \mathbf{c}$ ,  $\mathbf{M} \in \mathbb{R}^{m,\ell}$ ,  $\mathbf{c} \in \mathbb{R}^m$ ,  $m \geq \ell$ , boil down to the linear system of equations

$$\begin{bmatrix} -\mathbf{I} & \mathbf{M} \\ \mathbf{M}^\top & \mathbf{O} \end{bmatrix} \begin{bmatrix} \mathbf{r} \\ \mathbf{x} \end{bmatrix} = \begin{bmatrix} \mathbf{c} \\ \mathbf{0} \end{bmatrix}. \quad (4.10.5)$$

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 (4.10.1) 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

$$[\mathbf{D}]_{i,j} := d_{i,j}, \quad 1 \leq i < j \leq n.$$

HIDDEN HINT 1 for (4-10.c) → 4-10-3-0:h3tt.pdf

SOLUTION for (4-10.c) → 4-10-3-1:s3.pdf ▲

**(4-10.d)** ☐ (15 min.) [ depends on Sub-problem (4-10.b) ]

The coefficient matrix  $\mathbf{M} \in \mathbb{R}^{n-1,n-1}$  of the **normal equations** [Lecture → Eq. (3.1.2.2)] associated with the overdetermined linear system of equations (4.10.1) is of the form

$$\mathbf{M} = \mathbf{D} - \mathbf{z}\mathbf{z}^T \quad \text{with} \quad \begin{array}{l} \text{a diagonal matrix } \mathbf{D} \in \mathbb{R}^{n-1,n-1}, \\ \text{and a column vector } \mathbf{z} \in \mathbb{R}^{n-1}. \end{array} \quad (4.10.8)$$

Compute  $\mathbf{D}$  and  $\mathbf{z}$ , and find a simple expression for the inverse  $\mathbf{M}^{-1}$ .

HIDDEN HINT 1 for (4-10.d) → 4-10-4-0:smw.pdf

SOLUTION for (4-10.d) → 4-10-4-1:s4.pdf ▲

**(4-10.e)** ☐ (15 min.) [ depends on Sub-problem (4-10.b) and Sub-problem (4-10.d) ]

Write an *efficient* C++ function

```
Eigen::VectorXd solveNormalEquations(
    const Eigen::MatrixXd &D);
```

that returns the least-squares solution of (4.10.1) by solving the corresponding normal equations. The signature is the same as for `solveExtendedNormalEquations()` from Sub-problem (4-10.c).

HIDDEN HINT 1 for (4-10.e) → 4-10-5-0:h4iA.pdf

SOLUTION for (4-10.e) → 4-10-5-1:s5.pdf ▲

**(4-10.f)** ☐ (5 min.) [ depends on Sub-problem (4-10.e) ]

What is the asymptotic complexity of your implementation of `solveNormalEquations()` in Sub-problem (4-10.e) for  $n \rightarrow \infty$ ?

SOLUTION for (4-10.f) → 4-10-6-0:s6.pdf ▲

**End Problem 4-10 , 88 min.**

### Problem 4-11: 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.

– COMMENT –This problem is connected with [Lecture → Section 1.4.3] and [Lecture → Section 3.4.4].

Recall the following result from elementary linear algebra:

#### Theorem 4.11.1. Representation of low-rank matrices

Every matrix  $\mathbf{M} \in \mathbb{R}^{m,n}$ ,  $m, n \in \mathbb{N}$ , with rank  $r := \text{rank}(\mathbf{M}) > 0$  can be factored as

$$\mathbf{M} = \mathbf{AB}^\top \quad \text{with} \quad \begin{aligned} \mathbf{A} &\in \mathbb{R}^{m,r}, \\ \mathbf{B} &\in \mathbb{R}^{n,r}. \end{aligned}$$

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;      // no. of rows
    unsigned int _n;      // no. 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.

(4-11.a) (15 min.)

Give a proof of Thm. 4.11.1.

HIDDEN HINT 1 for (4-11.a) → 4-11-1-0:h1.pdf

SOLUTION for (4-11.a) → 4-11-1-1:s1.pdf ▲

(4-11.b) (10 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 (4-11.b) → [4-11-2-0:s3.pdf](#)

**(4-11.c)** (5 min.) [ depends on Sub-problem (4-11.b) ]

What is the asymptotic complexity of your implementation of `MatrixLowRank::operator *` from Sub-problem (4-11.b) in terms of  $m, n, k \rightarrow \infty$ ,  $r := \text{rank}(\mathbf{M})$  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 (4-11.c) → [4-11-3-0:s3.pdf](#)

**(4-11.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 (4-11.d) → [4-11-4-0:s4.pdf](#)

**(4-11.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  $\text{trunc}_{\text{tol}}(\mathbf{M} + \mathbf{X})$ , where

- $\mathbf{X} \in \mathbb{R}^{m,n}$  is the matrix passed through the argument  $\mathbf{X}$ ,
- and, for  $\mathbf{Y} \in \mathbb{R}^{m,n}$ ,

$$\text{trunc}_{\text{tol}}(\mathbf{Y}) := \underset{\mathbf{R} \in \mathbb{R}^{m,n}}{\operatorname{argmin}} \left\{ \begin{array}{l} \text{rank}(\mathbf{R}) : \mathbf{R} \in \mathbb{R}^{m,n}, \\ \|\mathbf{Y} - \mathbf{R}\|_2 \leq \text{rtol} \|\mathbf{Y}\|_2, \\ \quad \text{or} \\ \|\mathbf{Y} - \mathbf{R}\|_2 \leq \text{atol} \end{array} \right\},$$

where  $\|\cdot\|_2$  designates the Euclidean matrix norm. The “internal rank”  $_r$  must be updated.

Your implementation may assume that

$$\text{rank}(\mathbf{M}) + \text{rank}(\mathbf{X}) \leq \min\{m, n\}.$$

HIDDEN HINT 1 for (4-11.e) → [4-11-5-0:h5a.pdf](#)

HIDDEN HINT 2 for (4-11.e) → [4-11-5-1:h5t.pdf](#)

SOLUTION for (4-11.e) → [4-11-5-2:s4.pdf](#)

**End Problem 4-11 , 80 min.**

# Chapter 5

## Filtering Algorithms

### Problem 5-1: 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 4.2.4].

This problem takes for granted that you know about the discrete Fourier transform and its connection with discrete convolutions

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. 4.2.4.14]; 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 → Eq. (4.2.4.15)], 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`):

```
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 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 as supplied by the function `fft2r` found in “FFT/fft.hpp”, see also [Lecture → Code 4.2.4.6] and ??.

$$V(\mathbf{C}) = \sum_{k_1=0}^{n-1} \sum_{k_2=0}^{m-1} \left( \left( \frac{n}{2} - |k_1 - \frac{n}{2}| \right)^2 + \left( \frac{m}{2} - |k_2 - \frac{m}{2}| \right)^2 \right) |\hat{\mathbf{C}}_{k_1, k_2}|^2.$$

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

find  $f \in \mathbb{R}$  such that  $V(\mathbf{B}(f))$  becomes maximal.

- (5-1.a)  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 **Portable Graymap (PGM)** format.

For this you can use objects of type `PGMObject` (found in “`pgm.hpp`”). You can find example of usages of this class in “`examples/pgm_example.cpp`”.

SOLUTION for (5-1.a) → [5-1-1-0:render.pdf](#)



- (5-1.b)  The two-dimensional discrete Fourier transform can be performed using the C++ function `fft2r`. Write a C++ function

```
void plot_freq(double focus);
```

that creates 3D plots of the (modulus of the) 2D DFTs of the images obtained in sub-problem (5-1.a). Clamp the data between 0 and 8000.

SOLUTION for (5-1.b) → [5-1-2-0:plotfft2.pdf](#)



- (5-1.c)  In plots from sub-problem Sub-problem (5-1.b), mark (or explain) the regions corresponding to the high and low frequencies.

HIDDEN HINT 1 for (5-1.c) → [5-1-3-0:afh1.pdf](#)

SOLUTION for (5-1.c) → [5-1-3-1:explain.pdf](#)



- (5-1.d)  Write a C++ function

```
double void high_frequency_content(const MatrixXd & C);
```

that returns the value  $V(\mathbf{C})$  for a given matrix  $\mathbf{C}$ . Write a C++ function

```
void plotV();
```

that 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 (5-1.d) → [5-1-4-0:plotV.pdf](#)



- (5-1.e)  Write an *efficient* (you want the auto-focus procedure to take not longer than a few seconds!) 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  $f_0$ ?

To locate the change of slope of  $V(\mathbf{B}(f))$  in  $[0, 5]$  you should use the *bisection algorithm* [Lecture → Section 8.3.1] for finding a zero of the derivative  $\partial V(\mathbf{B}(f))/\partial f$ , which will, hopefully, mark the location of the maximum of  $V(f)$ .

The bisection algorithm for finding a zero of a continuous function  $\varphi : [a, b] \rightarrow \mathbb{R}$  with  $\varphi(a) \cdot \varphi(b) < 0$  is rather simple and lucidly demonstrated in [Lecture → Code 8.3.1.2].

Of course, the derivative is not available, so that we have to approximate it crudely by means of a difference quotient

$$\frac{\partial V(f)}{\partial f} \approx \frac{V(\mathbf{B}(f + \delta f)) - V(\mathbf{B}(f - \delta f))}{2\delta f}.$$

You can assume that the smallest step of the auto-focus mechanism is **0.05** and so the natural choice for  $\delta f$  is  $\delta f = 0.05$ . This maximal resolution also tells you a priori how many bisection steps should be performed.

Use the structure of the function  $V(\mathbf{B}(f))$  that you observed in sub-problem (5-1.d). Use as few evaluations of  $V(\mathbf{B}(f))$  as possible!

SOLUTION for (5-1.e) → [5-1-5-0:bisect.pdf](#)



**End Problem 5-1**

### Problem 5-2: 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 and the idea of least-squares fitting

A planet orbits the sun in a closed, planar curve. For equispaced polar angles  $\varphi_j = 2\pi \frac{j}{n}$ ,  $j = 0, \dots, n$ ,  $n \in \mathbb{N}$  its distance from the sun is measured and found to be  $d_j$ ,  $j = 0, \dots, n - 1$ . An approximation of the planet's trajectory can be found as follows.

Write  $\mathcal{P}_m^C$ ,  $m \in \mathbb{N}$  for the space of real trigonometric polynomials of the form

$$p(t) = \sum_{k=0}^m c_k \cos(2\pi kt), \quad c_k \in \mathbb{R}. \quad (5.2.1)$$

The aim is to approximate the distance from the sun as a function of the angle with a trigonometric polynomial  $p^* \in \mathcal{P}_m^C$  that solves

$$p^* = \underset{p \in \mathcal{P}_m^C}{\operatorname{argmin}} \sum_{j=0}^{n-1} \left| p\left(\frac{\varphi_j}{2\pi}\right) - d_j \right|^2. \quad (5.2.2)$$

Throughout, we assume  $m < n$ .

(5-2.a) □ Recast this task as a standard linear least squares problem

$$\mathbf{x}^* = \underset{\mathbf{x} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{Ax} - \mathbf{b}\|_2 \quad (5.2.3)$$

with suitable matrix  $\mathbf{A}$  and vectors  $\mathbf{b}, \mathbf{x}$ .

SOLUTION for (5-2.a) → 5-2-1-0:relsq.pdf

(5-2.b) □ State the normal equations for this linear least squares problem in explicit form.

With  $i^2 = -1$  use the identity  $\cos(2\pi x) = \frac{1}{2}(e^{2\pi ix} + e^{-2\pi ix})$  and the geometric sum formula

$$\sum_{\ell=0}^{n-1} q^\ell = \frac{1 - q^n}{1 - q}. \quad (5.2.4)$$

SOLUTION for (5-2.b) → 5-2-2-0:normleq.pdf

(5-2.c) □ Write a C++ function

```
VectorXd find_c(const VectorXd & d, unsigned int m);
```

that computes the coefficients  $c_k$ ,  $k = 0, \dots, m$  of  $p^*$  in the form (??) for arbitrary inputs  $d_j \in \mathbb{R}$ ,  $j = 0, \dots, n - 1$ . These are passed as vector  $d$ . The function must not have asymptotic complexity worse than  $O(n \log n)$  in terms of the problem size parameter  $n$ .

Use the provided function `VectorXd fftr(const VectorXd &)` that realises the discrete Fourier transform (defined in "FFT/fft.hpp"). Also note the identity

$$\sum_{k=0}^{n-1} e^{\frac{2\pi i}{n} d_k} = \overline{\sum_{k=0}^{n-1} e^{-\frac{2\pi i}{n} j k} d_k}, \quad (5.2.6)$$

where the over-line denotes the complex conjugation.

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

(5-2.d)  Test your implementation. To that end, test your routine by fitting a trigonometric polynomial of degree 3 with the data values `d` found in `main()` of `fftlsq.cpp`. ([Get it on GitLab \(fftlsq.cpp\)](#)). Store the result in the vector `g`.

As comparison, we obtain the vector  $g = [0.984988, -0.00113849, -0.00141515]$  when running our code. ▲

**End Problem 5-2**

### Problem 5-3: Multiplication and division of polynomials based on FFT

In [Lecture → Rem. 4.1.0.24] we saw that the formula of discrete convolution [Lecture → Def. 4.1.0.22] also gives the coefficients of products of polynomials. Thus, in light of the realization of discrete convolutions by means of DFT [Lecture → Section 4.2.1], the Fast Fourier Transform (FFT) becomes relevant for efficiently multiplying polynomials of very high degree.

Knowledge about DFT and discrete convolutions

Let two large numbers  $m, n \gg 1$  and two polynomials

$$u(x) = \sum_{j=0}^{m-1} \alpha_j x^j, \quad v(x) = \sum_{j=0}^{n-1} \beta_j x^j, \quad \alpha_j, \beta_j \in \mathbb{C} \quad (5.3.1)$$

be given. We consider their product

$$uv(x) = u(x)v(x) \quad (5.3.2)$$

The tasks are:

1. Efficiently compute the coefficients of the polynomial  $uv$  (with degree  $m + n - 1$ ) from (5.3.2) is fulfilled.
2. Given  $uv$  and  $u$  in terms of their coefficients, find the polynomial  $v$  again, if it exists (polynomial division).

**(5-3.a)**  Given polynomials  $u$  and  $v$  with degrees  $m - 1$  and  $n - 1$ , respectively, their coefficients can be stored as C++ vectors `Eigen::VectorXd u` and `Eigen::VectorXd v` of size  $m$  and  $n$  (the known term is stored in position 0).

Write a C++ function that “naively”, i.e. using a simple loop-based implementation, computes the vector of coefficients of the polynomial which is the multiplication between polynomials  $u$  and  $v$ :

```
VectorXd polyMult_naive(const VectorXd & u, const VectorXd & v);
```

You can use EIGEN classes. Also determine the asymptotic complexity of your algorithm for  $mn \rightarrow \infty$  (separately).

HIDDEN HINT 1 for (5-3.a) → 5-3-1-0:PolyDiv1h.pdf

SOLUTION for (5-3.a) → 5-3-1-1:PolyDiv1s.pdf ▲

**(5-3.b)**  Write a C++ function that efficiently computes the vector of coefficients of the polynomial which is the multiplication between polynomials  $u$  and  $v$ :

```
VectorXd polyMult_fast(const VectorXd & u, const VectorXd & v);
```

You can use EIGEN classes.

HIDDEN HINT 1 for (5-3.b) → 5-3-2-0:PolyDiv2h.pdf

SOLUTION for (5-3.b) → 5-3-2-1:PolyDiv2s.pdf ▲

**(5-3.c)**  Determine the complexity of your algorithm in (5-3.b).

HIDDEN HINT 1 for (5-3.c) → 5-3-3-0:PolyDiv3h.pdf

SOLUTION for (5-3.c) → 5-3-3-1:PolyDiv3s.pdf ▲

- (5-3.d)  What does it mean to apply a discrete Fourier transform to the coefficients of a polynomial? In other words, what is an equivalent operation that can be performed on a polynomial which leads to the same result?

HIDDEN HINT 1 for (5-3.d) → [5-3-4-0:PolyDiv4h.pdf](#)

SOLUTION for (5-3.d) → [5-3-4-1:PolyDiv4s.pdf](#) ▲

- (5-3.e)  Now we will handle with polynomial division.

Consider the division between polynomials  $uv$  and  $u$ . How can you check whether  $u$  divides  $uv$ ?

HIDDEN HINT 1 for (5-3.e) → [5-3-5-0:PolyDiv5h.pdf](#)

SOLUTION for (5-3.e) → [5-3-5-1:PolyDiv5s.pdf](#) ▲

- (5-3.f)  Write a C++ function that efficiently computes the vector of coefficients of the polynomial  $v$ , division between polynomials  $uv$  and  $u$  from the problems above:

```
VectorXd polyDiv(const VectorXd & uv, const VectorXd & u);
```

You can use EIGEN classes.

HIDDEN HINT 1 for (5-3.f) → [5-3-6-0:PolyDiv6h.pdf](#)

SOLUTION for (5-3.f) → [5-3-6-1:PolyDiv6s.pdf](#) ▲

**End Problem 5-3**

### Problem 5-4: Solving triangular Toeplitz systems

In [Lecture → Section 4.5] we learned about **Toeplitz matrices**, the class of matrices with constant diagonals [Lecture → Def. 4.5.0.8]. Obviously,  $m \times n$  Toeplitz matrices are ***data-sparse*** in the sense that it takes only  $m + n - 1$  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 4.5.1] 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].

#### DFT and circulant matrices

In [Lecture → § 4.1.0.1] 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 → Eq. (4.1.0.14)]; see also [Lecture → Def. 4.1.0.22]. 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:

$$\begin{bmatrix} y_0 \\ \vdots \\ y_{n-1} \end{bmatrix} = \underbrace{\begin{bmatrix} h_0 & 0 & \cdots & \cdots & 0 \\ h_1 & h_0 & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \ddots & 0 \\ h_{n-1} & \cdots & \cdots & h_1 & h_0 \end{bmatrix}}_{=: \mathbf{H}_n \in \mathbb{R}^{n,n}} \begin{bmatrix} x_0 \\ \vdots \\ x_{n-1} \end{bmatrix} \quad (5.4.1)$$

The matrix  $\mathbf{H}_n \in \mathbb{R}^{n,n}$  is a lower triangular matrix with constant diagonals:

$$(H)_{l,j} = \begin{cases} h_{l-j} & \text{if } l \geq j \\ 0 & \text{else} \end{cases}$$

This matrix is clearly not circulant: see [Lecture → Def. 4.1.0.38]. At the same time, it still belongs to the class of **Toeplitz matrices**, see [Lecture → Def. 4.5.0.8], which generalizes the class of circulant matrices. (Recall [Lecture → Def. 4.5.0.8]:  $\mathbf{T} \in \mathbb{K}^{m,n}$  is a Toeplitz matrix if there is a vector  $\mathbf{u} = (u_{-m+1}, \dots, u_{n-1}) \in \mathbb{K}^{m+n-1}$  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 → Eq. (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.

- (5-4.a)**  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 composed 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 (5-4.a) → 5-4-1-0:Toeplitz1h.pdf

SOLUTION for (5-4.a) → 5-4-1-1:Toeplitz1s.pdf

(5-4.b) ☐ Write the following C++ function:

```
Eigen::MatrixXd toeplitz(const VectorXd &c, const 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**  $\mathbf{T} \in \mathbb{R}^{m,n}$  such that:

$$(T)_{i,j} = \begin{cases} (c)_{i-j+1} & , \text{if } i \geq j \\ (r)_{j-i+1} & , \text{if } j > i \end{cases} \quad 1 \leq i \leq m, 1 \leq j \leq n$$

SOLUTION for (5-4.b) → [5-4-2-0:tplcpp.pdf](#)

(5-4.c) ☐ Show that the following two C++ functions realize the same linear mapping  $\mathbf{x} \mapsto \mathbf{y}$ , when supplied with the same arguments.

#### C++11-code 5.4.3: Function `toepmatmult`

```
2 VectorXd toepmatmult(const VectorXd & c, const VectorXd & r,
3                               const VectorXd & x)
4 {
5     assert(c.size() == r.size() &&
6            c.size() == x.size() &&
7            "c, r, x have different lengths!");
8
9     MatrixXd T = toeplitz(c,r);
10
11    VectorXd y = T*x;
12
13    return y;
14 }
```

Get it on  [GitLab \(toeplitz.cpp\)](#).

#### C++11-code 5.4.4: Function `toepmult`

```
2 VectorXd toepmult(const VectorXd & c, const VectorXd & r,
3                               const VectorXd & x)
4 {
5     assert(c.size() == r.size() &&
6            c.size() == x.size() &&
7            "c, r, x have different lengths!");
8     int n = c.size();
9
10    VectorXcd cr_tmp = c.cast<std::complex<double>>();
11    cr_tmp.conservativeResize(2*n); cr_tmp.tail(n) = VectorXcd::Zero(n);
12    cr_tmp.tail(n-1).real() = r.tail(n-1).reverse();
13
14    VectorXcd x_tmp = x.cast<std::complex<double>>();
15    x_tmp.conservativeResize(2*n); x_tmp.tail(n) = VectorXcd::Zero(n);
16
17    VectorXd y = pconvfft(cr_tmp, x_tmp).real();
18    y.conservativeResize(n);
19
20    return y;
21 }
```

Get it on  [GitLab \(toeplitz.cpp\)](#).

Function `pconvfft` is defined in [Lecture → Section 4.2.1], [Lecture → Code 4.2.1.3], and `toeplitz` is the simple C++ function from Sub-problem (5-4.b).

HIDDEN HINT 1 for (5-4.c) → [5-4-3-0:Toeplitz1h.pdf](#)

SOLUTION for (5-4.c) → [5-4-3-1:Toeplitz2s.pdf](#)

(5-4.d) ☐ What is the asymptotic complexity of either `toepmatmult` or `toepmult`?

SOLUTION for (5-4.d) → [5-4-4-0:Toeplitz3s.pdf](#)

(5-4.e) ☒ 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++11-code 5.4.6: Function `ttmatssolve`

```

2  VectorXd ttmatssolve(const VectorXd & h, const VectorXd & y)
3  {
4      assert(h.size() == y.size() &&
5          "h and y have different lengths!");
6      int n = h.size();
7
8      VectorXd h_tmp = VectorXd::Zero(n);
9      h_tmp(0) = h(0);
10
11     MatrixXd T = toeplitz(h,h_tmp);
12
13     VectorXd x = T.fullPivLu().solve(y);
14
15     return x;
16 }
```

Get it on  [GitLab \(toeplitz.cpp\)](#).

#### C++11-code 5.4.7: Function `ttrecslove`

```

2  VectorXd ttrecslove(const VectorXd & h, const VectorXd & y, int l)
3  {
4      assert(h.size() == y.size() &&
5          "h and y have different lengths!");
6
7      VectorXd x;
8
9      if(l == 0) {
10          x.resize(1);
11          x(0) = y(0)/h(0);
12      } else {
13          int n = std::pow(2,l);
14          int m = n/2;
15
16          assert(h.size() == n && y.size() == n &&
17              "h and y have length different from 2^l!");
18
19          VectorXd x1 = ttrecslove(h.head(m), y.head(m), l-1);
20          VectorXd y2 = y.segment(m,m) - toepmult(h.segment(m,m),
21              h.segment(1,m).reverse(), x1);
22          VectorXd x2 = ttrecslove(h.head(m), y2, l-1);
23
24          x.resize(n);
25          x.head(m) = x1;
26          x.tail(m) = x2;
```

```

27     }
28
29     return x;
30 }
```

Get it on  [GitLab \(toeplitz.cpp\)](#).

HIDDEN HINT 1 for (5-4.e) → [5-4-5-0:Toeplitz4h.pdf](#)

SOLUTION for (5-4.e) → [5-4-5-1:Toeplitz4s.pdf](#) ▲

(5-4.f) ☐ Why is the algorithm implemented in `ttreccsolve` called a “divide & conquer” method?

SOLUTION for (5-4.f) → [5-4-6-0:Toeplitz5s.pdf](#) ▲

(5-4.g) ☐ Perform a timing comparison of `ttmatsolve` and `ttreccsolve` for

```
h = VectorXd::LinSpaced(n, 1, n).cwiseInverse();
```

and  $n = 2^l$ ,  $l = 3, \dots, 11$ . Plot the timing results for both functions in double logarithmic scale.

SOLUTION for (5-4.g) → [5-4-7-0:Toeplitz6s.pdf](#) ▲

(5-4.h) ☐ Derive the asymptotic complexity of both functions `ttmatsolve` and `ttreccsolve` in terms of the problem size parameter  $n$ .

SOLUTION for (5-4.h) → [5-4-8-0:Toeplitz7s.pdf](#) ▲

(5-4.i) ☐ For the case that  $n = h.size()$  is not a power of 2, implement a wrapper function for `ttreccsolve` that, in the absence of roundoff errors, would behave exactly like `ttmatsolve`, i.e.  $ttsolve(h, y) = ttmatsolve(h, y)$ .

```
VectorXd ttsolve(const VectorXd & h, const VectorXd & y);
```

HIDDEN HINT 1 for (5-4.i) → [5-4-9-0:tplhext.pdf](#)

SOLUTION for (5-4.i) → [5-4-9-1:Toeplitz8s.pdf](#) ▲

**End Problem 5-4**

# Chapter 6

## Data Interpolation and Data Fitting in 1D

### Problem 6-1: Evaluating the derivatives of interpolating polynomials

In [Lecture → Ex. 5.1.0.7] 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 8.3.2.2] often require information about the derivative  $f'$  as well. Therefore, 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 → Rem. 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].

(6-1.a) (15 min.) Using the Horner scheme, 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  $c$ :

```
template <typename CoeffVec>
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,

$$p(t) = c[0]t^k + c[1]t^{k-1} + \cdots + c[k-1]t + c[k], \quad t \in \mathbb{R}. \quad (6.1.1)$$

As indicated above, the type **CoeffVec** must provide component access via **operator [] (int) const** and a **size()** method.

SOLUTION for (6-1.a) → [6-1-1-0:Horner1s.pdf](#)

(6-1.b) For the sake of testing, write a naive C++ implementation of the evaluation requested in (6-1.a)

```
template <typename CoeffVec>
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 (6-1.b) → [6-1-2-0:Horner2s.pdf](#)

(6-1.c)  (5 min.) [ depends on Sub-problem (6-1.a), Sub-problem (6-1.b) ]

What are the asymptotic complexities of the two functions you implemented in (6-1.a) and (6-1.b) in terms of raising the polynomial degree?

SOLUTION for (6-1.c) → [6-1-3-0:Horner3s.pdf](#)

(6-1.d)  (15 min.) [ depends on Sub-problem (6-1.a), Sub-problem (6-1.b) ]

Implement a function

```
bool polyTestTime(unsigned int d);
```

that

- (i) validates your implementation of the two functions `evaldp()`/`evaldp_naive()` from (6-1.a)/(6-1.b) by comparing their results for  $x = 1.23$  and a polynomial of degree  $n$  with monomial coefficients  $c_0 = 1, c_1 = 2, \dots, c_n = n$  and  $n = 2, 4, 8, \dots, 2^d$ . 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 (6-1.d) → [6-1-4-0:h4h1.pdf](#)

SOLUTION for (6-1.d) → [6-1-4-1: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.

(6-1.e)  Write an efficient C++ function

```
VectorXd dipoleval(const VectorXd & t,
                     const VectorXd & y,
                     const VectorXd & x);
```

that returns the row vector  $[p'(x_1), \dots, p'(x_m)]^\top$ , when the argument  $x$  passes  $[x_1, \dots, x_m]$ ,  $m \in \mathbb{N}$  small. Here,  $p'$  denotes the *derivative* of the polynomial  $p \in \mathcal{P}_n$  interpolating the data points  $(t_i, y_i)$ ,  $i = 0, \dots, n$ , for pairwise different  $t_i \in \mathbb{R}$  and data values  $y_i \in \mathbb{R}$ .

To that end differentiate the recursion formula [Lecture → Eq. (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 (6-1.e) → [6-1-5-0:hornxh1.pdf](#)

SOLUTION for (6-1.e) → [6-1-5-1:dipoleval.pdf](#)

(6-1.f)  For validation purposes devise an alternative, less efficient, implementation of `dipoleval()`

```
VectorXd dipoleval_alt(const VectorXd & t,
                      const VectorXd & y,
                      const 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 testDipoleEval(void);
```

Return `false` in case the test fails, `true` otherwise.

SOLUTION for (6-1.f) → [6-1-6-0:test.pdf](#)



(6-1.g)  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 (6-1.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 (6-1.g) → [6-1-7-0:hx1.pdf](#)

SOLUTION for (6-1.g) → [6-1-7-1:sx.pdf](#)



**End Problem 6-1 , 35 min.**

### Problem 6-2: Lagrange interpolant

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 → Eq. (5.2.2.4)] we denote by  $L_i$  the  $i$ -th Lagrange polynomial for given nodes  $t_j \in \mathbb{R}$ ,  $j = 0, \dots, n$ ,  $t_i \neq t_j$ , if  $i \neq j$ . Then the polynomial Lagrange interpolant  $p$  through the data points  $(t_i, y_i)_{i=0}^n$  has the representation

$$p(x) = \sum_{i=0}^n y_i L_i(x) \quad \text{with} \quad L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^n \frac{x - t_j}{t_i - t_j}. \quad (6.2.1)$$

**(6-2.a)** Show that

$$\sum_{i=0}^n L_i(t) = 1 \quad \forall t \in \mathbb{R}. \quad (6.2.2)$$

HIDDEN HINT 1 for (6-2.a) → [6-2-1-0:glgh1.pdf](#)

SOLUTION for (6-2.a) → [6-2-1-1:glgs1.pdf](#)

**(6-2.b)** Show that

$$\sum_{i=0}^n L_i(0) t_i^j = \begin{cases} 1 & \text{for } j = 0, \\ 0 & \text{for } j = 1, \dots, n. \end{cases} \quad (6.2.3)$$

(Read  $t_i$  raised to the power  $j$ .)

HIDDEN HINT 1 for (6-2.b) → [6-2-2-0:glgh1.pdf](#)

SOLUTION for (6-2.b) → [6-2-2-1:glgs1.pdf](#)

**(6-2.c)** Show that  $p(x)$  can also be written as

$$p(x) = \omega(x) \sum_{i=0}^n \frac{y_i}{(x - t_i)\omega'(t_i)} \quad \text{with} \quad \omega(t) := \prod_{j=0}^n (t - t_j). \quad (6.2.4)$$

HIDDEN HINT 1 for (6-2.c) → [6-2-3-0:LagrangePoly1h.pdf](#)

SOLUTION for (6-2.c) → [6-2-3-1:LagrangePoly1s.pdf](#)

**End Problem 6-2**

### Problem 6-3: 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 → Eq. (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 → Eq. (5.2.2.4)].

This is a purely theoretical problem.

Let  $n+1$  different nodes  $t_i, i = 0, \dots, n, n \in \mathbb{N}$ , be given. For numbers  $y_j, c_j \in \mathbb{R}, j = 0, \dots, n$ , Hermite interpolation seeks a polynomial  $p \in \mathcal{P}_{2n+1}$  satisfying  $p(t_i) = y_i$  and  $p'(t_i) = c_i, i = 0, \dots, n$ . Here  $p'$  designates the derivative of  $p$ .

The **generalized Lagrange polynomials** for Hermite interpolation  $L_i, K_i \in \mathcal{P}_{2n+1}$  satisfy

$$L_i(t_j) = \delta_{ij}, \quad L'_i(t_j) = 0, \quad (6.3.1a)$$

$$K_i(t_j) = 0, \quad K'_i(t_j) = \delta_{ij}, \quad (6.3.1b)$$

for  $i, j \in \{0, \dots, n\}$ .

**(6-3.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*  $\{x \mapsto x^k, k = 0, \dots, 2n+1\}$  of  $\mathcal{P}_{2n+1}$ .

HIDDEN HINT 1 for (6-3.a) → 6-3-1-0:gagh0.pdf

SOLUTION for (6-3.a) → 6-3-1-1:glgs1.pdf ▲

**(6-3.b)** □ (15 min.) [ depends on Sub-problem (6-3.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 (6-3.b) → 6-3-2-0:glgh1.pdf

,

SOLUTION for (6-3.b) → 6-3-2-1:glgs1.pdf ▲

**(6-3.c)** □ (5 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 (6-3.c) → 6-3-3-0:glgs1.pdf ▲

**(6-3.d)** □ (10 min.) Let  $g_1, \dots, g_m, m \in \mathbb{N}$ , 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 (6-3.d) → 6-3-4-0:glgs1.pdf ▲

**(6-3.e)** □ (30 min.) [ depends on Sub-problem (6-3.c), Sub-problem (6-3.d) ]

Find general formulas for the polynomials  $L_i, K_i$  as defined through (6.3.1). Of course, these formulas will involve the nodes  $t_i$ .

HIDDEN HINT 1 for (6-3.e) → [6-3-5-0:glgh1.pdf](#)

HIDDEN HINT 2 for (6-3.e) → [6-3-5-1:glghx.pdf](#)

SOLUTION for (6-3.e) → [6-3-5-2:glgs1.pdf](#)



**End Problem 6-3 , 75 min.**

### Problem 6-4: Piecewise linear interpolation

[Lecture → Ex. 5.1.0.12] 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 → Eq. (5.1.0.13)].

Study [Lecture → Ex. 5.1.0.12] before tackling this problem. Problem involves coding in C++.

#### (6-4.a)

Write a C++ class `LinearInterpolant` representing the piecewise linear interpolant. Make sure your class has an efficient internal representation of a basis. Provide a constructor and an evaluation operator() as described in the following template:

##### C++11-code 6.4.1:

```

2  class LinearInterpolant {
3  public:
4
5  /*!
6   * \brief LinearInterpolant builds interpolant from data.
7   * Sort the array for the first time:
8   * the data is not assumed to be sorted
9   * sorting is necessary for binary search
10  * \param TODO
11  */
12  LinearInterpolant(/* TODO: pass data here */);
13
14 /*!
15  * \brief operator () Evaluation operator.
16  * Return the value of I at x, i.e. I(x).
17  * Performs bound checks (i.e. if x < t0 or x >= tn).
18  * \param x Value x ∈ ℝ.
19  * \return Value I(x).
20  */
21  double operator() (double x);
22 private:
23  // TODO: your data there
24 };

```

Get it on  GitLab (`linearinterpolant.cpp`).

SOLUTION for (6-4.a) → [6-4-1-0:impl.pdf](#)

### End Problem 6-4

### Problem 6-5: 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.

Involves coding in C++.

The (ordered) real trigonometric basis of  $\mathcal{P}_{2n}^T$  is

$$\mathcal{B}_{\text{Re}} := \left\{ C_0 := t \mapsto 1, S_1 := t \mapsto \sin(2\pi t), C_1 := t \mapsto \cos(2\pi t), S_2 := t \mapsto \sin(4\pi t), \dots, C_{2t} \mapsto \cos(4\pi t), \dots, S_n := t \mapsto \sin(2n\pi t), C_n := t \mapsto \cos(2n\pi t) \right\}. \quad (6.5.1)$$

Another (ordered) basis of  $\mathcal{P}_{2n}^T$  is

$$\mathcal{B}_{\text{exp}} := \{ B_k := t \mapsto \exp(2\pi k i t) : k = -n, \dots, n \}. \quad (6.5.2)$$

Both bases have  $2n + 1$  elements, which agrees with the dimension of  $\mathcal{P}_{2n}^T$  [Lecture → Cor. 5.6.1.6].

**(6-5.a)** ☐ Give the matrix  $S \in \mathbb{C}^{2n+1, 2n+1}$  that effects the transformation of a coefficient representation with respect to  $\mathcal{B}_{\text{Re}}$  into a coefficient representation with respect to  $\mathcal{B}_{\text{exp}}$ .

HIDDEN HINT 1 for (6-5.a) → 6-5-1-0:trh1.pdf

SOLUTION for (6-5.a) → 6-5-1-1:trisol1.pdf ▲

**(6-5.b)** ☐ The shift operator  $S_\tau : C^0(\mathbb{R}) \rightarrow C^0(\mathbb{R})$  acts on a function according to  $S_\tau f(t) = f(t - \tau)$ , 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  $\mathcal{B}_{\text{exp}}$ , what is the coefficient vector  $\tilde{\mathbf{c}} \in \mathbb{C}^{2n+1}$  of  $S_\tau p$ ?

HIDDEN HINT 1 for (6-5.b) → 6-5-2-0:tch2.pdf

SOLUTION for (6-5.b) → 6-5-2-1:trisol2.pdf ▲

Now we consider the set of interpolation nodes

$$\mathcal{T}_n := \left\{ \frac{j + 1/2}{2n + 1} : j = 0, \dots, 2n \right\}. \quad (6.5.3)$$

The **cardinal basis functions**  $b_0, \dots, b_{2n} \in \mathcal{P}_{2n}^T$  of  $\mathcal{P}_{2n}^T$  with respect to  $\mathcal{T}_n$  are defined by

$$b_j(t_k) = \delta_{kj} \quad [\text{Kronecker symbol}], \quad j, k = 0, \dots, 2n \quad (6.5.4)$$

**(6-5.c)** ☐ Use the function `trigpolyvalequid`, see [Lecture → Code 5.6.3.8] (can be included from “`trigpolyvalequid.hpp`”) to plot the cardinal basis function  $t \mapsto b_0(t)$  in  $[0, 1]$  for  $n = 5$ . To that end evaluate  $b_0(t)$  in 1000 equidistant points  $\frac{k}{1000}$ ,  $k = 0, \dots, 999$ .

SOLUTION for (6-5.c) → 6-5-3-0:plotsol.pdf ▲

**(6-5.d)** ☐ Show that

$$b_{k+1} = S_\delta b_k \quad \text{with} \quad \delta := \frac{1}{2n + 1}. \quad (6.5.6)$$

HIDDEN HINT 1 for (6-5.d) → 6-5-4-0:h1.pdf

SOLUTION for (6-5.d) → 6-5-4-1:trisol1.pdf ▲

- (6-5.e) ☐ Describe the entries of the “interpolation matrix”, that is, the system matrix of [Lecture → Eq. (5.1.0.17)], for the trigonometric interpolation problem with node set  $\mathcal{T}_n$  and with respect to the basis  $\mathcal{B}_{\text{exp}}$  of  $\mathcal{P}_{2n}^T$ .

HIDDEN HINT 1 for (6-5.e) → 6-5-5-0:tklh1.pdf

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

SOLUTION for (6-5.e) → 6-5-5-2:trisoll.pdf



- (6-5.f) ☐ What is the inverse of the interpolation matrix found in the previous problem?

HIDDEN HINT 1 for (6-5.f) → 6-5-6-0:h1.pdf

SOLUTION for (6-5.f) → 6-5-6-1:trisoll.pdf



- (6-5.g) ☐ Give a closed-form expression for the cardinal basis functions  $b_k$  defined in (6.5.4).

HIDDEN HINT 1 for (6-5.g) → 6-5-7-0:h1.pdf

SOLUTION for (6-5.g) → 6-5-7-1:trisoll.pdf



- (6-5.h) ☐ Write a function

```
double trigIpL(std::size_t n);
```

that approximately computes the **Lebesgue constant**  $\lambda(n)$ , see [Lecture → § 5.2.4.7], for trigonometric interpolation based on the node set  $\mathcal{T}_n$ . The Lebesgue constant is available through the formula

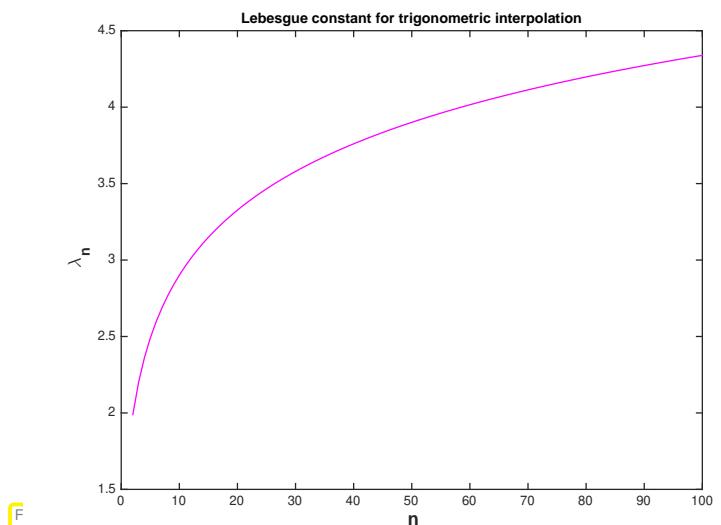
$$\lambda(n) = \sup_{\mathbf{y} \in \mathbb{C}^{2n+1} \setminus \{\mathbf{0}\}} \frac{\|I_n \mathbf{y}\|_{L^\infty([0,1])}}{\|\mathbf{y}\|_\infty} = \sup_{t \in [0,1]} \sum_{k=0}^{2n} |b_k(t)| , \quad (6.5.9)$$

where  $I_n : \mathbb{C}^{2n+1} \rightarrow \mathcal{C}^0(\mathbb{R})$  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  $[0,1]$ . Maximum efficiency of the implementation need not be a concern. Use the formula involving sum of cosines.

SOLUTION for (6-5.h) → 6-5-8-0:trisoll.pdf

Graph of  $n \mapsto \lambda(n)$



- (6-5.i) ☐ The following table gives the values  $\lambda(n)$  for  $n = 2^k$ ,  $k = 2, 3, \dots, 8$ .

$k$	$\lambda(2^k)$
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  $n \rightarrow \infty$ . Is it  $O(\log n)$ ,  $O(n)$ ,  $O(n \log n)$ , or  $O(n^2)$ ?

SOLUTION for (6-5.i) → [6-5-9-0:xtrsoll.pdf](#)



**End Problem 6-5**

### Problem 6-6: Quadratic Splines

[Lecture → Def. 5.4.0.1] 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 5.4.1] 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.

Familiarity with [Lecture → Section 5.4.1] is required. This exercise involves substantial implementation in C++ and EIGEN.

Consider a 1-periodic function  $f : \mathbb{R} \rightarrow \mathbb{R}$ , that is,  $f(t+1) = f(t)$  for all  $t \in \mathbb{R}$ , and a set of knots

$$\mathcal{M} := \{0 = t_0 < t_1 < t_2 < \dots < t_{n-1} < t_n = 1\} \subset [0, 1].$$

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  $[t_{j-1}, t_j]$ ,  $j = 0, \dots, n$ .

In analogy to the local representation of a cubic spline function according to [Lecture → Eq. (5.4.1.4)], we parametrize a quadratic spline function  $s \in \mathcal{S}_{2,\mathcal{M}}$  according to

$$s|_{[t_{j-1}, t_j]}(t) = d_j \tau^2 + c_j 4 \tau(1 - \tau) + d_{j-1} (1 - \tau)^2, \quad \tau := \frac{t - t_{j-1}}{t_j - t_{j-1}}, \quad j = 1, \dots, n, \quad (6.6.1)$$

with  $c_j, d_k \in \mathbb{R}$ ,  $j = 1, \dots, n$ ,  $k = 0, \dots, n$ .

**(6-6.a)**  (5 min.) How are the coefficients  $d_j$ ,  $c_j$ , and  $d_{j-1}$  in (6.6.1) 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 (6-6.a) → 6-6-1-0:s0h1.pdf

SOLUTION for (6-6.a) → 6-6-1-1:s0.pdf ▲

**(6-6.b)**  (5 min.) What is the dimension of the subspace of 1-periodic spline functions in  $\mathcal{S}_{2,\mathcal{M}}$ ?

HIDDEN HINT 1 for (6-6.b) → 6-6-2-0:s1h1.pdf

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

**(6-6.c)**  (5 min.) [ depends on Sub-problem (6-6.a) ]

What kind of continuity is already guaranteed by the mere use of the representation (6.6.1)?

SOLUTION for (6-6.c) → 6-6-3-0:s2.pdf ▲

**(6-6.d)**  (25 min.) Derive a  $n \times n$  linear system of equations (system matrix and right hand side) whose solution provides the coefficients  $c_j$ ,  $j = 1, \dots, n$ , in the representation (6.6.1) based on the function values  $y_j := f(\frac{1}{2}(t_{j-1} + t_j))$ ,  $j = 1, \dots, n$ .

HIDDEN HINT 1 for (6-6.d) → 6-6-4-0:s3h1.pdf

SOLUTION for (6-6.d) → 6-6-4-1:s3.pdf ▲

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

Show that the system matrix found in Sub-problem (6-6.d) can be written as a rank-1 modification of a tridiagonal matrix.

SOLUTION for (6-6.e) → 6-6-5-0:s3a.pdf ▲

(6-6.f)  (15 min.) [ depends on Sub-problem (6-6.d) and Sub-problem (6-6.e) ]

Outline an efficient algorithm for the solution of the linear system of equations found in Sub-problem (6-6.d). What is the asymptotic complexity of the algorithm in terms of the number  $n$  of data values for  $n \rightarrow \infty$

SOLUTION for (6-6.f) → [6-6-6-0:s6.pdf](#) ▲

(6-6.g)  (25 min.) [ depends on Sub-problem (6-6.d) ]

Implement a C++ function

```
Eigen::VectorXd compute_c(const Eigen::VectorXd &t,
                           const Eigen::VectorXd &y);
```

that returns the coefficient vector  $[c_1, \dots, c_n]^T \in \mathbb{R}^n$  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  $y_j$ ,  $j = 1, \dots, n$ , that specify the function values of the quadratic spline  $s$  at the midpoints of the knot intervals.

Use a suitable sparse direct solver provided by EIGEN. You need not employ the techniques of [Lecture → § 2.6.0.12].

SOLUTION for (6-6.g) → [6-6-7-0:s3b.pdf](#) ▲

(6-6.h)  (20 min.) [ depends on Sub-problem (6-6.d) ]

Realize a C++ function

```
Eigen::VectorXd compute_d ( const Eigen::VectorXd &c, const
                           Eigen::VectorXd &t);
```

that determines the coefficients  $d_j$ ,  $j = 0, \dots, n$ , in the local representation (6.6.1) assuming that the values  $c_j$ ,  $j = 1, \dots, n$ , have already been computed. The argument  $t$  is to pass the variable knot positions  $t_j$ ,  $j = 1, \dots, n - 1$ .

SOLUTION for (6-6.h) → [6-6-8-0:3c.pdf](#) ▲

(6-6.i)  (25 min.) [ depends on Sub-problem (6-6.e), Sub-problem (6-6.g), Sub-problem (6-6.h) ]

Based on `compute_c()` from Sub-problem (6-6.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 of a function  $f$  at the midpoints  $\frac{1}{2}(t_{j-1} + t_j)$ ,  $j = 1, \dots, n$ , 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 (6-6.i) → [6-6-9-0:s4.pdf](#) ▲

(6-6.j)  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

$s\left(\frac{1}{2}(t_j + t_{j-1})\right) = f\left(\frac{1}{2}(t_j + t_{j-1})\right)$ ,  $j = 1, \dots, 10$ . 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 (6-6.j) → 6-6-10-0:s5.pdf ▲

(6-6.k) (30 min.) [ depends on Sub-problem (6-6.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  $N = 10.000$ .

The function is supposed to return the error norms  $E_n$  for  $n = 2, 4, 8, \dots, 2^q$ , where  $q$  is passed in the argument  $q$ . Describe quantitatively and qualitatively the convergence of the error norms in terms of  $n \rightarrow \infty$ .

SOLUTION for (6-6.k) → 6-6-11-0:s7.pdf ▲

**End Problem 6-6 , 170 min.**

### Problem 6-7: 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 → § 5.1.0.15] and [Lecture → Def. 5.1.0.19] together with [Lecture → Section 5.3.3.2].

The goal of this problem is to prove the following theorem.

#### Theorem 6.7.1. Linear monotonicity preserving interpolation operator

Given the node set  $\mathcal{T} = \{t_0, \dots, t_n\} \subset [t_0, t_n]$ , with  $t_0 < t_1 < \dots < t_n$ , consider an interpolation operator  $I_{\mathcal{T}} : \mathbb{R}^{n+1} \rightarrow C^0([t_0, t_n])$ , for which, by definition, we have  $(I_{\mathcal{T}}(\mathbf{y}))(t_j) = y_j$  for every  $j = 0, \dots, n$  and every vector of data values  $\mathbf{y} = [y_0, \dots, y_n]^{\top} \in \mathbb{R}^{n+1}$ . 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  $y_j \leq y_{j+1}, \forall j = 0, \dots, n-1$ , 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:

$$(I_{\mathcal{T}}(\mathbf{y}))'(t_j) = 0, \quad \forall j \in \{1, \dots, n-1\}, \quad \forall \mathbf{y} \in \mathbb{R}^{n+1}. \quad (6.7.2)$$

**(6-7.a)** (15 min.) Show that the assertion (6.7.2) holds for the special data vectors

$$\mathbf{y} = \mathbf{y}^{(j)} := \underbrace{[0, \dots, 0]}_{j \text{ times}}, 1, \dots, 1]^{\top}, \quad j = 0, \dots, n.$$

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)** (10 min.) Prove that  $\text{span}\{\mathbf{y}^{(j)}, j = 0, \dots, n\} = \mathbb{R}^{n+1}$ .

SOLUTION for (6-7.b) → 6-7-2-0:s1a.pdf ▲

**(6-7.c)** (15 min.) [ depends on Sub-problem (6-7.a) and Sub-problem (6-7.b) ]

Invoke the insight gained in Sub-problem (6-7.a) to complete the proof of the theorem.

SOLUTION for (6-7.c) → 6-7-3-0:s2.pdf ▲

**End Problem 6-7**, 40 min.

### Problem 6-8: Approximating $\pi$ by extrapolation

In [Lecture → Section 5.2.3.3] we learned about the approximation of a limit  $\lim_{h \rightarrow 0} \psi(h)$  based on the evaluation of the function  $\psi$  “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 5.2.3.20], 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

$$x^* = \lim_{n \rightarrow \infty} T(n) ,$$

where the function  $T : \{n_{\min}, n_{\min} + 1, \dots\} \mapsto \mathbb{R}$  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 → § 5.2.3.17].

The idea of **extrapolation** is, firstly, to compute a few values  $T(n_0), T(n_1), \dots, T(n_k)$ ,  $k \in \mathbb{N}$ , and to consider them as the values  $g(1/n_0), g(1/n_1), \dots, g(1/n_k)$  of a continuous function

$$g : (0, 1/n_{\min}] \mapsto \mathbb{R}, \quad \text{for which, obviously } x^* = \lim_{h \rightarrow 0} g(h) .$$

Secondly, the function  $g$  is approximated by an interpolating polynomial  $p_k \in \mathcal{P}_k$  with  $p_k(n_j^{-1}) = T(n_j)$ ,  $j = 0, \dots, k$ . In many cases we can expect that  $p_k(0)$  will provide a good approximation for  $x^*$ .

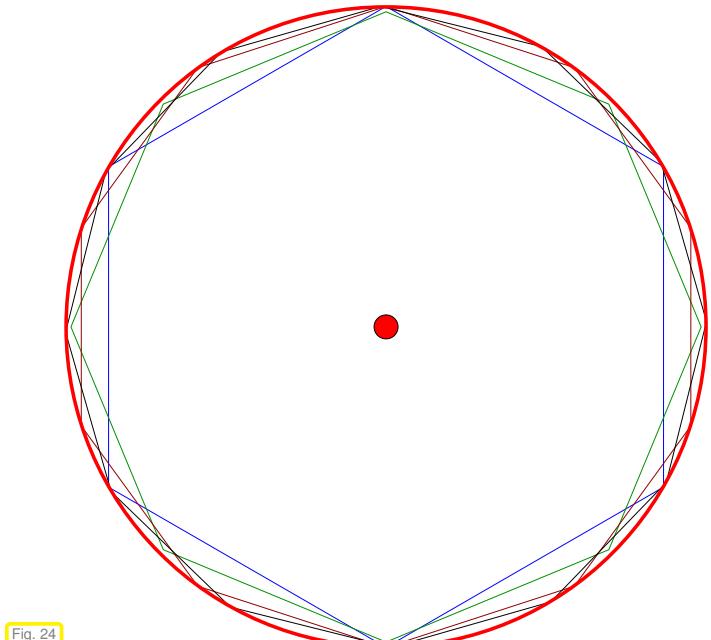


Fig. 24

(6-8.a) (10 min.)

Show that for any  $L \in \mathbb{N}$  and for  $k \rightarrow \infty$

$$\frac{1}{2}s_k = \pi + \sum_{\ell=1}^L c_\ell k^{-2\ell} + O(k^{-2(L+1)}) \quad \text{for some } c_\ell \in \mathbb{R} \quad \text{independent of } L . \quad (6.8.2)$$

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 (6.8.2) 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 (6-8.a) → [6-8-1-0:s0h1.pdf](#)

SOLUTION for (6-8.a) → [6-8-1-1:s0.pdf](#) ▲

(6-8.b) ☐ (30 min.) Implement a C++ function

```
double extrapolate_to_pi(unsigned int k);
```

that uses the *Aitken-Neville scheme*, see [Lecture → Code 5.2.3.10], to approximate  $\pi$  by extrapolation from the data points  $(j^{-1}, \frac{1}{2}s_j)$ , for  $j = 2, \dots, k$ . Use the values  $\frac{1}{2}s_j$  as given in (6.8.1).

SOLUTION for (6-8.b) → [6-8-2-0:s1.pdf](#) ▲

(6-8.c) ☐ (20 min.) Write a C++ function

```
void plotExtrapolationError(void);
```

that generates a *linear-logarithmic* plot of the extrapolation errors

$$\text{err}(k) := |\pi - p_k(0)|, \quad k = 2, \dots, 10,$$

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.

SOLUTION for (6-8.c) → [6-8-3-0:s2.pdf](#) ▲

(6-8.d) ☐ (10 min.) [ depends on Sub-problem (6-8.c) ]

Which kind of convergence of  $\text{err}(k) \rightarrow 0$  for  $k \rightarrow \infty$  can you conclude from the data generated in Sub-problem (6-8.c)? The main types of asymptotic convergence to 0 are introduced in [Lecture → Def. 6.1.2.7].

HIDDEN HINT 1 for (6-8.d) → [6-8-4-0:s3h1.pdf](#)

SOLUTION for (6-8.d) → [6-8-4-1:s3.pdf](#) ▲

(6-8.e) ☐ Modify your implementation of `plotExtrapolationError()` from Sub-problem (6-8.c) such that it plots and tabulates  $\text{err}_k$  for  $k$  up to 30. Describe and explain your observations.

SOLUTION for (6-8.e) → [6-8-5-0:s4.pdf](#) ▲

**End Problem 6-8 , 70 min.**

### Problem 6-9: 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  $\mathcal{M} := \{a = t_0 < t_1 < \dots < t_{n-1} < t_n = b\}$  we consider the following three spaces of piecewise polynomials:

$$V_{-1} := \left\{ v : [a, b] \rightarrow \mathbb{R} : v|_{[t_{j-1}, t_j]} \in \mathcal{P}_2, j = 1, \dots, n \right\}, \quad (6.9.1)$$

$$V_0 := \left\{ v \in C^0([a, b]) : v|_{[t_{j-1}, t_j]} \in \mathcal{P}_2, j = 1, \dots, n \right\}, \quad (6.9.2)$$

$$V_1 := \left\{ v \in C^1([a, b]) : v|_{[t_{j-1}, t_j]} \in \mathcal{P}_2, j = 1, \dots, n \right\}, \quad (6.9.3)$$

$$V_2 := \left\{ v \in C^2([a, b]) : v|_{[t_{j-1}, t_j]} \in \mathcal{P}_2, j = 1, \dots, n \right\}. \quad (6.9.4)$$

Here,  $\mathcal{P}_d$  stands for the space of polynomials  $\mathbb{R} \rightarrow \mathbb{R}$  of degree  $\leq d$ .

**(6-9.a)** (8 min.) Determine the dimensions of the four spaces defined above:

$$\dim V_{-1} = \boxed{\phantom{00}},$$

$$\dim V_0 = \boxed{\phantom{00}},$$

$$\dim V_1 = \boxed{\phantom{00}},$$

$$\dim V_2 = \boxed{\phantom{00}}.$$

SOLUTION for (6-9.a) → [6-9-1-0:s1.pdf](#)

**(6-9.b)** (3 min.) Which of the spaces  $V_k$ ,  $k = -1, 0, 1$ , coincides with one of the spline spaces  $S_{d, \mathcal{M}}$  introduced in [Lecture → Def. 5.4.0.1] in class?

$$V_{\boxed{\phantom{0}}} = S_{\boxed{\phantom{0}}, \mathcal{M}}.$$

SOLUTION for (6-9.b) → [6-9-2-0:s2.pdf](#)

**(6-9.c)** (8 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  $j = 1, \dots, n$

<input type="radio"/> necessary	<input type="radio"/> sufficient	<input type="radio"/> neither
---------------------------------	----------------------------------	-------------------------------

(ii)  $v'(x) \geq 0$  for all  $x \in [a, b]$

<input type="radio"/> necessary	<input type="radio"/> sufficient	<input type="radio"/> neither
---------------------------------	----------------------------------	-------------------------------

(iii)  $v'(\frac{1}{2}(t_{j-1} + t_j)) \geq 0$  for all  $j \in \{1, \dots, n\}$

<input type="radio"/> necessary	<input type="radio"/> sufficient	<input type="radio"/> neither
---------------------------------	----------------------------------	-------------------------------

(iv)  $v'(t_j) \geq 0$  for all  $j \in \{0, \dots, n\}$

<input type="radio"/> necessary	<input type="radio"/> sufficient	<input type="radio"/> neither
---------------------------------	----------------------------------	-------------------------------

SOLUTION for (6-9.c) → [6-9-3-0:s3.pdf](#) ▲

In the sequel we consider the mesh (knot set)  $\mathcal{M} = \{0, 1, 2\}$ .

**(6-9.d)** (2 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  $v \in V_1$ .

$$a = \boxed{\phantom{000}} , \quad b = \boxed{\phantom{000}} .$$

SOLUTION for (6-9.d) → [6-9-4-0:s4.pdf](#) ▲

**(6-9.e)** (4 min.) Write down a  $v \in V_1$  such that

$$v(0) = 0 , \quad v(1) = 1 , \quad v(2) = 0 .$$

$$v(x) = \begin{cases} \boxed{\phantom{000}} & \text{for } 0 \leq x < 1 , \\ \boxed{\phantom{000}} & \text{for } 1 \leq x \leq 2 . \end{cases}$$

SOLUTION for (6-9.e) → [6-9-5-0:s5.pdf](#) ▲

**End Problem 6-9**, 25 min.

### Problem 6-10: Adaptive Interpolation of Closed Curves

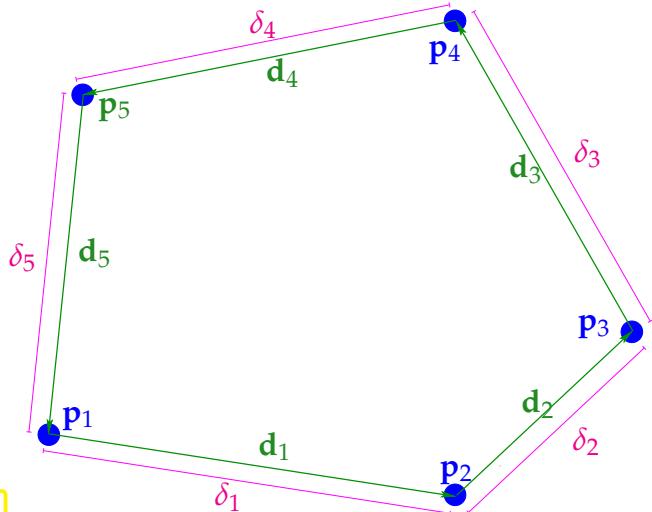
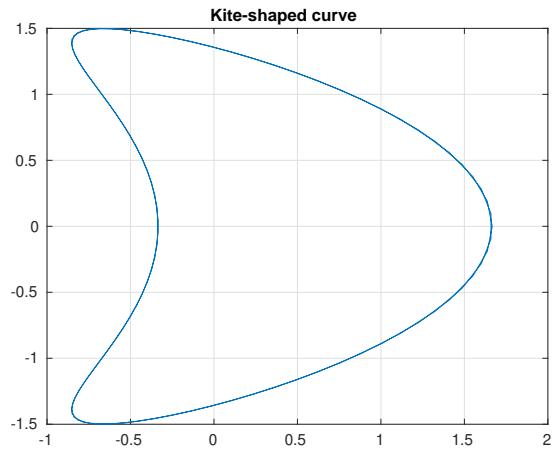
Often, for the sake of efficiency, closed curves (loops) need to be approximated by piecewise polynomials. This problem presents a simple adaptive algorithm for constructing such an approximation.

Related to [Lecture → Section 5.3.3]; the idea of adaptive approximation is borrowed from [Lecture → Section 7.5].

A **1-periodic** continuously differentiable function  $\mathbf{c} : \mathbb{R} \rightarrow \mathbb{R}^2$  parameterizes a closed curve in the plane.

For instance, the “kite-shaped” curve displayed beside is described by

$$\mathbf{c}(t) = \begin{bmatrix} \cos(2\pi t) + \frac{2}{3} \cos(4\pi t) \\ \frac{3}{2} \sin(2\pi t) \end{bmatrix}. \quad (6.10.1)$$



**Notations.** Based on a sequence of  $n \geq 2$  mutually distinct points  $(\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n)$  in the plane ( $\mathbf{p}_k \in \mathbb{R}^2$ ) the following notations are defined:

- $\delta_i := \|\mathbf{p}_{i+1} - \mathbf{p}_i\|$ ,  $i = 1, \dots, n-1$ ,
- $\delta_n := \|\mathbf{p}_1 - \mathbf{p}_n\|$ ,
- $\lambda_0 := 0$ ,
- $\lambda_k := \sum_{j=1}^k \delta_j$ ,  $k = 1, \dots, n$ ,
- $\mathbf{d}_i := \mathbf{p}_{i+1} - \mathbf{p}_i$ ,  $i = 1, \dots, n-1$ ,
- $\mathbf{d}_n := \mathbf{p}_1 - \mathbf{p}_n$ ,
- $\mathbf{t}_i := \frac{\mathbf{d}_i}{\|\mathbf{d}_i\|}$ ,  $i = 1, \dots, n$ .

We study two ways to connect points in the plane by a piecewise polynomial closed curve.

#### Definition 6.10.2. Closed polygonal interpolant

Given a sequence  $\Sigma := (\mathbf{p}_1, \dots, \mathbf{p}_n)$ ,  $n \in \mathbb{N}$ , of mutually distinct points  $\in \mathbb{R}^2$ , their **interpolating closed polygon**  $\Pi_\Sigma$  is a function  $\Pi_\Sigma : [0, \lambda_n] \rightarrow \mathbb{R}^2$  that fulfills:

- (i)  $\Pi_\Sigma(\lambda_k) = \mathbf{p}_{k+1}$ ,  $k = 1, \dots, n-1$ , and  $\Pi_\Sigma(0) = \Pi_\Sigma(\lambda_n) = \mathbf{p}_1$ .
- (ii)  $\Pi_\Sigma|_{[\lambda_{k-1}, \lambda_k]}$ ,  $k = 1, \dots, n$ , is componentwise an affine linear function, that is,  $\Pi_\Sigma|_{[\lambda_{k-1}, \lambda_k]} \in (\mathcal{P}_1)^2$ .

### Definition 6.10.3. Interpolating closed cubic Hermite curve

Given a sequence  $\Sigma := (\mathbf{p}_1, \dots, \mathbf{p}_n)$ ,  $n \in \mathbb{N}$ , of mutually distinct points  $\in \mathbb{R}^2$ , their **interpolating closed cubic Hermite curve**  $\mathbf{H}_\Sigma$  is a function  $\Pi_\Sigma : [0, \lambda_n] \rightarrow \mathbb{R}^2$  that satisfies:

- (i)  $\mathbf{H}_\Sigma(\lambda_k) = \mathbf{p}_{k+1}$ ,  $k = 1, \dots, n-1$ , and  $\mathbf{H}_\Sigma(0) = \mathbf{H}_\Sigma(\lambda_n) = \mathbf{p}_1$ .
- (ii)  $\mathbf{H}'(\lambda_k) = \frac{\mathbf{s}_k}{\|\mathbf{s}_k\|}$ ,  $\mathbf{s}_k := \begin{cases} \frac{\delta_{k+1}}{\delta_k + \delta_{k+1}} \mathbf{t}_k + \frac{\delta_k}{\delta_k + \delta_{k+1}} \mathbf{t}_{k+1} & , \text{ if } k = 1, \dots, n-1 \\ \frac{\delta_n}{\delta_1 + \delta_n} \mathbf{t}_1 + \frac{\delta_1}{\delta_1 + \delta_n} \mathbf{t}_n & , \text{ if } k = 0, n \end{cases}$ .
- ( $\mathbf{H}'$  denotes the derivative of  $\mathbf{H}$ .)
- (iii)  $\mathbf{H}_\Sigma|_{[\lambda_{k-1}, \lambda_k]}$ ,  $k = 1, \dots, n$ , is componentwise a cubic polynomial, that is  $\mathbf{H}_\Sigma|_{[\lambda_{k-1}, \lambda_k]} \in (\mathcal{P}_3)^2$ .

(6-10.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  $\Pi_\Sigma(x_k \lambda_n)$ ,  $k = 1, \dots, M$ ,  $M$  the length of the vector  $x$  with *sorted* entries  $0 \leq x_1 < x_2 < \dots < x_M \leq 1$ .  $\Pi_\Sigma$  is the interpolating closed polygon for a set  $\Sigma$  of points passed through the vector argument Sigma.

SOLUTION for (6-10.a) → [6-10-1-0:s1.pdf](#)

(6-10.b) (20 min.) [ depends on Sub-problem (6-10.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  $\mathbf{H}_\Sigma(x_k \lambda_n) \in \mathbb{R}^2$ ,  $k = 1, \dots, M$ ,  $M$  the length of the vector  $x$  with *sorted* entries  $0 \leq x_1 < x_2 < \dots < x_M \leq 1$ . In this case  $\mathbf{H}_\Sigma$  is the interpolating closed cubic Hermite curve according to Def. 6.10.3 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 → Eq. (5.3.3.4)] for evaluating a cubic Hermite interpolating polynomial, see also [Lecture → Code 5.3.3.6].

HIDDEN HINT 1 for (6-10.b) → [6-10-2-0:h2.pdf](#)

SOLUTION for (6-10.b) → [6-10-2-1:s1.pdf](#)

Now, given a 1-periodic function  $c : \mathbb{R} \rightarrow \mathbb{R}$  providing the parameterization of a closed curve, we pursue the following policy aiming for its adaptive piecewise polynomial approximation.

**Pseudocode 6.10.6: Algorithm for adaptive approximation of closed curve**


---

```

1 Given:  $\text{tol} > 0$ ,  $n_{\min} \in \mathbb{N} \setminus \{1\}$ .
2  $\mathcal{M} := \{t_0 := 0, t_1 := \frac{1}{n_{\min}}, \dots, t_{n-1} := \frac{n_{\min}-1}{n_{\min}}\}$  // An ordered set
3 repeat {
4    $n := \#\mathcal{M}$ ; // Notation:  $\mathcal{M} := \{t_0, \dots, t_{n-1}\}$ 
5    $\Sigma := (\mathbf{c}(t))_{t \in \mathcal{M}}$ ; // sequence of points; also defines  $\lambda_k$ 
6   //  $\Pi_\Sigma / \mathbf{H}_\Sigma$  according to Def. 6.10.2, Def. 6.10.3
7    $\sigma_k := \left\| (\mathbf{H}_\Sigma - \Pi_\Sigma) \left( \frac{1}{2}(\lambda_{k-1} + \lambda_k) \right) \right\|$ ,  $k = 1, \dots, n$ ;
8    $\alpha := \frac{1}{n} \sum_{k=1}^n \sigma_k$ ;
9   for  $j = 1, \dots, n$  do {
10    if ( $\sigma_j > 0.9\alpha$ )  $\mathcal{M} \leftarrow \mathcal{M} \cup \{\frac{1}{2}(t_{j-1} + t_j)\}$ ; // Convention  $t_n := 1$ 
11    //  $\mathcal{M}$  is always assumed to be sorted!
12   }
13 }
14 until ( $\max_{k=1, \dots, n} \sigma_k \leq \text{tol}$ );

```

---

(6-10.c) (30 min.) [ depends on Sub-problem (6-10.a) and Sub-problem (6-10.b) ]

Based on the C++ functions `closedPolygonalInterpolant()` and `closedHermiteInterpolant()` implement a C++ function

```

template <typename CurveFunctor>
std::pair<std::vector<Eigen::Vector2dstd::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 6.10.6. The 1-periodic continuous function  $\mathbf{c} : \mathbb{R} \rightarrow \mathbb{R}$  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  $\mathbf{H}_\Sigma(x_k L)$ ,  $k = 1, \dots, M$ ,  $M$  the length of the vector `x` with `sorted` entries  $0 \leq x_1 < x_2 < \dots < x_M \leq 1$ . In this case  $\mathbf{H}_\Sigma$  is the interpolating closed cubic Hermite curve according to Def. 6.10.3 for the set  $\Sigma$  at the termination of the algorithm,
2. the coordinates of the points in the sequence  $\Sigma$  as a second sequence of 2-vectors.

SOLUTION for (6-10.c) → [6-10-3-0:s3.pdf](#)



**End Problem 6-10 , 70 min.**

# Chapter 7

## Approximation of Functions in 1D

### Problem 7-1: Piecewise linear approximation on graded meshes

One of the main point made in [Lecture → Section 6.1.3] 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  $\mathcal{M} = \{0 = t_0 < t_1 < \dots < t_{n-1} < t_n = 1\}$  on the unit interval  $I = [0, 1]$ ,  $n \in \mathbb{N}$ , we define the *piecewise linear interpolant*:

$$I_{\mathcal{M}} : C^0(I) \rightarrow \mathcal{P}_{1,\mathcal{M}} = \{s \in C^0(I), s|_{[t_{j-1}, t_j]} \in \mathcal{P}_1 \forall j\}, \quad \text{s.t. } (I_{\mathcal{M}}f)(t_j) = f(t_j), \quad j = 0, \dots, n \quad (7.1.1)$$

See also [Lecture → Section 5.3.2].

(7-1.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  $\|f - I_{\mathcal{M}}f\|_{L^\infty(I)}$  when  $n \rightarrow \infty$ ?

HIDDEN HINT 1 for (7-1.a) → 7-1-1-0:GradedMeshes1h.pdf

SOLUTION for (7-1.a) → 7-1-1-1:GradedMeshes1s.pdf ▲

(7-1.b)  (5 min.) What is the smoothness class of the function  $f : I := [0, 1] \rightarrow \mathbb{R}$ ,  $f(t) = t^\alpha$ ,  $0 < \alpha < 2$ ? In other words, for which  $k \in \mathbb{N}$  do we have  $f \in C^k(I)$ ?

HIDDEN HINT 1 for (7-1.b) → 7-1-2-0:GradedMeshes2h.pdf

SOLUTION for (7-1.b) → 7-1-2-1:GradedMeshes2s.pdf ▲

(7-1.c)  (30 min.) Implement a templated C++ function

```
template <typename FUNCTION>
double pwlintpMaxError(FUNCTION &&f, const Eigen::VectorXd &t);
```

that computes  $\|f - I_{\mathcal{M}}f\|_{L^\infty(I)}$  when  $f$  is passed through the functor  $f$  (with evaluation operator `double operator ()(double) const`) and the mesh  $\mathcal{M}$  through its *sorted* vector  $[t_0, t_1, \dots, t_n]^T \in \mathbb{R}^{n+1}$ ,  $n \in \mathbb{N}$ , 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  $\text{in } I$ , see, e.g., [Lecture → Ex. 6.5.1.7].

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

(7-1.d) ☐ (15 min.) [ depends on Sub-problem (7-1.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 error norms  $\|f - I_M f\|_{L^\infty(I)}$  as a function of  $n := \#M - 1 \rightarrow \infty$  for families of *equidistant meshes*. Use meshes with  $n = 2^k$ ,  $k = 5, 6, \dots, 12$ , and values for  $\alpha$  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 (7-1.d) → 7-1-4-0:sa1.pdf ▲

(7-1.e) ☐ (30 min.) [ depends on Sub-problem (7-1.c) ]

Create a C++ function

```
Eigen::VectorXd cvgRateEquidistantMesh(const Eigen::VectorXd
&alpha);
```

that estimates rates of asymptotic algebraic convergence (→ [Lecture → Rem. 6.1.2.9]) of  $\|f - I_M f\|_{L^\infty(I)}$  for  $f(t) := t^\alpha$  in terms of  $n := \#M - 1 \rightarrow \infty$  for families of *equidistant meshes* for the passed values of the parameter  $\alpha$  and returns the estimates. These regression-based estimates should rely on meshes with  $n = 2^k$ ,  $k = 5, 6, \dots, 12$ .

HIDDEN HINT 1 for (7-1.e) → 7-1-5-0:gm3h1.pdf

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

(7-1.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(void);
```

that plots (use **MATPLOTLIBCPP**) and tabulates the measured convergence rates based on  $\alpha = 0.05, 0.15, 0.25, \dots, 2.95$ . Save the plot in the file `cvgRateEquidistant.png`.

SOLUTION for (7-1.f) → 7-1-6-0:GradedMeshes3s.pdf ▲

(7-1.g) ☐ (5 min.) In which mesh interval do you expect  $|f - I_M f|$  to attain its maximum?

HIDDEN HINT 1 for (7-1.g) → 7-1-7-0:GradedMeshes4hb.pdf

SOLUTION for (7-1.g) → 7-1-7-1:GradedMeshes4s.pdf ▲

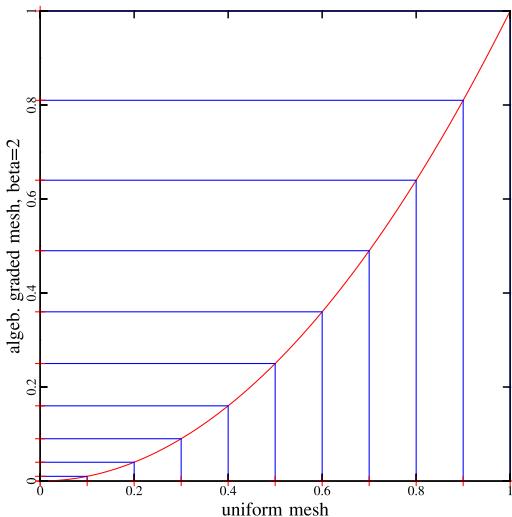
(7-1.h) ☐ (20 min.) [ depends on Sub-problem (7-1.g) ]

Compute (“on paper”) the exact value of  $\|f - I_M f\|_{L^\infty(I)}$  for  $f(t) := t^\alpha$ ,  $0 < \alpha < 2$ . Compare the order of convergence obtained with the one observed numerically in (7-1.b).

HIDDEN HINT 1 for (7-1.h) → 7-1-8-0:GradedMeshes5h.pdf

SOLUTION for (7-1.h) → 7-1-8-1:GradedMeshes5s.pdf ▲

(7-1.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.



A suitable choice is an **algebraically graded mesh**, defined as

$$\mathcal{G} = \left\{ t_j = \left( \frac{j}{n} \right)^\beta, j = 0, \dots, n \right\} \quad (7.1.8)$$

for the **grading parameter**  $\beta > 1$ .

▷ Algebraically graded mesh for  $\beta = 2$ .

Fig. 35

Realize a C++ function

```
Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
                                     const Eigen::VectorXd &beta);
```

that estimates rates of asymptotic algebraic convergence ( $\rightarrow$  [Lecture  $\rightarrow$  Rem. 6.1.2.9]) of  $\|f - I_G f\|_{L^\infty(I)}$  for  $f(t) := t^\alpha$  in terms of  $n := \#\mathcal{M} - 1 \rightarrow \infty$  for families of *graded meshes* for the passed values of the parameter  $\alpha$  (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  $\alpha$ -value, a row for each  $\beta$ -value. The regression-based estimates should rely on meshes with  $n = 2^k$ ,  $k = 5, 6, \dots, 12$ .

SOLUTION for (7-1.i)  $\rightarrow$  [7-1-9-0:.pdf](#)

(7-1.j) (20 min.) [ depends on Sub-problem (7-1.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  $\alpha$  in order to recover an asymptotic algebraic convergence like  $O(n^{-2})$  of  $\|f - I_G f\|_{L^\infty(I)}$  for  $\#\mathcal{G} \sim n \rightarrow \infty$ ?

To arrive at a conjecture rely on the function `cvgrateGradedMesh()` from Sub-problem (7-1.i) and write another function

```
void testcvgGradedMesh(void);
```

that calls `cvgrateGradedMesh()` for

$$\begin{aligned} \alpha &\in \{0.05, 0.15, 0.25, \dots, 2.95\}, \\ \beta &\in \{0.1, 0.2, 0.3, \dots, 2.0\}, \end{aligned}$$

and creates a useful plot of the data (using `MATPLOTLIBCPP`) and stores it in the file `alphabeta.png`.

SOLUTION for (7-1.j)  $\rightarrow$  [7-1-10-0:c1s.pdf](#)

**End Problem 7-1 , 170 min.**

### Problem 7-2: Piecewise linear interpolation with knots different from nodes

In [Lecture → Ex. 5.1.0.12] 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**  $x_0 < x_1 < x_2 < \dots < x_n$ ,
- (II) the (interpolation) **nodes**  $t_0 < t_1 < \dots < t_n$ ,  $n \in \mathbb{N}$ , satisfying  $x_0 \leq t_j \leq x_n$ ,  $j = 0, \dots, n$ .

The space of piecewise linear **continuous** functions with respect to the knot set  $\mathcal{N} := \{x_j\}_{j=0}^n$  is defined as:

$$\mathcal{S}_{1,\mathcal{N}} := \{s \in C^0([x_0, x_n]) : s(t) = \gamma_j t + \beta_j \text{ for } t \in ]x_{j-1}, x_j], \gamma_j, \beta_j \in \mathbb{R} \text{ } i = 1, \dots, n\}. \quad (7.2.1)$$

Compare [Lecture → Ex. 5.1.0.12] 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  $s(t_i) = y_i$ ,  $i = 0, \dots, n$ , for given values  $y_i \in \mathbb{R}$ .

This fits the general framework of [Lecture → § 5.1.0.15].

Below we set  $I_0 := [x_0, x_1]$ ,  $I_j := ]x_{j-1}, x_{j+1}]$ ,  $j = 1, \dots, n-1$ ,  $I_n := [x_{n-1}, x_n]$ .

- (7-2.a)** Show that the interpolation problem may not have a solution for some values  $y_j$  if a single interval  $[x_j, x_{j+1}]$ ,  $j = 0, \dots, n-1$ , contains more than 2 nodes  $t_i$ .

SOLUTION for (7-2.a) → 7-2-1-0:lips1.pdf

- (7-2.b)** Show that the interpolation problem can have a unique solution for any data values  $y_j$  only if each interval  $I_j$ ,  $j = 0, \dots, n$ , contains at least one node.

HIDDEN HINT 1 for (7-2.b) → 7-2-2-0:glgh1.pdf

SOLUTION for (7-2.b) → 7-2-2-1:lips2.pdf

- (7-2.c)** Show that the interpolation problem has a unique solution for all data values  $y_j$  if each of the closed intervals  $]x_0, x_1[, ]x_1, x_2[, ]x_2, x_3[, \dots, ]x_{n-1}, x_n[$  contains at least one node  $t_j$ .

HIDDEN HINT 1 for (7-2.c) → 7-2-3-0:glgh1.pdf

SOLUTION for (7-2.c) → 7-2-3-1:lips3.pdf

- (7-2.d)** Implement a C++ function

```
VectorXd tentBasCoeff(const VectorXd &x, const VectorXd &t,
                      const VectorXd &y);
```

that returns the vector of values  $s(x_j)$  of the interpolant  $s$  of the data points  $(t_i, y_i)$ ,  $i = 0, \dots, n$  in  $\mathcal{S}_{1,\mathcal{N}}$  in the knots  $x_j$ ,  $j = 0, \dots, n$ . 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 (7-2.c) is satisfied. You must not take for granted that knots or nodes are sorted already.

HIDDEN HINT 1 for (7-2.d) → 7-2-4-0:glgh1.pdf

SOLUTION for (7-2.d) → [7-2-4-1:glgs1.pdf](#) ▲

(7-2.e) ☐ Implement a C++ class

```
class PwLinIP {
public:
    PwLinIP(const VectorXd &x, const VectorXd &t, const VectorXd &y);
    double operator()(double arg) const;
private:
    ...
};
```

that realizes an interpolator class in the spirit of [Lecture → Rem. 5.1.0.21]. The meanings of the arguments of the constructor are the same as for the function `tentBasCoeff` from Sub-problem (7-2.d). Pay attention to the efficient implementation of the evaluation operator!

HIDDEN HINT 1 for (7-2.e) → [7-2-5-0:lipx1.pdf](#)

HIDDEN HINT 2 for (7-2.e) → [7-2-5-1:lipzz.pdf](#)

SOLUTION for (7-2.e) → [7-2-5-2:dsfg1.pdf](#) ▲

(7-2.f) ☐ Implement a C++ code that creates plots of the **cardinal basis functions** for interpolation in  $\mathcal{S}_{1,\mathcal{N}}$

- with know set  $\mathcal{N} := \{0, 1, 2, 3, \dots, 9, 10\}$
- and for interpolation nodes  $t_0 := 0, t_j = j - \frac{1}{2}, j = 1, \dots, 10$ .

Recall that the cardinal basis function  $b_k \in \mathcal{S}_{1,\mathcal{N}}$  is characterized by the conditions:

$$b_k(t_j) = \delta_{kj}, \quad k, j = 0, \dots, 10 \quad (7.2.6)$$

HIDDEN HINT 1 for (7-2.f) → [7-2-6-0:liphx.pdf](#)

SOLUTION for (7-2.f) → [7-2-6-1:glgs1.pdf](#) ▲

**End Problem 7-2**

### Problem 7-3: Adaptive polynomial interpolation

In [Lecture → Section 6.1.3] 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 variety 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

$$\mathcal{T}_{n+1} = \mathcal{T}_n \cup \left\{ \operatorname{argmax}_{t \in \mathcal{S}} |f(t) - I_{\mathcal{T}_n}(t)| \right\}, \quad (7.3.1)$$

where  $I_{\mathcal{T}_n}$  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  $\text{tol} > 0$ :

$$\max_{t \in \mathcal{S}} |f(t) - I_{\mathcal{T}_n}(t)| \leq \text{tol} \cdot \max_{t \in \mathcal{S}} |f(t)|. \quad (7.3.2)$$

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  $p(x_i)$ ,  $i = 1, \dots, N$ , where  $p \in \mathcal{P}_n$  is the global polynomial interpolant through the data points  $(t_j, y_j)$ ,  $j = 0, \dots, n$ , see [Lecture → Code 5.2.3.7].

#### C++ code 7.3.3: Function `intpolyval()`

```
2  VectorXd intpolyval(const VectorXd& t, const VectorXd& y,
3                         const VectorXd& x) {
4     const unsigned n = t.size(), // no. of interpolation nodes = deg. of
5                   polynomial - 1
5     N = x.size(); // no. of evaluation points
6
7     VectorXd p = VectorXd(N);
8
9     // Precompute the weights λ_i with effort O(n^2)
10    VectorXd lambda(n);
11    for (unsigned k = 0; k < n; ++k) {
12        // little workaround: cannot subtract a vector from a scalar
13        // -> multiply scalar by vector of ones
14        lambda(k) =
```

```

15     1./((t(k)*VectorXd::Ones(k) - t.head(k)).prod() *
16         (t(k)*VectorXd::Ones(n - k - 1) - t.tail(n - k - 1)).prod()) ;
17 }
18 // Compute quotient of weighted sums of  $\frac{\lambda_i}{t-t_i}$ ,
19 // effort  $O(n)$ 
20 for (unsigned i = 0; i < N; ++i) {
21     VectorXd z = (x(i)*VectorXd::Ones(n) - t);
22
23     // check if we want to evaluate at a node <-> avoid division by zero
24     double* ptr = std::find(z.data(), z.data() + n, 0.0);
25     if (ptr != z.data() + n) { // if ptr = z.data + n = z.end no zero was
26         found
27         p(i) = y(ptr - z.data()); // ptr - z.data gives the position of the
28         zero
29     } else {
30         VectorXd mu = lambda.cwiseQuotient(z);
31         p(i) = (mu.cwiseProduct(y)).sum() / mu.sum();
32     }
33 }
34 return p;
}

```

(7-3.a) (45 min.) Write a C++ function

```

template <class Function>
Eigen::VectorXd adaptivepolyintp(Function&& f,
                                  double a, double b,
                                  double tol, 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

$$\mathcal{S} := \{a + (b - a)j/N : j = 0, \dots, N\}, \quad (7.3.4)$$

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 (7-3.a) → 7-3-1-0:s1h1.pdf

SOLUTION for (7-3.a) → 7-3-1-1:solfile.pdf

(7-3.b) (15 min.) [ depends on Sub-problem (7-3.a) ]

Extend the function `adaptivepolyintp()` from Sub-problem (7-3.a) to

```

template <class Function>
Eigen::VectorXd
adaptivepolyintp(Function &&f,
                  double a, double b,
                  double tol, int N,
                  std::vector<double> *errortab = nullptr);

```

so that it also reports (and returns in `errortab`) the quantity

$$\epsilon_n := \max_{t \in \mathcal{S}} |f(t) - T_{\mathcal{T}_n}(t)| \quad (7.3.5)$$

for each intermediate node set  $\mathcal{T}_n$ .

SOLUTION for (7-3.b) → [7-3-2-0:solfile.pdf](#)



(7-3.c)  (15 min.) [ depends on Sub-problem (7-3.b) ]

For  $f_1(t) := \sin(e^{2t})$  and  $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$  write a C++ function

```
void plotInterpolationError(void);
```

that creates (employing [MATPLOTLIBCPP](#)) and stores (in `intperrplot.png`) a plot of  $e_n$  versus  $n$  (the number of interpolation nodes). Choose axis scales that reveal the qualitative decay (types of convergence as given in [Lecture → Def. 6.1.2.7]) of this error norm as the number of interpolation nodes is increased. Use interval  $[a, b] = [0, 1]$ ,  $N=1000$  sampling points, tolerance  $\text{tol} = 1e-6$ .

SOLUTION for (7-3.c) → [7-3-3-0:solfile.pdf](#)



**End Problem 7-3 , 75 min.**

### Problem 7-4: Chebyshev polynomials and their properties

Chebyshev polynomials as defined in [Lecture → Def. 6.1.3.3] are a sequence of orthogonal polynomials, which can be defined recursively, see [Lecture → Thm. 6.1.3.4]. 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. 6.1.3.3] and  $\xi_0^{(n)}, \dots, \xi_{n-1}^{(n)}$  be the  $n$  zeros of  $T_n$ . According to [Lecture → Eq. (6.1.3.10)], these are given by:

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \dots, n-1 \quad (7.4.1)$$

We define the family of discrete  $L^2$  semi-inner products (i.e. not conjugate symmetric), cf. [Lecture → Eq. (6.2.2.3)]:

$$(f, g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)})g(\xi_j^{(n)}), \quad f, g \in C^0([-1, 1]) \quad (7.4.2)$$

We also define the special weighted  $L^2$  semi-inner product:

$$(f, g)_w := \int_{-1}^1 \frac{1}{\sqrt{1-t^2}} f(t)g(t) dt \quad f, g \in C^0([-1, 1]) \quad (7.4.3)$$

**(7-4.a)** □ Show that the Chebyshev polynomials are an orthogonal family of polynomials with respect to the inner product defined in Eq. (7.4.3) according to [Lecture → Def. 6.2.2.6], namely  $(T_k, T_l)_w = 0$  for every  $k \neq l$ .

HIDDEN HINT 1 for (7-4.a) → 7-4-1-0:ChebPolyProperties1h.pdf

SOLUTION for (7-4.a) → 7-4-1-1:ChebPolyProperties1s.pdf



Consider the following statement:

#### Theorem 7.4.4.

The family of polynomials  $\{T_0, \dots, T_n\}$  is an orthogonal basis ([Lecture → Def. 6.2.1.14]) of  $\mathcal{P}_n$  with respect to the inner product  $( , )_{n+1}$  defined in Eq. (7.4.2).

**(7-4.b)** □ Write a C++ code to test the assertion of Thm. 7.4.4.

HIDDEN HINT 1 for (7-4.b) → 7-4-2-0:ChebPolyProperties2h.pdf

SOLUTION for (7-4.b) → 7-4-2-1:ChebPolyProperties2s.pdf



**(7-4.c)** □ Prove Thm. 7.4.4.

HIDDEN HINT 1 for (7-4.c) → 7-4-3-0:ChebPolyProperties3h.pdf

SOLUTION for (7-4.c) → 7-4-3-1:ChebPolyProperties3s.pdf



**(7-4.d)** □ Given a function  $f \in C^0([-1, 1])$ , find an expression for the best approximant  $q_n \in \mathcal{P}_n$  of  $f$  in the discrete  $L^2$ -norm:

$$q_n = \operatorname{argmin}_{p \in \mathcal{P}_n} |f - p|_{n+1},$$

$\| \cdot \|_{n+1}$  is the norm induced by the scalar product  $(\cdot, \cdot)_{n+1}$ . You should represent  $q_n$  through an expansion in Chebychev polynomials of the form:

$$q_n = \sum_{j=0}^n \alpha_j T_j, \quad (7.4.12)$$

for suitable coefficients  $\alpha_j \in \mathbb{R}$ , see also [Lecture → Eq. (6.1.3.27)].

HIDDEN HINT 1 for (7-4.d) → 7-4-4-0:ChebPolyProperties4h.pdf

SOLUTION for (7-4.d) → 7-4-4-1:ChebPolyProperties4s.pdf ▲

(7-4.e) ☐ Write a C++ function that returns the vector of coefficients  $(\alpha_j)_j$  in Eq. (7.4.12) given a function  $f$ :

```
template <typename Function>
void bestpolchebnodes(const Function &f, Eigen::VectorXd &alpha)
```

Note that the degree of the polynomial is indirectly passed with the length of the output `alpha`. The input `f` is a lambda-function, e.g.:

```
auto f = [] (double & x) { return 1 / (pow(5*x, 2)+1); };
```

SOLUTION for (7-4.e) → 7-4-5-0:ChebPolyProperties5s.pdf ▲

(7-4.f) ☐ Test `bestpolchebnodes` with the function  $f(x) = \frac{1}{(5x)^2+1}$  and  $n = 20$ . Approximate the supremum norm of the approximation error by sampling on an equidistant grid with  $10^6$  points.

HIDDEN HINT 1 for (7-4.f) → 7-4-6-0:ChebPolyProperties6h.pdf

SOLUTION for (7-4.f) → 7-4-6-1:ChebPolyProperties6s.pdf ▲

(7-4.g) ☐ Let  $L_j$ ,  $j = 0, \dots, n$ , be the Lagrange polynomials associated with the nodes  $t_j = \xi_j^{(n+1)}$  of the Chebyshev interpolation with  $n+1$  nodes on  $[-1, 1]$  (see [Lecture → Eq. (6.1.3.10)]). Show that:

$$L_j = \frac{1}{n+1} + \frac{2}{n+1} \sum_{l=1}^n T_l(\xi_j^{(n+1)}) T_l$$

HIDDEN HINT 1 for (7-4.g) → 7-4-7-0:ChebPolyProperties7h.pdf

SOLUTION for (7-4.g) → 7-4-7-1:ChebPolyProperties7s.pdf ▲

End Problem 7-4

### Problem 7-5: Chebychev interpolation of analytic functions

This problem concerns Chebychev interpolation as discussed in [Lecture → Section 6.1.3].

Problem involves implementation in C++ based on EIGEN.

Using techniques from complex analysis, notably the residue theorem [Lecture → Thm. 6.1.2.33], in class we derived an expression for the interpolation error [Lecture → Eq. (6.1.2.39)] and from it an error bound [Lecture → Eq. (6.1.2.40)], as much sharper alternative to [Lecture → ??] and [Lecture → Lemma 6.1.2.17] for *analytic* interpolants. The bound tells us that for all  $t \in [a, b]$

$$|f(t) - L_T f(t)| \leq \left| \frac{w(x)}{2\pi i} \int_{\gamma} \frac{f(z)}{(z-t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a \leq \tau \leq b} |w(\tau)|}{\min_{z \in \gamma} |w(z)|} \frac{\max_{z \in \gamma} |f(z)|}{d([a, b], \gamma)},$$

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  $\gamma$  must be contractible in the domain  $D$  of analyticity of  $f$  and must wind around  $[a, b]$  exactly once, see [Lecture → Fig. 202].

Now we consider the interval  $[-1, 1]$ . Following [Lecture → Rem. 6.1.3.22], 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, see [Lecture → Fig. 216],

$$\gamma_\rho(\theta) := \cos(\theta - i \log(\rho)), \quad \forall 0 \leq \theta \leq 2\pi, \quad \rho > 1. \quad (7.5.1)$$

**(7-5.a)**  Find an upper bound for the length  $|\gamma_\rho|$  of the contour  $\gamma_\rho$ .

HIDDEN HINT 1 for (7-5.a) → [prbfile.pdf](#)

SOLUTION for (7-5.a) → [7-5-1-1:solfile.pdf](#)



Now consider the S-curve function (the logistic function):

$$f(t) := \frac{1}{1 + e^{-3t}}, \quad t \in \mathbb{R}.$$

**(7-5.b)**  Determine the maximal domain of analyticity of the extension of  $f$  to the complex plane  $\mathbb{C}$ .

HIDDEN HINT 1 for (7-5.b) → [prbfile.pdf](#)

SOLUTION for (7-5.b) → [7-5-2-1:solfile.pdf](#)



**(7-5.c)**  Write a C++ function that computes an approximation  $M$  of:

$$\min_{\rho > 1} \frac{\max_{z \in \gamma_\rho} |f(z)|}{d([-1, 1], \gamma_\rho)}, \quad (7.5.10)$$

by sampling, where the distance of  $[a, b]$  from  $\gamma_\rho$  is formally defined as

$$d([a, b], \gamma) := \inf\{|z - t| \mid z \in \gamma, t \in [a, b]\}. \quad (7.5.11)$$

HIDDEN HINT 1 for (7-5.c) → [prbfile.pdf](#)

HIDDEN HINT 2 for (7-5.c) → [prbfile.pdf](#)

HIDDEN HINT 3 for (7-5.c) → [prbfile.pdf](#)

HIDDEN HINT 4 for (7-5.c) → [prbfile.pdf](#)

SOLUTION for (7-5.c) → [7-5-3-4:solfile.pdf](#)



(7-5.d) ☐ Based on the result of (7-5.c), and [Lecture → Eq. (6.1.3.24)], give an “optimal” bound for

$$\|f - L_n f\|_{L^\infty([-1,1])},$$

where  $L_n$  is the operator of Chebychev interpolation on  $[-1, 1]$  into the space of polynomials of degree  $\leq n$ .

SOLUTION for (7-5.d) → [7-5-4-0:solfile.pdf](#)



(7-5.e) ☐ Graphically compare your result from (7-5.d) with the measured supremum norm of the approximation error of Chebychev interpolation of  $f$  on  $[-1, 1]$  for polynomial degree  $n = 1, \dots, 20$ . To that end, write a C++-code and rely on the provided function `intpolyval` (cf. [Lecture → Code 5.2.3.14]).

HIDDEN HINT 1 for (7-5.e) → [prbfile.pdf](#)

HIDDEN HINT 2 for (7-5.e) → [prbfile.pdf](#)

SOLUTION for (7-5.e) → [7-5-5-2:solfile.pdf](#)



(7-5.f) ☒ Rely on pullback to  $[-1, 1]$  to discuss how the error bounds in [Lecture → Eq. (6.1.3.24)] will change when we consider Chebychev interpolation on  $[-a, a], a > 0$ , instead of  $[-1, 1]$ , whilst keeping the function  $f$  fixed.

SOLUTION for (7-5.f) → [7-5-6-0:solfile.pdf](#)



**End Problem 7-5**

### Problem 7-6: Polynomial Best Approximation

The famous Jackson Theorem [Lecture → Thm. 6.1.1.11] 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 6.1.1]

Throughout this problems we write Here

(7-6.a) (7 min.) The following is the assertion of Jackson's theorem of [Lecture → Thm. 6.1.1.11] 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.

$$\inf_{p \in \mathcal{P}_n} \|f - p\| \boxed{\quad} \leq \left(1 + \frac{\pi^2}{2}\right)^{\boxed{\quad}} \frac{(\boxed{\quad} - \boxed{\quad})!}{\boxed{\quad}!} \|f^{(\boxed{\quad})}\| \boxed{\quad}$$

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 (7-6.a) → [7-6-1-0:s1.pdf](#)



(7-6.b) (8 min.) For  $n \in \mathbb{N}$  we abbreviate

$$E_n(f) := \inf_{p \in \mathcal{P}_n} \|f - p\|_{L^\infty([-1, 1])} \quad \text{for } f \in C^0([-1, 1]). \quad (7.6.2)$$

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) $E_n(f+g) \leq E_n(f) + E_n(g)$	<input type="radio"/>	<input type="radio"/>
(ii) $E_n(\lambda f) = \lambda E_n(f)$ for all $\lambda \in \mathbb{R}$	<input type="radio"/>	<input type="radio"/>
(iii) $E_n$ is a norm on $C^0([-1, 1])$	<input type="radio"/>	<input type="radio"/>
(iv) $E_n(f) - E_n(g) \leq E_n(f-g)$	<input type="radio"/>	<input type="radio"/>

HIDDEN HINT 1 for (7-6.b) → [7-6-2-0:2norm.pdf](#)

SOLUTION for (7-6.b) → [7-6-2-1:s2.pdf](#)



**End Problem 7-6 , 15 min.**

# Chapter 8

## Numerical Quadrature

### Problem 8-1: Smooth integrand by transformation

In [Lecture → Rem. 7.3.0.47] 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.

Related problems are Problem 8-9 and Problem 8-7.

Given a smooth, odd function  $f : [-1, 1] \rightarrow \mathbb{R}$ , consider the integral

$$I := \int_{-1}^1 \arcsin(t) f(t) dt. \quad (8.1.1)$$

We want to approximate this integral using global Gauss quadrature. The nodes (vector  $\mathbf{x}$ ) and the weights (vector  $\mathbf{w}$ ) of an  $n$ -point Gaussian quadrature on  $[-1, 1]$  can be computed using the provided C++ routine `gaussquad` (found in `gaussquad.hpp`).

We have the following function:

```
QuadRule gaussquad(const unsigned n);
```

`QuadRule` is a struct containing the quadrature weights  $\mathbf{w}$  and nodes  $\mathbf{x}$ :

```
struct QuadRule {
    Eigen::VectorXd nodes, weight;
};
```

(8-1.a) Write a C++ routine

```
template <class Function>
void gaussConv(const Function & f);
```

that produces an appropriate convergence plot of the quadrature error versus the number  $n = 1, \dots, 50$  of quadrature points. Here  $f$  is a handle to the function  $f$ .

Save your convergence plot for  $f(t) = \sinh(t)$  as `GaussConv.eps`.

HIDDEN HINT 1 for (8-1.a) → [8-1-1-0:hcv.pdf](#)

SOLUTION for (8-1.a) → [8-1-1-1:gauq1.pdf](#)



- (8-1.b)**  Describe qualitatively and quantitatively the asymptotic (for  $n \rightarrow \infty$ ) convergence of the quadrature error you observe in (8-1.a).

HIDDEN HINT 1 for (8-1.b) → [8-1-2-0:h1.pdf](#)

SOLUTION for (8-1.b) → [8-1-2-1:gauq2.pdf](#) ▲

- (8-1.c)**  Transform the previous integral into an equivalent one, with a suitable change of variable, so that the Gauss quadrature applied to the transformed integral can be expected to converge exponentially if  $f \in C^\infty([-1,1])$ .

Use the transformation  $t = \sin(x)$ .

SOLUTION for (8-1.c) → [8-1-3-0:gauq3.pdf](#) ▲

- (8-1.d)**  Now, write a C++ function

```
template <class Function>
void gaussConvCV(const Function & f);
```

which plots (employing [MATPLOTLIBCPP](#)) the quadrature error versus the number  $n = 1, \dots, 50$  of quadrature points for the integral obtained in the previous subtask.

Again, as in Sub-problem (8-1.a), choose  $f(t) = \sinh(t)$  and save your convergence plot as `GaussConvCV.png`.

SOLUTION for (8-1.d) → [8-1-4-0:gauq4.pdf](#) ▲

- (8-1.e)**  Similar question as in Sub-problem (8-1.b): describe qualitatively the asymptotic (for  $n \rightarrow \infty$ ) convergence of the quadrature error you observe in Sub-problem (8-1.d).

SOLUTION for (8-1.e) → [8-1-5-0:gauq2.pdf](#) ▲

- (8-1.f)**  Explain the difference between the results obtained in Sub-problem (8-1.a) and Sub-problem (8-1.d).

SOLUTION for (8-1.f) → [8-1-6-0:gauq5.pdf](#) ▲

**End Problem 8-1**

### Problem 8-2: Generalized “Hermite-type” quadrature formula

In this exercise we will construct a new quadrature formula and then use it in an application.

This is a purely theoretical exercise. You may want to have a look at the methods used in [Lecture → Ex. 7.3.0.14], as they will come handy.

**(8-2.a)** Determine  $A, B, C, x_1 \in \mathbb{R}$  such that the quadrature formula

$$\int_0^1 f(x)dx \approx Af(0) + Bf'(0) + Cf(x_1) \quad (8.2.1)$$

is exact for polynomials of highest possible degree.

SOLUTION for (8-2.a) → [8-2-1-0:herm1s.pdf](#)

**(8-2.b)** Compute an approximation of  $z(2)$  using the quadrature rule of Eq. (8.2.1), where the function  $z$  is defined as the solution of the initial value problem

$$z'(t) = \frac{t}{1+t^2} \quad , \quad z(1) = 1 . \quad (8.2.2)$$

HIDDEN HINT 1 for (8-2.b) → [8-2-2-0:herm2h.pdf](#)

SOLUTION for (8-2.b) → [8-2-2-1:herm2s.pdf](#)

**End Problem 8-2**

### Problem 8-3: Numerical Quadrature of Improper Integrals

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  $f : \mathbb{R} \rightarrow \mathbb{R}$  that decay sufficiently fast for  $|t| \rightarrow \infty$  (such that they are integrable on  $\mathbb{R}$ ).

The problem makes extensive use of substitution rules for integrals, implementation in C++ is requested.

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) on  $[-b, b]$ .

**(8-3.a)**  (10 min.) For the integrand  $g(t) := 1/(1+t^2)$  determine  $b$  such that the truncation error  $E_T$  satisfies:

$$E_T := \left| \int_{-\infty}^{\infty} g(t)dt - \int_{-b}^b g(t)dt \right| \leq 10^{-6}. \quad (8.3.1)$$

SOLUTION for (8-3.a) → [8-3-1-0:oplsol.pdf](#)

**(8-3.b)**  (5 min.) What is the algorithmic difficulty faced in the implementation of the truncation approach for a generic integrand?

SOLUTION for (8-3.b) → [8-3-2-0: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  $t = \cot(s) := \frac{\cos s}{\sin s}$ ,  $\cot : ]0, \pi[ \rightarrow ]-\infty, \infty[$ .

**(8-3.c)**  (10 min.) Into which integral does the substitution  $t = \cot(s)$  convert  $\int_{-\infty}^{\infty} f(t)dt$ ?

SOLUTION for (8-3.c) → [8-3-3-0:op2sub.pdf](#)

**(8-3.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 (8-3.d) → [8-3-4-0:h1ti.pdf](#)

SOLUTION for (8-3.d) → [8-3-4-1:op2com.pdf](#)

**(8-3.e)**  (20 min.) 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. 7.3.0.36], see also [Lecture → Code 7.3.0.37].

HIDDEN HINT 1 for (8-3.e) → [8-3-5-0:hlf1.pdf](#)

SOLUTION for (8-3.e) → [8-3-5-1:op2imp.pdf](#) ▲

(8-3.f)  (15 min.) Implement a C++ function

```
void cvgQuadInf(void);
```

in order to study the convergence for  $n \rightarrow \infty$  of the quadrature method implemented in the previous subproblem for the integrand  $h(t) := \exp(-(t-1)^2)$  (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 (8-3.f) → [8-3-6-0:testsol.pdf](#) ▲

**End Problem 8-3 , 70 min.**

### Problem 8-4: 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

$$I(x, y) := \exp(-\alpha((x - p)^2 + (y - q)^2)), \quad x, y \in \mathbb{R} \quad (8.4.1)$$

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.

- (8-4.a)  (5 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 (8-4.a) → [8-4-1-0:neqls.pdf](#)



- (8-4.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 7.1] 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. 7.1.0.1].)

SOLUTION for (8-4.b) → [8-4-2-0:neq2s.pdf](#)



- (8-4.c)  (20 min.) [ depends on Sub-problem (8-4.a) ]

Write a C++ function

```
template <class Function>
double gaussquadtriangle(const Function &f, const unsigned N)
```

for the computation of the integral

$$\int_{\Delta} f(x, y) dx dy \quad (8.4.3)$$

using nested  $n$ -point, 1D **Gauss quadrature** [Lecture → Section 7.3] in combination with the function `evalquad()` of Sub-problem (8-4.b). Use the function `gaussquad()` from `gaussquad.hpp` to obtain weights and nodes of Gauss quadrature formulas on  $[-1, 1]$ .

HIDDEN HINT 1 for (8-4.c) → [8-4-3-0:neq3h1.pdf](#)

HIDDEN HINT 2 for (8-4.c) → [8-4-3-1:neq3h2.pdf](#)

SOLUTION for (8-4.c) → [8-4-3-2:neq3s.pdf](#)



(8-4.d)  Write a C++ function

```
void convtest2DQuad(unsigned int nmax = 20);
```

that applies the function `gaussquadtriangle()` from Sub-problem (8-4.c) to approximate  $\int_{\Delta} I(x, y) dx dy$  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  $n = 1, 2, 3, \dots, 20$  using the “exact” value of the integral  $I \approx 0.366046550000405$ . What kind of convergence do you observe and why?

HIDDEN HINT 1 for (8-4.d) → [8-4-4-0:neq4h.pdf](#)

SOLUTION for (8-4.d) → [8-4-4-1:neq4s.pdf](#)



**End Problem 8-4 , 40 min.**

### Problem 8-5: Quadrature 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 7.3.0.43], [Lecture → Ex. 7.3.0.46], [Lecture → § 7.4.0.9], [Lecture → Exp. 7.4.0.12].

We consider three different functions on the interval  $I = [0, 1]$ , which have the following properties:

- function A:  $f_A \in C^\infty(I)$ ,  $f_A \notin \mathcal{P}_k \forall k \in \mathbb{N}$ ;
- function B:  $f_B \in C^0(I)$ ,  $f_B \notin C^1(I)$ ;
- function C:  $f_C \in \mathcal{P}_{12}$ ,

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.

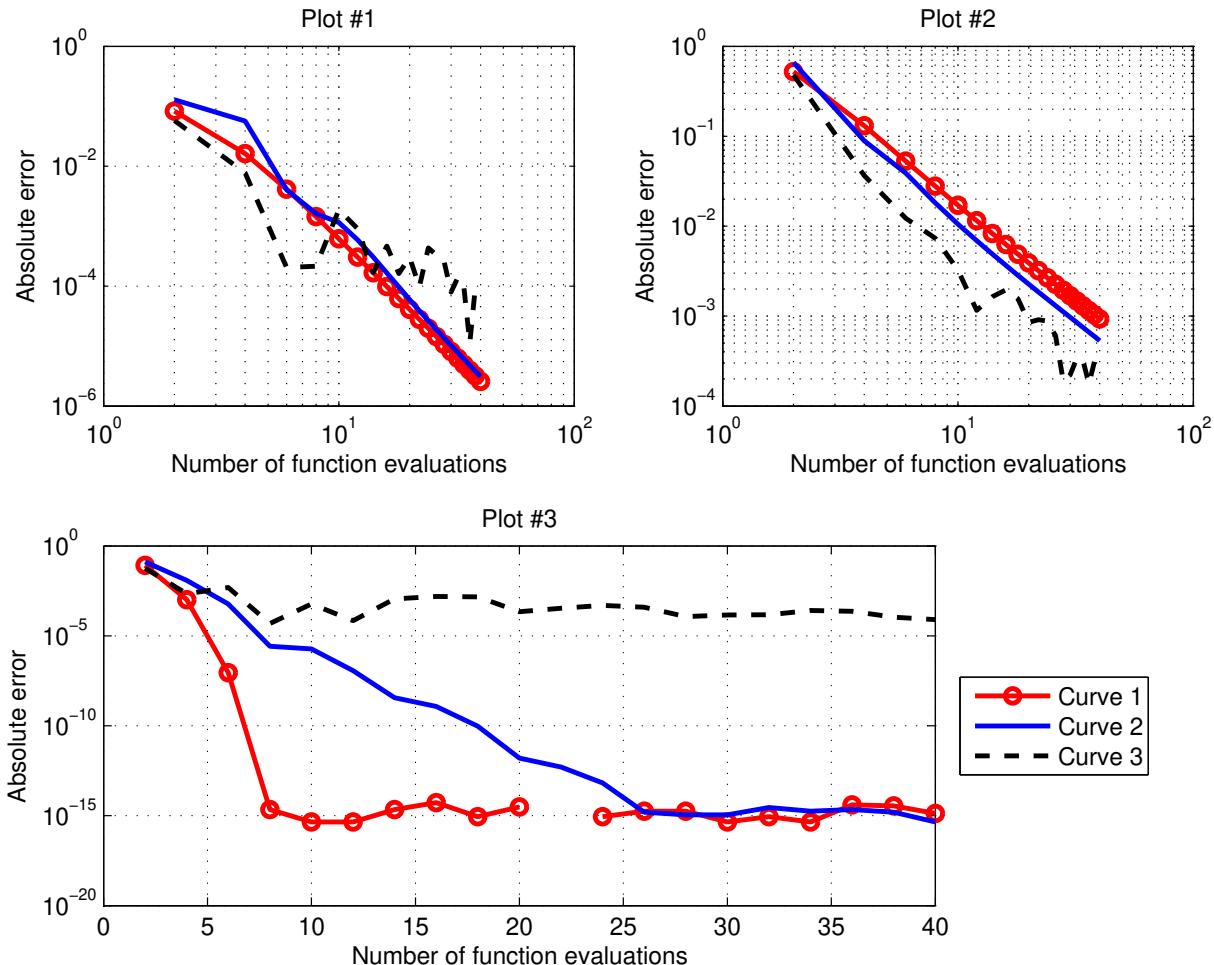


Fig. 44

- (8-5.a)  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 (8-5.a) → [8-5-1-0:plot1h.pdf](#)

SOLUTION for (8-5.a) → [8-5-1-1:plot1s.pdf](#) ▲

(8-5.b) ☺ 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 (8-5.b) → [8-5-2-0:plot2s.pdf](#) ▲

**End Problem 8-5**

### Problem 8-6: Weighted Gauss quadrature

In [Lecture → Rem. 7.3.0.47] 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 7.3] and requests a little implementation in C++.

The development of an alternative quadrature formula for a particular singular integral relies on the Chebychev polynomials of the second kind  $U_n$ , defined as

$$U_n(t) = \frac{\sin((n+1)\arccos t)}{\sin(\arccos t)}, \quad n \in \mathbb{N}.$$

The  $U_n$  are **orthogonal polynomials** with respect to a weighted  $L^2$  inner product (see [Lecture → Eq. (6.2.2.2)]) with weight function given by  $w(\tau) = \sqrt{1-\tau^2}$ .

(8-6.a)  (10 min.) Recall the role of the orthogonal Legendre polynomials in the derivation and definition of Gauss-Legendre quadrature rules (see [Lecture → § 7.3.0.27]). ▲

(8-6.b)  (15 min.) Show that the  $U_n$  satisfy the 3-term recursion

$$U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t), \quad U_0(t) = 1, \quad U_1(t) = 2t,$$

for every  $n \geq 1$ .

SOLUTION for (8-6.b) → [8-6-2-0:wgq1s.pdf](#)

(8-6.c)  (5 min.) Show that  $U_n \in \mathcal{P}_n$  and has leading coefficient  $2^n$ .

SOLUTION for (8-6.c) → [8-6-3-0:wgq2s.pdf](#)

(8-6.d)  (15 min.) Show that for every  $m, n \in \mathbb{N}_0$ , we have

$$\int_{-1}^1 \sqrt{1-t^2} U_m(t) U_n(t) dt = \frac{\pi}{2} \delta_{mn}.$$

SOLUTION for (8-6.d) → [8-6-4-0:wgq3s.pdf](#)

(8-6.e)  (10 min.) What are the zeros  $\xi_j^n$  ( $j = 1, \dots, n$ ) of  $U_n$ ,  $n \geq 1$ ? Give an explicit formula similar to the formula for the Chebyshev nodes in  $[-1, 1]$ .

SOLUTION for (8-6.e) → [8-6-5-0:wgq4s.pdf](#)

(8-6.f)  (30 min.) [ depends on Sub-problem (8-6.d) ]

Show that the choice of weights

$$w_j = \frac{\pi}{n+1} \sin^2\left(\frac{j}{n+1}\pi\right), \quad j = 1, \dots, n,$$

ensures that the quadrature formula

$$Q_n^U(f) = \sum_{j=1}^n w_j f(\xi_j^n) \quad (8.6.2)$$

provides the exact value of the integral

$$I := \int_{-1}^1 \sqrt{1-t^2} f(t) dt$$

for  $f \in \mathcal{P}_{n-1}$  (assuming exact arithmetic).

SOLUTION for (8-6.f) → 8-6-6-0:wqq5s.pdf



(8-6.g) ☐ Show that the quadrature formula (8.6.2) gives the exact value of (??) even for every  $f \in \mathcal{P}_{2n-1}$ .

HIDDEN HINT 1 for (8-6.g) → 8-6-7-0:wqq6h.pdf

SOLUTION for (8-6.g) → 8-6-7-1:wqq6s.pdf



(8-6.h) ☐ Show that the quadrature error

$$|Q_n^U(f) - W(f)|$$

decays to 0 exponentially as  $n \rightarrow \infty$  for every  $f \in C^\infty([-1, 1])$  that admits an analytic extension to an open subset of the complex plane.

HIDDEN HINT 1 for (8-6.h) → 8-6-8-0:wqq7h.pdf

SOLUTION for (8-6.h) → 8-6-8-1:wqq7s.pdf



(8-6.i) ☐ 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 an 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 (8-6.i) → 8-6-9-0:wqq8s.pdf



(8-6.j) ☐ Devise a C++ function

```
void testQuadU(unsigned int nmax = 20);
```

that, for the function  $f(t) = 1/(2 + e^{3t})$  and  $n = 1, \dots, n_{\text{max}}$  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. 6.1.2.7].

SOLUTION for (8-6.j) → 8-6-10-0:wqq9s.pdf



**End Problem 8-6 , 85 min.**

### Problem 8-7: Quadrature by transformation

In [Lecture → Rem. 7.3.0.47] a suitable transformation converted a singular integrand into a smooth one. In this problem we see another example. Also Problem 8-9 and Problem 8-1 cover this topic.

This problem does not contain an implementation part. Study [Lecture → Rem. 7.3.0.47] to learn about regularizing transformations.

For  $f \in C^0([0, 2])$  we consider the definite improper integral with *singular integrand*

$$I(f) := \int_0^2 \sqrt{\frac{2-t}{t}} f(t) dt. \quad (8.7.1)$$

**(8-7.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 (8-7.a) → 8-7-1-0:hsil1.pdf

HIDDEN HINT 2 for (8-7.a) → 8-7-1-1:hsi5.pdf

SOLUTION for (8-7.a) → 8-7-1-2:s1t.pdf



**(8-7.b)** (20 min.) For  $n \in \mathbb{N}^*$ , derive a family of  $2n$ -point quadrature formulas

$$Q_n(f) = \sum_{j=1}^{2n} w_j^n f(c_j^n), \quad f \in C([0, 2])$$

for which  $Q_n(f)$  converges to  $I(f)$  exponentially as  $n \rightarrow \infty$ , 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 (8-7.b) → 8-7-2-0:qtf1.pdf



**(8-7.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 (8-7.c) → 8-7-3-0:1tf2ZerosLegendre5s.pdf



**End Problem 8-7 , 30 min.**

### Problem 8-8: Discretization of an integral operator

In this problem we consider a completely new concept, an **integral operator** of the form

$$(\mathbf{T}f)(x) = \int_I k(x, y) f(y) dy, \quad x \in I \subset \mathbb{R}, \quad (8.8.1)$$

where the **kernel**  $k$  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 7.3] 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  $g_n(x)$ .

**(8-8.a)** ☐ 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  $j = 1, \dots, n - 1$ . We can write

$$(g_n(\xi_l^n))_{l=1}^n = M(f(\xi_j^n))_{j=1}^n,$$

for a suitable matrix  $M \in \mathbb{R}^{n,n}$ . Give a formula for the entries of  $M$ .

SOLUTION for (8-8.a) → 8-8-1-0:ongp1.pdf ▲

**(8-8.b)** ☐ Implement a function

```
template <typename Function>
Eigen::VectorXd comp_g_gausspts(Function f, unsigned int n)
```

that computes the  $n$ -vector  $(g_n(\xi_l^n))_{l=1}^n$  with optimal complexity  $O(n)$  (excluding the computation of the nodes and weights), where  $f$  is an object with an evaluation operator, 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 weights  $w$  and the ordered nodes  $xi$  relative to the  $n$ -point Gauss quadrature on the interval  $[-1, 1]$ .

SOLUTION for (8-8.b) → 8-8-2-0:ongp2.pdf ▲

**(8-8.c)** ☐ Test your implementation and write a C++ function

```
double testCompGGaussPts(void);
```

that computes  $g(\xi_{11}^{21})$  for  $f(y) = e^{-|0.5-y|}$ . What result do you expect?

SOLUTION for (8-8.c) → 8-8-3-0:ongp3.pdf ▲

### End Problem 8-8

### Problem 8-9: Efficient quadrature of singular integrands

We know that smoothness of the integrand is essential for fast convergence 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. 7.3.0.47]. For other problems on the same topic see Problem 8-7 and Problem 8-1.

Our task is to develop quadrature formulas for integrals of the form:

$$W(f) := \int_{-1}^1 \sqrt{1-t^2} f(t) dt, \quad (8.9.1)$$

where  $f$  possesses an analytic extension to a complex neighbourhood of  $[-1, 1]$ .

(8-9.a)  The function

```
QuadRule gauleg(unsigned int n);
```

(contained in `gauleg.hpp`) returns a structure `QuadRule` containing nodes ( $x_j$ ) and weights ( $w_j$ ) of a Gauss-Legendre quadrature [Lecture → Def. 7.3.0.30] on  $[-1, 1]$  with  $n$  nodes. ▲

(8-9.b)  Study [Lecture → § 7.3.0.39] 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 (8-9.b) → [8-9-2-0:effs1.pdf](#)

(8-9.c)  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  $n \rightarrow \infty$ , the error of your implementation has asymptotic exponential convergence to zero.

HIDDEN HINT 1 for (8-9.c) → [8-9-3-0:h21.pdf](#)

HIDDEN HINT 2 for (8-9.c) → [8-9-3-1:h23.pdf](#)

SOLUTION for (8-9.c) → [8-9-3-2:effs2.pdf](#)

(8-9.d)  Give formulas for the nodes  $c_j$  and weights  $\tilde{w}_j$  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.

SOLUTION for (8-9.d) → [8-9-4-0:effs3.pdf](#)

(8-9.e) 2 Implement a C++ function

```
void tabQuadErr(void)
```

that tabulates the quadrature error:

$$\epsilon_i := |W(f) - \text{quadsingint}(f, n)|$$

for  $f(t) := \frac{1}{2 + \exp(3t)}$  and  $n = 1, 2, \dots, 25$ . Then describe the type of convergence observed, see [Lecture → Def. 6.1.2.7].

SOLUTION for (8-9.e) → [8-9-5-0:effs5.pdf](#)



End Problem 8-9

### Problem 8-10: Zeros of orthogonal polynomials

This problem combines elementary methods for finding zeros from [Lecture → Section 8.3] and 3-term recursions satisfied by orthogonal polynomials, cf. [Lecture → Thm. 6.2.2.13]. This will permit us to find the zeros of Legendre polynomials, the so-called **Gauss nodes** (see [Lecture → Def. 7.3.0.30]).

Connected with [Lecture → Section 7.3] and [Lecture → Section 8.3]

The zeros of the Legendre polynomial  $P_n$  (see [Lecture → Def. 7.3.0.28]) are the  $n$  Gauss points  $\xi_j^n$ ,  $j = 1, \dots, n$ . In this problem we compute the Gauss points by zero-finding methods applied to  $P_n$ . The 3-term recursion [Lecture → Eq. (7.3.0.34)] for Legendre polynomials will play an essential role. Moreover, recall that, by definition, the Legendre polynomials are  $L^2([-1, 1])$ -orthogonal.

**(8-10.a)** Prove the following **interleaving property** of the zeros of the Legendre polynomials. For all  $n \in \mathbb{N}_0$  we have:

$$-1 < \xi_j^n < \xi_j^{n-1} < \xi_{j+1}^n < 1, \quad j = 1, \dots, n-1.$$

HIDDEN HINT 1 for (8-10.a) → 8-10-1-0:ZerosLegendre1h.pdf

SOLUTION for (8-10.a) → 8-10-1-1:ZerosLegendre1s.pdf

**(8-10.b)** By differentiating [Lecture → Eq. (7.3.0.34)] derive a combined 3-term recursion for the sequences  $(P_n)_n$  and  $(P'_n)_n$ .

SOLUTION for (8-10.b) → 8-10-2-0:ZerosLegendre2s.pdf

**(8-10.c)** Use the recursions obtained in (8-10.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  $\{P'_k(x_j)\}_{jk}$ ,  $j = 0, \dots, N-1$  and  $k = 0, \dots, n$ , for an input vector  $x \in \mathbb{R}^N$  (passed in  $x$ ):

SOLUTION for (8-10.c) → 8-10-3-0:ZerosLegendre3s.pdf

**(8-10.d)** We can compute the zeros of  $P_k$ ,  $k = 1, \dots, n$ , by means of the secant rule (see [Lecture → § 8.3.2.21]) using the endpoints  $\{-1, 1\}$  of the interval and the zeros of the previous Legendre polynomial as initial guesses; see (8-10.a). We opt for a correction-based termination criterion (see [Lecture → Section 8.1.2]) based on prescribed relative and absolute tolerance (see [Lecture → Code 8.3.2.24]).

Write a C++ function that computes the Gauss points  $\xi_j^k \in [-1, 1]$ ,  $j = 1, \dots, k$  and  $k = 1, \dots, n$ , 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 (8-10.d) → 8-10-4-0:ZerosLegendre4h.pdf

SOLUTION for (8-10.d) → [8-10-4-1:ZerosLegendre4s.pdf](#)



**(8-10.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 (8-10.e) → [8-10-5-0:ZerosLegendre5h.pdf](#)

SOLUTION for (8-10.e) → [8-10-5-1:ZerosLegendre5s.pdf](#)



**(8-10.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  $\text{sign}[f(x^{(k+1)})] \neq \text{sign}[f(z)]$ . 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 (8-10.f) → [8-10-6-0:ZerosLegendre6s.pdf](#)



**End Problem 8-10**

### Problem 8-11: Quadrature rules and quadrature errors

When sequences of quadrature rules are applied to evaluate  $\int_a^b f(t) dt$ , then the asymptotic convergence of quadrature error in terms of  $n \rightarrow \infty$ ,  $n \triangleq$  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 7.3], in particular [Lecture → Ex. 7.3.0.46], and also [Lecture → Exp. 7.4.0.12], [Lecture → Exp. 7.4.0.16]

In this problem we consider three different families of quadrature rules, for which we use the following abbreviations:

**(TR)**  $\triangleq$  Equidistant trapezoidal rule, see [Lecture → Eq. (7.4.0.17)],

**(SR)**  $\triangleq$  Equidistant Simpson rule, see [Lecture → Eq. (7.4.0.5)],

**(GR)**  $\triangleq$  Gauss-Legendre quadrature rules, as defined in [Lecture → Def. 7.3.0.30].

**(8-11.a)** (3 min.) The following C++ function implements the  $n$ -point equidistant trapezoidal rule,  $n \in \mathbb{N}$ ,  $n \geq 2$ , for the approximate evaluation of  $\int_a^b f(t) dt$ . The integrand is passed through the functor  $f$ .

```

1 template <typename Functor>
2 double equidTrapezoidalRule(Functor &&f, double a, double b, unsigned int n) {
3     assert(n>=2);
4     const double h = (b - a)/(n-1);
5     double t = a + h, s = 0.0;
6     for (unsigned int i = 1; i < n-1; t += h, ++i) { s += f(t); }
7     return ( [ ]  * f(a) + [ ]  * f(b) + [ ]  * s );
8 }
```

Supplement the missing C++ code in the boxes.

SOLUTION for (8-11.a) → [8-11-1-0:sa.pdf](#)

**(8-11.b)** (3 min.) The following plots display the quadrature errors for the approximation of the integral

$$\int_0^1 f_1(t) dt , \quad f_1(t) := |\sin(5t)| ,$$

by the three families of quadrature rules and numbers  $n = 4, \dots, 100$  of quadrature points.

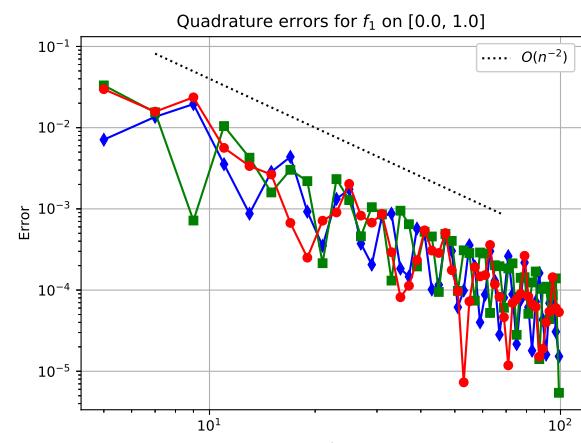
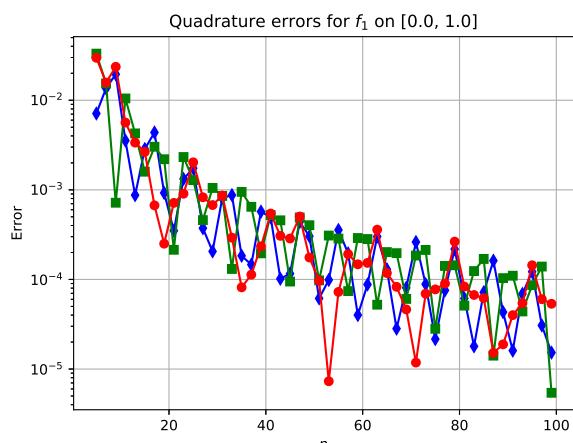
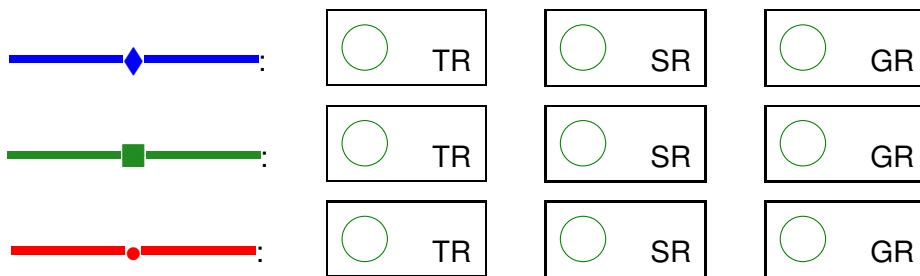


Fig. 47

Fig. 48

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 (8-11.b) → [8-11-2-0:s1.pdf](#)

**(8-11.c)** (3 min.)

Below we have plotted the quadrature errors for the approximation of the integral

$$\int_0^1 f_2(t) dt \quad , \quad f_2(t) := |\sin(5t)|^2 \quad ,$$

for the three families of quadrature rules and numbers  $n = 4, \dots, 100$  of quadrature points.

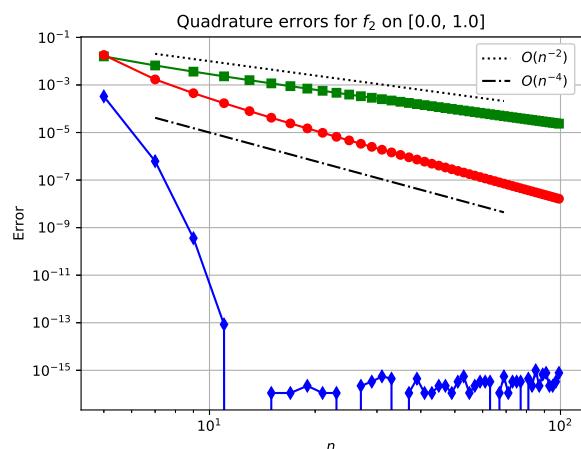
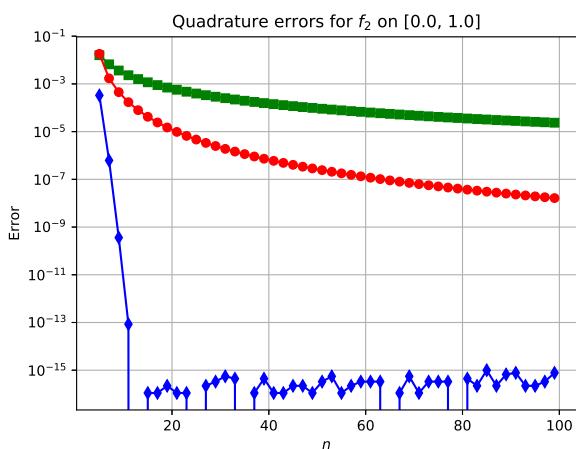
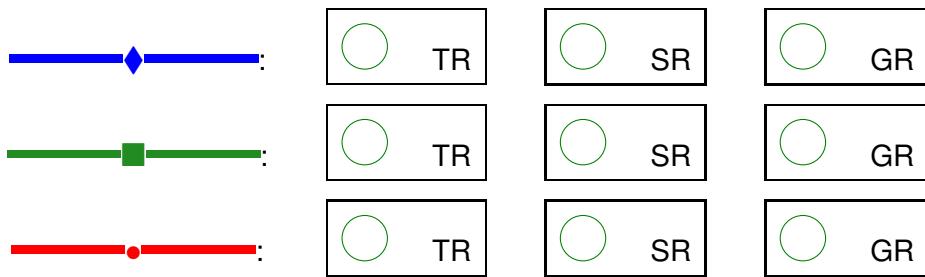


Fig. 51

Fig. 52

Select the quadrature rule, if the curve correctly represents the expected behavior of its quadrature error as a function of  $n$ .



SOLUTION for (8-11.c) → [8-11-3-0:s2.pdf](#)

(8-11.d) (3 min.)

The plots below display the quadrature errors for

$$\int_0^1 f_3(t) dt, \quad f_3(t) := |\sin(5t)|^5,$$

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,  $n = 4, \dots, 100$ .

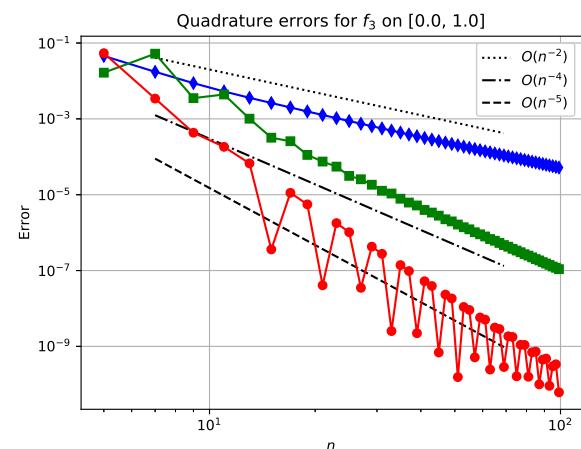
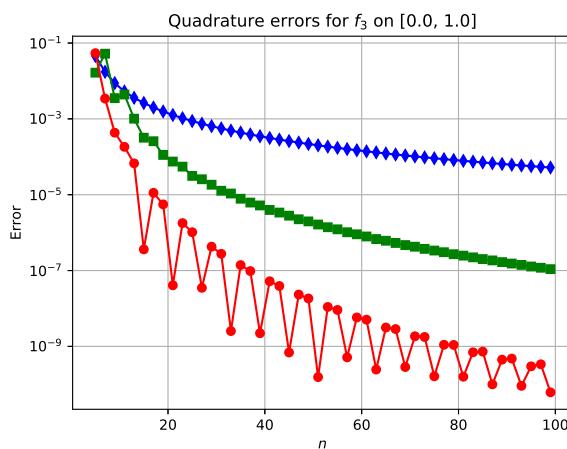
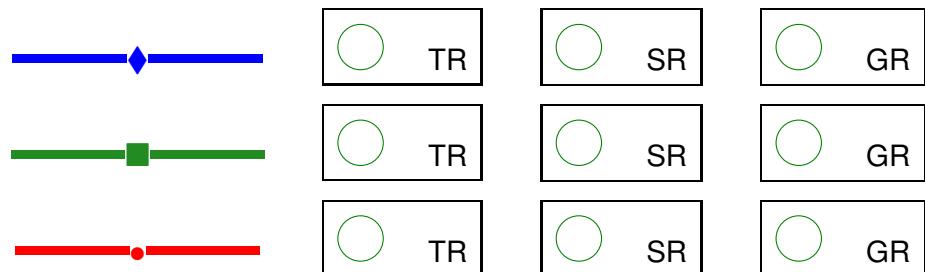


Fig. 55

Fig. 56

Select those quadrature rules, for which the curve correctly represents the expected behavior of its quadrature error as a function of  $n$ .



SOLUTION for (8-11.d) → [8-11-4-0:s3.pdf](#)

(8-11.e) (4 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  $\int_0^1 |\sin(\pi t)| dt$ .

true

false

SOLUTION for (8-11.e) → [8-11-5-0:sx.pdf](#)



**End Problem 8-11 , 16 min.**

# Chapter 9

## Iterative Methods for Non-Linear Systems of Equations

### Problem 9-1: Convergent Newton iteration

As explained in [Lecture → Section 8.3.2.1], 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. 151]: 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.

Give a “graphical proof” of the following statement:

**Theorem 9.1.1. 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 → Eq. (8.3.2.1)] 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 (9-1.) → [9-1-0-0:Conv1s.pdf](#)

**End Problem 9-1 , (25 min.)**

### Problem 9-2: 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 8.3.2.1], involves implementation in C++.

(9-2.a) (20 min.) What is the purpose of the following C++ code?

#### C++ code 9.2.1: Undocumented function.

```

2  double myfunction(double x) {
3      double log2=0.693147180559945;
4      double y=0;
5      while(x>std::sqrt(2)){x/=2; y+=log2;} //
6      while(x<1./std::sqrt(2)){x*=2; y-=log2;} //
7      double z=x-1; //
8      double dz=x*std::exp(-z)-1.0;
9      while(std::abs(dz/z)>std::numeric_limits<double>::epsilon()) {
10         z+=dz; dz=x*std::exp(-z)-1.0;
11     }
12     return y+z+dz; //
13 }
```

HIDDEN HINT 1 for (9-2.a) → [9-2-1-0:Code1ha.pdf](#)

HIDDEN HINT 2 for (9-2.a) → [9-2-1-1:Code1hb.pdf](#)

SOLUTION for (9-2.a) → [9-2-1-2:Code1s.pdf](#)

(9-2.b) (10 min.) Explain the rationale behind the first two `while` loops in the code Code 9.2.1, Lines 5 & 6, preceding the main iteration.

SOLUTION for (9-2.b) → [9-2-2-0:Code2s.pdf](#)

(9-2.c) (10 min.) Explain what the last loop body does.

SOLUTION for (9-2.c) → [9-2-3-0:Code3s.pdf](#)

(9-2.d) (5 min.) Explain the conditional expression in the last `while` loop.

SOLUTION for (9-2.d) → [9-2-4-0:Code4s.pdf](#)

(9-2.e) (40 min.) 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 (9-2.e) → [9-2-5-0:s5h1.pdf](#)

SOLUTION for (9-2.e) → [9-2-5-1:Code5s.pdf](#)

**End Problem 9-2 , 85 min.**

### Problem 9-3: Newton's method for $F(x) := \arctan x = 0$

The merely local convergence of Newton's method is notorious, see [Lecture → Section 8.4.2] and [Lecture → Ex. 8.4.4.1]. 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. 8.4.4.1].

Moderate implementation in C++ is requested.

- (9-3.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 (9-3.a) → [9-3-1-0:NewtonArctan1ha.pdf](#)

HIDDEN HINT 2 for (9-3.a) → [9-3-1-1:NewtonArctan1hb.pdf](#)

SOLUTION for (9-3.a) → [9-3-1-2:NewtonArctan1s.pdf](#) ▲

- (9-3.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 (9-3.a) suggest that we use an initial guess in  $[1, 2]$ , we opt for  $x^{(0)} = 2$ .

SOLUTION for (9-3.b) → [9-3-2-0:NewtonArctan2s.pdf](#) ▲

**End Problem 9-3 , 40 min.**

### Problem 9-4: A derivative-free iterative scheme for finding zeros

[Lecture → Rem. 8.1.1.12] 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 8.1.1], 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

$$x^{(n+1)} := x^{(n)} - \frac{f(x^{(n)})}{g(x^{(n)})}, \quad \text{where} \quad g(x) = \frac{f(x + f(x)) - f(x)}{f(x)}. \quad (9.4.1)$$

(9-4.a) (15 min.) Write a C++ function for the Steffensen's method:

```
template <class Function>
double steffensen(Function &&f, double x0)
```

$f$  is a handle to function  $f$ ,  $x0$  is the initial guess  $x^{(0)}$ . Terminate the iteration when the computed sequence of approximations becomes stationary, see [Lecture → Code 8.1.2.5] for an example.

SOLUTION for (9-4.a) → [9-4-1-0:Quad1s.pdf](#)

(9-4.b) (15 min.) [ depends on Sub-problem (9-4.a) ]

Write a C++ function

```
void testSteffensen(void);
```

that applies your implementation of `steffensen()` to find the zero of the function  $f(x) = xe^x - 1$  (see [Lecture → Exp. 8.2.1.3]). Use  $x^{(0)} = 1$  as initial guess.

SOLUTION for (9-4.b) → [9-4-2-0:sq2.pdf](#)

(9-4.c) (10 min.) [ depends on Sub-problem (9-4.a) ]

Extend your implementation of `steffensen()` to

```
template <typename Function, typename Logger>
VectorXd steffensen_log(Function &&f, double x0, Logger *logger_p
= nullptr);
```

so that it feeds the current iterates to an object of type `Logger` with through operator `(double)`. Then use this new facility to implement a function

```
void orderSteffensen(void);
```

that tabulates values from which you can read off the order of Steffensen's method when applied to the test case of Sub-problem (9-4.b).

HIDDEN HINT 1 for (9-4.c) → [9-4-3-0:stefh1.pdf](#)

SOLUTION for (9-4.c) → [9-4-3-1:sq3.pdf](#)

(9-4.d) (10 min.) For the test case of Sub-problem (9-4.b) the function  $g(x)$  from (9.4.1) contains a term like  $e^{xe^x}$ . Therefore it grows very fast in  $x$  and the method cannot start for a 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 (9-4.d) → [9-4-4-0:Quad2h.pdf](#)

SOLUTION for (9-4.d) → [9-4-4-1:Quad2s.pdf](#)



**End Problem 9-4 , 50 min.**

### Problem 9-5: Order- $p$ convergent iterations

In [Lecture → Section 8.1.1] we investigated the speed of convergence of iterative methods for the solution of a general non-linear problem  $F(\mathbf{x}) = \mathbf{0}$  and introduced the notion of convergence of order  $p \geq 1$ , see [Lecture → Def. 8.1.1.10]. This problem highlights the fact that for  $p > 1$  convergence may not be guaranteed, even if the error norm estimate of [Lecture → Def. 8.1.1.10] 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 with a little C++ coding

Given  $\mathbf{x}^* \in \mathbb{R}^n$ , suppose that a sequence  $\mathbf{x}^{(k)}$  satisfies [Lecture → Def. 8.1.1.10]:

$$\exists C > 0: \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C \|\mathbf{x}^{(k)} - \mathbf{x}^*\|^p \quad \forall k \quad \text{and} \quad p > 1. \quad (9.5.1)$$

(9-5.a) (20 min.) Determine  $\epsilon_0 > 0$  as large as possible such that

$$\|\mathbf{x}^{(0)} - \mathbf{x}^*\| \leq \epsilon_0 \implies \lim_{k \rightarrow \infty} \mathbf{x}^{(k)} = \mathbf{x}^*. \quad (9.5.2)$$

In other words,  $\epsilon_0$  tells us which distance of the initial guess from  $\mathbf{x}^*$  still guarantees **local convergence**.

SOLUTION for (9-5.a) → 9-5-1-0:OrdC1s.pdf ▲

(9-5.b) (15 min.) Provided that  $\|\mathbf{x}^{(0)} - \mathbf{x}^*\| < \epsilon_0$  is satisfied with  $\epsilon_0$  from Sub-problem (9-5.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 (9-5.b) → 9-5-2-0:OrdC2s.pdf ▲

(9-5.c) (15 min.) Implement a C++ function

```
void kminplot(void);
```

that uses **MATPLOTLIBCPP** to created a surface plot of  $k_{\min}(\epsilon_0, \tau)$  for the values  $p = 1.5$ ,  $C = 2$  and  $(\epsilon_0, \tau) \in \text{linspace}\left([0, C^{\frac{1}{1-p}}]^2\right)$ .

SOLUTION for (9-5.c) → 9-5-3-0:OrdC3s.pdf ▲

**End Problem 9-5 , 50 min.**

### Problem 9-6: Order of convergence from error recursion

In [Lecture → Exp. 8.3.2.25] we have observed *fractional* orders of convergence ([Lecture → Def. 8.1.1.10]) 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. 8.3.2.26], where a linearised error recursion is given in [Lecture → Eq. (8.3.2.30)].

For an iterative scheme producing the sequence  $(x^{(n)})_{n \in \mathbb{N}}$ ,  $x^{(n)} \in \mathbb{R}$  we suppose a recursive bound for the norms of the iteration errors of the form

$$\|e^{(n+1)}\| \leq \|e^{(n)}\| \sqrt{\|e^{(n-1)}\|}, \quad (9.6.1)$$

where  $e^{(n)} = x^{(n)} - x^*$  is the error of  $n$ -th iterate.

**(9-6.a)** (20 min.) Write C++ function

```
double testOrder(void);
```

that guesses the maximal order of convergence of the method from a numerical experiment.

HIDDEN HINT 1 for (9-6.a) → [9-6-1-0:RecursionOrder1h.pdf](#)

SOLUTION for (9-6.a) → [9-6-1-1:RecursionOrder1s.pdf](#)

**(9-6.b)** (30 min.) Find the maximal guaranteed order of convergence of this method by mathematical analysis as in [Lecture → Rem. 8.3.2.26].

HIDDEN HINT 1 for (9-6.b) → [9-6-2-0:RecursionOrder2ha.pdf](#)

HIDDEN HINT 2 for (9-6.b) → [9-6-2-1:RecursionOrder2hb.pdf](#)

HIDDEN HINT 3 for (9-6.b) → [9-6-2-2:RecursionOrder2hc.pdf](#)

SOLUTION for (9-6.b) → [9-6-2-3:RecursionOrder2s.pdf](#)

**End Problem 9-6**, 50 min.

### Problem 9-7: 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. 8.0.0.1]. 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 through a diode as a function of the applied voltage can be modelled by the relationship  $I_{kj} = \alpha \left( e^{\beta \frac{U_k - U_j}{U_T}} - 1 \right)$ , with suitable parameters  $\alpha, \beta$  and the thermal voltage  $U_T$ .

Now we consider the circuit depicted in Fig. 64 and assume that all resistors have resistance  $R = 1$ .

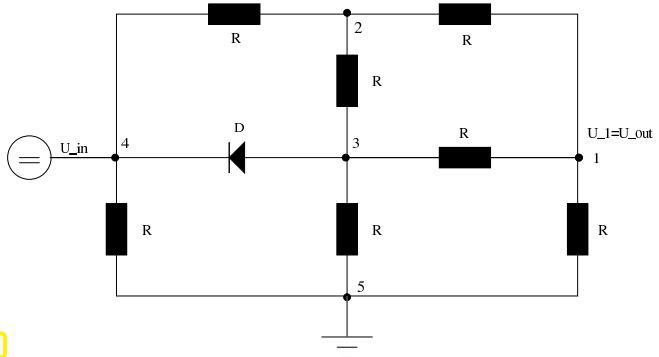


Fig. 64

- (9-7.a) ☐ 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 voltages in nodes 1, 2 and 3, cf. [Lecture → Eq. (8.0.0.2)]. Note that the voltages in nodes 4 and 5 are known (input voltage and ground voltage 0).

SOLUTION for (9-7.a) → [9-7-1-0:NonL1s.pdf](#)

- (9-7.b) ☐ Write an EIGEN-based C++ function

```
void circuit(const double & alpha, const double & beta,
            const VectorXd & Uin, VectorXd & Uout)
```

that computes the output voltages  $U_{\text{out}}$  (at node 1 in Fig. 64) for a *sorted* vector of input voltages  $U_{\text{in}}$  (at node 4) and for a thermal voltage  $U_T = 0.5$ . The parameters  $\alpha, \beta$  pass the (non-dimensional) diode parameters.

Use Newton's method to solve  $\mathbf{F}(\mathbf{u}) = \mathbf{0}$  with a relative tolerance of  $\tau = 10^{-6}$ .

SOLUTION for (9-7.b) → [9-7-2-0:NonL2s.pdf](#)

- (9-7.c) ☐ We are interested in the nonlinear effects introduced by the diode. Implement a C++ function

```
void plotU(void);
```

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 (9-7.c) → [9-7-3-0:NonL3s.pdf](#)

**End Problem 9-7**

### Problem 9-8: 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 8.4]

In the space  $\mathbb{C}$  of complex numbers the equation

$$z^3 = 1 \quad (9.8.1)$$

has three solutions:

$$z_1 = 1 , \quad z_2 = -\frac{1}{2} + \frac{1}{2}\sqrt{3}i , \quad z_3 = -\frac{1}{2} - \frac{1}{2}\sqrt{3}i , \quad (9.8.2)$$

the cubic roots of unity.

**(9-8.a)** (10 min.) As you know from the analysis course, the complex plane  $\mathbb{C}$  can be identified with  $\mathbb{R}^2$  via  $(x, y) \mapsto z = x + iy$ . Using this identification, convert Eq. (9.8.1) 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 (9-8.a) → [9-8-1-0:JuliaSet1s.pdf](#) ▲

**(9-8.b)** (15 min.) [ depends on Sub-problem (9-8.a) ]

Formulate Newton iteration [Lecture → Eq. (8.4.1.1)] for the non-linear equation  $\mathbf{F}(\mathbf{x}) = \mathbf{0}$  with  $\mathbf{x} = (x, y)^T$  and  $\mathbf{F}$  from Sub-problem (9-8.a).

SOLUTION for (9-8.b) → [9-8-2-0:JuliaSet2s.pdf](#) ▲

**(9-8.c)** (10 min.) [ depends on Sub-problem (9-8.a) ]

Implement two C++ functions

```
Eigen::Vector2d F(const Eigen::Vector2d& x);
Eigen::Matrix2d DF(const Eigen::Vector2d& x);
```

that return  $F(\mathbf{x})$  and the Jacobian  $D F(\mathbf{x})$  for the function  $F: \mathbb{R}^2 \rightarrow \mathbb{R}^2$  found in Sub-problem (9-8.a).

SOLUTION for (9-8.c) → [9-8-3-0:sa1.pdf](#) ▲

**(9-8.d)** (30 min.) [ depends on Sub-problem (9-8.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.

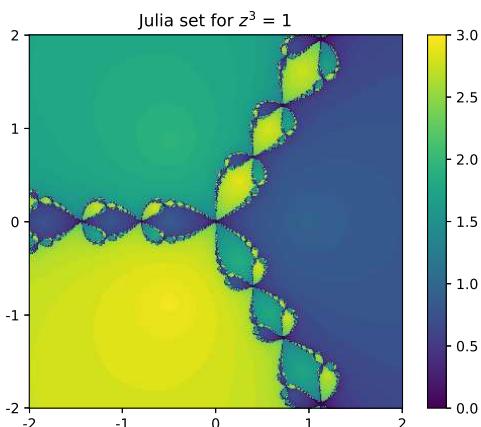
Analyze the behavior of Newton iterations by means of a C++ code using the following approach:

- 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(void);
```

supplied in `julia.hpp`. It creates and saves a plot in `julia.png`.



▷ The Julia set for  $z^3 - 1 = 0$  on a mesh containing  $780 \times 780$  points,  $N_{\text{it}}=20$ .

This is how your final plot should look like.

Fig. 66

HIDDEN HINT 1 for (9-8.d) → [9-8-4-0:JuliaSet3h.pdf](#)

SOLUTION for (9-8.d) → [9-8-4-1:JuliaSet3s.pdf](#)



**End Problem 9-8 , 65 min.**

### Problem 9-9: Modified Newton method

The following problem consists in EIGEN implementation of a modified version of the Newton method (in one dimension [Lecture → Section 8.3.2.1] and many dimensions [Lecture → Section 8.4]) for the solution of a nonlinear system.

Involves coding in C++. Refresh your knowledge of stopping criteria for iterative methods [Lecture → Section 8.1.2].

For the solution of the non-linear system of equations  $\mathbf{F}(\mathbf{x}) = \mathbf{0}$  (with  $\mathbf{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ ), the following iterative method has been proposed:

$$\begin{aligned}\mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + D\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{x}^{(k)}), \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - D\mathbf{F}(\mathbf{x}^{(k)})^{-1} \mathbf{F}(\mathbf{y}^{(k)}),\end{aligned}\tag{9.9.1}$$

where  $D\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$  is the Jacobian matrix of  $\mathbf{F}$  evaluated in the point  $\mathbf{x}$ .

**(9-9.a)**  (15 min.) Show that the iteration is *consistent* with  $\mathbf{F}(\mathbf{x}) = \mathbf{0}$  in the sense of [Lecture → Def. 8.2.1.1], that is, show that  $\mathbf{x}^{(k)} = \mathbf{x}^{(0)}$  for every  $k \in \mathbb{N}$  if and only if  $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$  and  $D\mathbf{F}(\mathbf{x}^{(0)})$  is regular.

SOLUTION for (9-9.a) → [9-9-1-0:Mod1s.pdf](#)

**(9-9.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 (9-9.b) → [9-9-2-0:Modi2s.pdf](#)

**(9-9.c)**  (15 min.) What is the order of convergence of the method described in (9.9.1)?

To investigate it, write a C++ function

```
void mod_newt_ord(void);
```

that

- uses the function `mod_newt_step` to the following scalar equation:  $\arctan(x) - 0.123 = 0$ ,
- determines empirically the order of convergence, in the sense of [Lecture → Rem. 8.1.1.12],
- implements meaningful stopping criteria, see [Lecture → Section 8.1.2].

Use  $x_0 = 5$  as initial guess and tabulate data from which you can read off the order.

HIDDEN HINT 1 for (9-9.c) → [9-9-3-0:Modi3ha.pdf](#)

HIDDEN HINT 2 for (9-9.c) → [9-9-3-1:Modi3hb.pdf](#)

SOLUTION for (9-9.c) → [9-9-3-2:Modi3s.pdf](#)

(9-9.d) (15 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 (9.9.1) for a function  $\mathbf{F}: \mathbb{R}^n \rightarrow \mathbb{R}^n$  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 (9-9.d) → [9-9-4-0:3xh1.pdf](#)

SOLUTION for (9-9.d) → [9-9-4-1:s3x.pdf](#) ▲

In the sequel we consider the non-linear system of equations

$$\mathbf{F}(\mathbf{x}) := \mathbf{Ax} + \begin{bmatrix} c_1 e^{x_1} \\ \vdots \\ c_n e^{x_n} \end{bmatrix} = \mathbf{0}, \quad (9.9.5)$$

where  $\mathbf{A} \in \mathbb{R}^{n,n}$  is symmetric positive definite,  $\mathbf{x} = [x_j]_{j=1}^n \in \mathbb{R}^n$  is the solution vector, and  $c_i \geq 0$ ,  $i = 1, \dots, n$ .

(9-9.e) Compute the Jacobian matrix  $D\mathbf{F}(\mathbf{x})$  for  $\mathbf{F}$  as given in (9.9.5).

SOLUTION for (9-9.e) → [9-9-5-0:s4x.pdf](#) ▲

(9-9.f) (20 min.) [ depends on Sub-problem (9-9.d) ]

Based on your implementation of `mod_newt_step_system()`, create a C++ function

```
template <typename Vector, typename Matrix>
Vector mod_newt_sys(const Matrix &A,
                    const Vector &c,
                    double tol 1.0E-6, int maxit = 100);
```

that uses the modified Newton iteration (9.9.1) to solve (9.9.5). The argument  $A$  passes the matrix  $\mathbf{A}$  and  $c$  supplies the values  $c_j$ .

The type `Vector` can be expected to be compatible with `Eigen::VectorXd`, whereas `Matrix` can be thought of as `Eigen::MatrixXd`.

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 (9-9.f) → [9-9-6-0:s4m.pdf](#) ▲

(9-9.g) (20 min.) [ depends on Sub-problem (9-9.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 (9-9.g) → [9-9-7-0:s4h1.pdf](#)

SOLUTION for (9-9.g) → [9-9-7-1:Modi4s.pdf](#)



**End Problem 9-9 , 105 min.**

### Problem 9-10: Solving a quasi-linear system

In [Lecture → § 8.4.1.18] we studied Newton's method for a so-called quasi-linear system of equations, see [Lecture → Eq. (8.4.1.19)]. In [Lecture → Ex. 8.4.1.23] 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 8.2].

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:

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}, \quad (9.10.1)$$

as in [Lecture → Ex. 8.4.1.23]. Here,  $\mathbf{A} : \mathbb{R}^n \rightarrow \mathbb{R}^{n,n}$  is a matrix-valued function:

$$\mathbf{A}(\mathbf{x}) := \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & \\ 1 & \gamma(\mathbf{x}) & 1 & \\ & \ddots & \ddots & \ddots \\ & & \ddots & \ddots & \ddots \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \end{bmatrix}, \quad \gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2 \quad (9.10.2)$$

where  $\|\cdot\|_2$  is the Euclidean norm.

**(9-10.a)** □ (10 min.) A fixed point iteration can be obtained from (9.10.1) 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 (9-10.a) → [9-10-1-0:s1.pdf](#)

**(9-10.b)** □ (10 min.) [ depends on Sub-problem (9-10.a) ]

We consider the fixed point iteration derived in Sub-problem (9-10.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 (9-10.b) → [9-10-2-0:q2h1.pdf](#)

SOLUTION for (9-10.b) → [9-10-2-1:QuasiLinear2s.pdf](#) ▲

**(9-10.c)** □ (10 min.) Write a C++ function

```
Eigen::VectorXd solveQuasiLinSystem(double rtol, double atol
                                       const Eigen::VectorXd &b);
```

that finds the solution  $\mathbf{x}^*$  of (9.10.1), (9.10.2) 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 8.1.2] and pass absolute and relative tolerance as arguments *atol* and *rtol*.

SOLUTION for (9-10.c) → [9-10-3-0:QuasiLinear3s.pdf](#) ▲

(9-10.d)  (10 min.) Let  $\mathbf{b} \in \mathbb{R}^n$  be given. Write the recursion formula for the solution of

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \quad (9.10.6)$$

with the Newton method.

SOLUTION for (9-10.d) → [9-10-4-0:QuasiLinear4s.pdf](#) ▲

(9-10.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 (9-10.d) exploiting the Sherman-Morrison-Woodbury inversion formula for rank-one modifications from [Lecture → Lemma 2.6.0.21].

SOLUTION for (9-10.e) → [9-10-5-0:QuasiLinear5s.pdf](#) ▲

(9-10.f)  (30 min.) [ depends on Sub-problem (9-10.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 (9-10.e).

HIDDEN HINT 1 for (9-10.f) → [9-10-6-0:QuasiLinear6ha.pdf](#)

SOLUTION for (9-10.f) → [9-10-6-1:QuasiLinear6s.pdf](#) ▲

(9-10.g)  (10 min.) [ depends on Sub-problem (9-10.f) ]

Repeat Sub-problem (9-10.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 (9.10.1), (9.10.2) 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 8.4.3].

SOLUTION for (9-10.g) → [9-10-7-0:QuasiLinear7s.pdf](#) ▲

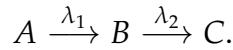
**End Problem 9-10 , 95 min.**

### Problem 9-11: Radioactive Decay

This exercise studies a (non-linear) least-squares parameter estimation problem.

Requires [Lecture → Section 8.5]

For radioactive substances  $A$  and  $B$  consider the decay chain



That is  $A$  decays into  $B$  with a decay rate  $\lambda_1$ .  $B$  itself decays into a third substance  $C$  with decay rate  $\lambda_2$ . 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 [**STRLN09**]:

$$\begin{aligned}\dot{\Phi}_A(t) &= -\lambda_1 \Phi_A(t) \\ \dot{\Phi}_B(t) &= -\lambda_2 \Phi_B(t) + \lambda_1 \Phi_A(t).\end{aligned}\tag{9.11.1}$$

**(9-11.a)**  Calculate the analytical solution of the system (9.11.1) using the method of variation of constants, see [**Struwe**].

You are encouraged to use Maple (`dsolve`), Mathematica (`DSolve`), or Maxima (`desolve`), which should always be used when routine, but tedious and error prone computations have to be carried out.

Assume general initial values  $\Phi_A(0) = A_0, \Phi_B(0) = B_0$  and verify your solution by plugging it into the system (9.11.1).

SOLUTION for (9-11.a) → [9-11-1-0:rd1.pdf](#)



**(9-11.b)**  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  $\lambda_1$  and  $\lambda_2$ . 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 8.5]. Here, the vector  $\mathbf{x}$  of unknown parameters is  $(A_0, B_0, \lambda_1, \lambda_2)^T$ .

The measurements  $\{t_i, m_i\}, i \in \{1, \dots, N\}$ , are stored in the file `decay.txt`:  $t_i$  are the times of measurements and  $m_i$  are the measured values of  $\Phi_B$ .

Write down the function  $\mathbf{F}$  for the current problem.

HIDDEN HINT 1 for (9-11.b) → [9-11-2-0:Radioactive3h.pdf](#)

SOLUTION for (9-11.b) → [9-11-2-1:rd2.pdf](#)



**(9-11.c)**  We want to use the Gauss-Newton method (see [Lecture → Section 8.5.2]) for solving the non-linear problem described above.

Write down the concrete iteration formulas [Lecture → Eq. (8.5.2.1)] 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 (9-11.c) → [9-11-3-0:rd3.pdf](#)



**(9-11.d)**  Implement the Gauss-Newton method for this problem in C++. Use the data provided in `decay.txt` and find an estimate for the initial quantities  $A_0, B_0$  and the decay rates  $\lambda_1, \lambda_2$ .

Which kind of convergence do you observe? Since you don't know the exact parameters, how can you check if your code works properly?

HIDDEN HINT 1 for (9-11.d) → [9-11-4-0:Radioactive4h.pdf](#)

HIDDEN HINT 2 for (9-11.d) → [9-11-4-1:Radioactive4h.pdf](#)

HIDDEN HINT 3 for (9-11.d) → [9-11-4-2:Radioactive4h.pdf](#)

SOLUTION for (9-11.d) → [9-11-4-3:rd4.pdf](#)



**End Problem 9-11**

### Problem 9-12: Approximation of a circle

In this problem we study how a fitting problem arising in computational geometry can be solved using least squares in several ways, leading to different results.

Requires [Lecture → Section 8.5]

Let us consider a sequence of  $N$  points 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  $(m_1, m_2)$  and radius  $r$  is described by

$$(x - m_1)^2 + (y - m_2)^2 = r^2. \quad (9.12.1)$$

#### 9-12.I: linear algebraic fit

**(9-12.a)**  Plugging the point coordinates  $(x_i, y_i)$  into (9.12.1) 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 → Eq. (3.1.3.7)].

SOLUTION for (9-12.a) → [9-12-1-0:cala1.pdf](#)

**(9-12.b)**

Write a C++ function

```
Vector3d circl_alg_fit(const VectorXd &x, const VectorXd & y);
```

that receives point coordinates in the vectors  $\mathbf{x}$  and  $\mathbf{y}$  and solves the overdetermined system in least squares sense and returns a 3-dimensional vector  $(m_1, m_2, r)$ .

SOLUTION for (9-12.b) → [9-12-2-0:cala2.pdf](#)

#### 9-12.II: geometric fit

The algebraic approach lacks an intuitive geometrical meaning: minimising the equation residual of (9.12.1) 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

$$d_i = \left| \sqrt{(m_1 - x_i)^2 + (m_2 - y_i)^2} - r \right| \quad i = 1, \dots, N,$$

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

$$\mathbf{z}^* = \underset{\mathbf{z}}{\operatorname{argmin}} \|\mathbf{F}(\mathbf{z})\|^2,$$

see [Lecture → Section 8.5].

**(9-12.c)**  Write down the concrete function  $\mathbf{F} : \mathbb{R}^{n_1} \rightarrow \mathbb{R}^{n_2}$  for this non-linear least squares problem.

You can use the auxiliary values

$$R_i = \sqrt{(x_i - m_1)^2 + (y_i - m_2)^2}, \quad i = 1, \dots, N.$$

SOLUTION for (9-12.c) → [9-12-3-0:cage1.pdf](#) ▲

**(9-12.d)** ☐

Define the functional

$$\Phi(\mathbf{z}) := \frac{1}{2} \|\mathbf{F}(\mathbf{z})\|^2.$$

Compute the Jacobian  $\mathbf{DF}$  of  $\mathbf{F}$ , the gradient  $\mathbf{grad} \Phi(\mathbf{z})$  of  $\Phi$  and the Hessian  $\mathbf{H}\Phi(\mathbf{z})$ .

SOLUTION for (9-12.d) → [9-12-4-0:cage2.pdf](#) ▲

**(9-12.e)** ☐ Use C++ 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

```
Vector3d circl_geo_fit(const VectorXd &x, const VectorXd & y);
```

that uses the Gauss-Newton method introduced in [Lecture → Section 8.5.2] to minimize the functional  $\Phi$ .

SOLUTION for (9-12.e) → [9-12-5-0:cage3.pdf](#) ▲

**(9-12.f)** ☐ Now solve the same problem using the Newton method for least squares equations as described in [Lecture → Section 8.5.1].

SOLUTION for (9-12.f) → [9-12-6-0:cage4.pdf](#) ▲

**(9-12.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.

Measure the error in the parameters in the maximum norm.

SOLUTION for (9-12.g) → [9-12-7-0:cage5.pdf](#) ▲

### 9-12.III: constrained fit using SVD

As you may know, for  $a \neq 0$  the solution set of the quadratic equation

$$a\mathbf{x}^T\mathbf{x} + \mathbf{b}^T\mathbf{x} + c = 0, \quad \mathbf{b} \in \mathbb{R}^2, a, c \in \mathbb{R}, \quad (9.12.7)$$

(solved with respect to  $\mathbf{x} \in \mathbb{R}^2$ ) describes a circle.

**(9-12.h)** ☐ Derive an expression in terms of  $a, \mathbf{b}, c$  for the center  $\mathbf{m}$  and the radius  $r$  of the circle defined by (9.12.7).

SOLUTION for (9-12.h) → [9-12-8-0:cavsd1.pdf](#) ▲

**(9-12.i)** ☐ According to equation (9.12.7), 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:

$$a\mathbf{x}_i^T\mathbf{x}_i + \mathbf{b}^T\mathbf{x}_i + c = 0 \quad i = 1, \dots, N, \quad \|(a, b_1, b_2, c)^T\|_2^2 = 1.$$

Write a MATLAB function

```
[m, r] = circ_svd_fit(x, y)
```

that solves this constrained overdetermined linear system of equations in least squares sense. Use the data in the table from the previous subtasks.

To learn how to use SVD to solve this problem, study carefully the hyperplane fitting problem [Lecture → Ex. 3.4.4.5] and [Lecture → Eq. (3.4.4.1)].

SOLUTION for (9-12.i) → [9-12-9-0:casvd2.pdf](#) ▲

#### 9-12.IV: comparison of the results

(9-12.j)  Draw the data points and the fitted circles computed in the previous subtasks. Compare the centers and the radii.

SOLUTION for (9-12.j) → [9-12-10-0:caco.pdf](#) ▲

**End Problem 9-12**

### Problem 9-13: Computing a Level Set

For a real-valued function  $f : D \subset \mathbb{R}^d \rightarrow \mathbb{R}$  the level sets  $\mathcal{L}_c := \{x \in D : f(x) \leq c\}$ ,  $c \in \mathbb{R}$ , 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 8.3].

Throughout this problem let  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$  be a continuously differentiable *strictly convex* function which has a *global minimum* in  $x = \mathbf{0}$  and satisfies the growth condition  $|f(x)| \rightarrow \infty$  uniformly for  $\|x\| \rightarrow \infty$

Then the level sets

$$\mathcal{L}_c := \{x \in \mathbb{R}^2 : f(x) \leq c\}, \quad c > 0, \quad (9.13.1)$$

will be closed and convex subsets of  $\mathbb{R}^2$ , and every ray

$$R_d := \{\xi d : \xi \geq 0\}, \quad d \in \mathbb{R}^2 \setminus \{0\}, \quad (9.13.2)$$

will intersect the boundary of  $\mathcal{L}_c$ ,

$$\partial\mathcal{L}_c := \{x \in \mathbb{R}^2 : f(x) = c\}, \quad (9.13.3)$$

in exactly one point.

**(9-13.a)** (15 min.) Based on point evaluations of  $f$  and  $\text{grad } f$  state the **Newton iteration** for the *scalar* non-linear equation that has to be solved in order to find the intersection of the ray  $R_d$  and  $\partial\mathcal{L}_c$ . The vector  $d \in \mathbb{R}^2 \setminus \{0\}$  and the number  $c > 0$  should be regarded as parameters.

SOLUTION for (9-13.a) → [9-13-1-0:s1.pdf](#) ▲

**(9-13.b)** Now, write down the recursion for the **secant method** [Lecture → § 8.3.2.21] for solving the *scalar* non-linear equation tackled in Sub-problem (9-13.a). Your formula should involve only  $f$ -evaluations and  $d \in \mathbb{R}^2 \setminus \{0\}$  and  $c > 0$  as parameters.

SOLUTION for (9-13.b) → [9-13-2-0:s3.pdf](#) ▲

**(9-13.c)** (15 min.) [ depends on Sub-problem (9-13.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  $R_d$  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  $x_0$  passes (a guesses for) the coordinates of a point  $x_0 \in \partial\mathcal{L}_c$ . This information may be used to obtain initial guesses for the secant iteration, which should start from the points  $\frac{\|x_0\|_2}{\|d\|_2}d$  and  $1.1 \cdot \frac{\|x_0\|_2}{\|d\|_2}d$ .

Employ a correction-based termination criterion with relative tolerance and absolute tolerance supplied by the arguments  $rtol$  and  $atol$ .

SOLUTION for (9-13.c) → [9-13-3-0:s4.pdf](#) ▲

(9-13.d) (20 min.) [ depends on Sub-problem (9-13.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 ( $n > 2$ )

$$\mathbf{p}_j \in \mathbb{R}^2: \quad \{\mathbf{p}_j\} = R_{\mathbf{d}_j} \cap \partial \mathcal{L}_c, \quad \mathbf{d}_j = \begin{bmatrix} \cos(2\pi j/n) \\ \sin(2\pi j/n) \end{bmatrix}, \quad j = 0, \dots, n-1.$$

HIDDEN HINT 1 for (9-13.d) → 9-13-4-0:hcp.pdf

HIDDEN HINT 2 for (9-13.d) → 9-13-4-1:hcp2.pdf

SOLUTION for (9-13.d) → 9-13-4-2:s4a.pdf



(9-13.e) (10 min.) We consider

$$f(\mathbf{x}) = x_1^2 + 2x_2^4, \quad \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \in \mathbb{R}^2.$$

<i>n</i>	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  $n \rightarrow \infty$ , 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  $\frac{1}{n} \rightarrow 0$ .

SOLUTION for (9-13.e) → 9-13-5-0:s5.pdf



**End Problem 9-13 , 60 min.**

### Problem 9-14: 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 problem is based on [Lecture → Section 8.5], in particular [Lecture → Section 8.5.2], 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. 3.4.4.16]:

$$\mathbf{x} \in \operatorname{argmin}_{\mathbf{z} \in \mathbb{R}^n} \left\| \mathbf{M} - \mathbf{z}\mathbf{z}^\top \right\|_F = \operatorname{argmin}_{\mathbf{z} \in \mathbb{R}^n} \|\Phi(\mathbf{z})\|_2, \quad (9.14.1)$$

$$\text{with } \Phi : \mathbb{R}^n \rightarrow \mathbb{R}^{n^2}, \quad \Phi(\mathbf{x}) := \operatorname{vec}(\mathbf{M}) - \mathbf{x} \otimes \mathbf{x}, \quad \mathbf{x} \in \mathbb{R}^n, \quad (9.14.2)$$

where  $\|\cdot\|_2$  denotes the Euclidean vector norm,  $\operatorname{vec}$  the vectorization of matrices

$$\operatorname{vec} : \mathbb{R}^{n,m} \rightarrow \mathbb{R}^{n \cdot m}, \quad \operatorname{vec}(\mathbf{A}) := \begin{bmatrix} (\mathbf{A})_{:,1} \\ (\mathbf{A})_{:,2} \\ \vdots \\ (\mathbf{A})_{:,m} \end{bmatrix} \in \mathbb{R}^{n \cdot m}, \quad [\text{Lecture} \rightarrow \text{Eq. (1.2.3.5)}]$$

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].

**(9-14.a)** (18 min.)

In the file `symrank1.hpp` implement an EIGEN-based C++ function

```
Eigen::VectorXd symRankOneBestApproxSym(const Eigen::MatrixXd &M);
```

that solves (9.14.1) for a *symmetric* matrix  $\mathbf{M}$ , that is, provided that  $\mathbf{M}^\top = \mathbf{M}$ .

HIDDEN HINT 1 for (9-14.a) → 9-14-1-0:s1h1.pdf

SOLUTION for (9-14.a) → 9-14-1-1:s1.pdf

For general  $\mathbf{M}$  we intend to solve (9.14.1) by means of the **Gauss-Newton method**, that is, by means of the iteration [Lecture → Eq. (8.5.2.1)]

$$\mathbf{x}^{(k+1)} := \mathbf{x}^{(k)} + \operatorname{argmin}_{\mathbf{h} \in \mathbb{R}^n} \left\| \Phi(\mathbf{x}^{(k)}) + D\Phi(\mathbf{x}^{(k)})\mathbf{h} \right\|_2, \quad (9.14.4)$$

with  $\Phi$  from (9.14.2).

**(9-14.b)** (15 min.)

Give an expression for the linear mapping  $\mathbf{h} \in \mathbb{R}^n \mapsto D\Phi(\mathbf{x})\mathbf{h}$ ,  $\mathbf{x} \in \mathbb{R}^n$  fixed.

HIDDEN HINT 1 for (9-14.b) → 9-14-2-0:kpbl.pdf

SOLUTION for (9-14.b) → 9-14-2-1:s2a.pdf

**(9-14.c)** (15 min.) [ depends on Sub-problem (9-14.b) ]

Derive a formula for the Jacobian  $D\Phi(\mathbf{x}) \in \mathbb{R}^{n^2, n}$  of  $\Phi$  as defined in (9.14.2).

Write down  $D\Phi(\mathbf{x})$  explicitly for  $n = 3$ , abbreviating  $x_i = (\mathbf{x})_i$ ,  $i = 1, 2, 3$ .

HIDDEN HINT 1 for (9-14.c) → 9-14-3-0:handy2.pdf

SOLUTION for (9-14.c) → 9-14-3-1:s2b.pdf

(9-14.d) (20 min.) [ depends on Sub-problem (9-14.c) ]

The minimization problem in (9.14.4) involves a linear least-squares problem of the form

$$\Delta \mathbf{x} := \underset{\mathbf{h} \in \mathbb{R}^n}{\operatorname{argmin}} \|\mathbf{A}\mathbf{h} - \mathbf{b}\|_2, \quad (9.14.11)$$

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 (9.14.11).

HIDDEN HINT 1 for (9-14.d) → 9-14-4-0:3hI.pdf

SOLUTION for (9-14.d) → 9-14-4-1:s4.pdf

(9-14.e) (10 min.) [ depends on Sub-problem (9-14.d) ]

Give a simple explicit formula for the inverse  $\mathbf{T}^{-1}$  of the system matrix of the normal equations.

HIDDEN HINT 1 for (9-14.e) → 9-14-5-0:smw.pdf

HIDDEN HINT 2 for (9-14.e) → 9-14-5-1:fails.pdf

SOLUTION for (9-14.e) → 9-14-5-2:s5.pdf

(9-14.f) (20 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

$$(\mathbf{v}^\top \otimes \mathbf{I}_n + \mathbf{I}_n \otimes \mathbf{v}^\top) \mathbf{b} \quad \text{for } \mathbf{v} \in \mathbb{R}^n, \quad \mathbf{b} \in \mathbb{R}^{n^2}, \quad (9.14.14)$$

for any  $n \in \mathbb{N}$ .

HIDDEN HINT 1 for (9-14.f) → 9-14-6-0:hrs.pdf

SOLUTION for (9-14.f) → 9-14-6-1:sx.pdf

(9-14.g) (30 min.) [ depends on Sub-problem (9-14.f) and Sub-problem (9-14.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** (9.14.4) for solving (9.14.1) 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 `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 (9-14.g) → [9-14-7-0:mcgn.pdf](#)

HIDDEN HINT 2 for (9-14.g) → [9-14-7-1:mcgn2.pdf](#)

HIDDEN HINT 3 for (9-14.g) → [9-14-7-2:fail2.pdf](#)

SOLUTION for (9-14.g) → [9-14-7-3:s6.pdf](#)



**End Problem 9-14 , 128 min.**

# **Chapter 10**

## **Computation of Eigenvalues and Eigenvectors**

# **Chapter 11**

## **Krylov Methods for Linear Systems of Equations**

# Chapter 12

## Numerical Integration – Single Step Methods

### Problem 12-1: 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 12-5.

Mainly relies on the information contained in [Lecture → Section 11.2].

We consider the *linear* matrix differential equation

$$\dot{\mathbf{Y}} = \mathbf{AY} =: \mathbf{f}(\mathbf{Y}) \quad \text{with} \quad \mathbf{A} \in \mathbb{R}^{n \times n}. \quad (12.1.1)$$

whose solutions are *matrix-valued functions*  $\mathbf{Y} : \mathbb{R} \rightarrow \mathbb{R}^{n \times n}$ .

(12-1.a) (10 min.) Show that for *skew-symmetric*  $\mathbf{A}$ , i.e.  $\mathbf{A} = -\mathbf{A}^\top$  we have:

$$\mathbf{Y}(0) \text{ orthogonal} \implies \mathbf{Y}(t) \text{ orthogonal } \forall t.$$

HIDDEN HINT 1 for (12-1.a) → 12-1-1-0:Mat01h1.pdf

HIDDEN HINT 2 for (12-1.a) → 12-1-1-1:Mat01h2.pdf

SOLUTION for (12-1.a) → 12-1-1-2:Mat01s.pdf

(12-1.b) (20 min.) Implement three C++ functions

1. a single step of the *explicit Euler method*, see [Lecture → Section 11.2.1]:

```
Eigen::MatrixXd eulstep(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 (12.1.1)

HIDDEN HINT 1 for (12-1.b) → [12-1-2-0:Mat02h.pdf](#)

SOLUTION for (12-1.b) → [12-1-2-1:Mat02s.pdf](#) ▲

(12-1.c) ☐ (30 min.) [ depends on Sub-problem (12-1.b) ]

Investigate numerically, which one of the implemented methods preserves orthogonality for the ODE (12.1.1) 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(void);
```

that also returns the Frobenius norms  $\mathbf{Y}_N^\top \mathbf{Y}_N - \mathbf{I}$  for each of the three methods.

SOLUTION for (12-1.c) → [12-1-3-0:Mat03s.pdf](#) ▲

**End Problem 12-1 , 60 min.**

### Problem 12-2: 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. 11.4.0.9]. They are usually described by giving their coefficients in the form of a Butcher scheme [Lecture → Eq. (11.4.0.11)].

Related to [Lecture → Section 11.4], [Lecture → Def. 11.4.0.9], and Butcher schemes as defined in [Lecture → Eq. (11.4.0.11)]

(12-2.a) (30 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. 11.4.0.9]) given by a Butcher scheme [Lecture → Eq. (11.4.0.11)] to solve the autonomous initial-value problem  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ,  $\mathbf{y}(t_0) = \mathbf{y}_0$ . The matrix  $A \in \mathbb{R}^{s,s}$  and the vector  $b \in \mathbb{R}^s$  from the Butcher scheme are passed to the constructor, whereas the right-hand side vector field  $f$ , the final time  $T > 0$ , the number  $N$  of *equidistant timesteps*, and the initial state  $y_0$  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 (12-2.a) → [12-2-1-0:RK3P1s.pdf](#)

(12-2.b) (30 min.) [ depends on Sub-problem (12-2.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. 11.1.1.5]):

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t)) y_1(t), \quad (12.2.2)$$

$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2) y_2(t), \quad (12.2.3)$$

$$\mathbf{y}(0) = [100, 5]^\top, \quad (12.2.4)$$

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. 11.4.0.9]):

0	0	
$\frac{1}{3}$	$\frac{1}{3}$	0
$\frac{2}{3}$	$\frac{2}{3}$	0
$\frac{1}{4}$	0	$\frac{3}{4}$

(12.2.5)

Compute an approximated solution up to time  $T = 10$  for the number of equidistant time steps  $N = 2^j$ ,  $j = 7, \dots, 14$ .

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 (12-2.b) → [12-2-2-0:RK3P2s.pdf](#)



**End Problem 12-2** , 60 min.

### Problem 12-3: 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 → Rem. 11.5.0.15].

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  $\Psi^h$  define the **discrete evolution** of an order  $p$  Runge-Kutta single step method for the autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ,  $\mathbf{f} : D \subseteq \mathbb{R}^d \rightarrow \mathbb{R}^d$ . We define a new discrete evolution operator:

$$\tilde{\Psi}^h := \frac{1}{1 - 2^p} (\Psi^h - 2^p \cdot (\Psi^{h/2} \circ \Psi^{h/2})) , \quad (12.3.1)$$

where  $\circ$  denotes the composition of mappings.

**(12-3.a)**  $\square$  (10 min.) If  $\Psi^h$  belongs to the explicit/forward Euler method as introduced in [Lecture → Section 11.2.1], give the explicit formula for  $\tilde{\Psi}^h$ .

SOLUTION for (12-3.a) → [12-3-1-0:ODESolve1s.pdf](#)

**(12-3.b)**  $\square$  (10 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  $\tilde{\Psi}^h y_0$  when given the underlying  $\Psi$  through the functor  $\text{Psi}$ . Objects of type **DiscEvlOp** must provide an evaluation operator:

```
Eigen::VectorXd operator() (double h, const Eigen::VectorXd &y);
```

providing the result of  $\Psi^h(\mathbf{y})$ . For instance, suitable C++ lambda functions satisfy this requirement, see [Lecture → Section 0.3.3].

SOLUTION for (12-3.b) → [12-3-2-0:ODESolve2s.pdf](#)

**(12-3.c)**  $\square$  (10 min.) Implement a C++ function

```
template <class DiscEvlOp>
std::vector<Eigen::Vector>
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  $\Psi$  passed through the functor object  $\text{Psi}$ . 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  $\text{y}_0$ ).

HIDDEN HINT 1 for (12-3.c) → [12-3-3-0:ODESolve3h.pdf](#)

SOLUTION for (12-3.c) → [12-3-3-1:ODESolve3s.pdf](#)

(12-3.d)  (15 min.) [ depends on Sub-problem (12-3.c), Sub-problem (12-3.a) ]

For the particular scalar initial-value problem (IVP)

$$\dot{y} = 1 + y^2 \quad , \quad y(0) = 0, \quad (12.3.4)$$

with exact solution  $y(t) = \tan(t)$ , determine empirically the order of the single step method induced by  $\tilde{\Psi}^h$  from (12.3.1), when  $\Psi$  arises from the explicit Euler method, recall Sub-problem (12-3.a). For that purpose, write a C++ function

```
double testcvpExtrapolatedEuler(void);
```

that monitors and tabulates the error  $|y_N(1) - y_{ex}(1)|$  at final time  $T = 1$  for  $N$  uniform steps with  $N = 2^q, q = 2, \dots, 12$ , also returns an estimate of the rate of algebraic convergence based on linear regression.

HIDDEN HINT 1 for (12-3.d) → 12-3-4-0:ODESolve4h.pdf

SOLUTION for (12-3.d) → 12-3-4-1:ODESolve4s.pdf ▲

(12-3.e)  (30 min.) In general, the method defined by  $\tilde{\Psi}^h$  from (12.3.1) 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<Vector>>
odeintssctrl(DiscEvlOp &&Psi, double T,
              const Vector &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  $\Psi$  and  $\tilde{\Psi}$ , where  $\Psi$  is passed through the argument  $\text{Psi}$ .

Step rejection and stepsize correction and prediction as explained in [Lecture → Section 11.5] is to be employed as in [Lecture → Code 11.5.0.21]. The argument  $T$  supplies the final time,  $y_0$  the initial state,  $h_0$  an initial stepsize,  $p$  the order of the discrete evolution  $\Psi$ ,  $\text{reltol}$  and  $\text{abstol}$  the respective tolerances, and  $h_{\min}$  a minimal stepsize that will trigger premature termination.

The function should return a vector of tuples  $(t_k, y_k)$ ,  $k = 0, \dots, M$ , 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  $(y_k)_k$  the sequence of states generated by the single-step method.

HIDDEN HINT 1 for (12-3.e) → 12-3-5-0:EPH.pdf

SOLUTION for (12-3.e) → 12-3-5-1:ODESolve5s.pdf ▲

(12-3.f)  (20 min.) [ depends on Sub-problem (12-3.e) ]

Implement a C++ function

```
void solveTangentIVP(void)
```

that relies on `odeintssctrl` from Sub-problem (12-3.e) to compute the solution of the IVP (12.3.4) up to time  $T = 1.5$  with the following data:  $h_0 = 1/100$ ,  $\text{reltol} = 10e - 4$ ,  $\text{abstol} = 10e - 6$ ,  $h_{\min} = 10e - 5$ . 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 (12-3.f) → 12-3-6-0:sx.pdf ▲

**End Problem 12-3 , 95 min.**

### Problem 12-4: 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  $[0, T]$ :

$$\begin{aligned} 2\ddot{u}_1 - \ddot{u}_2 &= u_1(u_2 + u_1), \\ -\ddot{u}_{i-1} + 2\ddot{u}_i - \ddot{u}_{i+1} &= u_i(u_{i-1} + u_{i+1}), \quad i = 2, \dots, n-1, \\ 2\ddot{u}_n - \ddot{u}_{n-1} &= u_n(u_n + u_{n-1}), \\ u_i(0) &= u_{0,i} \quad i = 1, \dots, n, \\ \dot{u}_i(0) &= v_{0,i} \quad i = 1, \dots, n. \end{aligned} \tag{12.4.1}$$

Here the notation  $\ddot{w}$  designates the second derivative of a time-dependent function  $t \mapsto w(t)$ .

**(12-4.a)** (15 min.) Write (12.4.1) 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 (12-4.a) → [12-4-1-0:Syst1h1.pdf](#)

HIDDEN HINT 2 for (12-4.a) → [12-4-1-1:Syst1h2.pdf](#)

HIDDEN HINT 3 for (12-4.a) → [12-4-1-2:Syst1h3.pdf](#)

SOLUTION for (12-4.a) → [12-4-1-3:Syst1s.pdf](#) ▲

**(12-4.b)** (10 min.) Write down the discrete evolution for the classical Runge-Kutta method of order 4, whose Butcher scheme is given in [Lecture → Ex. 11.4.0.15] for the concrete case of the **autonomous linear ODE**  $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ ,  $\mathbf{M} \in \mathbb{R}^{d,d}$ .

HIDDEN HINT 1 for (12-4.b) → [12-4-2-0:h2.pdf](#)

SOLUTION for (12-4.b) → [12-4-2-1:Syst2s.pdf](#) ▲

**(12-4.c)** (15 min.) [ depends on Sub-problem (12-4.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 (12-4.a) (associated right-hand side function passed via the functor `odefun`).  $y_0$  contains the state before the step and the function has to return the state after the step in  $y_1$ .

SOLUTION for (12-4.c) → [12-4-3-0:Syst3s.pdf](#) ▲

**(12-4.d)** (20 min.) [ depends on Sub-problem (12-4.a), Sub-problem (12-4.c) ]

Implement a function

```
double testcvgRK4(void)
```

that applies the classical Runge-Kutta method of order 4 to solve a particular initial-value problem for the ODE derived in Sub-problem (12-4.a).

Use the parameters and initial values

$$n = 5, \quad u_i(0) = i/n, \quad v_i(0) = -1, \quad T = 1,$$

and tabulate the Euclidean norm of the error at final time  $T = 1$  for numbers  $N = 2, 4, 8, \dots, 1024$  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 (12-4.d) → [12-4-4-0 : Syst4s.pdf](#)



**End Problem 12-4 , 60 min.**

### Problem 12-5: 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 12-1.

We consider a *non-linear* ODE in the state space of  $n \times n$  matrices and study the associated initial value problem

$$\dot{\mathbf{Y}} = -(\mathbf{Y} - \mathbf{Y}^\top)\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) , \quad \mathbf{Y}(0) = \mathbf{Y}_0 \in \mathbb{R}^{n,n}, \quad (12.5.1)$$

whose solution is given by a *matrix-valued function*  $t \mapsto \mathbf{Y}(t) \in \mathbb{R}^{n \times n}$ .

**(12-5.a)** (15 min.) Write a C++ function

```
Eigen::MatrixXd matode(const Eigen::MatrixXd & Y0, double T)
```

which solves (12.5.1) 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  $\mathbf{Y}_0$ . Set the absolute tolerance to  $10^{-10}$  and the relative tolerance to  $10^{-8}$ . The output should be an approximation of  $\mathbf{Y}(T) \in \mathbb{R}^{n \times n}$ .

**Instructions on the use of `ode45`, see [Lecture → Code 12.0.0.3].**

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 → Rem. 11.5.0.24] and [Lecture → Ex. 11.5.0.25], with an adaptive stepsize control as presented in [Lecture → Rem. 11.5.0.15].

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  $10e-6$ )
- `atol`: absolute tolerance for error control (default is  $10e-8$ )

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>) >
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
      const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ( $y_0 = y_0$ )
- `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 (12-5.a) → [12-5-1-0:NMat01s.pdf](#)



**(12-5.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 (12.5.1).

HIDDEN HINT 1 for (12-5.b) → [12-5-2-0:NMat02h1.pdf](#)

HIDDEN HINT 2 for (12-5.b) → [12-5-2-1:NMat02h2.pdf](#)

SOLUTION for (12-5.b) → [12-5-2-2:NMat02s.pdf](#)



**(12-5.c)** ☺ (15 min.) [ depends on Sub-problem (12-5.a) ]

Write a C++ function

```
bool checkinvariant(const Eigen::MatrixXd & M, double T);
```

which (numerically) determines if the invariant  $t \mapsto \mathbf{Y}(t)^\top \mathbf{Y}(t)$  is preserved for approximate solutions of (12.5.1) 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 (12-5.c) → [12-5-3-0:NMat03s.pdf](#)



**(12-5.d)** ☺ (20 min.) [ depends on Sub-problem (12-5.a) ]

The so-called **discrete gradient method** for (12.5.1) reads

$$\mathbf{Y}_* = \mathbf{Y}_0 + \frac{1}{2}hf(\mathbf{Y}_0) , \quad \mathbf{Y}_1 = (\mathbf{I} + \frac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*)^\top)^{-1}(\mathbf{I} - \frac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*)^\top)\mathbf{Y}_0 . \quad (12.5.5)$$

Using the solution produced by `matode()` from Sub-problem (12-5.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 (12.5.1) 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(void);
```

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.

SOLUTION for (12-5.d) → [12-5-4-0:s5.pdf](#)



**End Problem 12-5 , 60 min.**

### Problem 12-6: 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 relies on a class implemented in Problem 12-2.

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.

In order to use the class `RKIntegrator`, you first need to construct an object of this class, passing as arguments the Butcher tableau matrices `A` and `b`, for instance:

```
RKIntegrator<VectorXd> 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<VectorXd>` containing the solution at each equidistant time step.

**(12-6.a)** (30 min.) Consider the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \quad (12.6.1)$$

where  $\mathbf{f}: \mathbb{R}^n \rightarrow \mathbb{R}^n$  and  $\mathbf{y}_0 \in \mathbb{R}^n$ . Using the class `RKIntegrator` write a C++ function

```
template <class Function>
double testcvgRKSSM(Function &&f, double T,
                      const VectorXd &y0,
                      const MatrixXd &A, const VectorXd &b)
```

that computes an approximated solution  $\mathbf{y}_N$  of (12.6.1) up to time  $T$  by means of an explicit Runge-Kutta method with  $N = 2^k$ ,  $k = 1, \dots, 15$ , uniform timesteps. The method is defined by the Butcher scheme described by the inputs `A` and `b`. The input `f` is an object with an evaluation operator (e.g. a lambda function) for arguments of type `const VectorXd &` representing `f`. The input `y0` passes the initial value `y0`.

For each  $k$ , the function should output the error at the final point  $E_N = |\mathbf{y}_N(T) - \mathbf{y}_{2^{15}}(T)|$ ,  $N = 2^k$ ,  $k = 1, \dots, 12$ , accepting  $\mathbf{y}_{2^{15}}(T)$  as exact value. Assuming algebraic convergence for  $E_N \approx CN^{-r}$ , at each step also print an approximation of the order of convergence  $r_k$  (recall that  $N = 2^k$ ). This will be an expression involving  $E_N$  and  $E_{N/2}$ .

Finally, compute and return an approximate order of convergence by linear regression. Only take into account results for which the error is larger than  $10^{-10}$  in order not to be misled by the impact of round-off errors.

SOLUTION for (12-6.a) → [12-6-1-0:OrdN1h.pdf](#)



**(12-6.b)** (20 min.) Calculate the analytical solutions of the logistic ODE (see [Lecture → Ex. 11.1.1.1])

$$\dot{y} = (1 - y)y, \quad y(0) = 1/2, \quad (12.6.4)$$

and of the initial value problem

$$\dot{y} = |1.1 - y| + 1, \quad y(0) = 1. \quad (12.6.5)$$

HIDDEN HINT 1 for (12-6.b) → 12-6-2-0:ONA2s.pdf

HIDDEN HINT 2 for (12-6.b) → 12-6-2-1:OMA6s.pdf

HIDDEN HINT 3 for (12-6.b) → 12-6-2-2:OMA8s.pdf

SOLUTION for (12-6.b) → 12-6-2-3:OrdN2h.pdf



**(12-6.c)** (20 min.) [ depends on Sub-problem (12-6.a) ]

Using `testcvgRKSSM()` write a C++ function

```
void cmpCvgRKSSM(void);
```

that empirically determines and prints the rates of convergence of

- the explicit Euler method [Lecture → Eq. (11.2.1.4)], a RK single step method of order 1,
- the explicit trapezoidal rule [Lecture → Eq. (11.4.0.6)], a RK single step method of order 2,
- an RK method of order 3 given by the Butcher tableau [Lecture → Eq. (11.4.0.11)]

0		
1/2	1/2	
1	-1	2
	1/6	2/3
	1/6	

- the classical RK method of order 4, see [Lecture → Ex. 11.4.0.15] for details.

when applied for the numerical integration of the initial-value problems (12.6.4) and (12.6.5). Use final time  $T = 0.1$  in each case.

Comment on the calculated order of convergence for the different methods and the two different initial value problems.

SOLUTION for (12-6.c) → 12-6-3-0:x1s.pdf



**End Problem 12-6 , 70 min.**

### Problem 12-7: 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 11.1.3] and still remember Newton's method from [Lecture → Section 8.4].

**A differential equation for the derivative w.r.t. initial state.** We consider IVPs for the autonomous ODE

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad (12.7.1)$$

with smooth right hand side  $\mathbf{f}: D \rightarrow \mathbb{R}^d$ , 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. 11.1.3.1]).

By its very definition given in [Lecture → Def. 11.1.3.2], the evolution operator

$$\Phi: \mathbb{R} \times D \rightarrow D, \quad (t, \mathbf{y}) \mapsto \Phi(t, \mathbf{y})$$

satisfies

$$\frac{\partial \Phi}{\partial t}(t, \mathbf{y}) = \mathbf{f}(\Phi(t, \mathbf{y})).$$

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  $D_{\mathbf{y}}$ ,

$$\frac{\partial D_{\mathbf{y}} \Phi}{\partial t}(t, \mathbf{y}) = D_{\mathbf{y}} \frac{\partial \Phi}{\partial t}(t, \mathbf{y}) = D_{\mathbf{y}}(\mathbf{f}(\Phi(t, \mathbf{y}))) = D \mathbf{f}(\Phi(t, \mathbf{y})) D_{\mathbf{y}} \Phi(t, \mathbf{y}).$$

Abbreviating  $\mathbf{W}(t, \mathbf{y}) := D_{\mathbf{y}} \Phi(t, \mathbf{y})$  we can rewrite this as the non-autonomous ODE

$$\dot{\mathbf{W}} = D \mathbf{f}(\Phi(t, \mathbf{y})) \mathbf{W} \quad .. \quad (12.7.2)$$

Here, the state  $\mathbf{y}$  can be regarded as a parameter. Since  $\Phi(0, \mathbf{y}) = \mathbf{y}$ , we also know  $\mathbf{W}(0, \mathbf{y}) = \mathbf{I}$  (identity matrix), which supplies an initial condition for (12.7.2). In fact, we can even merge (12.7.1) and (12.7.2) into the ODE

$$\frac{d}{dt} [\mathbf{y}(\cdot), \mathbf{W}(\cdot, \mathbf{y}_0)] = [\mathbf{f}(\mathbf{y}(t)), D \mathbf{f}(\mathbf{y}(t)) \mathbf{W}(t, \mathbf{y}_0)], \quad (12.7.3)$$

which is autonomous again.

Now let us apply (12.7.2)/(12.7.3). As in [Lecture → Ex. 11.1.5], we consider the following autonomous Lotka-Volterra differential equation of a predator-prey model

$$\begin{aligned} \dot{u} &= (2-v)u, \\ \dot{v} &= (u-1)v, \end{aligned} \quad (12.7.4)$$

on the state space  $D = \mathbb{R}_+^2$ ,  $\mathbb{R}_+ = \{\xi \in \mathbb{R} : \xi > 0\}$ . All the solutions of (12.7.4) 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.

**(12-7.a)**  (5 min.) For fixed state  $\mathbf{y} \in D$ , (12.7.2) represents an ODE. What is its state space?

SOLUTION for (12-7.a) → [12-7-1-0:Init1h.pdf](#)

**(12-7.b)**  (10 min.) What is the right hand side function for the ODE (12.7.2), when the underlying autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  is the Lotka-Volterra ODE (12.7.4)? You may write  $u(t), v(t)$  for solutions of (12.7.4).

SOLUTION for (12-7.b) → [12-7-2-0:Init2h.pdf](#)

**(12-7.c)**  (10 min.) From now on we write  $\Phi: \mathbb{R} \times \mathbb{R}_+^2 \rightarrow \mathbb{R}_+^2$  for the **evolution operator** (→ [Lecture → Def. 11.1.3.2]) associated with (12.7.4). Based on  $\Phi$  derive a non-trivial function  $\mathbf{F}: \mathbb{R}_+^2 \rightarrow \mathbb{R}^2$ ,  $\mathbf{F} \not\equiv \mathbf{0}$ , which evaluates to zero for the input  $\mathbf{y}_0$  if the period of the solution of system (12.7.4) 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 (12-7.c) → [12-7-3-0:Init4h.pdf](#)

**(12-7.d)**  (5 min.) [ depends on Sub-problem (12-7.c) ]

We write  $\mathbf{W}(T, \mathbf{y}_0)$ ,  $T \geq 0$ ,  $\mathbf{y}_0 \in \mathbb{R}_+^2$  for the solution of (12.7.2) for the underlying ODE (12.7.4). Express the Jacobian of  $\mathbf{F}$  by means of  $\mathbf{W}$ .

SOLUTION for (12-7.d) → [12-7-4-0:Init5h.pdf](#)

**(12-7.e)**  (10 min.) [ depends on Sub-problem (12-7.c) ]

Argue, why the solution of  $\mathbf{F}(\mathbf{y}) = \mathbf{0}$  will, in general, not be unique. When will it be unique?

HIDDEN HINT 1 for (12-7.e) → [12-7-5-0:Init1s.pdf](#)

SOLUTION for (12-7.e) → [12-7-5-1:Init6h.pdf](#)

A C++ implementation **ode45** of an adaptive embedded Runge-Kutta method has been presented in [Lecture → Code 12.0.0.3].

**Instructions on the use of **ode45**, see [Lecture → Code 12.0.0.3].**

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 → Rem. 11.5.0.24] and [Lecture → Ex. 11.5.0.25], with an adaptive stepsize control as presented in [Lecture → Rem. 11.5.0.15].

1. Construct an object of **ode45** type: create an instance of the class, passing the right-hand-side function  $f$  as a functor object to the constructor

```
template <class StateType,
          class RhsType = std::function<StateType(const StateType &) >>
class ode45 {
public:
    ode45(const RhsType &rhs);
    // .....
}
```

Template parameters are

- **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `**`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.

- **RhsType**: type of rhs function (automatically deduced).

The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator() (const StateType & vec) ;
```

It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is `10e-6`)
- `atol`: absolute tolerance for error control (default is `10e-8`)

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** (`y0 = 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.

**(12-7.f)** (25 min.) [ depends on Sub-problem (12-7.b) ]

Relying on **ode45**, implement a C++ function

```
std::pair<Vector2d, Matrix2d> PhiAndW(double u0, double v0, double
T)
```

that computes  $\Phi(T, [u_0, v_0]^T)$  and  $\mathbf{W}(T, [u_0, v_0]^T)$ . The first component of the output pair should contain  $\Phi(T, [u_0, v_0]^T)$  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 (12-7.f) → 12-7-6-0:Init2s.pdf

SOLUTION for (12-7.f) → 12-7-6-1:Init7h.pdf ▲

(12-7.g) ☺ (25 min.) [ depends on Sub-problem (12-7.f) and Sub-problem (12-7.d) ]

Relying on `PhiAndW()`, write a C++ routine

```
std::pair<double, double> findInitCond(void);
```

that determines and returns initial conditions  $u(0)$  and  $v(0)$  such that the solution of the system (12.7.4) has period  $T_P = 5$ . Use the multi-dimensional **Newton method** for  $\mathbf{F}(\mathbf{y}) = \mathbf{0}$  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  $\Phi(100, \mathbf{y})$ .

**Remark.** The residual based termination criterion recommended above [Lecture → § 8.1.2.4] 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 8.4.3].

HIDDEN HINT 1 for (12-7.g) → 12-7-7-0:lvxh.pdf

SOLUTION for (12-7.g) → 12-7-7-1:Init8h.pdf ▲

**End Problem 12-7**, 90 min.

### Problem 12-8: 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 has more options and this problem examines a class of single-step methods that also rely on  $\mathbf{D}\mathbf{f}$ .

The new class of methods is obtained by the following reasoning: if the right hand side  $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^n$  of an autonomous initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \quad (12.8.1)$$

with solution  $\mathbf{y} : \mathbb{R} \rightarrow \mathbb{R}^n$  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. 8.2.2.12],

$$\mathbf{y}(t) = \sum_{n=0}^m \frac{\mathbf{y}^{(n)}(0)}{n!} t^n + R_m(t), \quad (12.8.2)$$

with remainder term  $R_m(t) = O(t^{m+1})$  for  $t \rightarrow 0$ .

A single step method for the numerical integration of (12.8.1) can be obtained by choosing  $m = 3$  in (12.8.2), 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*.

**(12-8.a)** (10 min.) Express  $\frac{d\mathbf{y}}{dt}(t)$  and  $\frac{d^2\mathbf{y}}{dt^2}(t)$  in terms of  $\mathbf{f}$  and its Jacobian  $\mathbf{D}\mathbf{f}$ .

HIDDEN HINT 1 for (12-8.a) → 12-8-1-0:Tay11h.pdf

SOLUTION for (12-8.a) → 12-8-1-1:Tay11s.pdf

**(12-8.b)** (15 min.) [ depends on Sub-problem (12-8.a) ]

Verify the formula

$$\frac{d^3\mathbf{y}}{dt^3}(0) = \mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{f}(\mathbf{y}_0), \mathbf{f}(\mathbf{y}_0)) + \mathbf{D}\mathbf{f}(\mathbf{y}_0)^2\mathbf{f}(\mathbf{y}_0). \quad (12.8.3)$$

HIDDEN HINT 1 for (12-8.b) → 12-8-2-0:Tay10h.pdf

HIDDEN HINT 2 for (12-8.b) → 12-8-2-1:Tay12h.pdf

HIDDEN HINT 3 for (12-8.b) → 12-8-2-2:Tay124h.pdf

SOLUTION for (12-8.b) → 12-8-2-3:Tay12s.pdf

**(12-8.c)** Now we apply the Taylor expansion method introduced above to the following ODE for the *predator-prey* model, as introduced in [Lecture → Ex. 11.1.1.5]:

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t)) y_1(t), \quad (12.8.4)$$

$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2) y_2(t), \quad (12.8.5)$$

$$\mathbf{y}(0) = [100, 5]^\top. \quad (12.8.6)$$

To this end, in the template file `taylorintegrator.hpp` implement a templated C++ function

```
template <class State, class Function>
std::vector<State> solvePredPreyTaylor(double T,
                                         const State &y0,
                                         unsigned int N);
```

for the numerical integration of initial-value problems for (12.8.4) using the Taylor expansion method with  $N, N \in \mathbb{N}$  uniform timesteps on the temporal interval  $[0, T]$

HIDDEN HINT 1 for (12-8.c) → [12-8-3-0:Tayl3h.pdf](#)

SOLUTION for (12-8.c) → [12-8-3-1:Tayl3s.pdf](#) ▲

**(12-8.d)** (20 min.) [ depends on Sub-problem (12-8.c) ]

Based on the function `solvePredPreyTaylor()` implemented in Sub-problem (12-8.c) experimentally determine the order of convergence of the considered Taylor expansion method when it is applied to solve (12.8.4). Study the behaviour of the error at final time  $t = 10$  for the initial data  $\mathbf{y}(0) = [100, 5]^\top$ . Implement a function

```
double testCvgTaylorMethod(void);
```

that generates a suitable error table and returns the empiric rate of algebraic convergence in terms of the (uniform) timestep  $h \rightarrow 0$ . This rate should be determined by linear regression.

SOLUTION for (12-8.d) → [12-8-4-0:Tayl4s.pdf](#) ▲

**(12-8.e)** (5 min.) What is the disadvantage of the Taylor's method compared with a Runge-Kutta method?

SOLUTION for (12-8.e) → [12-8-5-0:Tayl5s.pdf](#) ▲

**End Problem 12-8 , 50 min.**

### Problem 12-9: 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 → Code 12.0.0.3].

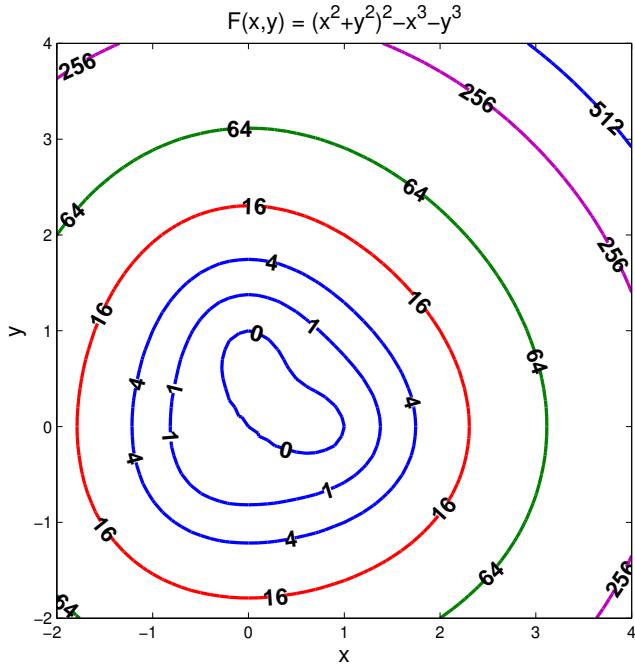


Fig. 69

If  $F$  is continuously differentiable, we can obtain connected components of  $\mathcal{C}$  as solution curves of the initial value problems

$$\dot{\mathbf{y}}(t) = \frac{\mathbf{grad} F(\mathbf{y}(t))^\perp}{\|\mathbf{grad} F(\mathbf{y}(t))\|}, \quad F(\mathbf{y}(0)) = 0. \quad (12.9.1)$$

Here  $^\perp$  stands for the counterclockwise rotation of a vector  $\in \mathbb{R}^2$  by  $90^\circ$ , that is,

$$\begin{bmatrix} x \\ y \end{bmatrix}^\perp = \begin{bmatrix} -y \\ x \end{bmatrix}, \quad x, y \in \mathbb{R}. \quad (12.9.2)$$

Then (12.9.1) is a consequence of the fact that an isoline always runs perpendicular to the direction of steepest ascent of  $F$ , which is provided by  $\mathbf{grad} F$ .

(12-9.a) (5 min.)

State the ordinary differential equation (12.9.1) for the concrete case of the function

$$F : \mathbb{R}^2 \rightarrow \mathbb{R}, \quad F(x, y) := (x^2 + y^2)^2 - x^3 - y^3. \quad (12.9.3)$$

SOLUTION for (12-9.a) → [12-9-1-0:s1.pdf](#)

(12-9.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 (12.9.1) with initial state  $\mathbf{y}_0 \in \mathbb{R}^2$  passed in  $y_0$  from intial time  $0$  up to final time  $T$  (given in  $T$ ). The argument  $gradF$  supplies a functor object equipped with an evaluation operator

```
Eigen::Vector2d operator(Eigen::Vector2d x) const;
```

that returns  $\mathbf{grad}\mathcal{F}(\mathbf{x})$ . The function `computeIsolinePoints()` should return the sequence of computed states  $\mathbf{y}_0, \mathbf{y}_1, \dots, \mathbf{y}_N$ ,  $N \in \mathbb{N}$ , 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 → Code 12.0.0.3].**

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 → Rem. 11.5.0.24] and [Lecture → Ex. 11.5.0.25], with an adaptive stepsize control as presented in [Lecture → Rem. 11.5.0.15].

1. Construct an object of `ode45` type: create an instance of the class, passing the right-hand-side function  $f$  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:

```
options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is  $10e-6$ )
- `atol`: absolute tolerance for error control (default is  $10e-8$ )

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>) >
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
      const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument **norm** is optional. The other arguments are

- $y_0$ : initial value of type **StateType** ( $y_0 = y_0$ )
- $T$ : 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 (12-9.b) → [12-9-2-0:s2.pdf](#)



(12-9.c) ☐ (10 min.) [ depends on Sub-problem (12-9.a) and Sub-problem (12-9.b) ]

The 0-isoline of  $F$  from (12.9.3)

$$\mathcal{C}_{\text{egg}} := \{x = (x, y) \in \mathbb{R}^2 \setminus \{\mathbf{0}\} : F(x) := (x^2 + y^2)^2 - x^3 - y^3 = 0\}.$$

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 (12-9.c) → [12-9-3-0:s3h1.pdf](#)

SOLUTION for (12-9.c) → [12-9-3-1:s3.pdf](#)



(12-9.d) ☒ (20 min.) [ depends on Sub-problem (12-9.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 (12-9.b). However, this time the argument  $F$  merely contains a functor object providing the function  $F$  itself.

To obtain an approximation of  $\text{grad } F$  rely on an approximation by **symmetric difference quotients**, e.g.,

$$\frac{\partial F}{\partial x_1}(\mathbf{x}) \approx \frac{F(\mathbf{x} + \begin{bmatrix} h \\ 0 \end{bmatrix}) - F(\mathbf{x} - \begin{bmatrix} h \\ 0 \end{bmatrix})}{2h} \quad \text{for some } h > 0.$$

Choose a suitable fixed  $h > 0$  and explain your choice in a comment in your code.

HIDDEN HINT 1 for (12-9.d) → [12-9-4-0:h4.pdf](#)

SOLUTION for (12-9.d) → [12-9-4-1:s4.pdf](#)



**End Problem 12-9 , 65 min.**

### Problem 12-10: 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 structural 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

$$\frac{d}{dt} \begin{bmatrix} \mathbf{p} \\ \mathbf{q} \end{bmatrix} = \begin{bmatrix} \mathbf{f}(\mathbf{q}, t) \\ \mathbf{g}(\mathbf{p}) \end{bmatrix} \quad \text{with continuous functions } \begin{aligned} \mathbf{f} : \mathbb{R}^n \times \mathbb{R} &\rightarrow \mathbb{R}^n, \\ \mathbf{g} : \mathbb{R}^n &\rightarrow \mathbb{R}^n. \end{aligned} \quad (12.10.1)$$

For the temporal discretization of (12.10.1), [RWR04] proposes the following class of  $s$ -stage methods,  $s \in \mathbb{N}$ , with uniform timestep  $h > 0$  for generating sequences  $\left( \begin{bmatrix} \mathbf{p}_j \\ \mathbf{q}_j \end{bmatrix} \right)_{j=0,\dots,M}$  of approximations of  $\begin{bmatrix} \mathbf{p}(jh) \\ \mathbf{q}(jh) \end{bmatrix}$ , starting from initial values  $\mathbf{p}_0 := \mathbf{p}(0)$  and  $\mathbf{q}_0 := \mathbf{q}(0)$ :

```

for  $j := 1$  to  $M$  do
     $\mathbf{p}_{\text{in}} := \mathbf{p}_{j-1}$ ,  $\mathbf{q}_{\text{in}} := \mathbf{q}_{j-1}$ 
    for  $k := 1$  to  $s$  do
         $t_k := j h + h \sum_{\ell=1}^{k-1} a_\ell$ ;
         $\mathbf{p}_{\text{out}} := \mathbf{p}_{\text{in}} + h b_k \mathbf{f}(\mathbf{q}_{\text{in}}, t_k)$ ;
         $\mathbf{q}_{\text{out}} := \mathbf{q}_{\text{in}} + h a_k \mathbf{g}(\mathbf{p}_{\text{out}})$ ;
         $\mathbf{p}_{\text{in}} := \mathbf{p}_{\text{out}}$ ,  $\mathbf{q}_{\text{in}} := \mathbf{q}_{\text{out}}$ ;
    end
     $\mathbf{p}_j := \mathbf{p}_{\text{out}}$ ,  $\mathbf{q}_j := \mathbf{q}_{\text{out}}$ ;
end

```

(12.10.2)

Here  $a_1, \dots, a_s$  and  $b_1, \dots, b_s$  are given coefficients, carefully chosen such that the single-step method achieves a certain order. A possible choice discussed in [RWR04] is, for  $s = 3$ ,

$$a_1 := \frac{2}{3}, \quad a_2 := -\frac{2}{3}, \quad a_3 := 1, \quad b_1 := \frac{7}{24}, \quad b_2 := \frac{3}{4}, \quad b_3 := -\frac{1}{24}. \quad (12.10.3)$$

(12-10.a) (10 min.) [ depends on Sub-problem (12-10.a) ]

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 (12.10.2) with  $s = 3$  and coefficients according to (12.10.3) for the “harmonic oscillator” ODE ( $n = 1$ )

$$\dot{\mathbf{p}} = \mathbf{q}, \quad \dot{\mathbf{q}} = -\mathbf{p}. \quad (12.10.4)$$

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 (12-10.a) → [12-10-1-0:symp4.pdf](#)



(12-10.b) (15 min.) [ depends on Sub-problem (12-10.a) ]

Experimentally determine the order of convergence of the single-step method implemented in Subproblem (12-10.a) in terms of the number of timesteps by

- integrating (12.10.4) 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:

$$\text{err}(M) := |p(2\pi) - p^{(M)}| + |q(2\pi) - q^{(M)}|, \quad M = 10, 20, 40, 80, 160, 320, 640, .$$

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 (12-10.b) → [12-10-2-0:smp5h1.pdf](#)

SOLUTION for (12-10.b) → [12-10-2-1:symp5.pdf](#) ▲

The symplectic timestepping method has especially been designed for **Hamiltonian ODEs**

$$\begin{aligned} \dot{\mathbf{p}}(t) &= -\frac{\partial H}{\partial \mathbf{q}}(\mathbf{p}(t), \mathbf{q}(t)), \\ \dot{\mathbf{q}}(t) &= \frac{\partial H}{\partial \mathbf{p}}(\mathbf{p}(t), \mathbf{q}(t)) \end{aligned} \quad \text{with } H : \mathbb{R}^n \times \mathbb{R}^n \rightarrow \mathbb{R}. \quad (12.10.7)$$

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.

(12-10.c) (15 min.) Show that for every solution  $t \mapsto (\mathbf{p}(t), \mathbf{q}(t))$ ,  $t \in I \subset \mathbb{R}$ , of (12.10.7) the function

$$t \mapsto E(t) := H(\mathbf{p}(t), \mathbf{q}(t)), \quad t \in I, \quad (12.10.8)$$

is constant.

SOLUTION for (12-10.c) → [12-10-3-0:s6.pdf](#) ▲

(12-10.d) (10 min.) Write down explicitly in the form (12.10.1) the Hamiltonian ODE (12.10.7) for the particular Hamiltonian

$$H(\mathbf{p}, \mathbf{q}) = \frac{1}{2}\|\mathbf{p}\|_2^2 + \|\mathbf{q}\|_2^4, \quad \mathbf{p}, \mathbf{q} \in \mathbb{R}^n, \quad n \in \mathbb{N}. \quad (12.10.9)$$

SOLUTION for (12-10.d) → [12-10-4-0:s7.pdf](#) ▲

(12-10.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 symplectic timestepping scheme (12.10.2) + (12.10.3) in order solve the Hamiltonian ODE (12.10.7) with Hamiltonian  $H$  as in (12.10.9) up to final time  $T > 0$  and with initial values  $\mathbf{p}_0$  and  $\mathbf{q}_0$  passed in the argument vectors  $\mathbf{p}_0$  and  $\mathbf{q}_0$  of equal length  $n \in \mathbb{N}$ .

The function should return the numerical solution

$$\begin{bmatrix} \mathbf{p}_0 \\ \mathbf{q}_0 \end{bmatrix}, \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{q}_1 \end{bmatrix}, \dots, \begin{bmatrix} \mathbf{p}_M \\ \mathbf{q}_M \end{bmatrix},$$

in the columns of a  $2n \times (M+1)$ -matrix.

SOLUTION for (12-10.e) → [12-10-5-0:s8.pdf](#)



(12-10.f) (10 min.) The following table gives the quantities

$$\mathcal{E}(M) := \max\{E(\mathbf{p}_k, \mathbf{q}_k), k = 0, \dots, M\}, \quad (12.10.12)$$

where  $E$  is the “energy” as defined in (12.10.8) 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  $M \rightarrow \infty$ .

SOLUTION for (12-10.f) → [12-10-6-0:s9.pdf](#)



**End Problem 12-10 , 80 min.**

# Chapter 13

## Single Step Methods for Stiff Initial Value Problems

### Problem 13-1: Implicit Runge-Kutta method

This problem addresses the implementation of general **implicit** Runge-Kutta methods [Lecture → Def. 12.3.3.1]. We will adapt all routines developed for the explicit method to the implicit case. This problem assumes familiarity with [Lecture → Section 12.3], and, especially, [Lecture → Section 12.3.3] and [Lecture → Rem. 12.3.3.7].

Related to Problem 12-2

Problem 12-2 introduced the class **RKIntegrator** that implemented the timestepping for a general **explicit** Runge-Kutta method according to [Lecture → Def. 11.4.0.9]. Keeping the interface we now extend this class so that it realizes a general Runge-Kutta timestepping method as given in [Lecture → Def. 12.3.3.1] for solving an autonomous initial value problem  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$ ,  $\mathbf{y}(0) = \mathbf{y}_0$ .

**(13-1.a)** (10 min.) Rederive the **stage form** [Lecture → Eq. (12.3.3.6)] of a general (implicit) Runge-Kutta method from the increment equations as given in [Lecture → Def. 12.3.3.1].

SOLUTION for (13-1.a) → [13-1-1-0:0s.pdf](#)



**(13-1.b)** (15 min.) Let  $\mathbf{g}_i$ ,  $i = 1, \dots, s$ , be the so-called stages of a general Runge-Kutta method as defined in [Lecture → Eq. (12.3.3.5)]. In [Lecture → Rem. 12.3.3.7] the stages are recovered by solving a non-linear system of equations  $\mathbf{F}(\mathbf{g}) = \mathbf{0}$  with a suitable  $\mathbf{F} : \mathbb{R}^{sd} \rightarrow \mathbb{R}^{sd}$ . Formulate the Newton iteration for it when only autonomous ODEs  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  with differentiable right-hand-side functions  $\mathbf{f}$  are considered.

SOLUTION for (13-1.b) → [13-1-2-0:0as.pdf](#)



**(13-1.c)** (45 min.) [ depends on Sub-problem (13-1.b) ]

By modifying the class **RKIntegrator** for the implementation of explicit Runge-Kutta methods, design a similar header-only C++ class **implicit\_RKIntegrator** which implements a general implicit RK method given through a Butcher scheme [Lecture → Eq. (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 implicit_RKIntegrator {
public:
    implicit_RKIntegrator(
        const Eigen::MatrixXd &A,
        const Eigen::VectorXd &b);
```

```

template <class Function, class Jacobian>
std::vector<Eigen::VectorXd> solve(
    Function &&f, Jacobian &&Jf, double T,
    const Eigen::VectorXd &y0, unsigned int 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  $\mathbf{A} \in \mathbb{R}^{s,s}$  and the weight vector  $\mathbf{b} \in \mathbb{R}^s$  that are passed to the constructor as arguments  $\mathbf{A}$  and  $\mathbf{b}$ .

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  $\mathbf{D}\mathbf{f}$  are passed through the functor objects  $\mathbf{f}$  and  $\mathbf{J}$  equipped with suitable evaluation operators.

In the `solve()` method the stages  $\mathbf{g}_i$  as introduced in ((13-1.a)) are to be computed with the **damped Newton method** (see [Lecture → Section 8.4.4]) applied to the nonlinear system of equations satisfied by the stages (see [Lecture → Rem. 12.3.3.4] and [Lecture → Rem. 12.3.3.7]). 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 8.4.4.5]. Note that we do not use the simplified Newton method as discussed in [Lecture → Rem. 12.3.3.7].

In the code template you will find large parts of **implicit\_RKIntegrator** already implemented. In fact, you only have to write the method `step()` for the actual implicit RK-SSM timestepping.

SOLUTION for (13-1.c) → [13-1-3-0:Impl1h.pdf](#)

(13-1.d) (15 min.) Examine the following code

#### C++11-code 13.1.5: Initialization of **implicit\_RKIntegrator** object

```

2 // Definition of coefficients in Butcher scheme
3 unsigned int s = 2;
4 MatrixXd A(s, s);
5 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 implicit_RKIntegrator RK(A, b);

```

Write down the complete Butcher scheme according to [Lecture → Eq. (12.3.3.3)] for the implicit Runge-Kutta method now available through `RK`. Which method is it? Is it A-stable [Lecture → Def. 12.3.4.6], L-stable [Lecture → Def. 12.3.4.12]?

HIDDEN HINT 1 for (13-1.d) → [13-1-4-0:Impl1s.pdf](#)

SOLUTION for (13-1.d) → [13-1-4-1:Impl2h.pdf](#)

**(13-1.e)** (15 min.) The file `main.cpp` uses your implementation `implicit_RKintegrator` of general implicit RK-SSMs to solve an initial value problem for the Lotka-Volterra ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_1(\alpha_1 - \beta y_2) \\ y_2(-\alpha_2 + \beta y_1) \end{bmatrix}. \quad (13.1.6)$$

with the particular parameters  $\alpha_1 = 3$ ,  $\alpha_2 = 2$ ,  $\beta = 0.1$ . 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 (13-1.e) → [13-1-5-0:Impl13h.pdf](#)



**End Problem 13-1 , 100 min.**

### Problem 13-2: 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.

Related to [Lecture → Chapter 12]

We consider the initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c \mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1, 1, 1]^\top, \quad (13.2.1)$$

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> required.

**(13-2.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 (13-2.a) → 13-2-1-0:Cros1s.pdf

SOLUTION for (13-2.a) → 13-2-1-1:Cros1h.pdf 

**(13-2.b)**  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 (13-2.b) → 13-2-2-0:Cros2h.pdf 

**(13-2.c)**  [ depends on Sub-problem (13-2.b) ]

For  $\mathbf{a} = [1, 0, 0]^\top$ , (13.2.1) 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.

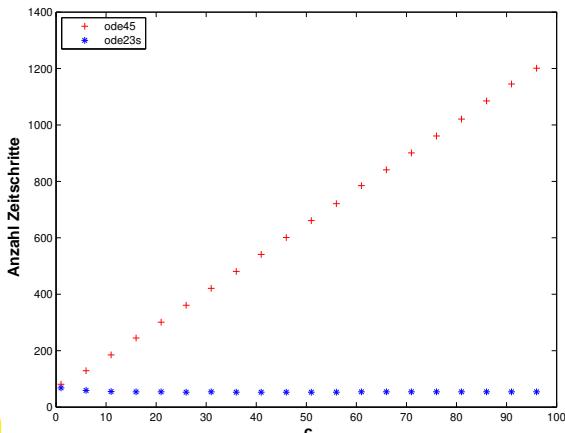


Fig. 71

SOLUTION for (13-2.c) → 13-2-3-0:Cros3h.pdf 

**(13-2.d)**  Formulate the non-linear equation given by the implicit mid-point method from [Lecture → Section 11.2.3] for the initial value problem (13.2.1).

SOLUTION for (13-2.d) → 13-2-4-0:Cros4h.pdf 

**(13-2.e)**  By means of a C++ function

```
void tab_crossprod(void);
```

solve (13.2.1) with  $\mathbf{a} = [1, 0, 0]^\top$ ,  $c = 1$  up to  $T = 10$ . Use  $N = 128$  uniform time steps of the implicit mid-point method. Tabulate  $\|\mathbf{y}_k\|_2$  for the sequence of approximate states generated by the implicit midpoint method. What do you observe?

For this task rely on the helper class `implicit_RKIntegrator`:

```
class implicit_RKIntegrator {
public:
    implicit_RKIntegrator(
        const Eigen::MatrixXd &A,
        const Eigen::VectorXd &b);
    template <class Function, class Jacobian>
    std::vector<Eigen::VectorXd> solve(
        Function &&f, Jacobian &&Jf, double T,
        const Eigen::VectorXd &y0, unsigned int N) 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  $N$  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  $J$  equipped with suitable evaluation operators. The `solve()` method relies on the damped Newton iteration to solve the stage equations, see [Lecture → Rem. 12.3.3.4] and [Lecture → Rem. 12.3.3.7].

SOLUTION for (13-2.e) → [13-2-5-0:Cros5h.pdf](#)



**(13-2.f)** ☺ 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  $\mathbf{f} : \mathbb{R}^d \rightarrow \mathbb{R}^d$ .

SOLUTION for (13-2.f) → [13-2-6-0:Cros6h.pdf](#)



**(13-2.g)** ☺ Implement the linear-implicit midpoint method in the C++ function

```
std::vector<VectorXd> solve_lin_mid(
    const Function &f,
    const Jacobian &Jf,
    double T,
    const Eigen::VectorXd & y0,
    unsigned int N);
```

The signature of this function is the same as that of the `solve()` method of `implicit_RKIntegrator`.

Extend your implementation of `tab_crossprod()` so that it uses `solve_lin_mid()` to solve (13.2.1) with  $\mathbf{a} = [1, 0, 0]^\top$ ,  $c = 1$  up to  $T = 10$  and  $N = 128$ . Tabulate  $\|\mathbf{y}_k\|_2$  for the sequence of approximate states generated by the linear implicit midpoint method. What do you observe?

SOLUTION for (13-2.g) → [13-2-7-0:Cros7h.pdf](#)



**End Problem 13-2 , 10 min.**

### Problem 13-3: 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:

$$\begin{array}{c|ccccc} & c_1 & \gamma & \cdots & 0 \\ \mathbf{c} & c_2 & a_{21} & \ddots & \vdots \\ & \vdots & \vdots & \ddots & \vdots \\ & \vdots & \vdots & \ddots & \vdots \\ & c_s & a_{s1} & \cdots & a_{s,s-1} & \gamma \\ \hline & b_1 & \cdots & b_{s-1} & b_s \end{array}, \quad (13.3.1)$$

with  $\gamma \neq 0$ .

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

$$\ddot{y} + \dot{y} + y = 0, \quad y(0) = 1, \quad \dot{y}(0) = 0 \quad (13.3.2)$$

should be solved numerically using a family of 2-stage SDIRK method described by the Butcher scheme

$$\begin{array}{c|cc} & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array}. \quad (13.3.3)$$

**(13-3.a)** (5 min.) Explain the benefit of using SDIRK-SSMs compared to using Gauss-Radau RK-SSMs as introduced in [Lecture → Ex. 12.3.4.18]. In what situations will this benefit matter much?

HIDDEN HINT 1 for (13-3.a) → [13-3-1-0:SDIR0h.pdf](#)

SOLUTION for (13-3.a) → [13-3-1-1:SDIR0s.pdf](#) ▲

**(13-3.b)** (10 min.) State the equations for the increments  $\mathbf{k}_1$  and  $\mathbf{k}_2$  of the Runge-Kutta method (13.3.3) applied to the initial value problem corresponding to the differential equation  $\dot{\mathbf{y}} = \mathbf{f}(t, \mathbf{y})$ .

SOLUTION for (13-3.b) → [13-3-2-0:SDIR1s.pdf](#) ▲

**(13-3.c)** (20 min.) Show that the stability function  $S(z)$  of the SDIRK method (13.3.3) 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 using the supplied MATLAB's function `stabdomSDIRK.m` or C++ code `stabdomSDIRK.cpp`.

SOLUTION for (13-3.c) → [13-3-3-0:SDIR2s.pdf](#) ▲

**(13-3.d)** (25 min.) Find out whether for  $\gamma = 1$  the SDIRK RK-SSM (13.3.3) is

- A-stable,
- L-stable.

Argue, based on the following version of the maximum principle for analytic functions:

**Lemma 13.3.4. 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 (13-3.d) → [13-3-4-0:SDIRxs.pdf](#)

(13-3.e)  (5 min.) Formulate (13.3.2) as an initial value problem for a linear first order system for the function  $\mathbf{z}(t) = (y(t), \dot{y}(t))^\top$ .

SOLUTION for (13-3.e) → [13-3-5-0:SDIR3s.pdf](#)

(13-3.f)  (20 min.) [ depends on Sub-problem (13-3.e) ]

Implement a C++ function

```
StateType sdirkStep(const Eigen::VectorXd &z0, double h, double gamma);
```

that realizes one step of the method (13.3.3) for the linear differential equation (13.3.2), starting from the initial state  $z_0$  and returning the state after a single step of size  $h$ .

SOLUTION for (13-3.f) → [13-3-6-0:SDIR4s.pdf](#)

(13-3.g)  (25 min.) [ depends on Sub-problem (13-3.f) ]

Realize a C++ function

```
double cvgSDIRK(void);
```

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 (13.3.3). Choose  $\mathbf{y}_0 = [1, 0]^\top$  as initial value,  $T=10$  as end time and  $N=20, 40, 80, \dots, 10240$  as numbers of equidistant timesteps. Track the maximal Euclidean error norm on the temporal mesh, tabulate it and use it as the basis for the estimation of the rate of algebraic convergence by means of the supplied function `polyfit`.

SOLUTION for (13-3.g) → [13-3-7-0:SDIR5s.pdf](#)

**End Problem 13-3 , 110 min.**

### Problem 13-4: 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. 12.3.3.7]. Semi-implicit Runge-Kutta single step methods, also known as Rosenbrock-Wanner (ROW) methods [Lecture → Eq. (12.4.0.6)] just require the solution of linear systems of equations. This problem deals with a concrete ROW method, its stability and aspects of implementation.

Requires the contents of [Lecture → Section 12.1]

We consider the following autonomous ODE:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \quad (13.4.1)$$

and discretize it with a *semi-implicit* Runge-Kutta SSM, a so-called **Rosenbrock method**,

$$\begin{aligned} \mathbf{Wk}_1 &= \mathbf{f}(\mathbf{y}_0), \\ \mathbf{Wk}_2 &= \mathbf{f}\left(\mathbf{y}_0 + \frac{1}{2}h\mathbf{k}_1\right) - ah\mathbf{Jk}_1, \\ \mathbf{y}_1 &= \mathbf{y}_0 + h\mathbf{k}_2, \end{aligned} \quad (13.4.2)$$

where

$$\mathbf{J} = D\mathbf{f}(\mathbf{y}_0), \quad \mathbf{W} = \mathbf{I} - ah\mathbf{J}, \quad a = \frac{1}{2 + \sqrt{2}}.$$

**(13-4.a)** □ (15 min.) Compute the stability function  $S$  of the Rosenbrock method (13.4.2), that is, compute the (rational) function  $S(z)$ , such that

$$y_1 = S(z)y_0, \quad z := h\lambda,$$

when we apply the method to perform one step of size  $h$ , starting from  $y_0$ , of the linear scalar model ODE  $\dot{y} = \lambda y, \lambda \in \mathbb{C}$ .

SOLUTION for (13-4.a) → [13-4-1-0:SemI1s.pdf](#)

**(13-4.b)** □ (15 min.) [ depends on Sub-problem (13-4.a) ]

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  $|z| \rightarrow 0$ ? Deduce the maximal possible order of the method Eq. (13.4.2).

HIDDEN HINT 1 for (13-4.b) → [13-4-2-0:SemI2h.pdf](#)

SOLUTION for (13-4.b) → [13-4-2-1:SemI2s.pdf](#)

**(13-4.c)** □ (15 min.) Implement a C++ function:

```
template <class Function, class Jacobian, class StateType>
std::vector<StateType> solveRosenbrock(
    Function &&f, Jacobian &&df,
    const StateType &y0, unsigned int N, double T);
```

that applies the Rosenbrock method (13.4.2) 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)  $\mathbf{y}_0 = \mathbf{y}(0)$ , a number of steps  $N$  and a final time  $T$ . The function returns the sequence of states generated by the single step method up to  $t = T$ , using  $N$  equidistant steps of the Rosenbrock method.

SOLUTION for (13-4.c) → [13-4-3-0:SemI3s.pdf](#)



(13-4.d) (15 min.) [ depends on Sub-problem (13-4.c) ]

Explore the order of the method (13.4.2) empirically by applying it to the IVP for the limit cycle [Lecture → Ex. 12.2.0.4]:

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2) \mathbf{y}, \quad (13.4.4)$$

with  $\lambda = 1$  and initial state  $\mathbf{y}_0 = [1, 1]^\top$  on  $[0, 10]$ . Use uniform timesteps of size  $h = 2^{-k}, k = 4, \dots, 10$  and compute a reference solution  $\mathbf{y}_{\text{ref}}$  with timestep size  $h = 2^{-12}$ . Monitor the maximal error on the temporal mesh

$$\max_j \|\mathbf{y}_j - \mathbf{y}_{\text{ref}}(t_j)\|_2,$$

and use it to estimate a rate of algebraic convergence by means of linear regression.

To do this write a C++ function

```
double cvgRosenbrock(void);
```

that tabulates the errors and returns the estimated rate of convergence.

SOLUTION for (13-4.d) → [13-4-4-0:SemI4s.pdf](#)



In complex analysis you might have heard about the maximum principle for holomorphic/analytic functions. The following is a special version of it.

### Theorem 13.4.5. Maximum principle for holomorphic functions

Let

$$\mathbb{C}^- := \{z \in \mathbb{C} \mid \operatorname{Re}(z) < 0\}.$$

Let  $f : D \subset \mathbb{C} \rightarrow \mathbb{C}$  be non-constant, defined on  $\overline{\mathbb{C}^-}$ , and analytic in  $\mathbb{C}^-$ . Furthermore, assume that  $w := \lim_{|z| \rightarrow \infty} f(z)$  exists and  $w \in \mathbb{C}$ , then:

$$\forall z \in \mathbb{C}^-: |f(z)| < \sup_{\tau \in \mathbb{R}} |f(i\tau)|.$$

(13-4.e) (20 min.) Appealing to Thm. 13.4.5 show that the method (13.4.2) is  $L$ -stable (cf. [Lecture → § 12.3.4.11]).

HIDDEN HINT 1 for (13-4.e) → [13-4-5-0:SemI5h.pdf](#)

SOLUTION for (13-4.e) → [13-4-5-1:SemI5s.pdf](#)



**End Problem 13-4 , 80 min.**

### Problem 13-5: 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 **gradient-flow ODE**

$$\dot{\mathbf{y}}(t) = -\mathbf{grad} V(\mathbf{y}(t)) , \quad (13.5.1)$$

where  $V : D \subset \mathbb{R}^d \rightarrow \mathbb{R}$  is a continuously differentiable function.

(13-5.a)  $\square$  (10 min.)

Show that for a solution  $t \mapsto \mathbf{y}(t)$  of (13.5.1) defined on  $I \subset \mathbb{R}$  the function  $t \mapsto V(\mathbf{y}(t))$  is non-increasing.

SOLUTION for (13-5.a) → [13-5-1-0:s1.pdf](#)

(13-5.b)  $\square$  (6 min.)

The scalar ( $d = 1$ ) linear ODE  $\dot{y} = -\lambda y$ ,  $\lambda \in \mathbb{R}$ , belongs to the class of gradient-flow ODEs. What is  $V$  in this case?

SOLUTION for (13-5.b) → [13-5-2-0: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

$c$	$\alpha$	$b^\top$	$\hat{=}$	$\begin{array}{c cccccc} 1 & 1 & 0 & 0 & 0 & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\ \frac{3}{4} & \frac{1}{2} & \frac{1}{4} & 0 & 0 & 0 \\ \hline \frac{11}{20} & \frac{17}{50} & -\frac{1}{25} & \frac{1}{4} & 0 & 0 \\ \frac{1}{2} & \frac{371}{1360} & -\frac{137}{2720} & \frac{15}{544} & \frac{1}{4} & 0 \\ \hline 1 & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \\ \hline & \frac{25}{24} & -\frac{49}{48} & \frac{125}{16} & -\frac{85}{12} & \frac{1}{4} \end{array}$	(13.5.2)
-----	----------	----------	-----------	--	----------

In the file `gradientflow.hpp` you find a function

`Eigen::MatrixXd ButcherMatrix();`

that gives you the  $6 \times 5$ -matrix  $\begin{bmatrix} \alpha \\ b^\top \end{bmatrix}$  of the Butcher scheme (13.5.2). Its bottom row is the vector  $b^\top$ .

(13-5.c)  $\square$  (10 min.)

Give the **stage form** [Lecture → Eq. (12.3.3.6)] of the increment equations for the implicit Runge-Kutta single-step method (timestep  $h > 0$ ) described by the Butcher scheme (13.5.2) and applied to a generic autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  with  $\mathbf{f} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ ,  $d \in \mathbb{N}$ .

SOLUTION for (13-5.c) → [13-5-3-0:s4.pdf](#)

(13-5.d)  (30 min.) [ depends on Sub-problem (13-5.c) ]

Implement a C++ function

```
template <typename Functor, typename Jacobian>
std::array<Eigen::VectorXd, 5> computeStages(
    Functor &&f, Jacobian &&J,
    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*  $\mathbf{g}_i$  ( $\rightarrow$  [Lecture  $\rightarrow$  Eq. (12.3.3.6)]) required for one step of size  $h > 0$  of the implicit Runge-Kutta single-step method described by the Butcher scheme (13.5.2) and applied to a generic autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  with  $\mathbf{f} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ ,  $d \in \mathbb{N}$ .

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.

SOLUTION for (13-5.d)  $\rightarrow$  [13-5-4-0:s5.pdf](#)



(13-5.e)  (10 min.) [ depends on Sub-problem (13-5.d) ]

Based on `computeStages()` realize a C++ function

```
template <typename Functor, typename Jacobian>
Eigen::VectorXd discEvolSDIRK(
    Functor &&f, Jacobian &&J,
    const Eigen::VectorXd &y, double h,
    double rtol = 1E-6, double atol = 1E-8);
```

that evaluates the discrete evolution operator  $\Psi^h$  for the implicit Runge-Kutta single-step method described by the Butcher scheme (13.5.2) and applied to a generic autonomous ODE  $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$  with  $\mathbf{f} : D \subset \mathbb{R}^d \mapsto \mathbb{R}^d$ ,  $d \in \mathbb{N}$ . Of course, the parameters  $y$  and  $h$  supply the arguments  $\mathbf{y}$  and  $h$  for  $\Psi$ . The meaning of  $f$ ,  $J$ ,  $rtol$  and  $atol$  has been explained before

SOLUTION for (13-5.e)  $\rightarrow$  [13-5-5-0:s6.pdf](#)



Finally we consider the concrete gradient-flow ODE (13.5.1) spawned by the potential function

$$V(\mathbf{y}) = \sin(\|\mathbf{y}\|^2) + \lambda(\mathbf{d}^\top \mathbf{y})^2, \quad \lambda \in \mathbb{R}, \quad \mathbf{y}, \mathbf{d} \in \mathbb{R}^d, \quad \|\mathbf{d}\| = 1. \quad (13.5.10)$$

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  $[0, 0.1]$ .

(13-5.f)  (10 min.)

Write down the autonomous gradient-flow ODE induced by  $\mathbf{y} \mapsto V(\mathbf{y})$  from (13.5.10).

SOLUTION for (13-5.f)  $\rightarrow$  [13-5-6-0:s6a.pdf](#)



(13-5.g)  (20 min.) [ depends on Sub-problem (13-5.f) ]

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 (13-5.g)  $\rightarrow$  [13-5-7-0:h7s.pdf](#)

SOLUTION for (13-5.g)  $\rightarrow$  [13-5-7-1:s7.pdf](#)



(13-5.h)  (15 min.) [ depends on Sub-problem (13-5.e) and Sub-problem (13-5.f) ]

Based on your implementation of `discEvolSDIRK()` write a C++ function

```
std::vector<Eigen::VectorXd>
solveGradientFlow(
    const Eigen::VectorXd &d, double lambda,
    const Eigen::VectorXd &y,
    double T, unsigned int N);
```

that uses  $N$  equidistant steps of the SDIRK RK-SSM with the Butcher scheme (13.5.2) to solve the gradient-flow ODE (13.5.1) spawned by  $V$  from (13.5.10) up to final time  $T > 0$ . The arguments  $d$  and  $\lambda$  supply the parameters, and  $y$  the initial state  $y^{(0)}$ . Use the default tolerance parameters for Newton's method as specified in the signature of `computeStages()`.

HIDDEN HINT 1 for (13-5.h) → 13-5-8-0:h8lf.pdf

SOLUTION for (13-5.h) → 13-5-8-1:s8.pdf ▲

(13-5.i)  (5 min.)

We have solved the gradient-flow ODE (13.5.1) with  $V$  from (13.5.10) and  $d = 2$ ,  $d = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ ,  $\lambda = 10$ ,  $y_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ ,  $T = 0.1$ .

$N$	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 norm of the error  $y_N - y(T)$  at final time for different numbers  $N$  of equidistant timesteps.

Which empirical order of convergence of the SDIRK single-step method do the data suggest?

SOLUTION for (13-5.i) → 13-5-9-0:s9.pdf ▲

End Problem 13-5 , 116 min.



---

SOLUTION of (1-1.a):

You have to add **#include <Eigen/Dense>** in the beginning of every file that uses EIGEN tools. Of course the compiler must be told the directory, where EIGEN's header files reside.

---

HINT 1 for (1-1.b): The test case implemented in main.cpp is smallTriangular(1, 2, 3) and returns

$$\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix}.$$

□

---

## SOLUTION of (1-1.b):

Matrix initialization with the `<<` operator [Lecture → § 1.2.1.3] is convenient. However also the entry access operator `double &operator (int, int)` can be used directly.

### C++ code 1.1.1: Initialization of fixed size $2 \times 2$ -matrix

Get it on  GitLab (MatrixClass.hpp).

```
2 Eigen::Matrix<double, 2, 2> smallTriangular(double a, double b, double c) {
3     /*
4      * This function returns a 2 by 2 triangular matrix of doubles a, b, c.
5      */
6
7     // We know in advance that we want to create a matrix of doubles with
8     // a fixed
9     // size of 2 by 2. Therefore, we pass the parameters <double, 2, 2>
10    // to the
11    // template class Eigen::Matrix.
12    Eigen::Matrix<double, 2, 2> A;
13
14    // We have declared the variable A of the type
15    // Eigen::Matrix<double, 2, 2>,
16    // but we have not initialized its entries.
17    // We can do this using comma-initialization:
18    A << a, b, 0, c;
19
20    // Question: Is A now an upper triangular matrix, or a lower
21    // triangular
22    // matrix?
23
24    return A;
25 }
```

HINT 1 for (1-1.c): The test call `constantTriangular(3, 20)` from `main.cpp` should yield the matrix

$$\begin{bmatrix} 20 & 20 & 20 \\ 0 & 20 & 20 \\ 0 & 0 & 20 \end{bmatrix}.$$

□

## SOLUTION of (1-1.c):

Of course, one can simply set the matrix entries using the direct access to matrix entries and two nested loops.

### C++ code 1.1.2: Initialization of upper triangular part of a matrix Get it on GitLab (MatrixClass.hpp).

```
2 Eigen::MatrixXd constantTriangular(int n, double val) {
3     /*
4      * This function returns an n by n upper triangular matrix with the constant
5      * value val in the upper triangular part.
6      */
7
8     // Now we do not know the size of our matrix at compile time.
9     // Hence, we use the special value Eigen::Dynamic to set the size of A.
10    Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A;
11
12    // Eigen::Matrix has a method Zero(n_rows,n_cols) which returns the n_rows by
13    // n_cols matrix whose entries are all equal to zero.
14    A = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>::Zero(n, n);
15
16    // To get a triangular matrix, we must set the values above the diagonal to
17    // val. This we can do by a nested for-loop.
18    for (int row = 0; row < n; row++) {
19
20        // TO DO: Write the inner for-loop.
21        // Hint: We can access and change the entries of A using parentheses, i.e.
22        // A(row,col) = val; Note that indexing starts at the value 0 (as usual),
23        // and not 1 (as in Matlab).
24
25        // START
26        for (int col = row; col < n; col++) {
27            A(row, col) = val;
28        }
29        // END
30    }
31    return A;
32 }
```

Alternative: One can also use the `triangularView()` member function for `Eigen::Matrix`, see [Lecture → Code 1.2.1.6],  EIGEN documentation.

## SOLUTION of (1-1.d):

If in EIGEN you want to mix matrices/vectors with different scalar types in arithmetic expressions, you have to do explicit typecasts using the `cast()` methods.

### C++ code 1.1.3: Casting the scalar type of a matrix

Get it on  GitLab ([MatrixClass.hpp](#)).

```
2  double casting() {
3      /*
4      * This function does not do anything meaningful.
5      * It is only meant to introduce vectors and how to typecast.
6      */
7
8      // Because the syntax Eigen::Matrix< type, n_rows, n_cols > is very
9      // cumbersome, Eigen provides convenience classes that work as shorthands.
10     // For example, Eigen::Matrix2d is shorthand for Eigen::Matrix< double, 2, 2
11     // >.
12
13     // Vectors are just a special type of matrices that have only one column.
14     // Thus, VectorXi is a shorthand for Eigen::Matrix< int, Eigen::Dynamic, 1 >.
15     // Constant(2,1) creates a vector of size 2 and initializes the entries with
16     // the value 1.
17     Eigen::VectorXi u = Eigen::VectorXi::Constant(2, 1);
18
19     // std::complex is a template class for complex numbers.
20     // Here we declare two complex numbers, with real and imaginary parts
21     // represented by doubles. z0 = 1 - i z1 = 5 + i
22     std::complex<double> z0(1, -1), z1(5, 1);
23
24     // Declare and initialize a size 2 vector of std::complex<double>.
25     Eigen::Vector2cd v;
26     v(0) = z0;
27     v(1) = z1;
28
29     double x;
30     // TO DO: Calculate the inner product of u and v, and store the result in x.
31     // Hint: The inner product of two vectors is given by u.dot(v), but
32     // first we need to cast the "int" vector u to a "std::complex<double>" vector.
33     // Use u.cast< NEW TYPE >() to achieve this. The result of the inner
34     // product will be 1*(1-i) + 1*(5+i) = 6 + 0i, a real number. You can get the
35     // real part of an std::complex<double> using the method "real()". START
36     std::complex<double> z = u.cast<std::complex<double>>().dot(v);
37     x = z.real();
38     // END
39     return x;
40 }
```

HINT 1 for (1-1.e): The test call in `main.cpp` is `arithmetics(4)` and should return the vector

$$\begin{bmatrix} -41.4706 + 27.1176i \\ -23.7692 + 42.8462i \\ -6.47692 + 48.4154i \\ 7.2 + 48.4i \end{bmatrix}.$$

□

SOLUTION of (1-1.e):

Refer to the comments in the code and notice the use of the `cast()` method.

#### C++ code 1.1.4: Evaluating a simple matrix-vector expression

Get it on  GitLab (MatrixClass.hpp).

```
2 Eigen::VectorXcd arithmetics(int n) {
3     /*
4      * This function does not do anything meaningful.
5      * It is only meant to show some simple Eigen arithmetics and matrix
6      * initializations.
7      */
8
9     // This declares dynamic size (signified by the letter 'X') matrices of
10    // complex numbers (signified by the letter c) with real and imaginary
11    // parts represented by doubles (signified by 'd').
12    Eigen::MatrixXcd A, C, I;
13
14    // We initialize the matrices arbitrarily:
15    // an n by n lower triangular matrix,
16    A = constantTriangular(n, 5.0).transpose().cast<std::complex<double>>();
17    // The n by n identity matrix,
18    I = Eigen::MatrixXcd::Identity(n, n);
19
20    // Declare the n by n matrix B.
21    Eigen::MatrixXcd B(n, n);
22
23    // TO DO: Fill in the matrix B such that B(k,l) = (k+l*i)/(k-l*i),
24    // where i is the imaginary unity, i^2 = -1.
25    // START
26    for (int k = 0; k < n; k++) {
27        for (int l = 0; l < n; l++) {
28            std::complex<double> tmp(k + 1, l + 1);
29            B(k, l) = tmp / std::conj(tmp);
30        }
31    }
32    // END
33
34    // We can perform arithmetics on matrices: +, -, *
35    // Note that for + and -, the left hand matrix and the right hand matrix have
36    // to be the same size, and that matrix multiplication * is only defined if
37    // the number of columns in the left hand matrix is the same as the number of
38    // rows in the right hand matrix.
39    C = B * (A - 5.0 * I);
40
41    // Eigen::VectorXcd is shorthand for
42    // Eigen::Matrix<std::complex<double>, Eigen::Dynamic, 1>.
43    // Hence, all the same arithmetics work for vectors.
44    Eigen::VectorXcd u, v;
45
46    // TO DO: Initialize the entries of u as the integers from 1 to n,
47    // multiply u by the matrix C, and store the result in v.
48    // START
49    u = Eigen::VectorXcd::LinSpaced(n, 1, n); // Or use a for-loop.
50    v = C * u;
```

```
51 // END  
52  
53     return v;  
54 }
```

---

HINT 1 for (1-2.a): In **CODEEXPERT** the following test case is implemented in the `main.cpp` file:  
`zero_row_col(Matrix3d::Constant(-1), 0, 1);`, which should produce the matrix

$$\begin{bmatrix} 0 & 0 & 0 \\ -1 & 0 & -1 \\ -1 & 0 & -1 \end{bmatrix}.$$

□

---

SOLUTION of (1-2.a):

Use the write access methods `row()` and `col()` for a matrix.

### C++ code 1.2.1: Initialization of fixed size $2 \times 2$ -matrix

Get it on  GitLab (`MatrixBlocks.hpp`).

```
2 MatrixXd zero_row_col(const MatrixXd &A, int p, int q) {
3     /*
4      * This function takes a matrix A, and returns a matrix that is exactly the
5      * same, except that row p and column q are set to zero.
6      */
7     // Make a copy of A.
8     MatrixXd Anew(A);
9
10    // TO DO: Set the entries of row number p and column number q to zero.
11    // Hint: We can access rows and columns of A by A.row() and A.col().
12    // The setZero() is useful here. START
13    Anew.row(p).setZero();
14    Anew.col(q).setZero();
15    // END
16
17    return Anew;
18}
```

HINT 1 for (1-2.b): The test call issued in `main.cpp` is `swap_left_right_blocks(MatrixXd::Identity (4,3),2)` and yields the matrix

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

□

## SOLUTION of (1-2.b):

We can use the write-access version of the `block()` method of matrix data types.

### C++ code 1.2.2: Swapping left and right parts of a matrix in EIGEN.

Get it on  GitLab (MatrixBlocks.hpp).

```
2 MatrixXd swap_left_right_blocks(const MatrixXd &A, int p) {
3     /*
4      * Writing as a block matrix  $A = [B \ C]$ , where  $B$  denotes the first  $p$  columns of
5      *  $A$ , and  $C$  denotes the  $q = (A.cols() - p)$  last columns, this function returns
6      *  $D = [C \ B]$ .
7      */
8
9     MatrixXd B, C;
10
11    // We can use .rows() and .cols() to get the number of rows and
12    // columns in A.
13    int q = A.cols() - p;
14
15    // A.block( i, j, m, n ) returns the  $m$  by  $n$  block that has its
16    // top-left corner
17    // at the index  $(i, j)$  in A. Hence, the first  $p$  columns of A can be
18    // accessed in
19    // the following way:
20    B = A.block(0, 0, A.rows(), p);
21
22    // TO DO: Use A.block() to define C as the matrix containing the last
23    // q
24    // columns of A. START
25    C = A.block(0, p, A.rows(), q);
26    // END
27
28    // Make a copy of A.
29    MatrixXd Anew(A);
30    // The block() method can access arbitrary blocks within a matrix.
31    // For our purposes, it is actually simpler to use leftCols() and
32    // rightCols().
33    Anew.leftCols(q) = C;
34
35    // TO DO: Use A.rightCols() to fill in the remaining columns of the
36    // new matrix
37    // A. START
38    Anew.rightCols(p) = B;
39    // END
40
41    // Tip: Many more methods exist that are special cases of block(),
42    // e.g. topRows(), bottomRows(), topLeftCorner(), bottomLeftCorner(),
43    // For vectors we have head(), tail(), and segment().
44
45    return Anew;
46 }
```

HINT 1 for (1-2.c): As test case invoked from `main.cpp` we use tridiagonal  $(4, -1, 2, -1)$ ; and should get

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix}$$

□

---

## SOLUTION of (1-2.c):

The code below use a special submatrix access function of EIGEN: `diagonal()`. However, a simple loop-based implementation based on entry access via `operator ()` is hardly less efficient.

### C++ code 1.2.3: Initialization of a triangular matrix in EIGEN.

Get it on  GitLab (MatrixBlocks.hpp).

```
2 MatrixXd tridiagonal(int n, double a, double b, double c) {
3     /*
4      * This function creates an n by n tridiagonal matrix with the values
5      * a on the first subdiagonal,
6      * b on the diagonal, and
7      * c on the first superdiagonal.
8      * Example for n=5:
9      *      [ b c 0 0 0 ]
10     *      [ a b c 0 0 ]
11     * A = [ 0 a b c 0 ]
12     *      [ 0 0 a b c ]
13     *      [ 0 0 0 a b ]
14     */
15
16     MatrixXd A;
17
18     // TO DO: Fill the matrix A with zeros. Then fill the subdiagonal
19     // with the
20     // value a, the diagonal with b and the superdiagonal with c. Hint:
21     // You can
22     // get the diagonal of A by A.diagonal(). Moreover, you can get the
23     // super- and
24     // subdiagonals by passing +1 or -1 as arguments to A.diagonal().
25     // START
26     A = MatrixXd::Zero(n, n);
27     A.diagonal(-1) = VectorXd::Constant(n - 1, a);
28     A.diagonal() = VectorXd::Constant(n, b);
29     A.diagonal(1) = VectorXd::Constant(n - 1, c);
30     // END
31     return A;
32 }
```

HINT 1 for (1-3.a): Read about the `sum()` method available for matrices in EIGEN, see  [EIGEN documentation](#)

HINT 2 for (1-3.a): From main.cpp we call average(Matrix3d::Identity()) and should get 0.333333. ↴

---

SOLUTION of (1-3.a):

**C++ code 1.3.1: Computing the average of matrix entries**

Get it on  [GitLab \(MatrixReduce.hpp\)](#).

```
2 double average(const MatrixXd &A) {  
3     /*  
4      * Calculates the average of the entries of A  
5      */  
6     double m;  
7     m = A.sum() / A.size();  
8     return m;  
9 }
```

HINT 1 for (1-3.b): Gather information about Boolean reductions from  EIGEN documentation.

HINT 2 for (1-3.b): The test call `percent_zero(Matrix3d::Identity())` in `main.cpp` should result in **66.6667**. ↳

---

## SOLUTION of (1-3.b):

Note the expression `(Arr == 0).count()` and that the `size()` method returns the total number of entries for a dense matrix.

### C++ code 1.3.2: Determines the percentage of (exactly) zero entries Get it on GitLab (MatrixReduce.hpp).

```
2 double percent_zero(const MatrixXd &A) {
3     /*
4      * Calculates how many entries of A are exactly equal to zero,
5      * as a percentage of the total number of entries.
6      */
7
8     // Eigen provides an Array class, that is similar to Matrix,
9     // but has operations that are entry-wise rather than linear algebra
10    // operations. For example, multiplying two Arrays is entry-wise
11    // multiplication, and not matrix multiplication.
12    // The benefit of using an Array here is that we can do entry-wise
13    // boolean
14    // comparison, which is not available for matrices.
15    ArrayXXd Arr = A.array();
16
17    double ratio;
18    // TO DO: Calculate the ratio of zero entries in A.
19    // START
20    int zeros = (Arr == 0).count();
21    ratio = ((double)zeros) / A.size();
22    // END
23
24    return ratio * 100;
}
```

HINT 1 for (1-3.c): You can use **partial reductions** for your code, see  **EIGEN** documentation.

□

HINT 2 for (1-3.c): For `B=MatrixXd::Identity(4,5)` `has_zero_column(B)` should yield true, while `has_zero_column(B.transpose())` must return false. This is the test implemented in `main.cpp`. ↴

SOLUTION of (1-3.c):

EIGEN's `any()` method checks whether an expression evaluates to **true** for a single entry of a matrix.

### C++ code 1.3.3: Checking for an exactly vanishing column

Get it on  GitLab (MatrixReduce.hpp).

```
2  bool has_zero_column(const MatrixXd &A) {
3  /*
4   * Returns 1 if A has a column that is exactly equal to zero.
5   * Returns 0 otherwise.
6   */
7
8  bool result;
9
10 // A vector is the zero vector if and only if it has norm 0.
11 // The following vector contains the squared norms of the columns of
12 // A.
13 VectorXd norms = A.colwise().squaredNorm();
14 // The norm is 0 if and only if the norm squared is zero.
15 // We use squaredNorm() instead of norm(), because norm() =
16 // sqrt(squaredNorm()) calculates a square root that is not necessary
17 // for our
18 // purposes.
19
20 // TO DO: Check if any one of the norms is equal to zero.
21 // Hint: Use an array to perform entry-wise comparison.
22 // START
23 result = (norms.array() == 0).any();
24 // END
25
26 return result;
27 }
```

HINT 1 for (1-3.d): The test case implemented in `main.cpp` calls `columns_sum_to_zero(C)` where `C=Matrix3d::Random() + Matrix3d::Constant(1)`. Note that the matrix **C** is not really random, because the seed for the “random” number generator is set to a particular value. Thus we always get the matrix

$$\begin{bmatrix} -0.721194 & 0.160061 & 0.561133 \\ 0.0929355 & -0.989846 & 0.896911 \\ 1.98551 & 0.159289 & -2.1448 \end{bmatrix}$$

□

---

## SOLUTION of (1-3.d):

The solution gives a very compact implementation using submatrix access via `diagonal()` and the `rowwise` modifier for an ensuing reduction operation.

### C++ code 1.3.4: Setting diagonals to make row sums vanish

Get it on  [GitLab \(MatrixReduce.hpp\)](#).

```
2 MatrixXd columns_sum_to_zero(const MatrixXd &A) {
3     /*
4      * Returns a matrix that is like A, but the entries on the diagonal
5      * have been changed so that the sum of the columns is equal to 0.
6      */
7
8     MatrixXd B(A);
9
10    // TO DO: Replace the diagonal of B with values such that the columns
11    // of B sum
12    // up to zero. Hint: Use diagonal(), rowwise(), and sum(). START
13    int p = std::min(B.rows(), B.cols());
14    B.diagonal() = VectorXd::Zero(p);
15    B.diagonal() = -B.rowwise().sum();
16    // END
17
18    return B;
}
```

HINT 1 for (2-1.a): First learn how EIGEN overloads the <<-operator for the initialization of matrices, see [Lecture → § 1.2.1.3].

---

SOLUTION of (2-1.a):

From code `arrowmatvec.cpp` ([Get it on GitLab \(arrowmatvec.cpp\)](#).) we deduce that the argument vectors are copied into the diagonal, the bottom row and rightmost column of the matrix  $\mathbf{A}$ , which yields

$$\mathbf{A} = \begin{bmatrix} d_1 & & & a_1 \\ & d_2 & & a_2 \\ & & \ddots & \vdots \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \quad (2.1.2)$$

From this the pattern is clear.

---

---

SOLUTION of (2-1.b):

The standard matrix-matrix multiplication has complexity  $\mathcal{O}(n^3)$  and the standard matrix-vector multiplication has complexity  $\mathcal{O}(n^2)$ . Hence, the overall computational complexity is dominated by  $\mathcal{O}(n^3)$ .

In this case, the line affecting the complexity is  $y = A \cdot A \cdot x$ .

Remember that most of C++ operators have a **left-to-right** precedence. This means that, even if, mathematically,  $(A \cdot A) \cdot y = A \cdot (A \cdot y)$ , when implementing the expression in C++, the *complexity* of the code will be very different. The left hand side has complexity  $\mathcal{O}(n^3)$ , whilst the right hand side has complexity  $\mathcal{O}(n^2)$ .

Remark: Even EIGEN's own internal template expression mechanism is not able to efficiently exploit this fact.

---

## SOLUTION of (2-1.c):

Due to the special structure of the matrix, it is possible to write a function that is more efficient than the standard matrix-vector multiplication.

Define  $\mathbf{a}_- := (a_{(1)}, \dots, a_{(n-1)}, 0)$ . We can rewrite  $\mathbf{A}$  as follows:

$$\mathbf{A} = \begin{bmatrix} d_1 & & & a_1 \\ & d_2 & & a_2 \\ & & \ddots & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \quad (2.1.3)$$

$$= \text{diag}(\mathbf{d}) + [0, \dots, 0, \mathbf{a}_-] + [0, 0, 0, \mathbf{a}_-]^T =: \mathbf{D} + \mathbf{V} + \mathbf{H}. \quad (2.1.4)$$

and  $\mathbf{Ax} = \mathbf{Dx} + \mathbf{Vx} + \mathbf{Hx}$ . Each of these multiplications can obviously be done in  $O(n)$  operations, see ??.

### C++11-code 2.1.5: Solution of ??

```
2 void efficient_arrow_matrix_2_times_x(const VectorXd &d, const VectorXd &a,
3                                     const VectorXd &x, VectorXd &y) {
4     assert(d.size() == a.size() && a.size() == x.size() &&
5            "Vector size must be the same!");
6     int n = d.size();
7
8     // Notice that we can compute (A*A)*x more efficiently using
9     // A*(A*x). This is, in fact, performing two matrix vector
10    // multiplications
11    // instead of a more expensive matrix-matrix multiplication.
12    // Therefore, as first step, we need a way to efficiently
13    // compute A*x
14
15    // This function computes A*x. you can use it
16    // by calling A_times_x(x).
17    // This is the syntax for lambda functions: notice the extra
18    // [variables] code. Each variable written within [] brackets
19    // will be captured (i.e. seen) inside the lambda function.
20    // Without &, a copy of the variable will be performed.
21    // Notice that A = D + H + V, s.t. A*x = D*x + H*x + V*x
22    // D*x can be rewritten as d*x componentwise
23    // H*x is zero, except at the last component
24    // V*x is only affected by the last component of x
25    auto A_times_x = [&a, &d, n](const VectorXd &x) {
26        // This takes care of the diagonal (D*x)
27        // Notice: we use d.array() to tell Eigen to treat
28        // a vector as an array. As a result: each operation
29        // is performed componentwise.
30        VectorXd Ax = (d.array() * x.array()).matrix();
31        // H*x only affects the last component of A*x
32        // This is a dot product between a and x with the last
33        // component removed
34        Ax(n - 1) += a.head(n - 1).dot(x.head(n - 1));
```

```
35     // V*x is equal to the vector
36     // (a(0)*x(n-1), ..., a(n-2)*x(n-1), 0)
37     Ax.head(n - 1) += x(n - 1) * a.head(n - 1);
38     return Ax;
39 }
40
41 // <=> y = A*(A*x)
42 y = A_times_x(A_times_x(x));
43 }
```

Get it on  GitLab (arrowmatvec.cpp).

---

Note the use of lambda functions [Lecture → Section 0.3.3] in order to achieve a streamlined implementation. Also note the use of the method `head` to access the top components of a vector.

---

SOLUTION of (2-1.d):

The efficient implementation only needs two vector-vector element-wise multiplications, a dot-product, and two vector-scalar multiplication. Each of them has complexity  $\mathcal{O}(n)$ . Therefore the complexity is  $\mathcal{O}(n)$ .

---

SOLUTION of (2-1.e):

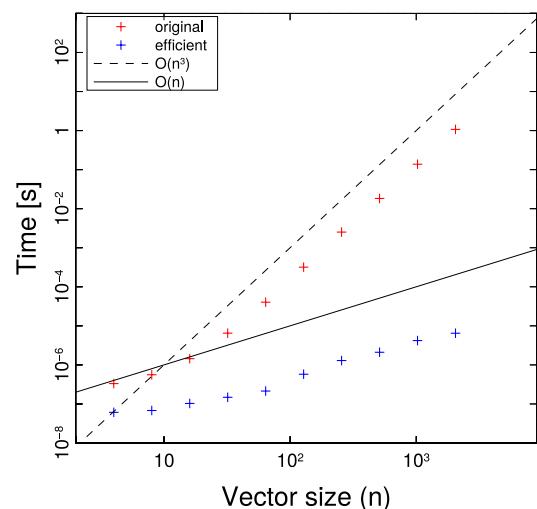
## Comparison of timings (Arrow Matrix)

The standard matrix multiplication has runtime growing with  $O(n^3)$ . The complexity of the more efficient implementation is  $O(n)$ . See ?? and ??, with data points almost lying on straight lines in doubly logarithmic plot. The slope of these lines indicates the power of  $n$ .

Comparison of runtimes of the two implementations in `arrowmatvec.cpp`



(The file `arrowmatvec.cpp` also contains an implementation using `chrono`.)



### C++11-code 2.1.10: Computation of timings of arrow matrix multiplication

```
2
3     std::cout << std::setw(8) << "n" << std::setw(15) << "original"
4             << std::setw(15) << "efficient" << std::endl;
5     for (unsigned n = 4; n <= 2048; n *= 2) {
6         // Number of repetitions
7         unsigned int repeats = 10;
8
9         Timer timer, timer_eff;
10        // Repeat test many times
11        for (unsigned int r = 0; r < repeats; ++r) {
12            // Create test input using random vectors
13            Eigen::VectorXd a = Eigen::VectorXd::Random(n),
14                            d = Eigen::VectorXd::Random(n),
15                            x = Eigen::VectorXd::Random(n), y;
16
17            // Compute times
18            timer.start();
19            arrow_matrix_2_times_x(d, a, x, y);
20            timer.stop();
21
22            // Compute times for efficient implementation
23            timer_eff.start();
24            efficient_arrow_matrix_2_times_x(d, a, x, y);
25            timer_eff.stop();
26        }
27
28        // Print results
29        std::cout << std::setw(8) << n << std::scientific << std::setprecision(3)
30                << std::setw(15) << timer.min() << std::setw(15)
31                << timer_eff.min() << std::endl;
32    }
```

Get it on  GitLab (arrowmatvec.cpp).

---

The file arrowmatvec.cpp also contains an implementation using chrono.

---

SOLUTION of (2-10.a):

As  $x \rightarrow 0$ , the terms  $t := \exp(x)$  and  $1/t = \exp(-x)$  become close to each other, thereby creating cancellations errors in the return value.

---

---

SOLUTION of (2-10.b):

Given  $m \in \mathbb{N}$  and  $x \in \mathbb{R}$ , there exists  $\xi_x \in [0, x]$  such that:

$$e^x = \sum_{k=0}^m \frac{x^k}{k!} + \frac{e^{\xi_x} x^{m+1}}{(m+1)!}. \quad (2.10.2)$$

---

---

SOLUTION of (2-10.c):

The claim is equivalent to proving that  $f(x) := e^x - e^{-x} - 2x \geq 0$  for every  $x \geq 0$ . This follows from the fact that  $f(0) = 0$  and  $f'(x) = e^x + e^{-x} - 2 \geq 0$  for every  $x \geq 0$ .

---

SOLUTION of (2-10.d):

The idea is to use the Taylor expansion given in (??). Inserting this identity in the definition of the hyperbolic sine yields

$$\sinh(x) = \frac{e^x - e^{-x}}{2} = \frac{1}{2} \sum_{k=0}^m (1 - (-1)^k) \frac{x^k}{k!} + \frac{\xi_x x^{m+1} + e^{\xi_x}(-x)^{m+1}}{2(m+1)!}.$$

The parameter  $m$  gives the precision of the approximation, since  $(m+1)! \rightarrow 0$  as  $m \rightarrow \infty$  (and, therefore, the remainder converges to zero as  $m \rightarrow \infty$ ). We will choose  $m$  later to obtain the desired error. Since  $1 - (-1)^k = 0$  when  $k$  is even, we set  $m = 2n$  for some  $n \in \mathbb{N}$  to be chosen later. From the above expression we obtain the new approximation given by

$$y_n(x) := \frac{1}{2} \sum_{k=0}^{2n} (1 - (-1)^k) \frac{x^k}{k!} = \sum_{j=0}^{n-1} \frac{x^{2j+1}}{(2j+1)!},$$

with remainder

$$y_n - \sinh(x) = \frac{e^{\xi_x} x^{2n+1} - e^{\xi_{-x}} x^{2n+1}}{2(2n+1)!} = \frac{(e^{\xi_x} - e^{\xi_{-x}}) x^{2n+1}}{2(2n+1)!}.$$

Therefore, by (??) and using the inequalities  $e^{\xi_x} \leq e^x$  and  $e^{\xi_{-x}} \leq e^{-x}$ , the new relative error can be bounded by

$$\epsilon'_{n,rel} := \frac{|y_n - \sinh(x)|}{\sinh(x)} \leq \frac{e^x x^{2n}}{(2n+1)!}.$$

We are now looking for an  $n$ , s.t. the r.h.s. is less than  $10^{-15}$ , which would imply that the relative error is less than  $10^{-15}$ . Calculating the right hand sides for  $n = 1, 2, 3$  and  $x = 10^{-3}$  (note that the r.h.s. is monotonically decreasing in  $x$ ), we obtain  $1.7 \cdot 10^{-7}$ ,  $8.3 \cdot 10^{-15}$  and  $2.0 \cdot 10^{-22}$ , respectively.

In conclusion,  $y_3(x)$  gives a relative error below  $10^{-15}$ , as required. This is the formula we can use for the stable approximation of  $\sinh$ .

---

SOLUTION of (2-11.a):

As  $u/v \rightarrow -\infty$  or when  $v = 0$  and  $u < 0$ , we have  $\sqrt{u^2 + v^2} \approx -u$ , and so cancellations may arise in  $\sqrt{u^2 + v^2} + u$ . Therefore, the implementation of  $x$  will be vulnerable to cancellation. Similarly, as  $u/v \rightarrow +\infty$  or when  $v = 0$  and  $u > 0$ , we have  $\sqrt{u^2 + v^2} \approx u$ , and so cancellations may arise in  $\sqrt{u^2 + v^2} - u$ . Therefore, the implementation of  $y$  will be vulnerable to cancellation.

In conclusion, the implementation of these formulas will be vulnerable to cancellation when  $|u|$  is much bigger than  $v$ .

---

---

SOLUTION of (2-11.b):

An immediate calculation shows that  $xy = v/2$ .

---

---

SOLUTION of (2-11.c):

Cf. complexroot.cpp.

### C++-code 2.11.3: Description

---

Get it on ❤️ GitLab (complexroot.cpp).

---

---

SOLUTION of (2-12.a):

The matrices  $\mathbf{U}_A + \mathbf{D}_A$  and  $\mathbf{U}_A + \mathbf{D}_A$  must be invertible, which is equivalent to  $\mathbf{D}_A$  being invertible. This is equivalent to the matrix  $\mathbf{A}$  having no zeros on the diagonal.

---

---

SOLUTION of (2-12.b):

Let  $\mathbf{x}$  be such fixed point, then

$$\mathbf{x} = (\mathbf{U}_A + \mathbf{D}_A)^{-1}\mathbf{b} - (\mathbf{U}_A + \mathbf{D}_A)^{-1}\mathbf{L}_A(\mathbf{L}_A + \mathbf{D}_A)^{-1}(\mathbf{b} - \mathbf{U}_A\mathbf{x}) \quad (2.12.3)$$

Left-multiply by  $\mathbf{U}_A + \mathbf{D}_A = \mathbf{A} - \mathbf{L}_A$  (invertible!):

$$(\mathbf{U}_A + \mathbf{D}_A)\mathbf{x} = \mathbf{b} - \mathbf{L}_A(\mathbf{L}_A + \mathbf{D}_A)^{-1}(\mathbf{b} - \mathbf{U}_A\mathbf{x}) \quad (2.12.4)$$

Let us now define  $\mathbf{y} := (\mathbf{L}_A + \mathbf{D}_A)^{-1}(\mathbf{b} - \mathbf{U}_A\mathbf{x})$ , then:

$$(\mathbf{L}_A + \mathbf{D}_A)\mathbf{y} = \mathbf{b} - \mathbf{U}_A\mathbf{x} \quad (2.12.5)$$

$$(\mathbf{U}_A + \mathbf{D}_A)\mathbf{x} = \mathbf{b} - \mathbf{L}_A\mathbf{y} \quad (2.12.6)$$

Subtract the first equation to the second:

$$(\mathbf{L}_A + \mathbf{D}_A)\mathbf{y} + \mathbf{U}_A\mathbf{x} - (\mathbf{U}_A + \mathbf{D}_A)\mathbf{x} - \mathbf{L}_A\mathbf{y} = \mathbf{0} \quad (2.12.7)$$

$$\Leftrightarrow \mathbf{D}_A(\mathbf{y} - \mathbf{x}) = \mathbf{0} \quad (2.12.8)$$

$$\Leftrightarrow \mathbf{y} = \mathbf{x} \quad (2.12.9)$$

Using this last identity in (??) and get  $(\mathbf{U}_A + \mathbf{D}_A + \mathbf{L}_A)\mathbf{x} = \mathbf{b}$ .  $\square$

---

SOLUTION of (2-12.c):

### C++11-code 2.12.10: Code

```
2  const auto U = TriangularView<const MatrixXd, StrictlyUpper>(A);
3  const auto L = TriangularView<const MatrixXd, StrictlyLower>(A);
4
5  const auto UpD = TriangularView<const MatrixXd, Upper>(A);
6  const auto LpD = TriangularView<const MatrixXd, Lower>(A);
7
8  // A temporary vector to store result of iteration
9  VectorXd temp(x.size());
10
11 // We'll use pointer magic to
12 VectorXd* xold = &x;
13 VectorXd* xnew = &temp;
14
15 // Iteration counter
16 unsigned int k = 0;
17 double err;
18
19 #if VERBOSE
20     std::cout << std::setw(10) << "it."
21             << std::setw(15) << "err" << std::endl;
22#endif // VERBOSE
23 do {
24     // Compute next iteration step
25     *xnew = UpD.solve(b) - UpD.solve(L*LpD.solve(b - U * (*xold)));
26
27     // Absolute error
28     err = (*xold - *xnew).norm();
29 #if VERBOSE
30     std::cout << std::setw(10) << k++
31             << std::setw(15) << std::setprecision(3) << std::scientific
32             << err << std::endl;
33#endif // VERBOSE
34
35     // Swap role of previous/next iteration
36     std::swap(xold, xnew);
37 } while( err > rtol * (*xnew).norm() );
38
39 x = *xnew;
```

Get it on ❤️ GitLab (gsit.cpp).

---

SOLUTION of (2-12.d):

### C++11-code 2.12.12: Code

```
2  VectorXd x = b;
3  GSIt(A, b, x, 10e-8);
4
5  double residual = (A*x - b).norm();
6
7  std::cout << "Residual = " << residual << std::endl;
```

Get it on ❤️ GitLab ([gsit.cpp](#)).

---

---

SOLUTION of (2-12.e):

Asymptotic linear convergence can be expected after an initial phase where the error decreases much faster.

---

HINT 1 for (2-13.a): A function template in C++ is a "blueprint" for a function that can be instantiated for different types. For example, our function

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

works for arguments of type `Eigen::VectorXd`, `Eigen::VectorXd`, `Eigen::Matrix<10, 1, double>`

↓

---

## SOLUTION of (2-13.a):

Knowing the rules of matrix  $\times$  vector multiplication the loop-based implementation is straightforward.

### C++11-code 2.13.3: Code

```
2 template<class Vector>
3 void xmatmult(const Vector& a, const Vector& y, Vector& x) {
4     assert(a.size() == y.size() && a.size() == x.size()
5         && "Input vector dimensions must match");
6     unsigned n = a.size();
7
8     // looping over half of a,
9     // if n is odd we treat the middle element differently
10    for (unsigned i = 0; i < n / 2; ++i) {    //integer division
11        x(i) = a(i) * y(i) + a(n-i-1) * y(n-i-1);
12        x(n-i-1) = x(i);
13    }
14
15    if (n%2) { //n is odd
16        x(n/2) = a(n/2) * y(n/2);
17    }
18 }
```

Get it on  GitLab (xmatmult.cpp).

---

---

SOLUTION of (2-13.b):

The efficient solution only does a constant amount of operations per vector element, so it's asymptotic complexity is  $O(n)$ .

Compare this to ordinary matrix-vector multiplication which is  $O(n^2)$ .

---

HINT 1 for (2-13.c): In C++ runtimes can be measured by means of the facilities provided by the **std::chrono** library.

HINT 2 for (2-13.c): You can find a possible initialization of **A** for the ordinary multiplication in the `test()` function. □

SOLUTION of (2-13.c):

### C++11-code 2.13.4: Code

```
2 void compare_times () {
3     std::cout << "Measuring runtimes for comparison" << std::endl;
4     unsigned repeats = 3;
5     Timer t_fast, t_slow;
6     MatrixXd results(10,3);
7
8     for (unsigned k = 14; k > 4; k--) {
9         //bitshift operator '<<': 1<<3 == pow(2,3)
10        unsigned n = 1<<k;
11        VectorXd a,y,x(n);
12        a = y = VectorXd::Random(n);
13
14        MatrixXd A = a.asDiagonal();
15        for (unsigned i = 0; i < n; ++i) {
16            A(n-i-1,i) = A(i,i);
17        }
18
19        //measure multiple times
20        for (unsigned i = 0; i < repeats; i++) {
21            t_fast.start();
22            xmatmult(a,y,x);
23            t_fast.stop();
24
25            t_slow.start();
26            x = A * y;
27            t_slow.stop();
28        }
29        results(k-5,0) = n;
30        results(k-5,1) = t_slow.min();
31        results(k-5,2) = t_fast.min();
32    }
33
34    // print results
35    std::cout << std::setw(8) << "n"
36    << std::setw(15) << "original"
37    << std::setw(15) << "efficient"
38    << std::setprecision(5) << std::endl;
39    for (int i = 0; i < results.rows(); i++) {
40        std::cout << std::setw(8) << results(i,0)
41        << std::setw(15) << results(i,1) << " s"
42        << std::setw(15) << results(i,2) << " s"
43        << std::endl;
44    }
45}
```

Get it on  GitLab (xmatmult.cpp).

---

SOLUTION of (2-2.a):

If the algorithm does not terminate prematurely, the two nested loops with loop indices  $i$  and  $j$  trigger execution of the innermost loop body  $\frac{1}{2}n(n - 1)$  times. This many times an inner product of two vectors of length  $n$  has to be computed, which leads to an overall cost of  $n^3$  for  $n \rightarrow \infty$ .

---

## SOLUTION of (2-2.b):

There are many ways to code the function `gram_schmidt()`. The following code makes haevy use of EIGEN's block operations.

### C++-code 2.2.1: Gram-Schmidt orthonormalization with EIGEN

```
2 MatrixXd gram_schmidt(const MatrixXd &A) {
3     // We create a matrix Q with the same size and data of A
4     MatrixXd Q(A);
5
6     // The first vector just gets normalized
7     Q.col(0).normalize();
8     // Iterate over all other columns of A
9     for (unsigned int j = 1; j < A.cols(); ++j) {
10        // See eigen documentation for usage of col and leftCols
11        Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() * A.col(j));
12        // Normalize vector, if possible
13        // (otherwise it means columns of A are
14        // almost linearly dependant)
15        double eps = std::numeric_limits<double>::denorm_min();
16        if (Q.col(j).norm() <= eps * A.col(j).norm()) {
17            std::cerr << "Gram-Schmidt failed because "
18                  << "A has (almost) linear dependant "
19                  << "columns. Bye." << std::endl;
20            break;
21        } else {
22            Q.col(j).normalize();
23        }
24    }
25    return Q; // Efficient due to return-value optimization
26 }
```

Get it on  GitLab (`gramschmidt.cpp`).

HINT 1 for (2-2.c): A simple test of how close a matrix is to the zero matrix can rely on computing a suitable norm of the matrix. In EIGEN you may use the `norm` method of the `Eigen::Matrix` class. □

HINT 2 for (2-2.c): Remember the warning given in [Lecture → Rem. 1.5.3.15] and design your test accordingly. □

HINT 3 for (2-2.c): If a matrix  $\mathbf{Q}$  has orthonormal columns, then  $\mathbf{Q}^T \mathbf{Q} = \mathbf{I}$  (identity matrix). Such matrices are called **orthogonal**. □

SOLUTION of (2-2.c):

We compute the matrix norm  $\|\mathbf{Q}^\top \mathbf{Q} - \mathbf{I}\|_F$  (Frobenius norm, Euclidean norm of the vectorized matrix, root of the sum of the squares of the matrix entries [Lecture → Def. 3.4.4.16]). This matrix norm is returned by EIGEN's `norm()` method.

This quantity is zero when  $\mathbf{Q}$  has orthonormal columns (**orthogonal** matrix), and is a measure of how far the matrix  $\mathbf{Q}$  is from being orthogonal.

### C++-code 2.2.2: Test of Gram-Schmidt orthonormalization with EIGEN

```
2 int main(void) {
3     // Orthonormality test
4     unsigned int n = 9;
5     MatrixXd A = MatrixXd::Random(n, n);
6     MatrixXd Q = gram_schmidt(A);
7     // Compute how far is  $\mathbf{Q}^\top * \mathbf{Q}$  from the identity
8     // i.e. "How far is  $\mathbf{Q}$  from being orthonormal?"
9     double err = (Q.transpose() * Q - MatrixXd::Identity(n, n)).norm();
10    // Error has to be small, but not zero (why?)
11    std::cout << "Error is: " << err << std::endl;
12    // If the error is too big, we exit with error
13    double eps = std::numeric_limits<double>::denorm_min();
14    exit(err < eps);
15 }
```

Get it on  GitLab (gramschmidt.cpp).

---

SOLUTION of (2-3.a):

The matrix  $\mathbf{C}$  is:

$$\mathbf{C} = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}.$$

---

---

## SOLUTION of (2-3.b):

The Kronecker product has the structure:

$$\mathbf{C} = \begin{bmatrix} a_{1,1}\mathbf{B} & \dots & a_{1,n}\mathbf{B} \\ \vdots & & \vdots \\ a_{n,1}\mathbf{B} & \dots & a_{n,n}\mathbf{B} \end{bmatrix}.$$

Here, each  $a_{(i,j)}\mathbf{B} \in \mathbb{R}^{n \times n}$  (for  $0 < i, j \leq n$ ) is a Matrix (we will call this matrix a block). The proposed code loops over each block and stores the result of the product withing the matrix.

### C++-code 2.3.1: Kronecker product

```
2 void kron(const MatrixXd &A, const MatrixXd &B, MatrixXd &C) {
3     // Allocate enough space for the matrix
4     C = MatrixXd(A.rows() * B.rows(), A.cols() * B.cols());
5     // TO DO: (2-3.b) Fill in the entries of C.
6     // Hint: Use a nested for-loop and C.block().
7     // START
8     for (unsigned int i = 0; i < A.rows(); ++i) {
9         for (unsigned int j = 0; j < A.cols(); ++j) {
10            // We use eigen block operations to set the values of
11            // each  $n \times n$  block.
12            C.block(i * B.rows(), j * B.cols(), B.rows(), B.cols()) =
13                A(i, j) * B; //  $\in \mathbb{R}^{(n \times n)}$ 
14        }
15    }
16    // END
17 }
```

Get it on  GitLab (kron.cpp).

---

In fact, EIGEN itself provides an implementation of the Kronecker product, see  EIGEN documentation.

---

SOLUTION of (2-3.c):

The code loops two times from  $0$  to  $n$ , i.e. is performing the inner portion of the code  $n^2$  times. Within the for loop, the code is multiplying a  $n \times n$  matrix by a scalar, this operation has complexity  $O(n^2)$ . In total, the complexity is  $O(n^2 \cdot n^2) = O(n^4)$ .

---

HINT 1 for (2-3.d): You can exploit the structure of the matrix and reuse computations as much as possible. The tricks how to do this are explained in detail in [Lecture → Ex. 1.4.3.8].

SOLUTION of (2-3.d):

Define  $\tilde{\mathbf{x}}_j := (x_{j,n+1}, \dots, x_{(j+1)\cdot n})^\top \in \mathbb{R}^n$  (a “segment” of  $\mathbf{x}$ ) for  $j = 0, \dots, n$ ). Notice the following:

$$\mathbf{C}\mathbf{x} = \begin{bmatrix} a_{1,1}\mathbf{B} & \dots & a_{1,n}\mathbf{B} \\ \vdots & & \vdots \\ a_{n,1}\mathbf{B} & \dots & a_{n,n}\mathbf{B} \end{bmatrix} \begin{bmatrix} \tilde{\mathbf{x}}_0 \\ \vdots \\ \tilde{\mathbf{x}}_{n-1} \end{bmatrix} = \begin{bmatrix} a_{1,1}\mathbf{B}\tilde{\mathbf{x}}_0 + \dots + a_{1,n}\mathbf{B}\tilde{\mathbf{x}}_{n-1} \\ \vdots \\ a_{n,1}\mathbf{B}\tilde{\mathbf{x}}_0 + \dots + a_{n,n}\mathbf{B}\tilde{\mathbf{x}}_{n-1} \end{bmatrix}.$$

The computation of  $\mathbf{z}_j := \mathbf{B}\tilde{\mathbf{x}}_j$ , can be reused.

### C++-code 2.3.2: Kronecker product times vector

```
2 void kron_mult(const MatrixXd &A, const MatrixXd &B, const VectorXd &x,
3                 VectorXd &y) {
4     assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
5            "Matrices A and B must be square matrices with same size!");
6     assert(x.size() == A.cols() * A.cols() &&
7            "Vector x must have length A.cols()^2");
8     unsigned int n = A.rows();
9
10    // Allocate space for output
11    y = VectorXd::Zero(n * n);
12
13    // TO DO: (2-3.d) Fill in the entries of y.
14    // Hint: Use a nested for-loop, x.segment(), and y.segment().
15    // In the outer loop, you can perform a computation based on
16    // B and x, and save the result in a variable that is reused in
17    // each iteration of the inner loop.
18    // START
19
20    // Note: this is like doing a matrix-vector multiplication
21    // where the entries of the matrix are smaller matrices
22    // and entries of the vector are smaller vectors
23
24    // Loop over all segments of x (tilde{x})
25    for (unsigned int j = 0; j < n; ++j) {
26        // Reuse computation of z
27        VectorXd z = B * x.segment(j * n, n);
28        // Loop over all segments of y
29        for (unsigned int i = 0; i < n; ++i) {
30            y.segment(i * n, n) += A(i, j) * z;
31        }
32    }
33    // END
34 }
```

Get it on  GitLab (kron.cpp).

HINT 1 for (2-3.e): In EIGEN, reshaping a matrix is done using the `MatrixXd::Map` class, see and  [EIGEN documentation](#) and  [EIGEN documentation](#). A constructor for the `Map` class takes three arguments. The first is a pointer to the data, and the second and third are the shape of the matrix represented by `Map`. Important: since `Map` is accessing a raw-pointer to the data, it is very important that the data is stored in column-major format. This is because the `Map` class has no idea of the underlying structure of the matrix.

↳

---

SOLUTION of (2-3.e):

See also [Lecture → Code 1.4.3.9].

### C++-code 2.3.3: Kronecker product times vector using Map ()

```
2 void kron_reshape (const MatrixXd &A, const MatrixXd &B, const VectorXd &x,
3                      VectorXd &y) {
4     assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
5            "Matrices A and B must be square matrices with same size!");
6     unsigned int n = A.rows();
7
8     // TO DO: (2-3.e) Fill in the entires of y.
9     // Hint: Use MatrixXd::Map() to reshape x into a n by n matrix.
10    // Then y is obtained by simple matrix multiplications and
11    // another reshape.
12    // START
13    MatrixXd t = B * MatrixXd::Map(x.data(), n, n) * A.transpose();
14    y = MatrixXd::Map(t.data(), n * n, 1);
15    // END
16 }
```

Get it on  GitLab (kron.cpp).

---

## SOLUTION of (2-3.f):

The following code is an example of runtime computation:

### C++-code 2.3.4: Runtime comparison

```
2 void kron_runtime() {
3
4     MatrixXd A, B, C;
5     VectorXd x, y;
6     // We repeat each runtime measurement 10 times
7     // (this is done in order to remove outliers).
8     unsigned int repeats = 10;
9
10    std::cout << "Runtime for each implementation." << std::endl;
11    std::cout << std::setw(5) << "n" << std::setw(15) << "kron" << std::setw(15)
12        << "kron_mult" << std::setw(15) << "kron_reshape" << std::endl;
13    // Loop from M = 2,...,28
14    for (unsigned int M = 2; M <= (1 << 8); M = M << 1) {
15        Timer tm_kron, tm_kron_mult, tm_kron_map;
16        // Run experiments "repeats" times
17        for (unsigned int r = 0; r < repeats; ++r) {
18            // Random matrices for testing
19            A = MatrixXd::Random(M, M);
20            B = MatrixXd::Random(M, M);
21            x = VectorXd::Random(M * M);
22
23            // Do not want to use kron for large values of M
24            if (M < (1 << 6)) {
25                // Kron using direct implementation
26                tm_kron.start();
27                kron(A, B, C);
28                y = C * x;
29                tm_kron.stop();
30            }
31
32            // TO DO: (2-3.f) Measure the runtime of kron_mult()
33            // and kron_reshape().
34            // START
35
36            // Kron matrix-vector multiplication
37            tm_kron_mult.start();
38            kron_mult(A, B, x, y);
39            tm_kron_mult.stop();
40
41            // Kron using reshape
42            tm_kron_map.start();
43            kron_reshape(A, B, x, y);
44            tm_kron_map.stop();
45
46            // END
47        }
48
49        double kron_time = (M < (1 << 6)) ? tm_kron.min() : std::nan("");
50        std::cout << std::setw(5) << M << std::scientific << std::setprecision(3)
51            << std::setw(15) << kron_time << std::setw(15)
```

```
52             << tm_kron_mult.min() << std::setw(15) << tm_kron_map.min()
53             << std::endl;
54     }
55 }
```

Get it on  GitLab (kron.cpp).

---

SOLUTION of (2-4.a):

The following code implements the Strassen's recursive algorithm:

### C++-code 2.4.1: Strassen's algorithm with EIGEN

```
2 MatrixXd strassenMatMult(const MatrixXd& A, const MatrixXd& B) {
3     // Ensure square matrix
4     assert(A.rows() == A.cols() && "Matrix A must be square");
5     assert(B.rows() == B.cols() && "Matrix B must be square");
6     // Matrix dimension must be a power of 2
7     assert(A.rows() % 2 == 0 && "Matrix dimensions must be a power of two.");
8
9     const unsigned n = A.rows();
10
11    // The function is recursive and acts on blocks of size n/2 × n/2
12    // i.e. exploits fast product of 2x2 block matrix
13    if (n==2) { // End of recursion
14        MatrixXd C(2, 2);
15        C << A(0,0)*B(0,0) + A(0,1)*B(1,0),
16                    A(0,0)*B(0,1) + A(0,1)*B(1,1),
17                    A(1,0)*B(0,0) + A(1,1)*B(1,0),
18                    A(1,0)*B(0,1) + A(1,1)*B(1,1);
19        return C;
20    }
21
22    MatrixXd Q0(n/2, n/2), Q1(n/2, n/2), Q2(n/2, n/2), Q3(n/2, n/2),
23                Q4(n/2, n/2), Q5(n/2, n/2), Q6(n/2, n/2);
24
25    MatrixXd A11 = A.topLeftCorner(n/2, n/2);
26    MatrixXd A12 = A.topRightCorner(n/2, n/2);
27    MatrixXd A21 = A.bottomLeftCorner(n/2, n/2);
28    MatrixXd A22 = A.bottomRightCorner(n/2, n/2);
29
30    MatrixXd B11 = B.topLeftCorner(n/2, n/2);
31    MatrixXd B12 = B.topRightCorner(n/2, n/2);
32    MatrixXd B21 = B.bottomLeftCorner(n/2, n/2);
33    MatrixXd B22 = B.bottomRightCorner(n/2, n/2);
34
35    Q0 = strassenMatMult(A11 + A22, B11 + B22);
36    Q1 = strassenMatMult(A21 + A22, B11);
37    Q2 = strassenMatMult(A11, B12 - B22);
38    Q3 = strassenMatMult(A22, B21 - B11);
39    Q4 = strassenMatMult(A11 + A12, B22);
40    Q5 = strassenMatMult(A21 - A11, B11 + B12);
41    Q6 = strassenMatMult(A12 - A22, B21 + B22);
42
43    MatrixXd C(n,n);
44
45    C << Q0 + Q3 - Q4 + Q6,
46                Q2 + Q4,
47                Q1 + Q3,
48                Q0 + Q2 - Q1 + Q5;
49
50    return C;
51 }
```

Get it on ❤️ GitLab (strassen.hpp).

---

HINT 1 for (2-4.b): The validation boils down to checking that a matrix is zero, or rather “close to zero”, because the impact of roundoff renders comparison with the zero matrix meaningless, see [Lecture → Rem. 1.5.3.15]. Closeness to zero can be inspected by computing some norm of the matrix, for instance by means of EIGEN’s `norm` method, which returns the so-called Frobenius norm of a matrix, see [Lecture → Def. 3.4.4.16]. □

SOLUTION of (2-4.b):

### C++-code 2.4.2: Strassen's algorithm with EIGEN

```
2 // Seed the random number generator
3     srand((unsigned int) time(0));
4
5 ///// Check algorithm for correctness
6
7 // Size of the matrix
8 int k = 2;
9 int n = std::pow(2, k);
10
11 // Generate random matrices
12 MatrixXd A = MatrixXd::Random(n,n);
13 MatrixXd B = MatrixXd::Random(n,n);
14
15 // Testing matrix multiplication
16 MatrixXd AB = strassenMatMult(A,B);
17 // Eigen Matrix multiplication
18 MatrixXd AxB = A*B;
19
20 std::cout << "Using Strassen's method, A*B=" << std::endl
21     << AB << std::endl;
22 std::cout << "Using standard method, A*B=" << std::endl
23     << AxB << std::endl;
24 std::cout << "The norm of the error is: "
25     << (AB-AxB).norm() << std::endl;
```

Get it on  GitLab (test.cpp).

SOLUTION of (2-4.c):

### C++-code 2.4.3: Runtime of Strassen's algorithm EIGEN

```
2 // Store seed for rng
3 unsigned seed = (unsigned) time(0);
4 // seed random number generator
5 srand(seed);
6
7 // Minimum number of repetitions
8 unsigned int repetitions = 10;
9
10 // Display header column
11 std::cout << std::setw(4) << "k"
12     << std::setw(15) << "A*B"
13     << std::setw(15) << "Strassen" << std::endl;
14 for(unsigned k = 4; k <= 9; k++) {
15     unsigned int n = std::pow(2, k);
16
17     // Initialize random input matrixies
18     MatrixXd A = MatrixXd::Random(n, n);
19     MatrixXd B = MatrixXd::Random(n, n);
20
21     // Timer to collect runtime of each individual run
22     Timer timer, timer_own;
23
24     for(unsigned int r = 0; r < repetitions; ++r) {
25         // initialize memory for result matrix
26         MatrixXd Ax_B, Ax_B2;
27
28         // Benchmark eigens matrix multiplication
29         timer.start(); // start timer
30         Ax_B=(A*B); // do the multiplication
31         timer.stop(); // stop timer
32
33         // Benchmark Stassens matrix multiplication
34         timer_own.start(); // start timer
35         Ax_B2=strassenMatMult(A, B); // do the multiplication
36         timer_own.stop(); // stop timer
37
38         // volatile double a = (Ax_B+Ax_B2).norm();
39     }
40
41     // Print runtimes
42     std::cout << std::setw(4) << k // power
43         << std::setprecision(3) << std::setw(15) << std::scientific
44         << timer.min() // eigen timing
45         << std::setprecision(3) << std::setw(15) << std::scientific
46         << timer_own.min() // strassing timing
47         << std::endl;
48
49 }
```

Get it on  GitLab (timing.cpp).

HINT 1 for (2-5.a): The EIGEN matrix/vector methods `normalized()`, `rightCols` and `transpose()` may be useful. ↳

SOLUTION of (2-5.a):

The following code implements Algorithm ?? making use of EIGEN's matrix-block operations.

### C++-code 2.5.2: C++ implementation of Code ??

```
2 void houseref(const Eigen::VectorXd & v, Eigen::MatrixXd & Z)
3 {
4     unsigned int n = v.size();
5     Eigen::VectorXd w = v.normalized();
6     Eigen::VectorXd u=w;
7     u(0) += 1;
8     Eigen::VectorXd q = u.normalized();
9     Eigen::MatrixXd X = Eigen::MatrixXd::Identity(n, n) - 2*q*q.transpose();
10    Z = X.rightCols(n-1);
11 }
```

Get it on  GitLab (houseref.cpp).

HINT 1 for (2-5.b): Notice that  $|\mathbf{q}|^2 = 1$ . □

---

SOLUTION of (2-5.b):

By straightforward computations:

$$\begin{aligned}\mathbf{X}^T \mathbf{X} &= (\mathbf{I}_n - 2\mathbf{q}\mathbf{q}^T)(\mathbf{I}_n - 2\mathbf{q}\mathbf{q}^T) \\ &= \mathbf{I}_n - 4\mathbf{q}\mathbf{q}^T + 4\mathbf{q} \underbrace{\mathbf{q}^T \mathbf{q}}_{=|\mathbf{q}|=1} \mathbf{q}^T \\ &= \mathbf{I}_n - 4\mathbf{q}\mathbf{q}^T + 4\mathbf{q}\mathbf{q}^T \\ &= \mathbf{I}_n.\end{aligned}$$

This is what we wanted to prove. □

---

HINT 1 for (2-5.c): Use the previous hint, and the fact that

$$\mathbf{u} = \mathbf{w} + \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

and  $|\mathbf{w}| = 1$ .

□

SOLUTION of (2-5.c):

Let  $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_n] \mathbb{R}^{n,n}$  be the matrix defined in Code ???. Let  $\mathbf{e}^{(1)} := [1, 0, \dots, 0]^\top$ . In view of the identity  $\mathbf{x}_1 = \mathbf{e}^{(1)} - 2q_1\mathbf{q}$  we have:

$$\mathbf{x}_1 = \begin{bmatrix} 1 - 2q_1^2 \\ -2q_1 q_2 \\ \vdots \\ -2q_1 q_n \end{bmatrix} = \begin{bmatrix} 1 - 2\frac{u_1^2}{\sum_{i=1}^n u_i^2} \\ -2\frac{u_1 u_2}{\sum_{i=1}^n u_i^2} \\ \vdots \\ -2\frac{u_1 u_n}{\sum_{i=1}^n u_i^2} \end{bmatrix} \stackrel{\text{HINT}}{=} \begin{bmatrix} \frac{(w_1+1)^2 + w_2^2 + \dots + w_n^2 - 2(w_1+1)^2}{(w_1+1)^2 + w_2^2 + \dots + w_n^2} \\ -\frac{2(w_1+1)w_2}{(w_1+1)^2 + w_2^2 + \dots + w_n^2} \\ \dots \\ -\frac{2(w_1+1)w_n}{(w_1+1)^2 + w_2^2 + \dots + w_n^2} \end{bmatrix} \stackrel{|\mathbf{w}|=1}{=} \begin{bmatrix} -\frac{2w_1(w_1+1)}{2(w_1+1)} \\ -\frac{2(w_1+1)w_2}{2(w_1+1)} \\ \dots \\ -\frac{2(w_1+1)w_n}{2(w_1+1)} \end{bmatrix} = -\mathbf{w},$$

gather\* which is a multiple of  $\mathbf{v}$ , since  $\mathbf{w} = \frac{\mathbf{v}}{|\mathbf{v}|}$ . □

---

SOLUTION of (2-5.d):

The columns of  $\mathbf{X} = [\mathbf{x}_1, \dots, \mathbf{x}_n]$  are an orthonormal basis (ONB) of  $\mathbb{R}^n$ . Thus, the columns of  $\mathbf{Z} = [\mathbf{z}_2, \dots, \mathbf{z}_n]$  are an ONB of the complement of  $\text{Span}(\mathbf{x}_1) \stackrel{??}{=} \text{Span}(\mathbf{v})$ . The function `houseref1` computes an ONB of the complement of  $\mathbf{v}$ .

---

---

SOLUTION of (2-5.e):

The complexity is  $O(n^2)$ : this is the asymptotic complexity of the construction of the product  $\mathbf{q}\mathbf{q}^\top$  in Code ??.

---

HINT 1 for (2-6.a): Reuse matrix products. Express the exponent  $k$  using a *binary representation*. □

SOLUTION of (2-6.a):

Write  $k$  in binary format:  $k = \sum_{j=0}^M b_j 2^j$ ,  $b_j \in \{0, 1\}$ . Then

$$\mathbf{A}^k = \prod_{j=0}^M \mathbf{A}^{2^j b_j} = \prod_{j \text{ s.t. } b_j=1} \mathbf{A}^{2^j}.$$

We compute  $\mathbf{A}$ ,  $\mathbf{A}^2$ ,  $\mathbf{A}^4$ ,  $\dots$ ,  $\mathbf{A}^{2^M}$  (one matrix-matrix multiplication each) and we multiply only the matrices  $\mathbf{A}^{2^j}$  such that  $b_j \neq 0$ .

### C++11-code 2.6.1: Efficient matrix power

```
2 MatrixXcd matPow( MatrixXcd &A, unsigned int k) {
3     // Identity matrix
4     MatrixXcd X = MatrixXcd::Identity(A.rows(), A.cols());
5
6     // p is used as binary mask to check whether,
7     // given k = \sum_{i=0}^M b_i 2^i, b_i = 1
8     // (i.e. if k has a 1 in the i-th binary digit).
9     // Obtaining the binary representation of p can be done in many ways,
10    // here we use ~k&p to check if b_i = 1
11    unsigned int p = 1;
12
13    // Cycle all the way up to the 1st 1 in the binary
14    // representation of k (this is called the "most significant bit")
15    // If you are interested in other methods to compute the "most
16    // significant bit", consult the file "most_significant_bit.cpp"
17    for (unsigned int j = 1; j <= std::ceil(std::log2(k)); ++j) {
18
19        if ((~k & p) == 0) { // if( b_i != 0 )
20            X = X * A;
21        }
22        A = A * A;
23        p = p << 1; // p = p*2;
24    }
25    return X;
26 }
```

Get it on  GitLab (matPow.cpp).

---

SOLUTION of (2-6.b):

Using a “naive” implementation according to

$$\mathbf{A}^k = \underbrace{\left( \dots \left( (\mathbf{A} \cdot \mathbf{A}) \cdot \mathbf{A} \right) \dots \cdot \mathbf{A} \right)}_k \cdot \mathbf{A} ,$$

the complexity is  $O((k - 1)n^3)$ .

Using the efficient implementation from Code ??, for each  $j \in \{1, 2, \dots, \lceil \log_2(k) \rceil\}$  we have to perform at most two multiplications ( $\mathbf{X} \star \mathbf{A}$  and  $\mathbf{A} \star \mathbf{A}$ ):

$$\text{asymptotic computational cost} = 2 \cdot M \cdot C = 2 \cdot \lceil \log_2 k \rceil \cdot O(n^3) .$$

where  $M := \lceil \log_2(k) \rceil$  is the loop size, and  $C$  is the cost of a matrix-matrix multiplication. (Here  $\lceil a \rceil = \text{ceil}(a) := \min\{b \in \mathbb{Z}, a \leq b\}$ .)

---

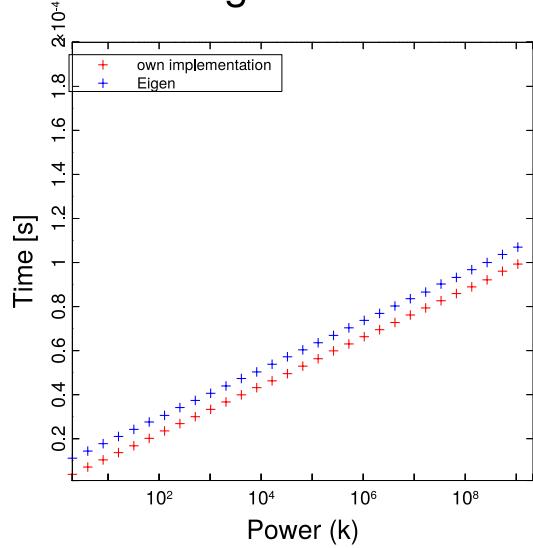
SOLUTION of (2-6.c):

### C++11-code 2.6.2: Runtime computation

```
2 // Will contain runtimes for each k, k stored in powers
3 std::vector<double> powers, times_matPow, times_Eigen_pow;
4
5 std::cout << "— Table of runtimes" << std::endl;
6 std::cout << std::setw(15) << "K" << std::setw(15) << "Own matPow"
7 << std::setw(15) << "Eigen pow()" << std::endl;
8 // Loop from 2 to 231
9 for (unsigned int K = 2; K <= (1u << 31); K = K << 1) {
10
11     unsigned int repeats = 10;
12     Timer tm_pow, tm_Eigen_pow;
13
14     // Repeat the test repeat times
15     for (unsigned int r = 0; r < repeats; ++r) {
16
17         // Build Vandermonde matrix of size n
18         MatrixXcd X, A = construct_matrix(n);
19
20         // Compute runtime if own implementation
21         tm_pow.start();
22         X = matPow(A, K);
23         tm_pow.stop();
24
25         // Compute runtime of eigen implementation
26         A = construct_matrix(n);
27         tm_Eigen_pow.start();
28         X = A.pow(K);
29         tm_Eigen_pow.stop();
30     }
31
32     // Output table
33     std::cout << std::setw(15) << K << std::scientific << std::setprecision(3)
34     << std::setw(15) << tm_pow.min() << std::setw(15)
35     << tm_Eigen_pow.min() << std::endl;
36
37     // Store data
38     powers.push_back(K);
39     times_matPow.push_back(tm_pow.min());
40     times_Eigen_pow.push_back(tm_Eigen_pow.min());
41 }
42 }
```

Get it on ❤️ GitLab (matPow.cpp).

## Timings of matPow



According to the result of ?? we expect an asymptotic behavior of the runtimes as  $O(\log k)$  for  $k \rightarrow \infty$ . This should best manifest itself as data points lying on a straight line in *logarithmic-linear* plot

**Remark:** All the eigenvalues of the Vandermonde matrix  $\mathbf{A}$  have absolute value 1, so the powers  $\mathbf{A}^k$  are “stable”: the eigenvalues of  $\mathbf{A}^k$  are not approaching neither 0 nor  $\infty$  when  $k$  grows.

---

---

SOLUTION of (2-7.a):

Notice the following:

- the outer product of two vectors has complexity  $\mathcal{O}(n^2)$ ;
- the Matrix–vector multiplication has quadratic complexity  $\mathcal{O}(n^2)$ ;
- component-wise minimum has complexity  $\mathcal{O}(n^2)$ .

Therefore, the overall complexity is  $\mathcal{O}(n^2)$ .

---

HINT 1 for (2-7.b): Write an expression for the components  $y_j$  of the result vector  $\mathbf{y}$ . Find a way to “reuse” your computations. □

SOLUTION of (2-7.b):

For every  $j = 1, \dots, n$  we have

$$y_j = \sum_{k=1}^j kx_k + j \sum_{k=j+1}^n x_k.$$

Recursively define for  $j = 2, \dots, n$ :

$$v_j := v_{j-1} + x_{n-j+1} = \sum_{k=n-j+1}^n x_k$$

and

$$w_j := w_{j-1} + jx_j = \sum_{k=1}^j kx_k$$

with  $v_1 = x_n$  and  $w_1 = x_1$ . You can easily show that:

$$v_j = \sum_{k=n-j+1}^n x_k, w_j = \sum_{k=1}^j kx_k$$

Then

$$y_j = \sum_{k=1}^j kx_k + j \sum_{k=j+1}^n x_k = w_j + jv_{n-j}. \quad (2.7.3)$$

We can compute  $w_j$  and  $v_j$  for  $j = 1, \dots, n$  with a for loop of size  $n$ . Once we computed  $w_j$  and  $v_j$ , we can use (??) to compute  $y_j$  for each  $j = 1, \dots, n$  using another for loop.

Particular care has to be taken when implementing EIGEN code, since indices start at 0.

#### C++11-code 2.7.4: Efficiently computing $\mathbf{Ax}$

```
2 void multAmin(const VectorXd & x, VectorXd & y) {
3     unsigned int n = x.size();
4     y = VectorXd::Zero(n);
5
6     // TO DO: (2-7.b) Fill in the entries of y.
7     // Hint: Find an expression  $y(j) = w(j) + j*x(j)$  such that
8     // the entries of w and u can be calculated recursively.
9     // START
10    VectorXd v = VectorXd::Zero(n);
11    VectorXd w = VectorXd::Zero(n);
12
13    v(0) = x(n-1);
14    w(0) = x(0);
15
16    for(unsigned int j = 1; j < n; ++j) {
17        v(j) = v(j-1) + x(n-j-1);
```

```
18     w(j) = w(j-1) + (j+1)*x(j);  
19 }  
20 for(unsigned int j = 0; j < n-1; ++j) {  
21     y(j) = w(j) + v(n-j-2)*(j+1);  
22 }  
23 y(n-1) = w(n-1);  
24 // END  
25 }
```

Get it on  GitLab (multAmin.cpp).

---

---

SOLUTION of (2-7.c):

Only simple loops of size  $n$  are performed. Therefore we have linear complexity:  $O(n)$ .

---

## SOLUTION of (2-7.d):

The matrix multiplication in (??) has complexity  $O(n^2)$ . The faster implementation has complexity  $O(n)$ .

### C++11-code 2.7.5: Runtime test for multAmin

```
2 // Timing from 24 to 210 repeating "nruns" times
3 unsigned int nruns = 10;
4
5 std::cout << "—> Timings:" << std::endl;
6 // Header, see iomanip documentation
7 std::cout << std::setw(15)
8     << "N"
9     << std::scientific << std::setprecision(3)
10    << std::setw(15) << "multAminSlow"
11    // << std::setw(15) << "multAminLoops"
12    << std::setw(15) << "multAmin"
13    << std::endl;
14 // From 24 to 210
15 for(unsigned int N = (1 << 4); N <= (1 << 10); N = N << 1) {
16     Timer tm_slow, tm_fast;
17     // TO DO: (2-7.d) Compute runtimes of multAminSlow(x,y) and
18     // multAmin(x,y) with x = VectorXd::Random(N). Repeat nruns times.
19     // START
20     for(unsigned int r = 0; r < nruns; ++r) {
21         VectorXd x = VectorXd::Random(N);
22         VectorXd y;
23
24         // Runtime of slow method
25         tm_slow.start();
26         multAminSlow(x, y);
27         tm_slow.stop();
28
29         // Runtime of fast method
30         tm_fast.start();
31         multAmin(x, y);
32         tm_fast.stop();
33     }
34     // END
35
36     std::cout << std::setw(15)
37         << N
38         << std::scientific << std::setprecision(3)
39         << std::setw(15) << tm_slow.min()
40         << std::setw(15) << tm_fast.min()
41         << std::endl;
42 }
```

Get it on  GitLab (multAmin.cpp).

---

SOLUTION of (2-7.e):

The matrix  $\mathbf{B}$  is the following:

$$\mathbf{B} := \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 1 \end{bmatrix} \in \mathbb{R}^{n,n}.$$

Notice the value 1 in the entry at position  $(n, n)$ .

---

## SOLUTION of (2-7.f):

You should have observed that you recover the same unit vector that you have multiplied with  $\mathbf{AB}$ . Hence, the matrix  $\mathbf{B}$  is the (unique) inverse of  $\mathbf{A}$ .

### C++11-code 2.7.7: Matrix multiplication $\mathbf{AB}$

```
2 Eigen::MatrixXd multABUnity() {
3     unsigned int n = 10;
4
5     /* SAM_LISTING_BEGIN_5 */
6     MatrixXd B = MatrixXd::Zero(n,n);
7     for(unsigned int i = 0; i < n; ++i) {
8         B(i,i) = 2;
9         if(i < n-1) B(i+1,i) = -1;
10        if(i > 0) B(i-1,i) = -1;
11    }
12    B(n-1,n-1) = 1;
13    /* SAM_LISTING_END_5 */
14
15    MatrixXd C(n,n);
16
17    // TO DO: (2-7.f) Set the columns of C to  $A \cdot B \cdot e_j$  for
18    //  $j=0, \dots, n-1$ , and print C.
19    // START
20    VectorXd y;
21    //  $B \cdot e_j$  is the  $j$ -th column of  $B \cdot Id = B$ , so we need only
22    // calculate  $A \cdot B.col(j)$  for  $j=0, \dots, n-1$ .
23    for( unsigned int i = 0; i < n; ++i ) {
24        multAmin( B.col(i), y );
25        C.col(i) = y;
26    }
27    std::cout << "C = \n" << C << std::endl;
28    // END
29
30    return C;
31 }
```

Get it on  GitLab (multAmin.cpp).

HINT 1 for (2-8.a): Use that (??) can be transformed, using the trigonometric identity

$$\sin(\varphi) - \sin(\psi) = 2 \cos\left(\frac{\varphi + \psi}{2}\right) \sin\left(\frac{\varphi - \psi}{2}\right). \quad (2.8.2)$$

□

---

SOLUTION of (2-8.a):

Based on (??) we get the equivalent formula

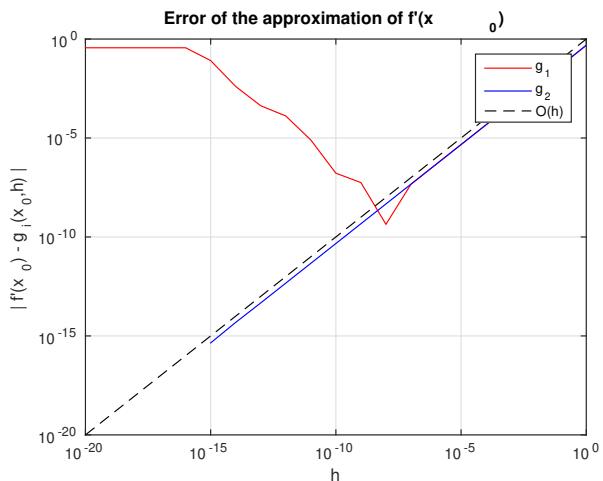
$$f_2(x_0, h) = 2 \cos\left(\frac{2x_0 + h}{2}\right) \sin\left(\frac{h}{2}\right).$$

---

HINT 1 for (2-8.b): For background information refer to [Lecture → Ex. 1.5.4.7].

## SOLUTION of (2-8.b):

Check the implementation in ?? and the plot in ??.  
 We can observe that the computation using  $f_1$  does not converge as  $h \rightarrow 0$ . This is due to the cancellation error given by the subtraction of two numbers of approximately same magnitude. The second implementation, using  $f_2$ , is very stable and does not display cancellation issues.



### C++-code 2.8.4: C++ implementation for ??

```

2 void sinederv() {
3     // Array of values of h
4     ArrayXd h = ArrayXd::LinSpaced(21, -20, 0.)
5         .unaryExpr([] (double i) {
6             return std::pow(10., i);
7         });
8     // Dummy array where to evaluate the derivative (1.2)
9     ArrayXd x = ArrayXd::Constant(h.size(), 1.2);
10
11    // Derivatives
12    ArrayXd g1, g2, ex;
13
14    // TO DO: (2-8.b) For each entry in h, fill in the arrays such that
15    // g1 approximates f' (1.2) using the difference quotient directly,
16    // g2 approximates f' (1.2) while avoiding cancellation, and
17    // ex is the "exact" value of f' (1.2).
18    // Hint: You can use Eigen's sin() and cos(), which calculate
19    // coefficient-wise results for Array inputs.
20    // START
21    g1 = (sin(x+h) - sin(x)) / h; // naive
22    g2 = 2 * cos(x+0.5*h) * sin(0.5 * h) / h; // better
23    ex = cos(x); // exact
24    // END
25
26    // Print error
27    // Table header
28    std::cout << std::setw(15) << "h"
29        << std::setw(15) << "exact"
30        << std::setw(15) << "cancellation"
31        << std::setw(15) << "error"
32        << std::setw(15) << "improved"
33        << std::setw(15) << "error" << std::endl;
34    for(unsigned int i = 0; i < h.size(); ++i) {
35        std::cout << std::setprecision(6);
36        // Table entries
37        // TO DO: (2-8.b) Print the i-th row of the table.

```

```
38 // Keep the order h, ex, g1, |g1 - ex|, g2, |g2 - ex|.
39 // Use std::setw(15) as in the table header.
40 // You can use std::abs() to calculate absolute values.
41 // START
42 std::cout << std::setw(15) << h(i)
43     << std::setw(15) << ex(i)
44     << std::setw(15) << g1(i)
45     << std::setw(15) << std::abs(g1(i) - ex(i))
46     << std::setw(15) << g2(i)
47     << std::setw(15) << std::abs(g2(i) - ex(i)) << std::endl;
48 // END
49 }
50
51 // Plots the contents of the table on a log-log scale.
52 py_plot( h, ex, g1, g2 );
53 }
```

Get it on  GitLab (cancellation.cpp).

---

HINT 1 for (2-8.c): Study the tricks applied in [Lecture → Ex. 1.5.4.21] to avoid cancellation. □

---

SOLUTION of (2-8.c):

We have

$$\ln(x - \sqrt{x^2 - 1}) = -\ln(x + \sqrt{x^2 - 1}).$$

We immediately derive  $\ln(x - \sqrt{x^2 - 1}) + \ln(x + \sqrt{x^2 - 1}) = \log(x^2 - (x^2 - 1)) = 0$ .

As  $x \rightarrow \infty$  the left **log** consists of subtraction of two numbers of almost equal value, whilst the right **log** consists of the addition of two numbers of approximately the same value (which does not incur in cancellation errors). Therefore, in the l.h.s. there may be cancellation for large values of  $x$ , making it worse for numerical computation. You can see the difference of the two expressions with  $x = 10^8$ .

---

---

SOLUTION of (2-8.d):

1. For  $x \gg 1$ , inside the square roots we have the addition (resp. subtraction) of a small number to a big number. The difference of the square roots incur in cancellation, since they have the same, large magnitude. Define  $A := x + \frac{1}{x}$ ,  $B := x - \frac{1}{x}$ . Then:

$$(A - B)(A + B)/(A + B) = \frac{2/x}{\sqrt{x + \frac{1}{x}} + \sqrt{x - \frac{1}{x}}} = \frac{2}{\sqrt{x}(\sqrt{x^2 + 1} + \sqrt{x^2 - 1})}$$

For  $x \approx 1$  cancellation occurs under the second root, but it is harmless, because the inaccurate, but small result will be added to a relative large number (the first root) and thus its relative error is mitigated.

2. The value of  $\frac{1}{a^2}$  becomes very large as  $a$  approaches 0, whilst  $\frac{1}{b^2} \rightarrow 1$  as  $b \rightarrow 1$ . Therefore, the relative size of  $\frac{1}{a^2}$  and  $\frac{1}{b^2}$  becomes so big, that, in computer arithmetic,  $\frac{1}{a^2} + \frac{1}{b^2} = \frac{1}{a^2}$ . On the other hand  $\frac{1}{a}\sqrt{1 + (\frac{a}{b})^2}$  avoids this problem by performing a division between two numbers with very different magnitude, instead of a summation.
-

---

SOLUTION of (2-9.a):

The output is  $\mathbf{y}$  s.t.  $\mathbf{y} = \mathbf{A}^k \mathbf{x}$ . In fact, the eigenvalue decomposition of the matrix  $\mathbf{A} = \mathbf{SDW}^{-1}$  (where  $\mathbf{D}$  is diagonal and  $\mathbf{W}$  invertible), allows us to write:

$$\mathbf{A}^k = (\mathbf{WDW}^{-1})^k = \mathbf{WD}^k\mathbf{W}^{-1}$$

$\mathbf{D}^k$  can be computed efficiently (component-wise) for diagonal matrices.

The code `getit.cpp` provides a line-by-line explanation. Get it on  GitLab ([getit.cpp](#)).

---

---

SOLUTION of (2-9.b):

The algorithm comprises the following operations:

- Diagonalization of a full-rank matrix  $\mathbf{A}$  is  $O(n^3)$ .
- Matrix-vector multiplication is  $O(n^2)$ .
- Vector-vector componentwise multiplication is  $O(n)$ .
- Raising a vector in  $\mathbb{R}^n$  for the power  $k$  has complexity  $O(n)$ .
- Solve a fully determined linear system:  $O(n^3)$ .

The complexity of the algorithm is dominated by the operations with higher exponent. Therefore the total complexity of the algorithm is  $O(n^3)$  for  $n \rightarrow \infty$ .

---