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

Homework Problems and Projects

Prof. R. Hiptmair, SAM, ETH Zurich

Autumn Term 2019

(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 in a difficult problem and you tried to find a solution for a sufficient amount of time.

0.1.3 Corrections and Grading of Assignments

Homework assignments will not be graded.

However, you may submit your handwritten solutions by putting them in designated trays in front of HG G 53.2, in order to get feedback from the assistants. 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.

Further details will be published on the course Moodle page.

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 to the group corresponding to your TA. Please, do not register to any other group. No setup is required in your machine and you do not need to install Eigen or other 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. Code Expert 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 and discuss alternatives.

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

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.

▲

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

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

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

HINT 1 for (1-1.b): The test case implemented in `main.cpp` is `smallTriangular(1,2,3)` and returns

$$\begin{bmatrix} 1 & 2 \\ 0 & 3 \end{bmatrix} .$$

Matrix initialization with the << operator is convenient. However also the entry access operator **double &operator (int, int)** can be used directly.

SOLUTION of (1-1.b):

C++ code 1.1.1: Initialization of fixed size 2×2-matrix
Get it on GitLab (MatrixClass.hpp).

```
Eigen::Matrix<double, 2, 2> smallTriangular(double a, double b, double c) {
  /*
   * This functions returns a 2 by 2 triangular matrix of doubles a, b, c.
   */

  // We know in advance that we want to create a matrix of doubles with a fixed
  // size of 2 by 2.  Therefore, we pass the parameters <double, 2, 2> to the
  // template class Eigen::Matrix.
  Eigen::Matrix<double, 2, 2> A;

  // We have declared the variable A of the type Eigen::Matrix<double,2,2>,
  // but we have not initialized its entries.
  // We can do this using comma-initialization:
  A << a, b, 0, c;

  // Question:  Is A now an upper triangular matrix, or a lower triangular
  // matrix?

  return A;
}
```

▲

(1-1.c) 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.

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.

One can also use the `triangularView()` member function for **Eigen::Matrix**, see EIGEN documentation.

C++ code 1.1.2: Initialization of upper triangular part of a matrix
Get it on GitLab (MatrixClass.hpp).

```
Eigen::MatrixXd constantTriangular(int n, double val) {
  /*
   * This function returns an n by n upper triangular matrix with the constant
   * value val in the upper triangular part.
   */

  // Now we do not know the size of our matrix at compile time.
  // Hence, we use the special value Eigen::Dynamic to set the size of A.
  Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic> A;

  // Eigen::Matrix has a method Zero(n_rows,n_cols) which returns the n_rows by
  // n_cols matrix whose entries are all equal to zero.
  A = Eigen::Matrix<double, Eigen::Dynamic, Eigen::Dynamic>::Zero(n, n);

  // To get a triangular matrix, we must set the values above the diagonal to
  // val. This we can do by a nested for-loop.
  for (int row = 0; row < n; row++) {

    // TO DO: Write the inner for-loop.
    // Hint: We can access and change the entries of A using parentheses, i.e.
    // A(row,col) = val; Note that indexing starts at the value 0 (as usual),
    // and not 1 (as in Matlab).

    // START
    for (int col = row; col < n; col++) {
      A(row, col) = val;
    }
    // END
  }
  return A;
}
```

▲

(1-1.d) In the file `MatrixClass.hpp` supplement the missing parts of the function `casting()` such that it returns the real part of $2(1-\imath)+2(5+\imath)$. To that end use EIGEN's matrix multiplication.

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).

```
double casting() {
  /*
   * This function does not do anything meaningful.
   * It is only meant to introduce vectors and how to typecast.
   */

  // Because the syntax Eigen::Matrix< type, n_rows, n_cols > is very
  // cumbersome, Eigen provides convenience classes that work as shorthands.
  // For example, Eigen::Matrix2d is shorthand for Eigen::Matrix< double, 2, 2 >.

  // Vectors are just a special type of matrices that have only one column.
  // Thus, VectorXi is a shorthand for Eigen::Matrix< int, Eigen::Dynamic, 1 >.
  // Constant(2,1) creates a vector of size 2 and initializes the entries with
  // the value 1.
  Eigen::VectorXi u = Eigen::VectorXi::Constant(2, 1);

  // std::complex is a template class for complex numbers.
  // Here we declare two complex numbers, with real and imaginary parts
  // represented by doubles. z0 = 1 - i z1 = 5 + i
  std::complex<double> z0(1, -1), z1(5, 1);

  // Declare and initialize a size 2 vector of std::complex<double>.
  Eigen::Vector2cd v;
  v(0) = z0;
  v(1) = z1;

  double x;
  // TO DO: Calculate the inner product of u and v, and store the result in x.
  // Hint: The inner product of two vectors is given by u.dot(v), but
  // first we need to cast the "int" vector u to a "std::complex<double>" vector.
  // Use u.cast< NEW TYPE >() to achieve this.
  // The result of the inner product will be 1*(1-i) + 1*(5+i) = 6 + 0i,
  // a real number. You can get the real part of an std::complex<double>
  // using the method "real()".
  // START
  std::complex<double> z = u.cast<std::complex<double>>().dot( v );
  x = z.real();
  // END
  return x;
}
```

▲

(1-1.e) 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

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

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

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

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

HINT 1 for (1-1.e): The test call in `main.cpp` is `arithmetics(4)` and should return the vector

$$\begin{bmatrix} -41.4706 + 27.1176\imath \\ -23.7692 + 42.8462\imath \\ -6.47692 + 48.4154\imath \\ 7.2 + 48.4\imath \end{bmatrix}.$$

SOLUTION of (1-1.e):

C++ code 1.1.4: Evaluating a simple matrix-vector expression
Get it on GitLab (MatrixClass.hpp).

```
Eigen::VectorXcd arithmetics(int n) {
  /*
   * This function does not do anything meaningful.
   * It is only meant to show some simple Eigen arithmetics and matrix
   * initializations.
   */

  // This declares dynamic size (signified by the letter 'X') matrices of
  // complex numbers (signified by the letter c) with real and imaginary
  // parts represented by doubles (signified by 'd').
  Eigen::MatrixXcd A, C, I;

  // We initialize the matrices arbitrarily:
  // an n by n lower triangular matrix,
  A = constantTriangular(n, 5.0).transpose().cast<std::complex<double>>();
  // The n by n identity matrix,
  I = Eigen::MatrixXcd::Identity(n, n);

  // Declare the n by n matrix B.
  Eigen::MatrixXcd B(n,n);

  // TO DO: Fill in the matrix B such that B(k,l) = (k+l*i)/(k-l*i),
  // where i is the imaginary unity, i^2 = -1.
  // START
  for(int k=0; k<n; k++) {
    for(int l=0; l<n; l++) {
      std::complex<double> tmp(k+1,l+1);
      B(k,l) = tmp/std::conj(tmp);
    }
  }
  // END

  // We can perform arithmetics on matrices: +, -, *
  // Note that for + and -, the left hand matrix and the right hand matrix have
  // to be the same size, and that matrix multiplication * is only defined if
  // the number of columns in the left hand matrix is the same as the number of
  // rows in the right hand matrix.
  C = B * (A - 5.0 * I);

  // Eigen::VectorXcd is shorthand for
  // Eigen::Matrix< std::complex<double>, Eigen::Dynamic, 1>.
  // Hence, all the same arithmetics work for vectors.
  Eigen::VectorXcd u, v;

  // TO DO: Initialize the entries of u as the integers from 1 to n,
  // multiply u by the matrix C, and store the result in v.
```

```
  // START
  u = Eigen::VectorXcd::LinSpaced(n, 1, n); // Or use a for-loop.
  v = C * u;
  // END

  return v;
}
```

▲

End Problem 1-1

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 is connected with [Lecture → § 1.2.1.5].

Remark. The codes given as solutions may leave ample room for improvement.

(1-2.a) 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.

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 2×2-matrix
Get it on GitLab (MatrixBlocks.hpp).

```
MatrixXd zero_row_col(const MatrixXd &A, int p, int q) {
  /*
   * This function takes a matrix A, and returns a matrix that is exactly the
   * same, except that row p and column q are set to zero.
   */
  // Make a copy of A.
  MatrixXd Anew(A);

  // TO DO: Set the entries of row number p and column number q to zero.
  // Hint:  We can access rows and columns of A by A.row() and A.col().
  // The setZero() is useful here.  START
  Anew.row(p).setZero();
  Anew.col(q).setZero();
  // END

  return Anew;
}
```

(1-2.b) 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

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

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

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).

```
MatrixXd swap_left_right_blocks(const MatrixXd &A, int p) {
  /*
   * Writing as a block matrix A = [B C], where B denotes the first p columns of
   * A, and C denotes the q=(A.cols() - p) last columns, this functions returns
   * D = [C B].
   */

  MatrixXd B, C;

  // We can use .rows() and .cols() to get the number of rows and columns in A.
  int q = A.cols() - p;

  // A.block( i, j, m, n ) returns the m by n block that has its top-left corner
  // at the index (i,j) in A. Hence, the first p columns of A can be accessed in
  // the following way:
  B = A.block(0, 0, A.rows(), p);

  // TO DO: Use A.block() to define C as the matrix containing the last q
  // columns of A. START
  C = A.block(0, p, A.rows(), q);
  // END

  // Make a copy of A.
  MatrixXd Anew(A);
  // The block() method can access arbitrary blocks within a matrix.
  // For our purposes, it is actually simpler to use leftCols() and rightCols().
  Anew.leftCols(q) = C;

  // TO DO: Use A.rightCols() to fill in the remaining columns of the new matrix
  // A. START
  Anew.rightCols(p) = B;
  // END
```

```
  // Tip:  Many more methods exist that are special cases of block(),
  // e.g.  topRows(), bottomRows(), topLeftCorner(),
     bottomLeftCorner(), ...
  // For vectors we have head(), tail(), and segment().

  return Anew;
}
```

▲

(1-2.c) Supplement the missing lines of the function

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

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

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

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

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):

C++ code 1.2.3: Initialization of a triangular matrix in EIGEN.
Get it on GitLab (MatrixBlocks.hpp).

```
MatrixXd tridiagonal(int n, double a, double b, double c) {
  /*
   * This function creates an n by n tridiagonal matrix with the values
   *      a on the first subdiagonal,
   *      b on the diagonal, and
```

```
    *      c on the first superdiagonal.
    * Example for n=5:
    *     [ b c 0 0 0 ]
    *     [ a b c 0 0 ]
    * A = [ 0 a b c 0 ]
    *     [ 0 0 a b c ]
    *     [ 0 0 0 a b ]
    */

   MatrixXd A;

   // TO DO: Fill the matrix A with zeros.  Then fill the subdiagonal with the
   // value a, the diagonal with b and the superdiagonal with c.  Hint: You can
   // get the diagonal of A by A.diagonal().  Moreover, you can get the super- and
   // subdiagonals by passing +1 or -1 as arguments to A.diagonal().  START
   A = MatrixXd::Zero(n, n);
   A.diagonal(-1) = VectorXd::Constant(n - 1, a);
   A.diagonal() = VectorXd::Constant(n, b);
   A.diagonal(1) = VectorXd::Constant(n - 1, c);
   // END
   return A;
}
```

▲

End Problem 1-2

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) 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`.

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).

```
double average(const MatrixXd &A) {
  /*
   * Calculates the average of the entries of A
   */
  double m;
  m = A.sum() / A.size();
  return m;
}
```

▲

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

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

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

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):

C++ code 1.3.2: Determines the percentage of (exactly) zero entries
Get it on GitLab (MatrixReduce..hpp).

```
double percent_zero(const MatrixXd &A) {
  /*
   * Calculates how many entries of A are exactly equal to zero,
   * as a percentage of the total number of entries.
   */

  // Eigen provides an Array class, that is similar to Matrix,
  // but has operations that are entry-wise rather than linear algebra
  // operations.  For example, multiplying two Arrays is entry-wise
  // multiplication, and not matrix multiplication.
  // The benefit of using an Array here is that we can do entry-wise boolean
  // comparison, which is not available for matrices.
  ArrayXXd Arr = A.array();

  double ratio;
  // TO DO: Calculate the ratio of zero entries in A.
  // START
  int zeros = (Arr == 0).count();
  ratio = ((double)zeros) / A.size();
  // END

  return ratio * 100;
}
```

▲

(1-3.c) 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.

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):

C++ code 1.3.3: Checking for an exactly vanishing column
Get it on GitLab (MatrixReduce..hpp).

```
bool has_zero_column(const MatrixXd &A) {
  /*
   * Returns 1 if A has a column that is exactly equal to zero.
```

```
    * Returns 0 otherwise.
    */

  bool result;

  // A vector is the zero vector if and only if it has norm 0.
  // The following vector contains the squared norms of the columns of A.
  VectorXd norms = A.colwise().squaredNorm();
  // The norm is 0 if and only if the norm squared is zero.
  // We use squaredNorm() instead of norm(), because norm() =
  // sqrt(squaredNorm()) calculates a square root that is not necessary for our
  // purposes.

  // TO DO: Check if any one of the norms is equal to zero.
  // Hint:  Use an array to perform entry-wise comparison.
  // START
  result = (norms.array() == 0).any();
  // END

  return result;
}
```

▲

(1-3.d) 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.

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 $\mathbf{C}$ 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):

C++ code 1.3.4: Setting diagonals to make row sums vanish
Get it on GitLab (MatrixReduce..hpp).

```
MatrixXd columns_sum_to_zero(const MatrixXd &A) {
  /*
   * Returns a matrix that is like A, but the entries on the diagonal
   * have been changed so that the sum of the columns is equal to 0.
   */

  MatrixXd B(A);
```

```

  // TO DO: Replace the diagonal of B with values such that the columns of B sum
  // up to zero.  Hint:  Use diagonal(), rowwise(), and sum().  START
  int p = std::min(B.rows(), B.cols());
  B.diagonal() = VectorXd::Zero(p);
  B.diagonal() = -B.rowwise().sum();
  // END

  return B;
}
```

▲

End Problem 1-3

Chapter 2

Computing with Matrices and Vectors

Problem 2-1: Arrow matrix×vector multiplication

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

Related to [Lecture → 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++11-code 2.1.1: Computing $\mathbf{A}^2\mathbf{y}$ for an arrow matrix $\mathbf{A}$

```
void arrow_matrix_2_times_x(const VectorXd &d, const VectorXd &a,
                            const VectorXd &x, VectorXd &y) {
  assert(d.size() == a.size() && a.size() == x.size() &&
         "Vector size must be the same!");
  int n = d.size();
  VectorXd d_head = d.head(n - 1);
  VectorXd a_head = a.head(n - 1);
  MatrixXd d_diag = d_head.asDiagonal();
  MatrixXd A(n, n);
  A << d_diag, a_head, a_head.transpose(), d(n - 1);
  y = A * A * x;
}
```

Get it on GitLab (arrowmatvec.cpp).

(2-1.a) (10 min.) For general vectors $\mathbf{d} = [d_1, \dots, d_n]^\top$ and $\mathbf{a} = [a_1, \dots, a_n]^\top$ sketch the pattern, that is, the `spy`-plot, of the matrix $\mathbf{A}$ created in the function `arrow_matrix_2_times_x` in Code 2.1.1.

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

$$\mathbf{A} = \begin{bmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \tag{2.1.2}$$

From this the `spy`-plot is clear.

▲

(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].

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.

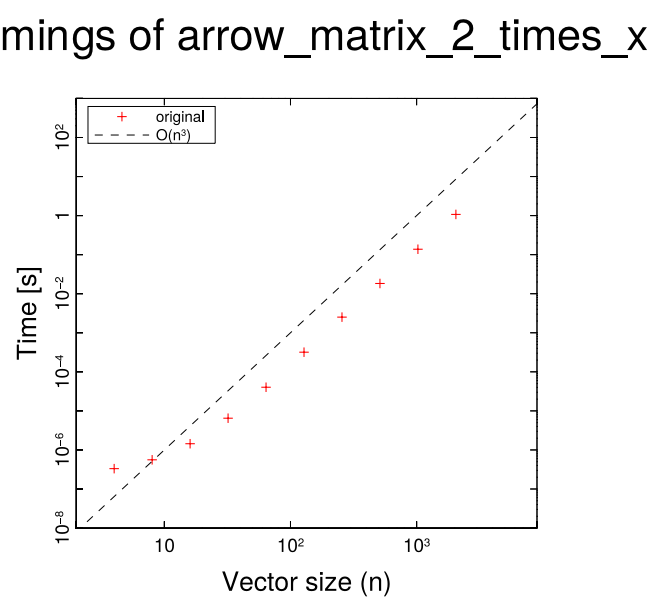


Fig. 1

SOLUTION of (2-1.b):

The standard matrix-matrix multiplication has complexity $O(n^3)$ and the standard matrix-vector multiplication has complexity $O(n^2)$. Hence, the overall computational complexity is dominated by $O(n^3)$.

In this case, the line affecting the complexity is `y = A*A*x`.

Remember that most of C++ operators have a **left-to-right** precedence. This means that, even if, mathematically, $(\mathbf{AA})\mathbf{y} = \mathbf{A}(\mathbf{Ay})$, when implementing the expression in C++, the *complexity* of the code will be very different. The left hand side has complexity $O(n^3)$, whilst the right hand size has complexity $O(n^2)$.

Remark: Even EIGEN's own internal template expression mechanism is not able to efficiently exploit this fact.

▲

(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 $[d_1, \dots, d_n]^\top$ and `a` passes the vector $[a_1, \dots, a_n]^\top$.

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:

$$\mathbf{A} = \begin{bmatrix} d_1 & & & & a_1 \\ & d_2 & & & a_2 \\ & & \ddots & & \vdots \\ & & & d_{n-1} & a_{n-1} \\ a_1 & a_2 & \dots & a_{n-1} & d_n \end{bmatrix} \tag{2.1.3}$$

$$= \operatorname{diag}(\mathbf{d}) + [\mathbf{0}, \dots, \mathbf{0}, \mathbf{a}_-] + [\mathbf{0}, \mathbf{0}, \mathbf{0}, \mathbf{a}_-]^\top =: \mathbf{D} + \mathbf{V} + \mathbf{H}. \tag{2.1.4}$$

and $\mathbf{Ax} = \mathbf{Dx} + \mathbf{Vx} + \mathbf{Hx}$. Each of these multiplications can obviously be done in $O(n)$ operations, see Code 2.1.5.

C++11-code 2.1.5: Solution of (2-1.c)

```
void efficient_arrow_matrix_2_times_x(const VectorXd &d, const VectorXd &a,
                                      const VectorXd &x, VectorXd &y) {
  assert(d.size() == a.size() && a.size() == x.size() &&
         "Vector size must be the same!");
  int n = d.size();

  // Notice that we can compute (A*A)*x more efficiently using
  // A*(A*x).  This is, in fact, performing two matrix vector
  // multiplications
  // instead of a more expensive matrix-matrix multiplication.
  // Therefore, as first step, we need a way to efficiently
  // compute A*x

  // This function computes A*x.  you can use it
  // by calling A_times_x(x).
  // This is the syntax for lambda functions:  notice the extra
  // [variables] code.  Each variable written within [] brackets
  // will be captured (i.e.  seen) inside the lambda function.
  // Without &, a copy of the variable will be performed.
  // Notice that A = D + H + V, s.t.  A*x = D*x + H*x + V*x
  // D*x can be rewritten as d*x componentwise
  // H*x is zero, except at the last component
```

```
  // V*x is only affected by the last component of x
  auto A_times_x = [&a, &d, n](const VectorXd &x) {
    // This takes care of the diagonal (D*x)
    // Notice:  we use d.array() to tell Eigen to treat
    // a vector as an array.  As a result:  each operation
    // is performed componentwise.
    VectorXd Ax = (d.array() * x.array()).matrix();
    // H*x only affects the last component of A*x
    // This is a dot product between a and x with the last
    // component removed
    Ax(n - 1) += a.head(n - 1).dot(x.head(n - 1));
    // V*x is equal to the vector
    // (a(0)*x(n-1), ..., a(n-2)*x(n-1), 0)
    Ax.head(n - 1) += x(n - 1) * a.head(n - 1);
    return Ax;
  };

  // <=> y = A*(A*x)
  y = A_times_x(A_times_x(x));
}
```

Get it on GitLab (arrowmatvec.cpp).

Note the use of lambda functions [Lecture → Section 0.2.3] in order to achieve a streamlined implementation. Also note the use of the method `head` to access the top components of a vector.

▲

(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 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 $O(n)$. Therefore the complexity is $O(n)$.

▲

(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:

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

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

§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

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

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

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

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

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

First include the `chrono` STL header file (note: this requires C++11). The `chrono` header provides the function:

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

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

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

C++11-code 2.1.9: `now()` function

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

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

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

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

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

SOLUTION 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 Code 2.1.10 and Fig. 2, 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 implementaitons in `arrowmatvec.cpp` ▷

(The file `arrowmatvec.cpp` also contains an implementation using `chrono`.)

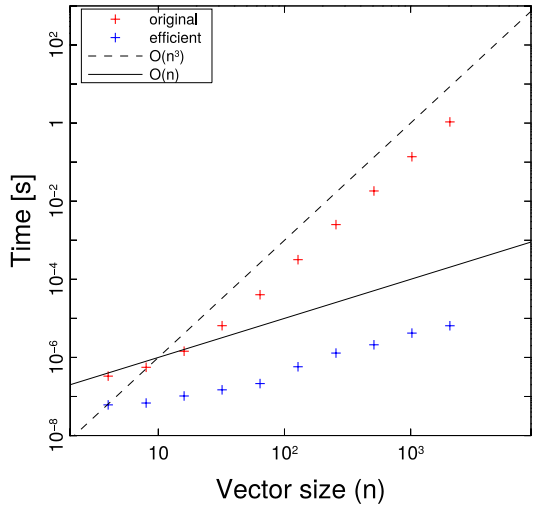


Fig. 2

C++11-code 2.1.10: Computation of timings of arrow matrix multiplication

```
std::cout << std::setw(8) << "n" << std::setw(15) << "original"
          << std::setw(15) << "efficient" << std::endl;
for (unsigned n = 4; n <= 2048; n *= 2) {
  // Number of repetitions
  unsigned int repeats = 10;

  Timer timer, timer_eff;
  // Repeat test many times
  for (unsigned int r = 0; r < repeats; ++r) {
    // Create test input using random vectors
    Eigen::VectorXd a = Eigen::VectorXd::Random(n),
                    d = Eigen::VectorXd::Random(n),
                    x = Eigen::VectorXd::Random(n), y;

    // Compute times
    timer.start();
    arrow_matrix_2_times_x(d, a, x, y);
    timer.stop();

    // Compute times for efficient implementation
    timer_eff.start();
    efficient_arrow_matrix_2_times_x(d, a, x, y);
    timer_eff.stop();
  }

  // Print results
```

```
    std::cout << std::setw(8) << n << std::scientific << std::setprecision(3)
              << std::setw(15) << timer.min() << std::setw(15)
              << timer_eff.min() << std::endl;
  }
```

Get it on GitLab (arrowmatvec.cpp).

The file `arrowmatvec.cpp` also contains an implementation using `chrono`.

▲

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. 0.2.5.29], involves C++ implementation with EIGEN.

In [Lecture → Ex. 0.2.5.29] and [Lecture → Code 0.2.5.30] 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 0.2.5.30] in terms of the matrix dimension n, if the argument matrix `A` has size $n \times n$ and no premature termination occurs.

SOLUTION 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 $n \to \infty$.

▲

(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 0.2.5.30]:

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

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

Use EIGENs block operations see EIGEN documentation.

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

```
MatrixXd gram_schmidt(const MatrixXd &A) {
  // We create a matrix Q with the same size and data of A
  MatrixXd Q(A);

  // The first vector just gets normalized
  Q.col(0).normalize();
  // Iterate over all other columns of A
  for (unsigned int j = 1; j < A.cols(); ++j) {
    // See eigen documentation for usage of col and leftCols
    Q.col(j) -= Q.leftCols(j) * (Q.leftCols(j).transpose() * A.col(j));
    // Normalize vector, if possible
    // (otherwise it means columns of A are
    // almost linearly dependant)
```

```
    double eps = std::numeric_limits<double>::denorm_min();
    if (Q.col(j).norm() <= eps * A.col(j).norm()) {
      std::cerr << "Gram-Schmidt failed because "
                << "A has (almost) linear dependant "
                << "columns. Bye." << std::endl;
      break;
    } else {
      Q.col(j).normalize();
    }
  }
  return Q; // Efficient due to return-value optimization
}
```

Get it on GitLab (gramschmidt.cpp).

▲

(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

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

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

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 $\mathbf{Q}^\top\mathbf{Q} = \mathbf{I}$ (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

```
int main(void) {
  // Orthonormality test
  unsigned int n = 9;
  MatrixXd A = MatrixXd::Random(n, n);
  MatrixXd Q = gram_schmidt(A);
  // Compute how far is Q^T * Q from the identity
  // i.e.  "How far is Q from being orthonormal?"
```

```
  double err = (Q.transpose() * Q - MatrixXd::Identity(n, n)).norm();
  // Error has to be small, but not zero (why?)
  std::cout << "Error is: " << err << std::endl;
  // If the error is too big, we exit with error
  double eps = std::numeric_limits<double>::denorm_min();
  exit(err < eps);
}
```

Get it on GitLab (gramschmidt.cpp).

▲

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].

(2-3.a) (5 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 of (2-3.a):

The matrix $\mathbf{C}$ is:

$$\mathbf{C} = \begin{bmatrix} 5 & 6 & 10 & 12 \\ 7 & 8 & 14 & 16 \\ 15 & 18 & 20 & 24 \\ 21 & 24 & 28 & 32 \end{bmatrix}.$$

▲

(2-3.b) (15 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 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

```
void kron(const MatrixXd &A, const MatrixXd &B, MatrixXd &C) {
  // Allocate enough space for the matrix
  C = MatrixXd(A.rows() * B.rows(), A.cols() * B.cols());
  // TO DO: (2-3.b) Fill in the entries of C.
  // Hint: Use a nested for-loop and C.block().
```

```
  // START
  for (unsigned int i = 0; i < A.rows(); ++i) {
    for (unsigned int j = 0; j < A.cols(); ++j) {
      // We use eigen block operations to set the values of
      // each n × n block.
      C.block(i * B.rows(), j * B.cols(), B.rows(), B.cols()) =
          A(i, j) * B; // ∈ ℝ^(n×n)
    }
  }
  // END
}
```

Get it on GitLab (kron.cpp).

In fact, EIGEN itself provides an implementation of the Kronecker product, see EIGEN documentation.

▲

(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 $n \to \infty$ 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 of (2-3.c):

The code loops two times from 0 to n, i.e. is performing the inner poriton 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 $O(n^2 \cdot n^2) = O(n^4)$.

▲

(2-3.d) 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.

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 $\tilde{\mathbf{x}}_j := (x_{j\cdot n+1}, \dots, x_{(j+1)\cdot n})^\top \in \mathbb{R}^n$ (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

```
void kron_mult(const MatrixXd &A, const MatrixXd &B, const VectorXd &x,
               VectorXd &y) {
  assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
         "Matrices A and B must be square matrices with same size!");
  assert(x.size() == A.cols() * A.cols() &&
         "Vector x must have length A.cols()^2");
  unsigned int n = A.rows();

  // Allocate space for output
  y = VectorXd::Zero(n * n);

  // TO DO: (2-3.d) Fill in the entires of y.
  // Hint: Use a nested for-loop, x.segment(), and y.segment().
  // In the outer loop, you can perform a computation based on
  // B and x, and save the result in a variable that is reused in
  // each iteration of the inner loop.
  // START

  // Note: this is like doing a matrix-vector multiplication
  // where the entries of the matrix are smaller matrices
  // and entries of the vector are smaller vectors

  // Loop over all segments of x (x̃)
  for (unsigned int j = 0; j < n; ++j) {
    // Reuse computation of z
    VectorXd z = B * x.segment(j * n, n);
    // Loop over all segments of y
    for (unsigned int i = 0; i < n; ++i) {
      y.segment(i * n, n) += A(i, j) * z;
    }
  }
  // END
}
```

Get it on GitLab (kron.cpp).

▲

(2-3.e) 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.

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 undelying 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()`

```
void kron_reshape(const MatrixXd &A, const MatrixXd &B, const VectorXd &x,
                  VectorXd &y) {
  assert(A.rows() == A.cols() && A.rows() == B.rows() && B.rows() == B.cols() &&
         "Matrices A and B must be square matrices with same size!");
  unsigned int n = A.rows();

  // TO DO: (2-3.e) Fill in the entires of y.
  // Hint: Use MatrixXd::Map() to reshape x into a n by n matrix.
  // Then y is obtained by simple matrix multiplications and
  // another reshape.
  // START
  MatrixXd t = B * MatrixXd::Map(x.data(), n, n) * A.transpose();
  y = MatrixXd::Map(t.data(), n * n, 1);
  // END
}
```

Get it on GitLab (kron.cpp).

▲

(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 of (2-3.f):

The following code is an example of runtime computation:

C++-code 2.3.4: Runtime comparison

```
void kron_runtime() {

  MatrixXd A, B, C;
  VectorXd x, y;
  // We repeat each runtime measurement 10 times
  // (this is done in order to remove outliers).
  unsigned int repeats = 10;

  std::cout << "Runtime for each implementation." << std::endl;
  std::cout << std::setw(5) << "n" << std::setw(15) << "kron" << std::setw(15)
```

```
              << "kron_mult" << std::setw(15) << "kron_reshape" << std::endl;
  // Loop from M = 2,...,2^8
  for (unsigned int M = 2; M <= (1 << 8); M = M << 1) {
    Timer tm_kron, tm_kron_mult, tm_kron_map;
    // Run experiments "repeats" times
    for (unsigned int r = 0; r < repeats; ++r) {
      // Random matrices for testing
      A = MatrixXd::Random(M, M);
      B = MatrixXd::Random(M, M);
      x = VectorXd::Random(M * M);

      // Do not want to use kron for large values of M
      if (M < (1 << 6)) {
        // Kron using direct implementation
        tm_kron.start();
        kron(A, B, C);
        y = C * x;
        tm_kron.stop();
      }

      // TO DO: (2-3.f) Measure the runtime of kron_mult()
      // and kron_reshape().
      // START

      // Kron matrix-vector multiplication
      tm_kron_mult.start();
      kron_mult(A, B, x, y);
      tm_kron_mult.stop();

      // Kron using reshape
      tm_kron_map.start();
      kron_reshape(A, B, x, y);
      tm_kron_map.stop();

      // END
    }

    double kron_time = (M < (1 << 6)) ? tm_kron.min() : std::nan("");
    std::cout << std::setw(5) << M << std::scientific << std::setprecision(3)
              << std::setw(15) << kron_time << std::setw(15)
              << tm_kron_mult.min() << std::setw(15) << tm_kron_map.min()
              << std::endl;
  }
}
```

Get it on GitLab (kron.cpp).

▲

End Problem 2-3 , 30 min.

Problem 2-4: Fast matrix multiplication with EIGEN

[Lecture → Rem. 1.4.2.1] 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. 1.4.2.1]) to multiply the two square matrices $\mathbf{A}$ and $\mathbf{B}$ of size $n = 2^k$, $k \in \mathbb{N}$, and returns the result as output.

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

SOLUTION of (2-4.a):

The following code implements the Strassen's recursive algorithm:

C++-code 2.4.1: Strassen's algorithm with EIGEN

```
MatrixXd strassenMatMult(const MatrixXd& A, const MatrixXd& B) {
    // Ensure square matrix
    assert(A.rows() == A.cols() && "Matrix A must be square");
    assert(B.rows() == B.cols() && "Matrix B must be square");
    // Matrix dimension must be a power of 2
    assert(A.rows() % 2 == 0 && "Matrix dimensions must be a power of two.");

    const unsigned n = A.rows();

    // The function is recursive and acts on blocks of size n/2 × n/2
    // i.e.  exploits fast product of 2x2 block matrix
    if ( n==2 ) { // End of recursion
        MatrixXd C(2, 2);
        C << A(0,0)*B(0,0) + A(0,1)*B(1,0),
             A(0,0)*B(0,1) + A(0,1)*B(1,1),
             A(1,0)*B(0,0) + A(1,1)*B(1,0),
             A(1,0)*B(0,1) + A(1,1)*B(1,1);
        return C;
    }

    MatrixXd Q0(n/2, n/2), Q1(n/2, n/2), Q2(n/2, n/2), Q3(n/2, n/2),
             Q4(n/2, n/2), Q5(n/2, n/2), Q6(n/2, n/2);

    MatrixXd A11 = A.topLeftCorner(n/2, n/2);
    MatrixXd A12 = A.topRightCorner(n/2, n/2);
    MatrixXd A21 = A.bottomLeftCorner(n/2, n/2);
    MatrixXd A22 = A.bottomRightCorner(n/2, n/2);

    MatrixXd B11 = B.topLeftCorner(n/2, n/2);
    MatrixXd B12 = B.topRightCorner(n/2, n/2);
    MatrixXd B21 = B.bottomLeftCorner(n/2, n/2);
    MatrixXd B22 = B.bottomRightCorner(n/2, n/2);

    Q0 = strassenMatMult(A11 + A22, B11 + B22);
```

```
    Q1 = strassenMatMult(A21 + A22, B11);
    Q2 = strassenMatMult(A11,       B12 - B22);
    Q3 = strassenMatMult(A22,       B21 - B11);
    Q4 = strassenMatMult(A11 + A12, B22);
    Q5 = strassenMatMult(A21 - A11, B11 + B12);
    Q6 = strassenMatMult(A12 - A22, B21 + B22);

    MatrixXd C(n,n);

    C << Q0 + Q3 - Q4 + Q6,
         Q2 + Q4,
         Q1 + Q3,
         Q0 + Q2 - Q1 + Q5;

    return C;
}
```

Get it on GitLab (strassen.hpp).

▲

(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).

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

```

    // Seed the random number generator
    srand((unsigned int) time(0));

    //// Check algorithm for correctness

    // Size of the matrix
    int k = 2;
    int n = std::pow(2, k);

    // Generate random matrices
    MatrixXd A = MatrixXd::Random(n,n);
    MatrixXd B = MatrixXd::Random(n,n);

    // Testing matrix multiplication
    MatrixXd AB = strassenMatMult(A,B);
    // Eigen Matrix multiplication
    MatrixXd AxB = A*B;

    std::cout << "Using Strassen's method, A*B=" << std::endl
```

```
                << AB << std::endl;
    std::cout << "Using standard method, A*B=" << std::endl
                << AxB << std::endl;
    std::cout << "The norm of the error is: "
                << (AB-AxB).norm() << std::endl;
```

Get it on GitLab (test.cpp).

▲

(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 of (2-4.c):

C++-code 2.4.3: Runtime of Strassen's algorithm EIGEN

```
    // Store seed for rng
    unsigned seed = (unsigned) time(0);
     // seed random number generator
    srand(seed);

    // Minimum number of repetitions
    unsigned int repetitions = 10;

    // Display header column
    std::cout << std::setw(4)  << "k"
              << std::setw(15) << "A*B"
              << std::setw(15) << "Strassen" << std::endl;
    for(unsigned k = 4; k <= 9; k++) {
        unsigned int n = std::pow(2, k);

        // Initialize random input matricies
        MatrixXd A = MatrixXd::Random(n, n);
        MatrixXd B = MatrixXd::Random(n, n);

        // Timer to collect runtime of each individual run
        Timer timer, timer_own;

        for(unsigned int r = 0; r < repetitions; ++r) {
            // initialize memory for result matrix
            MatrixXd AxB, AxB2;

            // Benchmark eigens matrix multiplication
            timer.start(); // start timer
            AxB=(A*B); // do the multiplication
```

```
            timer.stop(); // stop timer

            // Benchmark Stassens matrix multiplication
            timer_own.start(); // start timer
            AxB2=strassenMatMult(A, B); // do the multiplication
            timer_own.stop(); // stop timer

            // volatile double a = (AxB+AxB2).norm();
        }

        // Print runtimes
        std::cout << std::setw(4) << k // power
                  << std::setprecision(3) << std::setw(15) << std::scientific
                  << timer.min() // eigen timing
                  << std::setprecision(3) << std::setw(15) << std::scientific
                  << timer_own.min() // strassing timing
                  << std::endl;

    }
```

Get it on GitLab (timing.cpp).

▲

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

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

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

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

that implements the algorithm from Pseudocode 2.5.1.

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 2.5.1 making use of EIGEN's matrix-block operations.

C++-code 2.5.2: C++ implementation of Code 2.5.1

```
void houserefl(const Eigen::VectorXd & v, Eigen::MatrixXd & Z)
{
    unsigned int n = v.size();
    Eigen::VectorXd w = v.normalized();
    Eigen::VectorXd u=w;
    u(0) += 1;
    Eigen::VectorXd q = u.normalized();
    Eigen::MatrixXd X = Eigen::MatrixXd::Identity(n, n) - 2*q*q.transpose();
    Z = X.rightCols(n-1);
}
```

Get it on GitLab (houserefl.cpp).

▲

(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.

HINT 1 for (2-5.b): Notice that $|\mathbf{q}|^2 = 1$.

SOLUTION of (2-5.b):

By straightforward computations:

$$\begin{aligned}\mathbf{X}^\top \mathbf{X} &= (\mathbf{I}_n - 2\mathbf{q}\mathbf{q}^\top)(\mathbf{I}_n - 2\mathbf{q}\mathbf{q}^\top)\\ &= \mathbf{I}_n - 4\mathbf{q}\mathbf{q}^\top + 4\mathbf{q}\underbrace{\mathbf{q}^\top\mathbf{q}}_{=|\mathbf{q}|=1}\mathbf{q}^\top\\ &= \mathbf{I}_n - 4\mathbf{q}\mathbf{q}^\top + 4\mathbf{q}\mathbf{q}^\top\\ &= \mathbf{I}_n.\end{aligned}$$

This is what we wanted to prove. □

▲

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

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 $\mathbf{X} = [\mathbf{X}_1, \cdots, \mathbf{X}_n]\mathbb{R}^{n,n}$ be the matrix defined in Code 2.5.1. Let $\mathbf{e}^{(1)} := [1, 0, \cdots, 0]^\top$. In view of the identity $\mathbf{X}_1 = \mathbf{e}^{(1)} - 2q_1\mathbf{q}$ we have:

$$\mathbf{X}_1 = \begin{bmatrix}1-2q_1^2\\-2q_1q_2\\\vdots\\-2q_1q_n\end{bmatrix} = \begin{bmatrix}1-2\frac{u_1^2}{\sum_{i=1}^n u_i^2}\\-2\frac{u_1u_2}{\sum_{i=1}^n u_i^2}\\\vdots\\-2\frac{u_1u_n}{\sum_{i=1}^n u_i^2}\end{bmatrix} \overset{\text{HINT}}{=} \begin{bmatrix}\frac{(w_1+1)^2+w_2^2+\cdots+w_n^2-2(w_1+1)^2}{(w_1+1)^2+w_2^2+\cdots+w_n^2}\\-\frac{2(w_1+1)w_2}{(w_1+1)^2+w_2^2+\cdots+w_n^2}\\\cdots\\-\frac{2(w_1+1)w_n}{(w_1+1)^2+w_2^2+\cdots+w_n^2}\end{bmatrix} \overset{|\mathbf{w}|=1}{=} \begin{bmatrix}-\frac{2w_1(w_1+1)}{2(w_1+1)}\\-\frac{2(w_1+1)w_2}{2(w_1+1)}\\\cdots\\-\frac{2(w_1+1)w_n}{2(w_1+1)}\end{bmatrix} = -\mathbf{w},$$

gather* which is a multiple of $\mathbf{v}$, since $\mathbf{w} = \frac{\mathbf{v}}{|\mathbf{v}|}$. □

▲

(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 `houserefl`?

SOLUTION of (2-5.d):

The columns of $\mathbf{X} = [\mathbf{X}_1, \cdots, \mathbf{X}_n]$ are an orthonormal basis (ONB) of $\mathbb{R}^n$. Thus, the columns of $\mathbf{Z} = [\mathbf{X}_2, \cdots, \mathbf{X}_n]$ are an ONB of the complement of $\mathrm{Span}(\mathbf{X}_1) \overset{(2-5.c)}{=} \mathrm{Span}(\mathbf{v})$. The function `houserefl` computes an ONB of the complement of $\mathbf{v}$.

▲

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

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

Specify it in leading order using the Landau symbol O.

SOLUTION 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 2.5.2.

▲

End Problem 2-5 , 70 min.

Problem 2-6: Matrix powers

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

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

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

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

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

(2-6.a) Implement a C++ function

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

that, using only basic linear algebra operations (including matrix-vector or matrix-matrix multiplications), computes the kth 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 if contains elements of $\mathbb{C}$.

HINT 1 for (2-6.a): Reuse matrix products. Express the exponent k using a binary representation.

Remark: a `MatrixXcd` is defined internally (in EIGEN), as:

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

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

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

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

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

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

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

SOLUTION of (2-6.a):

Write k in binary format: $k = \sum_{j=0}^{M} b_j\, 2^j,\ b_j \in \{0,1\}$. Then

$$\mathbf{A}^k = \prod_{j=0}^{M} \mathbf{A}^{2^j\, b_j} = \prod_{j\ s.t.\ b_j=1} \mathbf{A}^{2^j}.$$

We compute $\mathbf{A},\ \mathbf{A}^2,\ \mathbf{A}^4,\ \ldots, \mathbf{A}^{2^M}$ (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

```
MatrixXcd matPow(MatrixXcd & A, unsigned int k) {
    // Identity matrix
    MatrixXcd X = MatrixXcd::Identity(A.rows(), A.cols());

    // p is used as binary mask to check whether,
    // given k = Σ_{i=0}^{M} b_i 2^i,  b_i = 1
    // (i.e.  if k has a 1 in the i-th binary digit).
    // Obtaining the binary representation of p can be done in many
       ways,
    // here we use ~k&p to check if b_i = 1
    unsigned int p = 1;

    // Cycle all the way up to the 1st 1 in the binary
    // representation of k (this is called the "most significant bit")
    // If you are interested in other methods to compute the "most
       significant
    // bit", consult the file "most_significant_bit.cpp"
    for(unsigned int j = 1; j <= std::ceil(std::log2(k)); ++j) {

      if( ( ~k & p ) == 0 ) { // if( b_i ≠ 0 )
            X = X*A;
        }

        A = A*A;
        p = p << 1; // p = p*2;
    }
    return X;
}
```

Get it on GitLab (matPow.cpp).

▲

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

SOLUTION of (2-6.b):

Using a "naive" implementation according to

$$\mathbf{A}^k = \underbrace{\Big(\ldots\big(\big(\mathbf{A}\cdot\mathbf{A}\big)\cdot\mathbf{A}\big)\ldots\cdot\mathbf{A}\Big)}_{k}\cdot\mathbf{A},$$

the complexity is $O\big((k-1)n^3\big)$.

Using the efficient implementation from Code 3.16.6, for each $j \in \{1,2,\ldots,\lceil\log_2(k)\rceil\}$ we have to perform at most two multiplications (`X*A` and `A*A`):

$$\text{complexity} = 2\cdot M\cdot C = 2\cdot\lceil\log_2 k\rceil\cdot O(n^3).$$

where $M := \lceil\log_2(k)\rceil$ is the loop size, and C is the cost of a matrix-matrix multiplication. (Here $\lceil a\rceil = \texttt{ceil}(a) := \min\{b\in\mathbb{Z},\ a\le b\}$.)

▲

(2-6.c) ⚁ Compute the runtime of the EIGEN power function (`MatrixXxd::pow(int)`) and find out the complexity w.r.t k. Compare this with the function `matPow` from (2-6.a). For $n > 0$, use the $n \times n$ matrix defined by

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

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

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

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

SOLUTION of (2-6.c):

Note: 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 ∞ when k grows.

C++11-code 2.6.2: Runtime computation

```

    // Will contain runtimes for each k, k stored in powers
    std::vector<double> powers, times_matPow, times_Eigen_pow;

    std::cout << "— Table of runtimes" << std::endl;
    std::cout << std::setw(15) << "K"
              << std::setw(15) << "Own matPow"
              << std::setw(15) << "Eigen pow()" << std::endl;
    // Loop from 2 to 2^31
    for(unsigned int K = 2; K <= (1u << 31); K = K << 1) {

        unsigned int repeats = 10;
        Timer tm_pow, tm_Eigen_pow;

        // Repeat the test repeat times
        for(unsigned int r = 0; r < repeats; ++r) {

            // Build Vandermonde matrix of size n
            MatrixXcd X, A = construct_matrix(n);

            // Compute runtime if own implementation
            tm_pow.start();
            X = matPow(A, K);
            tm_pow.stop();

            // Compute runtime of eigen implementation
            A = construct_matrix(n);
            tm_Eigen_pow.start();
            X = A.pow(K);
            tm_Eigen_pow.stop();
        }

        // Output table
        std::cout << std::setw(15) << K
```

```
                    << std::scientific << std::setprecision(3)
                    << std::setw(15) << tm_pow.min()
                    << std::setw(15) << tm_Eigen_pow.min() << std::endl;

        // Store data
        powers.push_back( K );
        times_matPow.push_back( tm_pow.min() );
        times_Eigen_pow.push_back( tm_Eigen_pow.min() );
    }
```

Get it on GitLab (matPow.cpp).

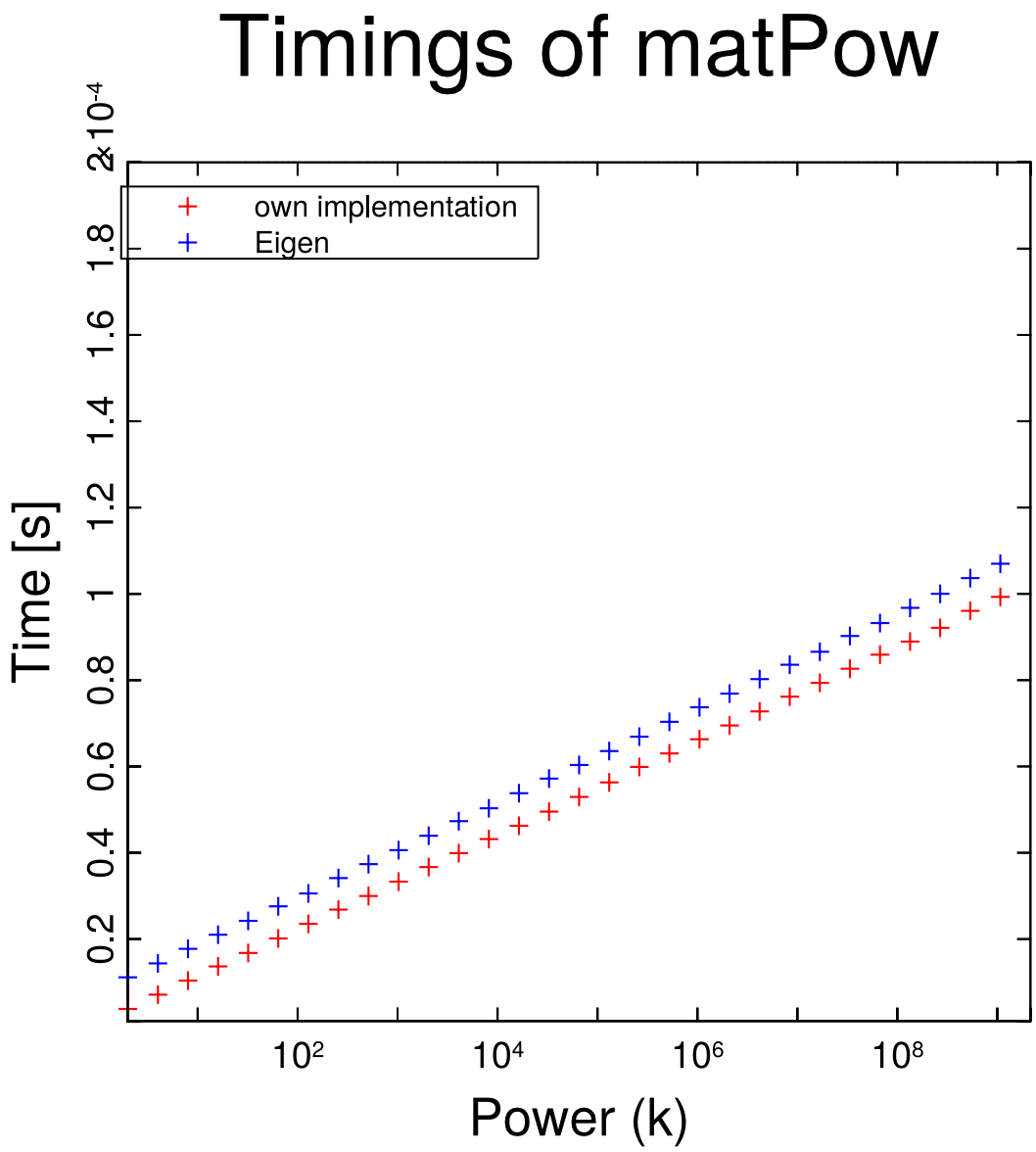


Fig. 3

▲

End Problem 2-6

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

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

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

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

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

Get it on GitLab (multAmin.cpp).

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

SOLUTION of (2-7.a):

Notice the following:

- the outer product of two vectors has complexity $O(n^2)$;
- the Matrix–vector multiplication has quadratic complexity $O(n^2)$;
- component-wise minimum has complexity $O(n^2)$.

Therefore, the overall complexity is $O(n^2)$.

▲

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

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

that computes the same multiplication as Code 2.7.5 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 2.7.5.

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 $j = 1, \cdots, n$ we have

$$y_j = \sum_{k=1}^{j} k x_k + j \sum_{k=j+1}^{n} x_k.$$

Recursively define for $j = 2, \ldots, n$:

$$v_j := v_{j-1} + x_{n-j+1} = \sum_{k=n-j+1}^{n} x_k$$

and

$$w_j := w_{j-1} + j x_j = \sum_{k=1}^{j} k x_k$$

with $v_1 = x_n$ and $w_1 = x_1$. You can easily show that:

$$v_j = \sum_{k=n-j+1}^{n} x_k, w_j = \sum_{k=1}^{j} k x_k$$

Then

$$y_j = \sum_{k=1}^{j} k x_k + j \sum_{k=j+1}^{n} x_k = w_j + j v_{n-j}. \tag{2.7.3}$$

We can compute w_j and v_j for $j = 1, \ldots, n$ with a for loop of size n. Once we computed w_j and v_j, we can use (2.7.3) to compute y_j for each $j = 1 \ldots, n$ 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 $\mathbf{Ax}$

```
void multAmin(const VectorXd & x, VectorXd & y) {
    unsigned int n = x.size();
    y = VectorXd::Zero(n);

    // TO DO: (2-7.b) Fill in the entries of y.
    // Hint: Find an expression y(j) = w(j) + j*u(j) such that
    // the entries of w and u can be calculated recursively.
    // START
    VectorXd v = VectorXd::Zero(n);
    VectorXd w = VectorXd::Zero(n);

    v(0) = x(n-1);
    w(0) = x(0);

    for(unsigned int j = 1; j < n; ++j) {
        v(j) = v(j-1) + x(n-j-1);
        w(j) = w(j-1) + (j+1)*x(j);
    }
    for(unsigned int j = 0; j < n-1; ++j) {
        y(j) = w(j) + v(n-j-2)*(j+1);
    }
    y(n-1) = w(n-1);
```

```
    // END
}
```

Get it on GitLab (multAmin.cpp).

▲

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

SOLUTION of (2-7.c):

Only simple loops of size n are performed. Therefore we have linear complexity: $O(n)$.

▲

(2-7.d) (20 min.) Compare the runtimes of your implementation and the implementation given in Code 2.7.5 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 of (2-7.d):

The matrix multiplication in (2.7.5) has complexity $O(n^2)$. The faster implementation has complexity $O(n)$.

C++11-code 2.7.5: Runtime test for `multAmin`

```
    // Timing from 2^4 to 2^10 repeating "nruns" times
    unsigned int nruns = 10;

    std::cout << "--> Timings:" << std::endl;
    // Header, see iomanip documentation
    std::cout << std::setw(15)
              << "N"
              << std::scientific << std::setprecision(3)
              << std::setw(15) << "multAminSlown"
              // << std::setw(15) << "multAminLoops"
              << std::setw(15) << "multAmin"
              << std::endl;
    // From 2^4 to 2^10
    for(unsigned int N = (1 << 4); N <= (1 << 10); N = N << 1) {
        Timer tm_slow, tm_fast;
        // TO DO: (2-7.d) Compute runtimes of multAminSlow(x,y) and
        // multAmin(x,y) with x = VectorXd::Random(N). Repeat nruns times.
        // START
        for(unsigned int r = 0; r < nruns; ++r) {
            VectorXd x = VectorXd::Random(N);
            VectorXd y;

            // Runtime of slow method
```

```
            tm_slow.start();
            multAminSlow(x, y);
            tm_slow.stop();

            // Runtime of fast method
            tm_fast.start();
            multAmin(x, y);
            tm_fast.stop();
        }
        // END

        std::cout << std::setw(15)
                  << N
                  << std::scientific << std::setprecision(3)
                  << std::setw(15) << tm_slow.min()
                  << std::setw(15) << tm_fast.min()
                  << std::endl;
    }
```

Get it on GitLab (multAmin.cpp).

▲

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

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

```
    MatrixXd B = MatrixXd::Zero(n,n);
    for(unsigned int i = 0; i < n; ++i) {
        B(i,i) = 2;
        if(i < n-1) B(i+1,i) = -1;
        if(i > 0) B(i-1,i) = -1;
    }
    B(n-1,n-1) = 1;
```

Get it on GitLab (multAmin.cpp).

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

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).

▲

(2-7.f) (15 min.) With $\mathbf{A}$ from (2.7.1) and $\mathbf{B}$ from (2-7.e) write a short EIGEN-based C++ function

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

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

Formulate a conjecture based on you observations.

SOLUTION 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 $\mathbf{AB}$

```
Eigen::MatrixXd multABunitv() {
    unsigned int n = 10;

    /* SAM_LISTING_BEGIN_5 */
    MatrixXd B = MatrixXd::Zero(n,n);
    for(unsigned int i = 0; i < n; ++i) {
        B(i,i) = 2;
        if(i < n-1) B(i+1,i) = -1;
        if(i > 0) B(i-1,i) = -1;
    }
    B(n-1,n-1) = 1;
    /* SAM_LISTING_END_5 */

    MatrixXd C(n,n);

    // TO DO: (2-7.f) Set the columns of C to A*B*e_j for
    // j=0,...,n-1, and print C.
    // START
    VectorXd y;
    // B*e_j is the j-th column of B*Id = B, so we need only
    // calculate A*B.col(j) for j=0,...,n-1.
    for( unsigned int i = 0; i < n; ++i ) {
        multAmin( B.col(i), y );
        C.col(i) = y;
    }
    std::cout << "C = \n" << C << std::endl;
    // END

    return C;
}
```

Get it on GitLab (multAmin.cpp).

▲

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

$$f_1(x_0,h) := \sin(x_0+h) - \sin(x_0)\ . \tag{2.8.1}$$

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.

HINT 1 for (2-8.a): Use that (2.8.1) can be transformed, using the trigonometric identity

$$\sin(\varphi) - \sin(\psi) = 2\cos\left(\frac{\varphi+\psi}{2}\right)\sin\left(\frac{\varphi-\psi}{2}\right)\ . \tag{2.8.2}$$

SOLUTION of (2-8.a):

Based on (2.8.2) we get the equivalent formula

$$f_2(x_0,h) = 2\cos\left(\frac{2x_0+h}{2}\right)\sin\left(\frac{h}{2}\right)\ .$$

▲

(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

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

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

Explain the observed behaviour of the error.

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 Code 2.8.4 and the plot in Fig. 4. We can observe that the computation using f_1 does not converge as $h \to 0$. This is due to the cancellation error given by the subtraction of two number of approximately same magnitude. The second implementation, using f_2, is very stable and does not display cancellation issues.

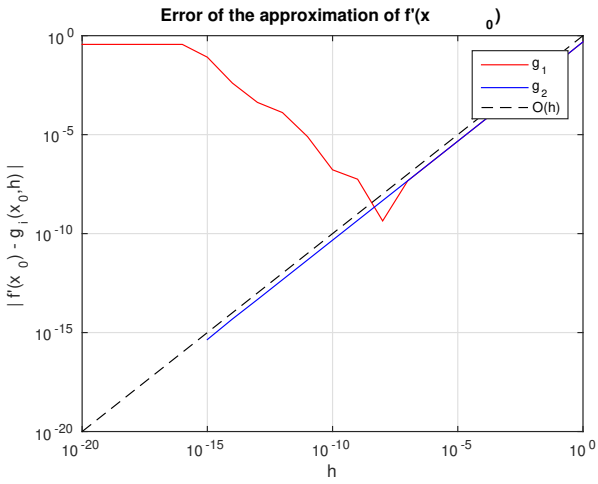


Fig. 4

C++-code 2.8.4: C++ implementation for Problem 2-8

```
void sinederv() {
    // Array of values of h
    ArrayXd h = ArrayXd::LinSpaced(21, -20, 0.)
        .unaryExpr([] (double i) {
            return std::pow(10., i);
        });
    // Dummy array where to evaluate the derivative (1.2)
    ArrayXd x = ArrayXd::Constant(h.size(), 1.2);

    // Derivatives
    ArrayXd g1, g2, ex;

    // TO DO: (2-8.b) For each entry in h, fill in the arrays such that
    // g1 approximates f'(1.2) using the difference quotient directly,
    // g2 approximates f'(1.2) while avoiding cancellation, and
    // ex is the "exact" value of f'(1.2).
    // Hint:  You can use Eigen's sin() and cos(), which calculate
    // coefficient-wise results for Array inputs.
    // START
    g1 = (sin(x+h) - sin(x)) / h; // naive
    g2 = 2 * cos(x+0.5*h) * sin(0.5 * h) / h; // better
    ex = cos(x); // exact
    // END

    // Print error
    // Table header
    std::cout << std::setw(15) << "h"
              << std::setw(15) << "exact"
              << std::setw(15) << "cancellation"
              << std::setw(15) << "error"
              << std::setw(15) << "improved"
              << std::setw(15) << "error" << std::endl;
```

```
    for(unsigned int i = 0; i < h.size(); ++i) {
        std::cout << std::setprecision(6);
        // Table entries
        // TO DO: (2-8.b) Print the i-th row of the table.
        // Keep the order h, ex, g1, |g1 - ex|, g2, |g2 - ex|.
        // Use std::setw(15) as in the table header.
        // You can use std::abs() to calculate absolute values.
        // START
        std::cout << std::setw(15) << h(i)
                  << std::setw(15) << ex(i)
                  << std::setw(15) << g1(i)
                  << std::setw(15) << std::abs(g1(i) - ex(i))
                  << std::setw(15) << g2(i)
                  << std::setw(15) << std::abs(g2(i) - ex(i)) << std::endl;
        // END
    }

    // Plots the contents of the table on a log-log scale.
    py_plot( h, ex, g1, g2 );
}
```

Get it on GitLab (cancellation.cpp).

▲

(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.

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 $x \to \infty$ 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 $x = 10^8$.

▲

(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 of (2-8.d):

1. For $x >> 1$, 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 $\frac{1}{b^2} \to 1$ as $b \to 1$. 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.

▲

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`.

```
VectorXd getit(const MatrixXd &A, const VectorXd &x, unsigned int k) {

    EigenSolver<MatrixXd> eig = EigenSolver<MatrixXd>(A);
    const VectorXcd & V = eig.eigenvalues();
    const MatrixXcd & W = eig.eigenvectors();

    VectorXcd cx = x.cast<std::complex<double>>();

    VectorXcd ret = W *
        (
          V.array().pow(k) *
          (W.partialPivLu().solve(cx)).array()
        ).matrix();

    return ret.real();
}
```

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*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 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 $\mathbf{A} = \mathbf{S}\mathbf{D}\mathbf{W}^{-1}$ (where $\mathbf{D}$ is diagonal and $\mathbf{W}$ invertible), allows us to write:

$$\mathbf{A}^k = (\mathbf{W}\mathbf{D}\mathbf{W}^{-1})^k = \mathbf{W}\mathbf{D}^k\mathbf{W}^{-1}$$

$\mathbf{D}^k$ 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).

▲

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

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

SOLUTION 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 $n \to \infty$.

▲

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].

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

Consider the C++ code given in Listing 2.10.1.

C++11-code 2.10.1: Implementation of sinh

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

Get it on GitLab (sinh.cpp).

(2-10.a) 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

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

with C++ for $x = 10^{-k}$, $k = 1, 2, \ldots, 10$. As "exact value" use the result of the C++ built-in function `std::sinh` (include `cmath`).

SOLUTION of (2-10.a):

As $x \to 0$, the terms $t := \exp(x)$ and $1/t = \exp(-x)$ become close to each other, thereby creating cancellations errors in the return value.

▲

(2-10.b) 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 of (2-10.b):

Given $m \in \mathbb{N}$ and $x \in \mathbb{R}$, there exists $\xi_x \in [0, x]$ such that:

$$e^x = \sum_{k=0}^{m} \frac{x^k}{k!} + \frac{e^{\xi_x} x^{m+1}}{(m+1)!}. \tag{2.10.2}$$

▲

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

$$\sinh x \geq x. \tag{2.10.3}$$

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 $x \geq 0$.

▲

(2-10.d) ⚁ 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 ϵ_{rel} is smaller than 10^{-15}. Follow the considerations of [Lecture → Ex. 1.5.4.26].

SOLUTION of (2-10.d):

The idea is to use the Taylor expansion given in (2.10.2). 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 $(m+1)! \to 0$ as $m \to \infty$ (and, therefore, the reminder converges to zero as $m \to \infty$). 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 (2.10.3) 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`.

▲

End Problem 2-10

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 $\sqrt{w} = x + \imath y$ can be defined by

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

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

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

(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 of (2-11.a):

As $u/v \to -\infty$ 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 $u/v \to +\infty$ 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.

▲

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

SOLUTION of (2-11.b):

An immediate calculation shows that $xy = v/2$.

▲

(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 of (2-11.c):

Cf. `complexroot.cpp`.

C++-code 2.11.3: Description

Get it on GitLab (complexroot.cpp).

▲

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

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

$$(\mathbf{D_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i = j, \\ 0, & i \neq j \end{cases}, (\mathbf{L_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i > j, \\ 0, & i \leq j \end{cases}, (\mathbf{U_A})_{i,j} := \begin{cases} (\mathbf{A})_{i,j}, & i < j, \\ 0, & i \geq j \end{cases}. \tag{2.12.1}$$

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

$$\mathbf{x}^{(k+1)} = (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{b} - (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}^{(k)}). \tag{2.12.2}$$

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

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.

▲

(2-12.b) 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 of (2-12.b):

Let $\mathbf{x}$ be such fixed point, then

$$\mathbf{x} = (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{b} - (\mathbf{U_A} + \mathbf{D_A})^{-1}\mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) \tag{2.12.3}$$

Left-multiply by $\mathbf{U_A} + \mathbf{D_A} = \mathbf{A} - \mathbf{L_A}$ (invertible!):

$$(\mathbf{U_A} + \mathbf{D_A})\mathbf{x} = \mathbf{b} - \mathbf{L_A}(\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x}) \tag{2.12.4}$$

Let us now define $\mathbf{y} := (\mathbf{L_A} + \mathbf{D_A})^{-1}(\mathbf{b} - \mathbf{U_A}\mathbf{x})$, then:

$$(\mathbf{L_A} + \mathbf{D_A})\mathbf{y} = \mathbf{b} - \mathbf{U_A}\mathbf{x} \tag{2.12.5}$$
$$(\mathbf{U_A} + \mathbf{D_A})\mathbf{x} = \mathbf{b} - \mathbf{L_A}\mathbf{y} \tag{2.12.6}$$

Subtract the first equation to the second:

$$(\mathbf{L_A} + \mathbf{D_A})\mathbf{y} + \mathbf{U_A}\mathbf{x} - (\mathbf{U_A} + \mathbf{D_A})\mathbf{x} - \mathbf{L_A}\mathbf{y} = \mathbf{0} \tag{2.12.7}$$
$$\Leftrightarrow \quad \mathbf{D_A}(\mathbf{y} - \mathbf{x}) = \mathbf{0} \tag{2.12.8}$$
$$\Leftrightarrow \quad \mathbf{y} = \mathbf{x} \tag{2.12.9}$$

Using this last identity in (2.12.5) and get $(\mathbf{U_A} + \mathbf{D_A} + \mathbf{L_A})\mathbf{x} = \mathbf{b}$. □

▲

(2-12.c) 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 trough `x`. The approximated solution given by the final iterate is then stored in `x` as an output.

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

SOLUTION of (2-12.c):

C++11-code 2.12.10: Code

```
    const auto U = TriangularView<const MatrixXd, StrictlyUpper>(A);
    const auto L = TriangularView<const MatrixXd, StrictlyLower>(A);

    const auto UpD = TriangularView<const MatrixXd, Upper>(A);
    const auto LpD = TriangularView<const MatrixXd, Lower>(A);

    // A temporary vector to store result of iteration
    VectorXd temp(x.size());

    // We'll use pointer magic to
    VectorXd* xold = &x;
    VectorXd* xnew = &temp;

    // Iteration counter
    unsigned int k = 0;
    double err;

#if VERBOSE
        std::cout << std::setw(10) << "it."
                  << std::setw(15) << "err" << std::endl;
#endif // VERBOSE
    do {
        // Compute next iteration step
        *xnew = UpD.solve(b) - UpD.solve(L*LpD.solve(b - U * (*xold) ));

        // Absolute error
        err = (*xold - *xnew).norm();
#if VERBOSE
        std::cout << std::setw(10) << k++
                  << std::setw(15) << std::setprecision(3) << std::scientific
                  << err << std::endl;
#endif // VERBOSE

        // Swap role of previous/next iteration
        std::swap(xold, xnew);
    } while( err > rtol * (*xnew).norm() );

    x = *xnew;
```

Get it on GitLab (gsit.cpp).

▲

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

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

output the l^2 norm of the residual of the approximated solution. Use $\mathbf{b}$ as your initial data.

SOLUTION of (2-12.d):

C++11-code 2.12.12: Code

```
    VectorXd x = b;
    GSIt(A, b, x, 10e-8);

    double residual = (A*x - b).norm();

    std::cout << "Residual = " << residual << std::endl;
```

Get it on GitLab (gsit.cpp).

▲

(2-12.e) Using the same matrix $\mathbf{A}$ and the same r.h.s. vector $\mathbf{b}$ as above (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 of (2-12.e):

Asymptotic linear convergence can be expected after an initial phase where the error decreases much faster.

▲

End Problem 2-12

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.

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

Consider the following Matrix-vector multiplication:

$$\mathbf{x} = \mathbf{A}\mathbf{y} \tag{2.13.1}$$

with

$$\mathbf{A} = \begin{bmatrix} a_1 & & & & a_n \\ & \ddots & & \iddots & \\ & & \vdots\cdot\vdots & & \\ & \iddots & & \ddots & \\ a_1 & & & & a_n \end{bmatrix} \in \mathbb{R}^{n,n}\,, \tag{2.13.2}$$

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) 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.

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::VectorXi`, `Eigen::Matrix<10,1,double>`... ⌟

SOLUTION of (2-13.a):

C++11-code 2.13.3: Code

```
template<class Vector>
void xmatmult(const Vector& a, const Vector& y, Vector& x) {
    assert(a.size() == y.size() && a.size() == x.size()
           && "Input vector dimensions must match");
    unsigned n = a.size();

    // looping over half of a,
    // if n is odd we treat the middle element differently
    for (unsigned i = 0; i < n / 2; ++i) {   //integer division
```

```
        x(i) = a(i) * y(i) + a(n-i-1) * y(n-i-1);
        x(n-i-1) = x(i);
    }

    if (n%2) { //n is odd
        x(n/2) = a(n/2) * y(n/2);
    }
}
```

Get it on GitLab (xmatmult.cpp).

▲

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

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 $O(n^2)$.

▲

(2-13.c) Compare the runtime of your solution to that of the matrix-vector multiplication provided by EIGEN.

HINT 1 for (2-13.c): You can find a possible initialization of $\mathbf{A}$ for the ordinary multiplication in the `test()` function.

SOLUTION of (2-13.c):

C++11-code 2.13.4: Code

```
void compare_times() {
    std::cout << "Measuring runtimes for comparison" << std::endl;
    unsigned repeats = 3;
    Timer t_fast, t_slow;
    MatrixXd results(10,3);

    for (unsigned k = 14; k > 4; k--) {
        //bitshift operator '«': 1«3 == pow(2,3)
        unsigned n = 1<<k;
        VectorXd a,y,x(n);
        a = y = VectorXd::Random(n);

        MatrixXd A = a.asDiagonal();
        for (unsigned i = 0; i < n; ++i) {
            A(n-i-1,i) = A(i,i);
        }

        //measure multiple times
        for (unsigned i = 0; i < repeats; i++) {
            t_fast.start();
            xmatmult(a,y,x);
```

```
            t_fast.stop();

            t_slow.start();
            x = A * y;
            t_slow.stop();
        }
        results(k-5,0) = n;
        results(k-5,1) = t_slow.min();
        results(k-5,2) = t_fast.min();
    }

    // print results
    std::cout << std::setw(8) << "n"
              << std::setw(15) << "original"
              << std::setw(15) << "efficient"
              << std::setprecision(5) << std::endl;
    for (int i = 0; i < results.rows(); i++) {
        std::cout << std::setw(8)<< results(i,0)
                  << std::setw(15) << results(i,1) << " s"
                  << std::setw(15) << results(i,2) << " s"
                  << std::endl;
    }
}
```

Get it on GitLab (xmatmult.cpp).

▲

End Problem 2-13

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 of (3-1.b):

From the information given above we have to extract a linear system of equations, whose solution gives the prices of the fruits. This linear system is evident from Code 3.1.1.

The following code uses a particular of the many direct solvers provided by EIGEN. You may use most other from the table in EIGEN documentation, especially **Eigen::PartialPivLU**, which is the standard choice.

C++ code 3.1.1: Code

```
int main() {
    // Initialize the matrix
    MatrixXd A(6,6);
```

```
    // See "Advanced Initialization" in the Eigen doc
    A << 3, 1, 7, 2, 0, 0,
         2, 0, 0, 0, 0, 1,
         1, 0, 0, 0, 3, 0,
         0, 5, 0, 1, 0, 0,
         0, 0, 0, 2, 0, 1,
         0, 1,20, 0, 0, 0;

    // Initialize the vector
    VectorXd b(6);
    b << 11.10, 17.00, 6.10, 5.25, 12.50, 7.00;

    // Initialize the solver; other decompositions also possible
    ColPivHouseholderQR<MatrixXd> solver;
    // Decompose A
    solver.compute(A);
    // Solve the LSE
    VectorXd x1 = solver.solve(b);

    // Notice how we split up initialization, decompositon and solving.
    // This means we can use the same solver for multiple matrices and the
    // same decompositions for multiple LSE with the same matrix.
    // These three steps can be combined:
    VectorXd x2 = A.colPivHouseholderQr().solve(b);

    // The result is the same:
    assert((x1-x2).norm() < 1e-10);

    //Print the result nicely
    std::vector<std::string> fruit_names {"Mango", "Kiwi", "Lychee", "Banana", "Pomegranate", "Pineapple"};
    std::cout << std::fixed << std::setprecision(2);    //stream formatting
    std::cout <<"The fruits cost:" << std::endl;
    for (int i = 0; i < 6; i++)
    {
        std::cout << fruit_names[i] << ": " << x1[i] << " sFr." << std::endl;
    }
}
```

Get it on GitLab (fruits.cpp).

▲

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

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

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

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

SOLUTION of (3-2.a):

We need to "interlace" the blocks of the matrix.

We start by moving the $n+k$-th row ($k > 1$) to $(2k-1)$-th position, and the k-th row ($k <= n$) to $2k$-th row.

We do the same for the columns: moving the $n+k$-th column ($k > 1$) to $(2k-1)$-th position, and the k-th column ($k <= n$) to $2k$-th column.

▲

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

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

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

SOLUTION of (3-2.b):

Let us decompose $\mathbf{x} := [\mathbf{x}_1; \mathbf{x}_2], \mathbf{x}_i \in \mathbb{R}^n$. Performing a block matrix vector multiplication, we obtain:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{d}_1 \odot \mathbf{x}_1 + \mathbf{c} \odot \mathbf{x}_2 \\ \mathbf{c} \odot \mathbf{x}_1 + \mathbf{d}_2 \odot \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \end{bmatrix} \odot \mathbf{x} + \begin{bmatrix} \mathbf{c} \\ \mathbf{c} \end{bmatrix} \odot \begin{bmatrix} \mathbf{x}_2 \\ \mathbf{x}_1 \end{bmatrix}.$$

Here, "$\odot$" denotes the component wise product of two vectors of equal length, and $\mathbf{d}_1, \mathbf{d}_2, \mathbf{c}$ are vectors containing the diagonal entries of the matrices $\mathbf{D}_1$, $\mathbf{D}_2$ and $\mathbf{C}$.

This can be efficiently implemented in C++.

C++11-code 3.2.1: Solution of (3-2.b)

```
    auto & x1 = x.head(n).array();
    auto & x2 = x.tail(n).array();
```

```
    y << d1.array() * x1 + c.array() * x2,
         d2.array() * x2 + c.array() * x1;

    return y;
```

Get it on GitLab (blocklsepiv.cpp).

▲

(3-2.c)

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

HINT 1 for (3-2.c): Take the cue from the block LU-factorization discussed in [Lecture → Rem. 2.3.2.19], in particular [Lecture → Eq. (2.3.2.20)].

SOLUTION of (3-2.c):

We compute the LU-factors using the block Gaussian elimination algorithm:

$$\mathbf{A} = \begin{pmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{C} & \mathbf{D}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{D}_1^{-1}\mathbf{C} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \mathbf{D}_1 & \mathbf{C} \\ \mathbf{0} & \mathbf{D}_2 - \mathbf{D}^{-1}\mathbf{C}\mathbf{C} \end{pmatrix} = \mathbf{L}\mathbf{U},$$

where $\mathbf{I}$ denotes $n \times n$ identity matrix and $\mathbf{0}$ indicates $n \times n$ zero matrix. Since all matrices are diagonal, the factor $\mathbf{L}$ is lower triangular with unit diagonal and the factor $\mathbf{U}$ is upper triangular.

▲

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

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

that solves $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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.

HINT 1 for (3-2.d): Recall the finding of (3-2.a), which permits you to reduce everything to the solution of n 2×2 linear systems of equations.

SOLUTION of (3-2.d):

C++11-code 3.2.2: Solution of (3-2.d)

```
    ArrayXd c1 = c_, c2 = c_, d1 = d1_, d2 = d2_, b = b_;

    double eps = std::numeric_limits<double>::epsilon();

    // For forward elimination + pivotisation we are
    // only required to loop the first half of the matrix
```

```
    // Loop over diagonal
    for(int k = 0; k < n; ++k) {
        // Check if need to pivot (i.e.  swap two rows)
        // Luckily we only need to check two rows
        double maxk = std::max(std::abs(d1(k)), std::abs(c1(k)));
        double maxnk = std::max(std::abs(c2(k)), std::abs(d2(k)));
        if( std::abs( c1(k) ) / maxk // max relative pivot at row k
            >
            std::abs( d1(k) ) / maxnk // max relative pivot at rok k+n
            ) {
            // Matrix
            std::swap(d1(k), c2(k));
            std::swap(c1(k), d2(k));
            // R.h.s.
            std::swap(b(n), b(n+k));
        }

        // Check if matrix is almost singuloar
        double piv = d1(k);
        // Norm of the block from k,k to n-1,n-1
        double norm = std::abs(d1(k)) + std::abs(d2(k))
                    + std::abs(c1(k)) + std::abs(c2(k));
        if( piv < eps * norm ) {
            std::cout << "Warning: matrix nearly singular!" << std::endl;
        }

        // Multiplication facot:
        double fac = c2(k) / piv;

        // Actually perform substitution
        // Bottom Right poriton changes
        d2(k) -= c1(k) * fac;
        // R.h.s
        b(n+k) -= b(k) * fac;
    }

    // Now the system has the form:
    // | d1 | c | | | | |
    // | 0 | d2 | * | x | = | b |
    // with d1, d2, c diagonal

    // Backward substitution

    // Lower potion
    b.tail(n) /= d2;
    // Upper portion
    b.head(n) = (b.head(n) - c1*b.tail(n)) / d1;

    return b;
```

Get it on GitLab (blocklsepiv.cpp).

▲

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

SOLUTION of (3-2.e):

In the implementation of `solveA` there is only one loop of size n where operations only simple floating point operations are performed. After this loop, there are two vector operations on vectors of length `n`. Hence, the computational complexity of the algorithm is linear, i.e. $O(n)$.

▲

(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 $b = [d1; d1]$. Tabulate the runtimes (in seconds, 3 decimal digit, scientific notation) for meaningful values of n.

SOLUTION of (3-2.f):

As a clear indication for linear complexity the data points is a problem size vs. runtime plot should be approximately located on a straight line, as in Fig. 5.

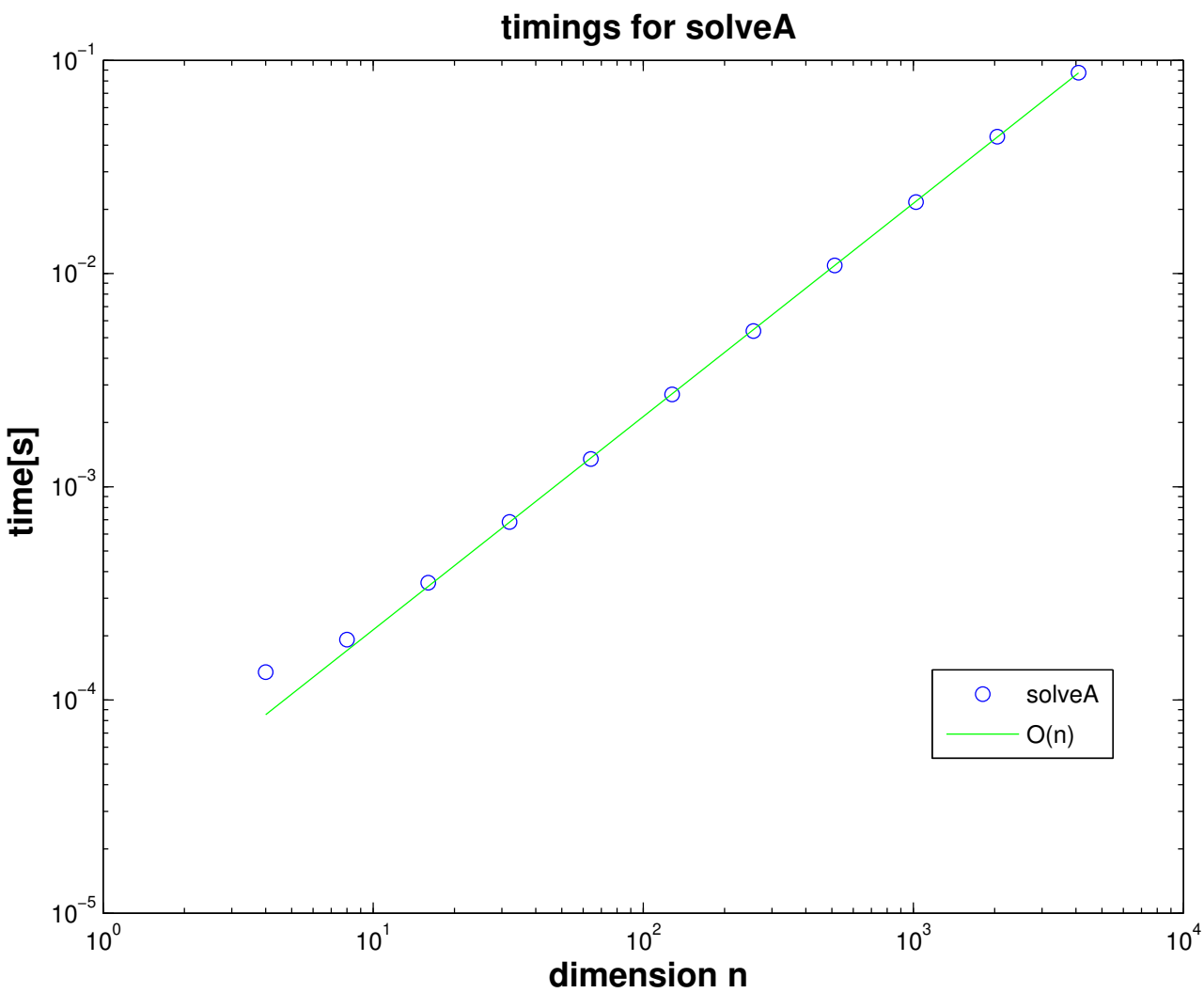


Fig. 5

C++11-code 3.2.3: Solution of (3-2.f)

```
    // Table header
    std::cout << std::setw(10) << "k"
              << std::scientific << std::setprecision(3)
              << std::setw(15) << "runtime [s]" << std::endl;
```

```
    // Loop from 2 to 2^7
    for(int k = 2; k <= (1<<7); k = k << 1) {
        buildvectors(k); // creates d1, d2, c, b

        // Repeat measurments
        Timer tm;
        for(int r = 0; r < repeats; ++r) {
            // Test run
            tm.start();
            yo = solveA(d1, d2, c, b);
            tm.stop();
        }

        // Print measurments
        std::cout << std::setw(10) << k
                  << std::scientific << std::setprecision(3)
                  << std::setw(15) << tm.duration() << std::endl;
    }
```

Get it on GitLab (blocklsepiv.cpp).

▲

End Problem 3-2

Problem 3-3: Block-wise LU-decompisiton

Using block matrix operations and the LU-decomposition as described in [Lecture → Rem. 2.3.2.19], you shouldimplement 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 of (3-3.a):

The LU-decomposition is:

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} \mathbf{I} & 0 \\ \mathbf{u}^T\mathbf{R}^{-1} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ 0 & -\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \end{bmatrix}$$

▲

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

SOLUTION of (3-3.b):

$\mathbf{A}$ is regular if and only if $\det(\mathbf{A}) \neq 0$.

$$\det(\mathbf{A}) = \det(\mathbf{L})\det(\mathbf{U}) = \det(\mathbf{L})\det(\mathbf{R})(\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v}) \tag{3.3.1}$$

$\det(\mathbf{L}) = 1$ and $\mathbf{R}$ is regular, so $\det(\mathbf{R}) \neq 0$, which leaves $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$ as necessary and sufficient condition.

▲

(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).

HINT 1 for (3-3.c): You will need to solve a LSE multiple times by backwards substitution. It can be useful to implement a function `VectorXd solve_R(const MatrixXd& R, const VectorXd& c);` that does that.

SOLUTION of (3-3.c):

C++11-code 3.3.2: Code

```
VectorXd solve_LSE(const MatrixXd& R,
                   const VectorXd& v,
                   const VectorXd& u,
                   const VectorXd& b)
{
    unsigned n = R.rows();
    assert(R.cols() == n && "R has to be square");
    assert(n == v.size() && n == u.size() && n+1 == b.size()
           && "Input dimensions must agree");
    // Initialize
    VectorXd y(n+1), x(n+1);
    // Solve the system Ax=b by LU-Decomposition.
    // Solve Ly = b through forward substitution.
    // Due to the special structure of our L,
    // the first n entries of y are easy:
    y.head(n) = b.head(n);
    // The last element of y is given by y_n = b_n - u^T R^{-1} y_{0...n-1}
    y(n) = b(n) - u.transpose() * solve_R(R, y.head(n));

    // Solve Ux = y by backward substitution.
    // First we build U
    MatrixXd U(n+1,n+1);
    U << R,                                v,
         VectorXd::Zero(n).transpose(), -u.transpose()*solve_R(R,v);
    // Then we solve Ux = y
    x = solve_R(U,y);
    //
    return x;
}
```

Get it on GitLab (blockLU.cpp).

This makes use of a function to solve a LSE by backwards substitution:

C++11-code 3.3.3: Code

```
VectorXd solve_R(const MatrixXd& R, const VectorXd& c)
{
    int n = R.rows();
    assert(n == R.cols() && n == c.size() &&
           "Input dimensions must agree");
    // Initialize
    VectorXd y(n);
    // Since R is upper triangular, we can solve by backwards
       substitution
    for (int i = n-1; i >= 0; --i)
    {
        y(i) = c(i);
        for (int j = n-1; j > i ; --j)
        {
            y(i) -= R(i,j) * y(j);
        }
        y(i) /= R(i,i);
    }
    return y;
```

```
}
```

Get it on GitLab (blockLU.cpp).

▲

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 $\operatorname{rank}(\mathbf{A}) = n$, the product matrix $\mathbf{A}^\top\mathbf{A}$ admits a Cholesky decomposition.

HINT 1 for (3-4.a): For a Cholesky decomposition to be possible, show that $\mathbf{A}^\top\mathbf{A}$ is symmetric and positive definite.

SOLUTION of (3-4.a):

$\mathbf{A}^\top\mathbf{A}$ has to be symmetric and positive definite.

$\left(\mathbf{A}^\top\mathbf{A}\right)^\top = \mathbf{A}^\top\left(\mathbf{A}^\top\right)^\top = \mathbf{A}^\top\mathbf{A}$, which proves the symmetry.

$\operatorname{rank}(\mathbf{A}) = n$ implies that $\mathbf{A}$ is injective. Hence, $\forall \mathbf{v} \in \mathbb{R}^n \mid \mathbf{v} \neq \mathbf{0}$, it holds that $\mathbf{A}\mathbf{v} \neq \mathbf{0} \Longrightarrow \mathbf{v}^\top\mathbf{A}^\top\mathbf{A}\mathbf{v} = |\mathbf{A}\mathbf{v}|_2^2 > 0$, which means that $\mathbf{A}^\top\mathbf{A}$ is positive definite.

▲

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

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

```
void CholeskyQR(const MatrixXd & A, MatrixXd & R, MatrixXd & Q) {

    MatrixXd AtA = A.transpose() * A;
    LLT<MatrixXd> L = AtA.llt();
    R = L.matrixL().transpose();
    Q = R.transpose().triangularView<Lower>().solve(A.transpose()).transpose();
    // .triangularView() template member only accesses the triangular part
    // of a dense matrix and allows to easily solve linear problem
}
```

Get it on GitLab (choleskyQR.cpp).

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

```
void DirectQR(const MatrixXd & A, MatrixXd & R, MatrixXd & Q) {

    size_t m = A.rows();
    size_t n = A.cols();

    HouseholderQR<MatrixXd> QR = A.householderQr();
   Q = QR.householderQ() * MatrixXd::Identity(m, std::min(m, n));
   R = MatrixXd::Identity(std::min(m, n), m) * QR.matrixQR().triangularView<Upper>();
   // If A: m x n, then Q: m x m and R: m x n.
   // If m > n, however, the extra columns of Q and extra rows of R are not needed.
```

```
    // Matlab returns this "economy-size" format calling "qr(A,0)",
    // which does not compute these extra entries.
    // With the code above, Eigen is smart enough to not compute the
       discarded vectors.
}
```

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).

HINT 1 for (3-4.b): If you test the code, ignore the signs of the entries in the resulting matrices $\mathbf{Q}$ and $\mathbf{R}$. The differences are due to the implementation of EIGEN. They are not however caused by the fact that EIGEN might use a permutation in the QR decomposition: the class we employ, `Eigen::HouseholderQR`, does not make use of them. (`Eigen::ColPivHouseholderQR` and `Eigen::FullPivHouseholderQR` do, on the other hand.)

Moreover, both functions only compute the same *economy-size* output of QR decompositions in MATLAB (`qr(A,0)`), which holds for $m > n$. Indeed, if $\mathbf{A} \in \mathbb{R}^{m,n}$, then orthogonal $\mathbf{Q}$ is a square matrix in $\mathbb{R}^{m,m}$ and upper triangular $\mathbf{R}$ is in $IR^{m,n}$. However, if $m > n$, we can obtain an equivalent QR decomposition for $\mathbf{A}$ by removing extra right columns of $\mathbf{Q}$ and extra bottom rows of $\mathbf{R}$, to end up with $\mathbf{Q} \in \mathbb{R}^{m,n}$ and $\mathbf{R} \in \mathbb{R}^{n,n}$. Both `CholeskyQR` and `DirectQR` only compute the entries which are needed. The former because it relies on a Cholesky decomposition of $\mathbf{A}^\top\mathbf{A} \in \mathbb{R}^{n,n}$. The latter because the underlying EIGEN QR decomposition is already optimized when one imposes that $\mathbf{Q} \in \mathbb{R}^{m,n}$ and $\mathbf{R} \in \mathbb{R}^{n,n}$. ⌟

SOLUTION of (3-4.b):

For `CholeskyQR` to return the same $\mathbf{R}$ and $\mathbf{Q}$ of a QR decomposition using a Cholesky decomposition, we need to verify three properties:

- $\mathbf{R}$ is upper triangular.
- $\mathbf{Q}$ is orthogonal.
- $\mathbf{QR} = \mathbf{A}$

This is done in the following:

- $\mathbf{R}$ is upper triangular from definition of Cholesky decomposition.
- Given that $\mathbf{Q} = \left(\mathbf{R}^{-\top}\mathbf{A}^\top\right)^\top = \mathbf{A}\mathbf{R}^{-1}$ and, from the Cholesky decomposition, $\mathbf{R}^\top\mathbf{R} = \mathbf{A}^\top\mathbf{A} \Rightarrow \mathbf{R}^{-\top}\mathbf{A}^\top\mathbf{A}\mathbf{R}^{-1} = \mathbf{I}$, we have that $\mathbf{Q}^\top\mathbf{Q} = \mathbf{R}^{-\top}\mathbf{A}^\top\mathbf{A}\mathbf{R}^{-1} = \mathbf{I}$. This means that $\mathbf{Q}$ is orthogonal.
- $\mathbf{QR} = \left(\mathbf{R}^{-\top}\mathbf{A}^\top\right)^\top\mathbf{R} = \mathbf{A}\mathbf{R}^{-1}\mathbf{R} = \mathbf{A}$

▲

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

HINT 1 for (3-4.c): Review [Lecture → Ex. 3.2.0.4]! ⌟

HINT 2 for (3-4.c): Are the hypothesis needed by the Cholesky decomposition satisfied? ⌟

SOLUTION of (3-4.c):

$\mathbf{A}$ has rank 2, but $\mathbf{A}^\top\mathbf{A} = \begin{bmatrix} 1+\frac{1}{4}\epsilon^2 & 1 \\ 1 & 1+\frac{1}{4}\epsilon^2 \end{bmatrix} \approx \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ has rank 1 in machine arithmetic. Hence, $\mathbf{A}^\top\mathbf{A}$ is symmetric but not positive definite: the hypothesis needed for the Cholesky decomposition are not satisfied.

▲

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 contain 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.

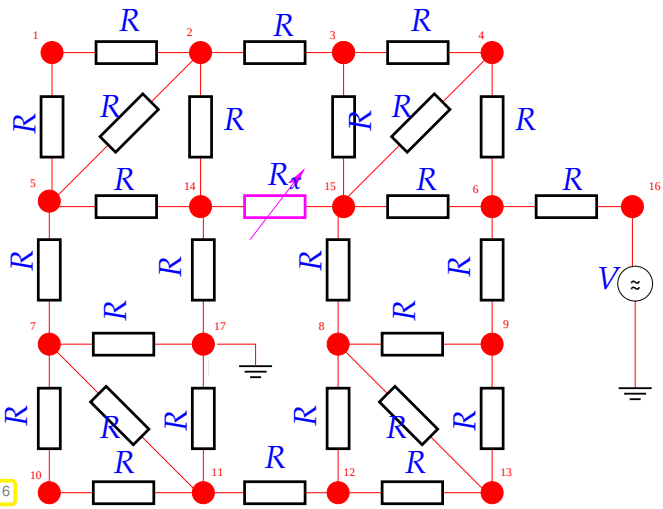


Fig. 6

(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 of (3-5.b):

We use the Kirchhoff's first law (as in [Lecture → Ex. 2.1.0.3]), stating that the sum the currents incident to a node is zero. We use Ohm's law to rewrite the current in terms of voltage and resistance: the current flowing in connection a to b is $I_{a,b} := \Delta_{a,b}W/R$. Here, we denote by $\Delta W_{i,j} := (W_i - W_j)$. Let $\mathbf{W} \in \mathbb{R}^{17}$ be the vector of voltages at the nodes. Set $f_{R_x} := R/R_x$. The system (obtained applying Kirchhoff's

law for each node $i = 1, \dots, 15$) becomes:

$$\begin{cases} \Delta W_{1,2} + \Delta W_{1,5} & = 0 \\ \Delta W_{2,1} + \Delta W_{2,3} + \Delta W_{2,5} + \Delta W_{2,14} & = 0 \\ \Delta W_{3,2} + \Delta W_{3,4} + \Delta W_{3,15} & = 0 \\ \Delta W_{4,3} + \Delta W_{4,6} + \Delta W_{4,15} & = 0 \\ \Delta W_{5,1} + \Delta W_{5,2} + \Delta W_{5,7} + \Delta W_{5,14} & = 0 \\ \Delta W_{6,4} + \Delta W_{6,9} + \Delta W_{6,15} + \Delta W_{6,16} & = 0 \\ \Delta W_{7,5} + \Delta W_{7,10} + \Delta W_{7,11} + \Delta W_{7,17} & = 0 \\ \Delta W_{8,9} + \Delta W_{8,12} + \Delta W_{8,13} + \Delta W_{8,5} & = 0 \\ \Delta W_{9,6} + \Delta W_{9,8} + \Delta W_{9,13} & = 0 \\ \Delta W_{10,7} + \Delta W_{10,11} & = 0 \\ \Delta W_{11,7} + \Delta W_{11,10} + \Delta W_{11,12} + \Delta W_{11,17} & = 0 \\ \Delta W_{12,8} + \Delta W_{12,11} + \Delta W_{12,13} & = 0 \\ \Delta W_{13,8} + \Delta W_{13,9} + \Delta W_{13,12} & = 0 \\ \Delta W_{14,2} + \Delta W_{14,5} + \Delta W_{14,17} + f_{R_x} \Delta W_{14,15} & = 0 \\ \Delta W_{15,3} + \Delta W_{15,4} + \Delta W_{15,6} + \Delta W_{15,8} + f_{R_x} \Delta W_{15,14} & = 0. \end{cases} \tag{3.5.1}$$

(We rescaled each equation multiplying by R.) We have the extra conditions $W_{16} := V$ and $W_{17} = 0$, due to the ground/source voltages. The unknowns of the system are voltages W_j for $j = 1, \dots, 15$. Define $\mathbf{x} := [W_1, \dots, W_{15}]^\top$: the vector of unknowns. We now have to obtain the system matrix. The system is rewritten in the following matrix notation (with $\mathbf{C}_{R_x} \in \mathbb{R}^{15 \times 17}$):

$$\mathbf{C}_{R_x} \cdot \mathbf{W} = [\mathbf{A}_{R_x},\ \mathbf{B}] \cdot \mathbf{W} = \mathbf{0} \ \Leftrightarrow\ \mathbf{A}_{R_x}\mathbf{x} = -\mathbf{B} \cdot [W_{16},\ W_{17}]^\top =: \mathbf{b}.$$

The matrix $\mathbf{A}_{R_x}$, is

$$\mathbf{A}(R_x) = \begin{bmatrix} 2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & -1 & 3 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & -1 & 3 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 4 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & -1 & 0 & 4 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 & -1 & 0 & 0 & -1 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 3 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 4 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 3 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & 3 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 + \frac{R}{R_x} & -\frac{R}{R_x} \\ 0 & 0 & -1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -\frac{R}{R_x} & 4 + \frac{R}{R_x} \end{bmatrix}$$

a square 15×15 matrix. The r.h.s matrix $\mathbf{B} \in \mathbb{R}^{15,2}$ is zero except $B_{6,1} = -1$.

(For each term with the form $\beta \cdot \Delta W_{i,j}$ in (3.5.1) we have add an entry β in the diagonal i, i and an entry $-\beta$ in the cell i, j (if $j \leq 15$). Entries with $j > 15$ will produce a corresponding positive entry $\beta \cdot U_j$ in the r.h.s.)

▲

(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".

HINT 1 for (3-5.c): Four entries of the circuit matrix will change. This amounts to a rank-1-modification in the sense of [Lecture → Eq. (2.6.0.16)] with suitable vectors $\mathbf{u}$ and $\mathbf{v}$. Read [Lecture → § 2.6.0.12].

SOLUTION of (3-5.c):

Define $f_{R_x} := R/R_x$. Define the vectors

$$\mathbf{u} = \mathbf{v} := \sqrt{f_{R_x}} \begin{bmatrix} 0 \\ \vdots \\ 0 \\ -1 \\ 1 \end{bmatrix}$$

we can write:

$$\mathbf{A}_{R_x} = \mathbf{A}_0 - \mathbf{u}\mathbf{v}^\top.$$

Therefore, if we already have the factorisation of $\mathbf{A}_0$, we can use the SMW formula to efficiently compute $\mathbf{A}_{R_x}^{-1}\mathbf{b}$, with $\mathbf{b} \in \mathbb{R}^{15}$:

$$\mathbf{A}_{R_x}^{-1}\mathbf{b} = \left(\mathbf{I} - \frac{\mathbf{A}_0^{-1}\mathbf{u}\mathbf{u}^\top}{1 + \mathbf{v}^\top \mathbf{A}_0^{-1}\mathbf{u}}\right)\mathbf{A}_0^{-1}\mathbf{b}.$$

▲

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

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

```
class ImpedanceMap {
    std::size_t nnodes; //< Number of nodes in the circuit
public:
    /* \brief Build system matrix and r.h.s. and perform a LU decomposition
     * The LU decomposition is stored in 'lu' and can be
     * reused in the SMW formula
     * to avoid expensive matrix solves
     * for repeated usages of the operator()
     * \param R Resistance (in Ohm) value of R
     * \param V Source voltage V at node 16 (in Volt),
     *          ground is set to 0V at node 17
     */
    ImpedanceMap(double R, double V) : R(R), V(V) {
        // TODO: build the matrix A_0.
        // Compute lu factorization of A_0.
        // Store LU factorization in 'lu'.
        // Compute the right hand side and store it in 'b'.
    };

    /* \brief Compute the impedance given the resistance R_x.
     * Use SMW formula for low rank perturbations to reuse LU
```

```
     * factorization.
     * \param Rx Resistence R_x > 0 between node 14 and 15
     * \return Impedance V/I of the system A_{R_x}
     */
    double operator()(double Rx) {
        // TODO: use SMW formula to compute the solution of A_{R_x} x = b
        // Compute and return the impedance of the system.
    }
private:
                MatrixXd  lu  //< Store LU decomp.  of matrix A.
    double      //< Resistance R and source voltage W.
    VectorXd    //< R.h.s vector prescribing sink/source voltages.
};
```

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 $R_x = 1, 2, 4, \ldots, 1024$.

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 of (3-5.d):

Cf. Code 3.5.4

C++11-code 3.5.3: Data structures

```
// Models a resistance between node i,j with i < j
// (in principle, it can handle different resistances)
typedef std::pair< int, int>            resistor;
// Vector containing all resistances in
// the circuit (excluded the variable one, which is modelled
    afterwards)
typedef std::vector< resistor >         resistor_topology;
// Models a ground or source of voltage, indexes store the node
// connected to the ground/source through a resistance
typedef std::tuple<int, int, double>    voltage;
// Vector containing a voltage object for each node connected to sink
    or source.
typedef std::vector< voltage >          voltage_topology;
/* SAM_LISTING_BEGIN_2 */

/* \brief Class implementing the topology of the circuit (cf. Figure)
 * Computes impedance of the entire circuit (between node 16 and 17) exploiting
 * the SMW formula for the inversion of low rank perturbations of a matrix A0,
 * whose factorization in known in advance
 */
/* SAM_LISTING_BEGIN_1 */
class ImpedanceMap {
    std::size_t nnodes; //< Number of nodes in the circuit
public:
    /* \brief Build system matrix and r.h.s. and perform a LU decomposition
```

```
     * The LU decomposition is stored in 'lu' and can be
     * reused in the SMW formula
     * to avoid expensive matrix solves
     * for repeated usages of the operator()
     * \param R Resistance (in Ohm) value of R
     * \param V Source voltage V at node 16 (in Volt),
     *          ground is set to 0V at node 17
     */
    ImpedanceMap(double R, double V) : R(R), V(V) {
        // In the following, instead of specifying directly
        // the entries of A_0 and of r.h.s.
        // we define some auxiliary structure and automatically
        // generate the right entries for
        // the structures we specify.
        // This way:  we avoid silly mistakes, avoid a very long list of matrix
        // entries have a flexible way (we may easily change the topology)
        // and automatically take care of shifting the indices
        // Of course, if you want, you can manually put each
        // entry in the matrix and r.h.s.
        // The result is the same.

        // Number of nodes
        nnodes = 15;

        // We implement the topology of the resistances by pushing each
        // Resistance between node i < j in a std::vector of
        // pair (i,j) ∈ ℕ^2
        // This way we can automatically build a symmetric
        // matrix, take care of indexing
        // and avoid mistakes during the matrix filling
        resistor_topology T;
        T.reserve(23);
        T.push_back(resistor(1,2));
        T.push_back(resistor(1,5));
        T.push_back(resistor(2,5));
        T.push_back(resistor(2,3));
        T.push_back(resistor(2,14));
        T.push_back(resistor(3,4));
        T.push_back(resistor(3,15));
        T.push_back(resistor(4,6));
        T.push_back(resistor(4,15));
        T.push_back(resistor(5,7));
        T.push_back(resistor(5,14));
        T.push_back(resistor(6,9));
        T.push_back(resistor(6,15));
        T.push_back(resistor(7,10));
        T.push_back(resistor(7,11));
        T.push_back(resistor(8,9));
        T.push_back(resistor(8,12));
        T.push_back(resistor(8,13));
        T.push_back(resistor(8,15));
        T.push_back(resistor(9,13));
        T.push_back(resistor(10,11));
        T.push_back(resistor(11,12));
        T.push_back(resistor(12,13));
        // The "base" matrix A_0 will have no resistance between 14 and 15

        // We implement voltage ground and sources by
```

```
        // specifying which node is connected to ground/source
        // trough a resistance.  This will be also part of the r.h.s.
        voltage_topology S;
        S.reserve(4);
        S.push_back(voltage(6, 16, V));
        S.push_back(voltage(7, 17, 0));
        S.push_back(voltage(11,17, 0));
        S.push_back(voltage(14,17, 0));

        // Automatically build A0 filling from topology
        // A0 is the matrix A_R (i.e.  with R_x = R)
        MatrixXd A0 = MatrixXd::Zero(nnodes, nnodes);
        for(resistor & res: T) {
            // Shift indices down by 1 (indices in Figure start from 1)
            std::size_t i = res.first - 1;
            std::size_t j = res.second - 1;
            // Fill diagonal:  each resistance contributes += R
            // into the diagonal of both nodes
            A0(i,i) += 1;
            A0(j,j) += 1;
            // Fill the off diagonal (negative part in ΔW_{i,j},
            // the matrix is kept symmetric
            A0(i,j) -= 1;
            A0(j,i) -= 1;
        }

        // Fill in the rest (source and ground), i.e.
        // components with ΔW_{i,j} with j > 15.
        // Each node i connected to ground or source contributes
        // to the r.h.s b with R·V
        // (R is the resistence between node i and ground/source
        // node, V is voltage at sink or source)
        // and to its own diagonal with R
        b = MatrixXd::Zero(nnodes, 1);
        for(voltage & volt: S) {
            // Shift index down by 1 and get voltage in W2
            int i = std::get<0>(volt) - 1;
            double source_voltage = std::get<2>(volt);
            // Add voltage to r.h.s.  (resistance assumed to be R)
            b(i) += source_voltage;
            // Add resistance to matrix diagonal:  contribution of
                source current
            // scaled by R.
            A0(i,i) += 1;
        }

        // Precompute the 'lu' factorization of A0
        lu = A0.lu();
    };

    /* \brief Compute the impedance given the resistance R_x.
     * Use SMW formula for low rank perturbations to reuse LU
     * factorization.
     * \param Rx Resistence R_x > 0 between node 14 and 15
     * \return Impedance V/I of the system A_{R_x}
     */
    double operator()(double Rx) {
        // Store the scaled factor for convenience
        double f = R/Rx;

```

```
        // Create u:  the vector in A + u * v^T
        VectorXd u = VectorXd::Zero(nnodes);
        u(13) = -std::sqrt(f);
        u(14) = std::sqrt(f);
        // Here:  v = u

        // Use SMW formula to compute (A + u · u^T)^{-1} · b.
        // Formula:
        // A^{-1} * b - A^{-1} * u * *(I + V * A^{-1} * U)^{-1} V * A^{-1} b

        // Start by precomputing A^{-1} b, needed twice
        VectorXd Ainvrhs = lu.solve(b);
        // Then, precompute A^{-1} u, needed twice
        VectorXd Ainvu = lu.solve(u);
        // Then, compute alpha.  Alpha is just a number.
        double alpha = 1 + u.dot(Ainvu);
        // Put the formula toghether, x is a column vector containing voltages
        // at each node (except 16 and 17, which are prescribed)
        VectorXd x = Ainvrhs - Ainvu * u.dot(Ainvrhs) / alpha;

        // Compute the current I = ΔW_{16,5}/R
        // and then impedance = V/I.
        // Here ΔW_{16,5} = W_16 - x_5 = V - x_5.
        return V * R / (V - x(5));
    }
private:
    PartialPivLU<MatrixXd> lu; //< Store LU decomp.  of matrix A.
    double R, V; //< Resistance R and source voltage W.
    VectorXd b; //< R.h.s vector prescribing sink/source voltages.
};
/* SAM_LISTING_END_1 */

int main(void) {
    // Create a first ImpedanceMap with resistance 1 and voltage 1
    ImpedanceMap IM = ImpedanceMap(1, 1);
    // Test Impedance at value one
    std::cout << "--> Impedance with R = 1: " << IM(1) << std::endl;

    // Print a table with various impedances
    std::cout << "--> Table of impedances" << std::endl;
    // Table header
    std::cout << std::setw(30) << "Impedance [Ohm]"
              << std::setw(30) << "R_x [Ohm]"
              << std::endl;
    // Table content:  print impedance for various resistance values
    std::cout << std::setw(30) << IM(0.1)
              << std::setw(30) << 0.1
              << std::endl;
    for(auto Rx = 1; Rx <= 1024; Rx *= 2) {
        std::cout << std::setw(30) << IM(Rx)
                  << std::setw(30) << Rx
                  << std::endl;
    }

    std::cout << "--> Runtime measurments" << std::endl;
    Timer tmr_build, tmr_compute;
    // Measuring time it takes to build the matrix
    tmr_build.start();
    ImpedanceMap IM2 = ImpedanceMap(1, 1);
```

```
    tmr_build.stop();
    // Measuring time it takes to compute the impedance
    tmr_compute.start();
    IM2(1);
    tmr_compute.stop();
    // Output time
    std::cout << "Building time: "
              << std::setprecision(3)
              << std::scientific
              << tmr_build.duration() << " s" << std::endl;
    std::cout << "Compute time: "
              << std::setprecision(3)
              << std::scientific
              << tmr_compute.duration() << " s" << std::endl;

}
```

Get it on GitLab (impedancemap.cpp).

C++11-code 3.5.4: `ImpedanceMap` Class signature

```
class ImpedanceMap {
    std::size_t nnodes; //< Number of nodes in the circuit
public:
    /* \brief Build system matrix and r.h.s. and perform a LU decomposition
     * The LU decomposition is stored in 'lu' and can be
     * reused in the SMW formula
     * to avoid expensive matrix solves
     * for repeated usages of the operator()
     * \param R Resistance (in Ohm) value of R
     * \param V Source voltage V at node 16 (in Volt),
     *          ground is set to 0V at node 17
     */
    ImpedanceMap(double R, double V) : R(R), V(V) {
        // In the following, instead of specifying directly
        // the entries of A_0 and of r.h.s.
        // we define some auxiliary structure and automatically
        // generate the right entries for
        // the structures we specify.
        // This way: we avoid silly mistakes, avoid a very long list of matrix
        // entries have a flexible way (we may easily change the topology)
        // and automatically take care of shifting the indices
        // Of course, if you want, you can manually put each
        // entry in the matrix and r.h.s.
        // The result is the same.

        // Number of nodes
        nnodes = 15;

        // We implement the topology of the resistances by pushing each
        // Resistance between node i < j in a std::vector of
        // pair (i,j) ∈ ℕ^2
        // This way we can automatically build a symmetric
        // matrix, take care of indexing
        // and avoid mistakes during the matrix filling
        resistor_topology T;
        T.reserve(23);
        T.push_back(resistor(1,2));
```

```
        T.push_back(resistor(1,5));
        T.push_back(resistor(2,5));
        T.push_back(resistor(2,3));
        T.push_back(resistor(2,14));
        T.push_back(resistor(3,4));
        T.push_back(resistor(3,15));
        T.push_back(resistor(4,6));
        T.push_back(resistor(4,15));
        T.push_back(resistor(5,7));
        T.push_back(resistor(5,14));
        T.push_back(resistor(6,9));
        T.push_back(resistor(6,15));
        T.push_back(resistor(7,10));
        T.push_back(resistor(7,11));
        T.push_back(resistor(8,9));
        T.push_back(resistor(8,12));
        T.push_back(resistor(8,13));
        T.push_back(resistor(8,15));
        T.push_back(resistor(9,13));
        T.push_back(resistor(10,11));
        T.push_back(resistor(11,12));
        T.push_back(resistor(12,13));
        // The "base" matrix A_0 will have no resistance between 14 and
            15

        // We implement voltage ground and sources by
        // specifying which node is connected to ground/source
        // trough a resistance.  This will be also part of the r.h.s.
        voltage_topology S;
        S.reserve(4);
        S.push_back(voltage(6, 16, V));
        S.push_back(voltage(7, 17, 0));
        S.push_back(voltage(11,17, 0));
        S.push_back(voltage(14,17, 0));

        // Automatically build A0 filling from topology
        // A0 is the matrix A_R (i.e.  with R_x = R)
        MatrixXd A0 = MatrixXd::Zero(nnodes, nnodes);
        for(resistor & res: T) {
            // Shift indices down by 1 (indices in Figure start from 1)
            std::size_t i = res.first - 1;
            std::size_t j = res.second - 1;
            // Fill diagonal:  each resistance contributes += R
            // into the diagonal of both nodes
            A0(i,i) += 1;
            A0(j,j) += 1;
            // Fill the off diagonal (negative part in ΔW_{i,j},
            // the matrix is kept symmetric
            A0(i,j) -= 1;
            A0(j,i) -= 1;
        }

        // Fill in the rest (source and ground), i.e.
        // components with ΔW_{i,j} with j > 15.
        // Each node i connected to ground or source contributes
        // to the r.h.s b with R·V
        // (R is the resistence between node i and ground/source
        // node, V is voltage at sink or source)
        // and to its own diagonal with R
        b = MatrixXd::Zero(nnodes, 1);
        for(voltage & volt: S) {
```

```
            // Shift index down by 1 and get voltage in W2
            int i = std::get<0>(volt) - 1;
            double source_voltage = std::get<2>(volt);
            // Add voltage to r.h.s.  (resistance assumed to be R)
            b(i) += source_voltage;
            // Add resistance to matrix diagonal:  contribution of source current
            // scaled by R.
            A0(i,i) += 1;
        }

        // Precompute the 'lu' factorization of A0
        lu = A0.lu();
    };

    /* \brief Compute the impedance given the resistance R_x.
     * Use SMW formula for low rank perturbations to reuse LU
     * factorization.
     * \param Rx Resistence R_x > 0 between node 14 and 15
     * \return Impedance V/I of the system A_{R_x}
     */
    double operator()(double Rx) {
        // Store the scaled factor for convenience
        double f = R/Rx;

        // Create u:  the vector in A + u*v^T
        VectorXd u = VectorXd::Zero(nnodes);
        u(13) = -std::sqrt(f);
        u(14) = std::sqrt(f);
        // Here:  v = u

        // Use SMW formula to compute (A + u·u^T)^{-1}·b.
        // Formula:
        // A^{-1}*b - A^{-1}*u**(I + V*A^{-1}*U)^{-1}V*A^{-1}b

        // Start by precomputing A^{-1}b, needed twice
        VectorXd Ainvrhs = lu.solve(b);
        // Then, precompute A^{-1}u, needed twice
        VectorXd Ainvu = lu.solve(u);
        // Then, compute alpha.  Alpha is just a number.
        double alpha = 1 + u.dot(Ainvu);
        // Put the formula toghether, x is a column vector containing voltages
        // at each node (except 16 and 17, which are prescribed)
        VectorXd x = Ainvrhs - Ainvu * u.dot(Ainvrhs) / alpha;

        // Compute the current I = ΔW_{16,5}/R
        // and then impedance = V/I.
        // Here ΔW_{16,5} = W_16 - x_5 = V - x_5.
        return V * R / (V - x(5));
    }
private:
    PartialPivLU<MatrixXd> lu; //< Store LU decomp.  of matrix A.
    double R, V; //< Resistance R and source voltage W.
    VectorXd b; //< R.h.s vector prescribing sink/source voltages.
};
```

Get it on GitLab (impedancemap.cpp).

▲

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 ($a_i, b_i \in \mathbb{R}$):

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

This matrix is an instance of a banded matrix.

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

```
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 $\mathbf{y} = \mathbf{A}\mathbf{x}$ (`Vector y` is the output).

SOLUTION of (3-6.a):

C++11-code 3.6.2: Solution of (3-6.a)

```
template <class Vector>
void multAx(const Vector & a, const Vector & b, const Vector & x, Vector & y) {
    unsigned int n = x.size();
    if( a.size() < n - 1 || b.size() < n - 2 ) {
        // Size check, error if do not match
        std::cerr << "Error: size mismatch!" << std::endl;
        return;
    }
    y.resize(n);

    // Handle first two rows
    if(n > 1) y(0) = 2*x(0) + a(0)*x(1);
    else { // Special case n = 1
        y(0) = 2*x(0);
        return;
    }
    if(n > 2) y(1) = 2*x(1) + a(1)*x(2);
    else { // Special case n = 2
        y(1) = 2*x(1);
        return;
    }
```

```
    // Row if n > 3 and without last row
    for(unsigned int i = 2; i < n-1; ++i) {
        y(i) = 2*x(i) + b(i-2)*x(i-2) + a(i)*x(i+1);
    }

    // Last row special case
    if(n > 2) y(n-1) = 2*x(n-1) + b(n-3)*x(n-3);
}
```

Get it on GitLab (efficientbandmult.cpp).

▲

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

HINT 1 for (3-6.b): Give a proof by contradiction that $\ker \mathbf{A} = \{\mathbf{0}\}$ by looking at the largest (in modulus) component of $\mathbf{x} \in \ker \mathbf{A}$.

SOLUTION of (3-6.b):

Assume by contradiction that $\ker \mathbf{A} \neq \{\mathbf{0}\}$.

Pick $\mathbf{x} \in \ker \mathbf{A}$ and consider its largest (in modulus) component $i = \operatorname{argmax}|x_i|$, with $x_i \neq 0$.

Since $\mathbf{A}\mathbf{x} = \{\mathbf{0}\}$, we have that $2x_i + a_i x_{i+1} + b_{i-2}x_{i-2} = 0 \Rightarrow 2 = -\left(\frac{x_{i+1}}{x_i}a_i + \frac{x_{i-2}}{x_i}b_{i-2}\right) \Rightarrow 2 \leq |\frac{x_{i+1}}{x_i}a_i + \frac{x_{i-2}}{x_i}b_{i-2}|$ (the last step is always true).

Then, given that $a_i, b_i \in [0,1]$ by hypothesis and $\frac{x_{i+1}}{x_i}, \frac{x_{i-2}}{x_i} \in [-1,1]$ from x_i being the largest (in modulus) component, we can say that $2 \leq |\frac{x_{i+1}}{x_i}a_i + \frac{x_{i-2}}{x_i}b_{i-2}| \leq a_i + b_{i-2} \leq 2$, unless all elements of $\mathbf{x}$ are equal. (But in this case, with $x \neq 0$, we have another contradiction, as $2x + a_i x + b_{i-2}x = 0 \Rightarrow 2 + a_i + b_{i-2} = 0$ cannot be true.)

By contradiction, $\ker \mathbf{A} = \{\mathbf{0}\}$.

Notice that $\ker \mathbf{A} = \{\mathbf{0}\}$ can also be proven in a more straightforward way from the general fact that kernel vectors of irreducible and diagonally dominant matrices (see [Lecture → Def. 2.8.0.8]), like $\mathbf{A}$ in our example, must be multiples of $[1,1,\dots,1]^\top$ (i.e. all their elements must be equal). Given this fact, we can immediately jump to the step where we see that $2x + a_i x + b_{i-2}x = 0$ cannot hold.

▲

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

```
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 $\mathbf{r} = \mathbf{A}\mathbf{x}$ (`Vector x` is the output).

HINT 1 for (3-6.c): Just think about the resulting linear system and if it can immediately be solved starting from one of the equations.

SOLUTION of (3-6.c):

C++11-code 3.6.3: Solution of (3-6.c)

```
template <class Vector>
void solvelseAupper(const Vector & a, const Vector & r, Vector & x) {
    // Set up dimensions
    unsigned int n = r.size();
    x.resize(n);

    // Backward substitution
    x(n-1) = 0.5 * r(n-1);
    for(int j = n-2; j >= 0; --j) {
        x(j) = 0.5*(r(j) - a(j)*x(j+1));
    }
}
```

Get it on GitLab (efficientbandmult.cpp).

▲

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

HINT 1 for (3-6.d): Thanks to the constraint $a_i, b_i \in [0,1]$, pivoting is not required in order to ensure the stability of Gaussian elimination (see [Lecture → Lemma 2.8.0.9]). Moreover, Gaussian elimination fills in zero entries of $\mathbf{A}$ only along one lower off-diagonal, which can be stored as vector `Vector c` (similarly to what was done in (3-6.a), (3-6.c)). If you store the diagonal (initially only made of 2s) as `Vector d`, you can perform a straightforward Gaussian elimination only operating on vectors `Vector c`, `Vector d` and `Vector r`.

SOLUTION of (3-6.d):

Try to perform Gaussian elimination on matrix $\mathbf{A}$ in Eq. (3.6.1), augmented by the known vector $\mathbf{r}$.

Start from cell $(3,1)$, where b_1 is. By subtracting the 1st row from the 3rd row, you fill in the zero cell $(3,2)$ with $-fracb_12a_1$: call this entry c_2 (c_1 is the zero cell $(2,1)$). On the other hand, r_3 (3rd element of the known vector $\mathbf{r}$, in position $(3,n+1)$) becomes $= r_3 - \frac{b_1}{2}r_1$.

Now in the 2nd column you have nonzero cells c_2 and b_2, in position $(3,2)$ and $(4,2)$ respectively. To make them zero, subtract the 2nd row from the 3rd and 4th row. You then make cell $(3,3)$, which we can call d_3 as it is the 3rd cell along the diagonal, $= 2 - fracc_22a_2$, while cell the zero cell $(4,3)$, which we can call c_3, becomes $= -fracb_22a_2$. On the other hand, r_3 becomes $= r_3 - fracc_22r_2$ while r_4 becomes $= r_4 - fracb_22r_2$.

Given these first two steps, you can derive the general algorithm in the solution code below.

C++11-code 3.6.4: Solution of (3-6.d)

```
template <class Vector>
void solvelseA(const Vector & a, const Vector & b, const Vector & r, Vector & x) {
    // Set up dimensions
    int n = r.size();
    Vector c(n-1,0);
    Vector d(n,  2);
    Vector y;
    x = r;
    y = r;

    // Plain vectors are enough:
    // Fill-in is confined to a single lower off-diagonal, which can be held in vector c
    for(int i = 0; i < n-2; ++i) {
     c(i+1) = -b(i)/d(i)*a(i);
     d(i+1) -= c(i)/d(i)*a(i);
     y(i+1) -= c(i)/d(i)*y(i);
     y(i+2) -= b(i)/d(i)*y(i);
    }

    x(n-1) = y(n-1) / d(n-1);
    for(int i = n-2; i >= 0; --i) {
     x(i) = (y(i) - a(i)*x(i+1)) / d(i);
    }
}
```

Get it on GitLab (efficientbandmult.cpp).

▲

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

HINT 1 for (3-6.e): Think about the complexity of each step of the algorithm you coded and "sum them up".

SOLUTION of (3-6.e):

Firstly, to build the matrix we need at most $O(3n)$ insertions (3 per row). Secondly, for the elimination stage we use three for-loops: one loop of size n and two loops of size, at most, 3 each (exploiting the banded structure of A), thus $O(n \cdot 3 \cdot 2)$ operations. Thirdly, for the backward substitution we use two loops, one of size n and the other of size, at most, 3, thus $O(n \cdot 3)$ operations. Therefore, the total complexity amounts to $O(12n)$ operations, which results in $O(n)$ complexity.

▲

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

HINT 1 for (3-6.f): The standard way of initialising a sparse EIGEN matrix efficiently is via the triplet format discussed in [Lecture → Section 2.7.2]. You may also use directly initialise a sparse matrix, provided that you `reserve()` (EIGEN's function) enough space for the nonzero entries of each column (see the EIGEN documentation).

SOLUTION of (3-6.f):

C++11-code 3.6.5: Solution of (3-6.f)

```
template <class Vector>
void solveIseAEigen(const Vector & a, const Vector & b, const Vector & r, Vector &
    x) {
    // Set up dimensions
    typedef typename Vector::Scalar Scalar;
    unsigned int n = r.size();

    // Fill in matrix:
    // We reserve 3 nonzero entries per row for Gaussian fill-in
    SparseMatrix<Scalar> A(n,n);
    A.reserve(3);
    for(unsigned int i = 0; i < n; ++i) {
        A.insert(i,i) = 2;
        if(i < n-1) A.insert(i,i+1) = a(i);
        if(i >= 2)  A.insert(i,i-2) = b(i-2);
    }
    A.makeCompressed();

    // Call SparseLU
    SparseLU< SparseMatrix<Scalar> > solver;
    solver.analyzePattern(A);
    solver.compute(A);
    x = solver.solve(r);
}
```

Get it on GitLab (efficientbandmult.cpp).

▲

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

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

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 of (3-7.a):

i) $\mathbf{H}\mathbf{H} = (\mathbf{I} - 2\mathbf{v}\mathbf{v}^H)^2 = \mathbf{I} - 4\mathbf{v}\mathbf{v}^H + 4(\mathbf{v}\mathbf{v}^H)(\mathbf{v}\mathbf{v}^H) \overset{\mathbf{v}^H\mathbf{v}=1}{=} \mathbf{I}$.

ii) $\mathbf{H}^H = (\mathbf{I} - 2\mathbf{v}\mathbf{v}^H)^H = \mathbf{I} - 2\mathbf{v}\mathbf{v}^H = \mathbf{H}, \quad \Longrightarrow \quad \mathbf{H}\mathbf{H}^H = \mathbf{H}\mathbf{H} \overset{i)}{=} \mathbf{I}$.

iii) $\mathbf{H}\mathbf{H}^H = \mathbf{I} \quad \Longrightarrow \quad \mathbf{H}^H = \mathbf{H}^{-1} \quad \Longrightarrow \quad \mathbf{H}$ is unitary

$\Longrightarrow \quad \forall\, \lambda_i$ eigenvalue of $\mathbf{H}$, $|\lambda_i| = 1 \quad \Longrightarrow \quad |\det \mathbf{H}| = |\prod_{i=1}^{n} \lambda_i| = 1$.

iv) $\mathbf{H}\mathbf{s} = \mathbf{s} - 2\mathbf{v}\mathbf{v}^H\mathbf{s} \overset{\mathbf{v}^H\mathbf{s}=0}{=} \mathbf{s}$.

v) $\mathbf{H}\mathbf{z} = c\mathbf{H}\mathbf{v} = c(\mathbf{v} - 2\mathbf{v}\mathbf{v}^H\mathbf{v}) \overset{\mathbf{v}^H\mathbf{v}=\mathbf{I}}{=} -c\mathbf{v} = -\mathbf{z}$.

▲

(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 of (3-7.b):

The matrix $\mathbf{H}$ is the following:

$$\mathbf{H} = \frac{1}{5}\begin{bmatrix} 4 & -3 \\ -3 & -4 \end{bmatrix}.$$

This transformation is a reflection across the "hyperplane" (= line, in $\mathbb{R}^2$) orthogonal to $\mathbf{v}$:

$$\begin{bmatrix} x \\ y \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 3 \end{bmatrix} = x + 3y = 0.$$

The line of the reflection is defined through $x + 3y = 0$.

▲

(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 of (3-7.c):

We call $\mathbf{c}_1$ and $\mathbf{c}_2$ the columns of $\mathbf{C}$. Since the reflection preserves the norm of vectors, and since the image of the first column has to be zero in the second component, there are two possibilities for the reflection $\mathbf{H}$:

$$\mathbf{H}_1 : \mathbf{c}_1 \mapsto \begin{bmatrix} \|\mathbf{c}_1\| \\ 0 \end{bmatrix}, \qquad \mathbf{H}_2 : \mathbf{c}_1 \mapsto \begin{bmatrix} -\|\mathbf{c}_1\| \\ 0 \end{bmatrix}.$$

Define $\mathbf{v}_1$ resp. $\mathbf{v}_2$ the vectors defining the transformation $\mathbf{H}_1$ resp. $\mathbf{H}_2$. Since $\|\mathbf{c}_1\| = \sqrt{5}$ we have, for $\mathbf{H}_1$:

$$\mathbf{v}_1 = \frac{\mathbf{c}_1 - \|\mathbf{c}_1\|\mathbf{e}_1}{\left\|\mathbf{c}_1 - \|\mathbf{c}_1\|\mathbf{e}_1\right\|} = \frac{1}{\sqrt{10 - 2\sqrt{5}}} \begin{bmatrix} 1 - \sqrt{5} \\ 2 \end{bmatrix},$$

therefore

$$\mathbf{H}_1 = \mathbf{I} - 2\mathbf{v}_1\mathbf{v}_1^H = \mathbf{I} - \frac{2}{5 - \sqrt{5}} \begin{bmatrix} 3 - \sqrt{5} & 1 - \sqrt{5} \\ 1 - \sqrt{5} & 2 \end{bmatrix} = \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix}.$$

And for $\mathbf{H}_2$:

$$\mathbf{v}_2 = \frac{\mathbf{c}_1 + \|\mathbf{c}_1\|\mathbf{e}_1}{\left\|\mathbf{c}_1 + \|\mathbf{c}_1\|\mathbf{e}_1\right\|} = \frac{1}{\sqrt{10 + 2\sqrt{5}}} \begin{bmatrix} 1 + \sqrt{5} \\ 2 \end{bmatrix},$$

therefore

$$\mathbf{H}_2 = \mathbf{I} - 2\mathbf{v}_2\mathbf{v}_2^H = \mathbf{I} - \frac{2}{5 + \sqrt{5}} \begin{bmatrix} 3 + \sqrt{5} & 1 + \sqrt{5} \\ 1 + \sqrt{5} & 2 \end{bmatrix} = -\mathbf{H}_1.$$

The column $\mathbf{c}_2$ is mapped to

$$\mathbf{H}_1\mathbf{c}_2 = \frac{1}{\sqrt{5}} \begin{bmatrix} 1 & 2 \\ 2 & -1 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \end{bmatrix} = \begin{bmatrix} 2\sqrt{5} \\ \sqrt{5} \end{bmatrix}.$$

Finally, the matrix $\mathbf{R}_1$ is

$$\mathbf{R}_1 = \mathbf{H}_1\mathbf{C} = \mathbf{H}_1\,[\mathbf{c}_1, \mathbf{c}_2] = \begin{bmatrix} \sqrt{5} & 2\sqrt{5} \\ 0 & \sqrt{5} \end{bmatrix}.$$

▲

(3-7.d) ⚀ Implement a function

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

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

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

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

SOLUTION of (3-7.d):

▲

(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.

HINT 1 for (3-7.e): Instead of **MatrixXd** we specify an abstract matrix type for the matrix argument, which also allows to pass matrices of other double-compatible scalar type and expressions derived from them. ⌟

HINT 2 for (3-7.e): The vector entry $(\mathbf{v}_i)_i$ can be computed, since we know $(\mathbf{v}_i)_i \geq 0$ and $\|\mathbf{v}_i\|_2 = 1$. ⌟

SOLUTION of (3-7.e):

▲

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

$$\mathbf{AX} + \mathbf{XA}^\top = \mathbf{I}\,, \tag{3.8.1}$$

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:

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

HINT 1 for (3-8.a): Recall the definition of a linear mapping between two vector spaces from linear algebra. ⌟

SOLUTION of (3-8.a):

Take $\alpha, \beta \in \mathbb{R}$ and $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n,n}$. We readily compute:

$$\begin{aligned} L(\alpha\mathbf{X} + \beta\mathbf{Y}) &= \mathbf{A}(\alpha\mathbf{X} + \beta\mathbf{Y}) + (\alpha\mathbf{X} + \beta\mathbf{Y})\mathbf{A}^\top \\ &= \alpha\mathbf{AX} + \beta\mathbf{AY} + \alpha\mathbf{XA}^\top + \beta\mathbf{YA}^\top \\ &= \alpha\left(\mathbf{AX} + \mathbf{XA}^\top\right) + \beta\left(\mathbf{AY} + \mathbf{YA}^\top\right) \\ &= \alpha L(\mathbf{X}) + \beta L(\mathbf{Y}) \end{aligned}$$

This concludes the demonstration.

▲

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

Owing to the observatin made in Sub-problem (3-8.a), (3.8.1) is equivalent to the following linear system of equations:

$$\mathbf{C}\,\mathrm{vec}(\mathbf{X}) = \mathbf{b} \tag{3.8.2}$$

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

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

HINT 1 for (3-8.c): Start by expressing X as $\left[\begin{smallmatrix} x_1 & x_3 \\ x_2 & x_4 \end{smallmatrix}\right]$.

SOLUTION of (3-8.c):

Writing $X = \left[\begin{smallmatrix} x_1 & x_3 \\ x_2 & x_4 \end{smallmatrix}\right]$, we have that $\mathrm{vec}(\mathbf{X}) = (x_1, x_2, x_3, x_4)^\top$. A direct calculation shows that (3.8.1) is equivalent to Eq. (3.8.2) with:

$$\mathbf{C} = \begin{bmatrix} 4 & 1 & 1 & 0 \\ -1 & 5 & 0 & 1 \\ -1 & 0 & 5 & 1 \\ 0 & -1 & -1 & 6 \end{bmatrix} \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

▲

(3-8.d) (25 min.) Use the Kronecker product $\bigotimes$ [Lecture → Def. 1.4.3.7] to find a general expression for $\mathbf{C}$ in terms of $\mathbf{A}$.

SOLUTION of (3-8.d):

By elementary properties of the matrix product:

$$\mathbf{AX} = \left[\mathbf{A}(\mathbf{X})_{:,1}, \dots, \mathbf{A}(\mathbf{X})_{:,n}\right] \quad \Leftrightarrow \quad \mathrm{vec}(\mathbf{AX}) = \left(\mathbf{I} \bigotimes \mathbf{A}\right) \mathrm{vec}(\mathbf{X}) \,,$$

$$\left(\mathrm{vec}(\mathbf{XA}^\top)\right)_k = \left(\mathbf{XA}^\top\right)_{i^*,j^*} = \sum_{\ell=1}^{n} (\mathbf{X})_{i^*,\ell} (\mathbf{A})_{j^*,\ell} = \sum_{\ell=1}^{n} (\mathbf{A})_{j^*,\ell} (\mathrm{vec}(\mathbf{X}))_{i^*+(\ell-1)n} \,,$$

where $(\mathrm{vec}(\mathbf{X}))_k = (\mathbf{X})_{i^*,j^*}$. Thus, we have

$$\boxed{\mathbf{C} = \mathbf{I} \bigotimes \mathbf{A} + \mathbf{A} \bigotimes \mathbf{I}} \,. \tag{3.8.3}$$

The first term is related to $\mathbf{AX}$, the second to $\mathbf{XA}^\top$.

▲

(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].

HINT 1 for (3-8.e): Read [Lecture → Section 2.7.2] carefully before starting.

SOLUTION of (3-8.e):

The argument matrix is assumed to be densely populated, but we have to return a matrix in sparse format. The simplest option for efficient initialization is to go via the triplet/COO format. Then go through the $n \times n$-blocks in (3.8.3) to fill the triplets

C++ code 3.8.4: Solution of (3-8.e)

```
SparseMatrix<double> buildC(const MatrixXd &A) {
  // Initialization
  int n = A.rows();
  SparseMatrix<double> C(n * n, n * n);
  std::vector<Triplet<double>> triplets;
  MatrixXd I = MatrixXd::Identity(n, n);

  // Iterate over n*n blocks.
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      // TO DO: (3-8.e) Fill in the (i,j)-th block of C.
      // Hint: The vector triplets should contain the position
      // and value the non-zero entries of C.
      // Use triplets.push_back() to add a new
      // Triplet<double> to the list.
      // START
      // Set diagonal
      if (i == j) {
        for (int k1 = 0; k1 < n; k1++) {
          for (int k2 = 0; k2 < n; k2++) {
            Triplet<double> triplet(i * n + k1, j * n + k2,
                                    A(i, j) * I(k1, k2) + A(k1, k2));
            triplets.push_back(triplet);
          }
        }
        // Set other cells
      } else {
        for (int k = 0; k < n; k++) {
          Triplet<double> triplet(i * n + k, j * n + k, A(i, j));
          triplets.push_back(triplet);
        }
      }
      // END
    }
  }
  C.setFromTriplets(triplets.begin(), triplets.end());
  C.makeCompressed();
  return C;
}
```

Get it on GitLab (lyapunov.cpp).

▲

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

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

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

HINT 1 for (3-8.f): Not every invertible matrix $\mathbf{A}$ allows a solution. If $\mathbf{A}$ and $-\mathbf{A}$ have a common

eigenvalue, the system $\mathbf{Cx} = \mathbf{b}$ is singular. This is for example the case of the matrix $\mathbf{A} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. For an efficient solution, see Chapter 15 of Higham's book. ⌟

SOLUTION of (3-8.f):

C++11-code 3.8.5: Solution of (3-8.f)

```
void solveLyapunov(const MatrixXd &A, MatrixXd &X) {
  // To solve A*X + X*A^T = I, we must first write
  // the LSE in "canonical form", C*vecX = b.
  int n = A.rows();
  SparseMatrix<double> C;
  MatrixXd I = MatrixXd::Identity(n, n);
  VectorXd b(n * n);
  VectorXd vecX(n * n);

  // TO DO: (3-8.f) Fill in the entries of C, b, and vecX.
  // Hint: Use SparseLU to solve the LSE.
  // START
  C = buildC(A);
  b = Map<MatrixXd>(I.data(), n * n, 1);
  SparseLU<SparseMatrix<double>> solver;
  solver.compute(C);
  vecX = solver.solve(b);
  // END

  X = Map<MatrixXd>(vecX.data(), n, n);
}
```

Get it on GitLab (lyapunov.cpp).

▲

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

```
bool testLyapunov();
```

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

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

SOLUTION of (3-8.g):

C++ code 3.8.6: Solution of (3-8.g)

```
bool testLyapunov() {
  // Initialization
  unsigned int n = 5;
  MatrixXd A(n, n);
  A << 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;

  double tol = 1e-8;
  bool works;

  // TO DO: (3-8.g)
  // i) Print the full system matrix C (with zeroes),
  // ii) print the solution X, and
  // iii) return true if and only if the error defined
  // by || A*X + X*A^T - I || is less than tol.
  // START
  // Test 'buildC'
  SparseMatrix<double> C = buildC(A);
  std::cout << "C = " << C << std::endl;

  // Test 'solveLyapunov'
  MatrixXd X(n, n);
  solveLyapunov(A, X);

  std::cout << "X = " << X << std::endl;

  // Verify the solution if you obtain zero
  MatrixXd I = MatrixXd::Identity(n, n);
  double error = (A * X + X * A.transpose() - I).norm();
  std::cout << "Correct if close to 0: " << error << std::endl;

  works = (error < tol);
  // END

  return works;
}
```

Get it on GitLab (lyapunov.cpp).

▲

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

SOLUTION of (3-8.h):

For general matrices $\mathbf{A}$ and $\mathbf{B}$ we have that $\mathrm{nnz}(\mathbf{A}\otimes\mathbf{B}) = \mathrm{nnz}(\mathbf{A})\,\mathrm{nnz}(\mathbf{B})$. This follows from the fact that the block in position (i,j) of the matrix $\mathbf{A}\otimes\mathbf{B}$ is $a_{ij}\mathbf{B}$.

In our case, we immediately obtain $\mathrm{nnz}(\mathbf{C}) = \mathrm{nnz}(\mathbf{I}\otimes\mathbf{A}+\mathbf{A}\otimes\mathbf{I}) \leq \mathrm{nnz}(\mathbf{I}\otimes\mathbf{A}) + \mathrm{nnz}(\mathbf{A}\otimes\mathbf{I}) \leq 2\,\mathrm{nnz}(\mathbf{I})\,\mathrm{nnz}(\mathbf{A})$. Summarising, $\mathrm{nnz}(\mathbf{C}) < 2n\,\mathrm{nnz}(\mathbf{A})$. The sharpness of this bound can be checked with the matrix $\mathbf{A} = \left[\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right]$.

This bound implies that, even if $\mathbf{A}$ is not sparse, we have that $\mathrm{nnz}(\mathbf{C}) < 2n^3 \ll n^4$. Therefore, $\mathbf{C}$, being and $n^2 \times n^2$ matrix, can be regarded as sparse for any $\mathbf{A}$.

Another way to see this is the following. The block in position (i,j) of matrix $\mathbf{C}$ is given by $a_{ij}\mathbf{I} + \delta_{ij}\mathbf{A}$ (δ_{ij} is Kronecker delta). Suppose now that $a_{ij} \neq 0$. We have two cases:

$i \neq j$ There are at most n nonzero entries in the block in position (i,j) of matrix $\mathbf{C}$, which is equal to

$a_{ij}\mathbf{I}$, and one nonzero entry in each diagonal block. This gives a total of, at most, $2n$ nonzero entries.

$i = j$ There is at most one nonzero entry in each diagonal block. Thus, a total of, at most, n nonzero entries.

▲

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

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

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.

HINT 1 for (3-9.a): Conceive a simple algorithm for solving a triangular linear system, and find out for which coefficient matrices it will run without break-down. These matrices must then be invertible.

Alternatively, you may appeal to the criteria of [Lecture → Thm. 2.2.1.4].

SOLUTION of (3-9.a):

The matrix $\mathbf{R}$ being upper triangular $\det(\mathbf{R}) = \prod_{i=1}^{n}(\mathbf{R})_{i,i}$, implies that all the diagonal elements must be non-zero for $\mathbf{R}$ to be invertible.

Alternatively, you may argue that backward elimination for solving the LSE $\mathbf{R}\mathbf{x} = \mathbf{b}$ will always produce a solution, if division by the diagonal elements of $\mathbf{R}$ can be carried out.

▲

(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

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

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 of (3-9.b):

From the first n equations of (3.9.2) we deduce

$$\mathbf{z} = \mathbf{R}^{-1}(\mathbf{b} - \xi\mathbf{v}) ,$$

from which we find, using the bottom equation of (3.9.2)

$$\mathbf{u}^\top\mathbf{R}^{-1}(\mathbf{b} - \xi\mathbf{v}) = \beta \quad \Rightarrow \quad \xi = \frac{\beta - \mathbf{u}^\top\mathbf{R}^{-1}\mathbf{b}}{\mathbf{u}^\top\mathbf{R}^{-1}\mathbf{v}} ,$$

assuming that the denominator does not vanish.

Alternatively, applying the block Gauss elimination as in [Lecture → § 2.6.0.2], we obtain:

$$\begin{bmatrix} \mathbf{1} & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{z} \\ \xi \end{bmatrix} = \begin{bmatrix} \mathbf{R}^{-1}(\mathbf{b} - \mathbf{v}s^{-1}b_s) \\ s^{-1}b_s \end{bmatrix},$$

with $s := -(\mathbf{u}^\top \mathbf{R}^{-1}\mathbf{v})$, $b_s := (\beta - \mathbf{u}^\top \mathbf{R}^{-1}\mathbf{b})$. Of course, we have to assume $s \neq 0$.

▲

(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 of (3-9.c):

The square matrix $\mathbf{A}$ is regular, if the corresponding linear system has a unique solution for every right hand side vector. If $\mathbf{u}^T\mathbf{R}^{-1}\mathbf{v} \neq 0$ the expressions derived in the previous sub-problem show that a solution can be found for any $\mathbf{b}$ and β, because $\mathbf{R}$ is already known to be invertible.

Ab alternative proof can be based on bock-LU decomposition [Lecture → Rem. 2.3.2.19]:

$$\mathbf{A} = \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{u}^T & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{I} & \mathbf{O} \\ \mathbf{u}^\top\mathbf{R}^{-1} & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{v} \\ \mathbf{O} & -\mathbf{u}^\top\mathbf{R}^{-1}\mathbf{v} \end{bmatrix},$$

and the upper triangular factor is invertible, once the bottom-right diagonal element does not vanish, *cf.* Sub-problem (3-9.a).

▲

(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 $\mathbf{A}\mathbf{x} = \mathbf{b}$ (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 of (3-9.d):

It is important to communicate to EIGEN that $\mathbf{R}$ is upper triangular so that the library knows that it can skip the costly elimination step.

C++11-code 3.9.3: Block Gauss based solution

```
void solvelse(const MatrixXd & R,
              const VectorXd & v, const VectorXd & u,
              const VectorXd & bb, VectorXd & x) {
    // Size of R, which is wize of u, v, and size of bb is n+1
    unsigned int n = R.rows();

    // TO DO: (3-9.d)
    // i) Use assert() to check that R, v, u, bb
    // all have the have the appropriate sizes.
    // ii) Use (3-9.b) to solve the LSE.
    // Hint:  Use R.triangularView<Upper>() to make
    // use of the triangular structure of R.
    // START
    assert( n == R.cols() && n == u.size()
            && n == v.size() && n+1  == bb.size()
            && "Size mismatch!");

    const TriangularView<const MatrixXd, Upper> & triR = R.triangularView<Upper>();

    // s is the Schur complement and, in this case, is a scalar
    // also b_s is a scalar
    // snv = s^{-1}, b_s as in lecture notes
    // sinvbs = s^{-1} * b_s
    double sinv = - 1. / u.dot(triR.solve(v));
    double bs = (bb(n) - u.dot(triR.solve(bb.head(n))));
    double sinvbs = sinv*bs;

    // Stack the vector (z,ξ)^T =: x
    x = VectorXd::Zero(n+1);
    x << triR.solve(bb.head(n) - v*sinvbs),
        sinvbs;
    // END
}
```

Get it on GitLab (blockdecomp.cpp).

▲

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

HINT 1 for (3-9.e): In order to obtain information about direct elimination solvers available in EIGEN consult EIGEN documentation. ⌟

HINT 2 for (3-9.e): Heed the warning of [Lecture → Rem. 1.5.3.15]: Never test whether a computed results of real type agrees exactly with zero. ⌟

SOLUTION of (3-9.e):

C++11-code 3.9.4: LU-decomposition based solution

```
bool testSolveLSE(const MatrixXd & R,
                  const VectorXd & v, const VectorXd & u,
                  const VectorXd & b, VectorXd & x) {
    bool areTheSame = false;
    // TO DO: (3-9.e)
    // i) Create the system matrix A and solve the LSE, using
    // an Eigen LU-solver.  Store the solution in x.
    // ii) Solve the LSE with solvelse(), and calculate the
    // difference between this solution and x.  Return true
    // if and only if the norm of this difference is close
    // enough to zero.
    // START
    // Size of R, which is wize of u, v, and size of bb is n+1
    unsigned int n = R.rows();
    assert( n == R.cols() && n == u.size()
            && n == v.size() && n+1  == b.size()
            && "Size mismatch!");

    MatrixXd A(n+1,n+1);
    // Build matrix using blocks
    A << R,              v,
         u.transpose(), 0;

    // Solve system using LU decpomposition
    x = A.partialPivLu().solve(b);

    // Solve system using solvelse()
    VectorXd y;
    solvelse(R, v, u, b, y);

    // We choose an acceptable tolerance.
    double tolerance = 1e-8;
    areTheSame = ( (x-y).norm() < tolerance*b.norm() );

    // END
    return areTheSame;
}
```

Get it on GitLab (blockdecomp.cpp).

▲

(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 $n \to \infty$?

SOLUTION of (3-9.f):

The complexity is $O(n^2)$. The backward substitution for $\mathbf{R}^{-1}\mathbf{x}$ is $O(n^2)$, vector dot product and subtraction of vectors is $O(n)$, so that the complexity is dominated by the backward substitution: $O(n^2)$.

▲

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}

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

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

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

(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.

HINT 1 for (3-10.a): Heed the advice of [Lecture → Rem. 2.2.1.6]! Instead of computing the inverse of a matrix, solve a linear system of equations.

SOLUTION of (3-10.a):

C++-code 3.10.2: Solution of (3-10.a)

```
double rankoneinvit(const VectorXd &d, const double &tol) {
  // Initialization
  double lmin = 0;
  VectorXd ev = d;
  double lnew = d.cwiseAbs().minCoeff();

  while (std::abs(lnew - lmin) > tol * lmin) {
    lmin = lnew;
    MatrixXd M = d.asDiagonal();
    M += ev * ev.transpose();
```

```
12      // Instead of calculating the inverse of M directly
13      // to get ev_new = M^{-1}*ev_old,
14      // we solve the LSE M*ev_new = ev_old.
15      ev = M.lu().solve(ev);
16      // Equivalent to ev <- ev/|ev|:
17      ev.normalize();
18      lnew = ev.transpose() * M * ev;
19    }
20
21    return lnew;
22  }
```

Get it on GitLab (rankoneinvit.cpp).

▲

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

HINT 1 for (3-10.b): Find which passage mostly contributes to the overall complexity of the code. What is the corresponding asymptotic complexity?

SOLUTION of (3-10.b):

Solving the linear system with matrix M (line 10) is the most relevant contributor to the asymptotic complexity. The complexity of solving a linear system is generally $O(n^3)$.

▲

(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.

HINT 1 for (3-10.c): Exploit the Sherman-Morrison-Woodbury formula (see [Lecture → § 2.6.0.12]).

SOLUTION of (3-10.c):

Since we want to solve $\mathbf{ev}_{new} = \mathbf{M}^{-1}\mathbf{ev}$ with $\mathbf{M} = \operatorname{diag}(\mathbf{d}) + \mathbf{ev}\cdot\mathbf{ev}^\top$, we can apply the Sherman-Morrison-Woodbury formula:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} - \frac{\mathbf{A}^{-1}\mathbf{u}(\mathbf{v}^H(\mathbf{A}^{-1}\mathbf{b}))}{1+\mathbf{v}^H(\mathbf{A}^{-1}\mathbf{u})}$$

In our case, assuming that $\mathbf{ev}$ is real, the formula becomes:

$$\begin{aligned}\mathbf{ev}_{new} &= \operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev} - \frac{\operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev}(\mathbf{ev}^\top(\operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev}))}{1+\mathbf{ev}^\top(\operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev})} \\ &= \operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev}\cdot\frac{1}{1+\mathbf{ev}^\top(\operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev})}\end{aligned}$$

As $\operatorname{diag}(\mathbf{d})^{-1} = \operatorname{diag}\left(d_1^{-1}, d_2^{-1}, \dots\right)$, this implementation is faster. More precisely, we simply need to perform a component-wise product between vectors $\operatorname{diag}\left(d_1^{-1}, d_2^{-1}, \dots\right)$ and $\mathbf{ev}$ and another one between $\mathbf{ev}^\top$ and $\operatorname{diag}(\mathbf{d})^{-1}\mathbf{ev}$. The asymptotic complexity is therefore $O(n)$.

C++-code 3.10.3: Solution of (3-10.c)

```
double rankoneinvit_fast(const VectorXd &d, const double &tol) {
  // Initialization
  double lmin = 0;
  VectorXd ev = d;
  double lnew = d.cwiseAbs().minCoeff();

  VectorXd dinv = (1 / d.array()).matrix();
  while (std::abs(lnew - lmin) > tol * lmin) {
    lmin = lnew;
    VectorXd ev0 = ev;

    // Here we solve the linear system
    // with the Sherman-Morrison-Woodbury formula
    // in the case of rank-1 perturbations.
    // This holds from M = diag(d) + ev * ev^t
    VectorXd Aib = dinv.cwiseProduct(ev);
    double temp = ev.transpose() * Aib;
    ev = Aib / (1 + temp);
    ev.normalize();
    // Better than the corresponding naive implementation.
    // This holds from M = diag(d) + ev * ev^t, too
    lnew = ev.transpose() * d.cwiseProduct(ev) + pow(ev.transpose() * ev0, 2);
  }

  return lnew;
}
```

Get it on GitLab (rankoneinvit.cpp).

▲

(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 of (3-10.d):

The loop body of the C++ function `rankoneinvit_fast` only consists in vector-vector multiplications. The new asymptotic complexity is therefore $O(n)$.

▲

(3-10.e) ⚁ (15 min.) Tabulate the runtimes of the two C++ implementations with different vector sizes $n = 2^p$, with $p = 2, 3, \ldots, 8$. As test vector `d` use `Eigen::VectorXd::LinSpaced(n,1,2)`. Estimate the laws of the different asymptotic complexities of the two implementations using the measured runtimes.

SOLUTION of (3-10.e):

C++-code 3.10.4: Solution of (3-10.e)

```
void rankoneinvit_runtime() {
  unsigned int repeats = 3;
  double tol = 1e-3;
  double lmin;
  VectorXd d;
  Timer tm_slow, tm_fast;

  std::cout << std::endl
            << std::setw(15) << "n" << std::setw(15) << "Slow" << std::setw(15)
            << "Fast" << std::endl;

  for (unsigned int p = 2; p <= 8; p++) {
    tm_slow.reset();
    tm_fast.reset();
    unsigned int n = pow(2, p);

    for (unsigned int r = 0; r < repeats; ++r) {
      d = VectorXd::LinSpaced(n, 1, 2);

      tm_slow.start();
      lmin = rankoneinvit(d, tol);
      tm_slow.stop();

      tm_fast.start();
      lmin = rankoneinvit_fast(d, tol);
      tm_fast.stop();
    }
    std::cout << std::setw(15) << n << std::scientific << std::setprecision(3)
              << std::setw(15) << tm_slow.min() << std::setw(15)
              << tm_fast.min() << std::endl;
  }
}
```

Get it on GitLab (rankoneinvit.cpp).

▲

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 $\mathbf{A}$: $\mathbf{A}\mathbf{x} = \mathbf{b}_i$. The computational cost of this problem may be reduced by performing an LU decomposition of $\mathbf{A}$ only once and reusing it for the different systems.

Related to [Lecture → Rem. 2.5.0.10]

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

Algorithm 3.11.1: Code `solvepermb`

Require: Matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$, vector $\mathbf{b} \in \mathbb{R}^n$
Ensure: Matrix $\mathbf{X} = [\mathbf{X}_1, \dots, \mathbf{X}_n] \in \mathbb{R}^{n\times n}$, $\mathbf{X}_i \in \mathbb{R}^n$
  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 `solvepermb` for $n \to \infty$?

HINT 1 for (3-11.a): Think about the complexity of solving a linear system (Gaussian elimination, see [Lecture → § 2.3.1.5]) for n times.

SOLUTION of (3-11.a):

This function solves the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}_i$ n times. In every iteration i (i going from 1 to n) $\mathbf{b}_i$ is the original vector shifted by $i-1$ elements. The output is the matrix $\mathbf{X} \in \mathbb{R}^{n\times n}$, where each column $\mathbf{X}_i$ corresponds to a solution of $\mathbf{A}\mathbf{X}_i = \mathbf{b}_i$.

The complexity of solving a linear system is generally $O(n^3)$.

The complexity of solving n linear systems is therefore $O(n^4)$.

▲

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

HINT 1 for (3-11.b): The C++ code should perform exactly the same computations in the same order. Consider the constructor of class template `FullPivLU` in EIGEN to solve the linear systems.

SOLUTION of (3-11.b):

C++11-code 3.11.2: Solution of (3-11.b)

```
void solvpermb(const MatrixXd & A, VectorXd & b, MatrixXd & X) {
    // Size of b, which is the size of A
    int n = b.size();
    assert( n == A.rows() && n == A.cols()
            && "Error: size mismatch!");
    X.resize(n,n);

    // For each loop iteration:
```

```
    // 1.  solve the linear system Ax = b,
    // 2.  store the result in a column of X
    // 3.  and shift b by one element for the next iteration.
    for(int l = 0; l < n; ++l) {
        X.col(l) = A.fullPivLu().solve(b);

        shift(b);
    }
}
```

Get it on GitLab (solvepermb.cpp).

▲

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

HINT 1 for (3-11.c): The complexity increases by solving the linear system n times *from scratch*. However, there are ways to represent the matrix $\mathbf{A}$ which makes the solution of the linear problem $\mathbf{Ax} = \mathbf{b}$ much easier to find. One of these ways is the LU decomposition. If we compute the LU decomposition of $\mathbf{A}$ once, we can reuse it when solving $\mathbf{Ax} = \mathbf{b}_i$ for different $\mathbf{b}_i$.

SOLUTION of (3-11.c):

C++11-code 3.11.3: Solution of (3-11.c)

```
void solvpermb_on3(const MatrixXd & A, VectorXd & b, MatrixXd & X) {
    // Size of b, which is the size of A
    int n = b.size();
    assert( n == A.cols() && n == A.rows()
            && "Error: size mismatch!");
    X.resize(n,n);

    // Notice that here we reuse the LU factorization
    FullPivLU<MatrixXd> LU = A.fullPivLu();

    for(int l = 0; l < n; ++l) {
        X.col(l) = LU.solve(b);

        shift(b);
    }
}
```

Get it on GitLab (solvepermb.cpp).

▲

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.

HINT 1 for (3-12.a): A matrix $\mathbf{A}$ is non-singular if and only if $\det|\mathbf{A}| \neq 0$ (which is also equivalent to $\mathbf{A}$ being invertible $\iff \operatorname{rank}(\mathbf{A}) = n$).

It can be proven by induction that the determinant of a triangular matrix is the product of its diagonal elements, i.e. $\det|\mathbf{A}| = \prod_{i=1}^{n} a_i$.

SOLUTION of (3-12.a):

$$a_i \neq 0 \quad \forall\, i = 1, \dots, n$$

▲

(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.

HINT 1 for (3-12.b): Consider a loop initialization.

SOLUTION of (3-12.b):

C++11-code 3.12.1: Solution of (3-12.b)

```
MatrixXd buildA(const VectorXd & a)
{
    // Initialization
    int n = a.size();
    MatrixXd A(n,n);

    for(int j=0; j<n; j++) {
     for(int i=j; i<n; i++) {
```

```
            A(i,j) = a(j);
        }
        }

        return A;
}
```

Get it on GitLab (structuredLSE.cpp).

▲

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

```
void solveA(const 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?

HINT 1 for (3-12.c): Remember what you did in Code 3.11.2.

The `main()` code to test your implementation is already provided in the template. ⌟

SOLUTION of (3-12.c):

C++11-code 3.12.2: Solution of (3-12.c)

```
void solveA(const VectorXd & a, const VectorXd & b, VectorXd & x) {
    // Size of b, which is the size of a
    int n = b.size();
    assert( n == a.size()
            && "Error: size mismatch!");
    x.resize(n);

     MatrixXd A = buildA(a);
    x = A.fullPivLu().solve(b);
}
```

Get it on GitLab (structuredLSE.cpp).

Even with a naive implementation, the solution of a lower triangular system corresponds to a forward substitution and should be of $O(n^2)$ complexity. However, EIGEN's methods are not optimized and follow the general complexity of solving a linear system, which is $O(n^3)$. This can be seen from the plot in Fig. 7.

Timings of naive solver

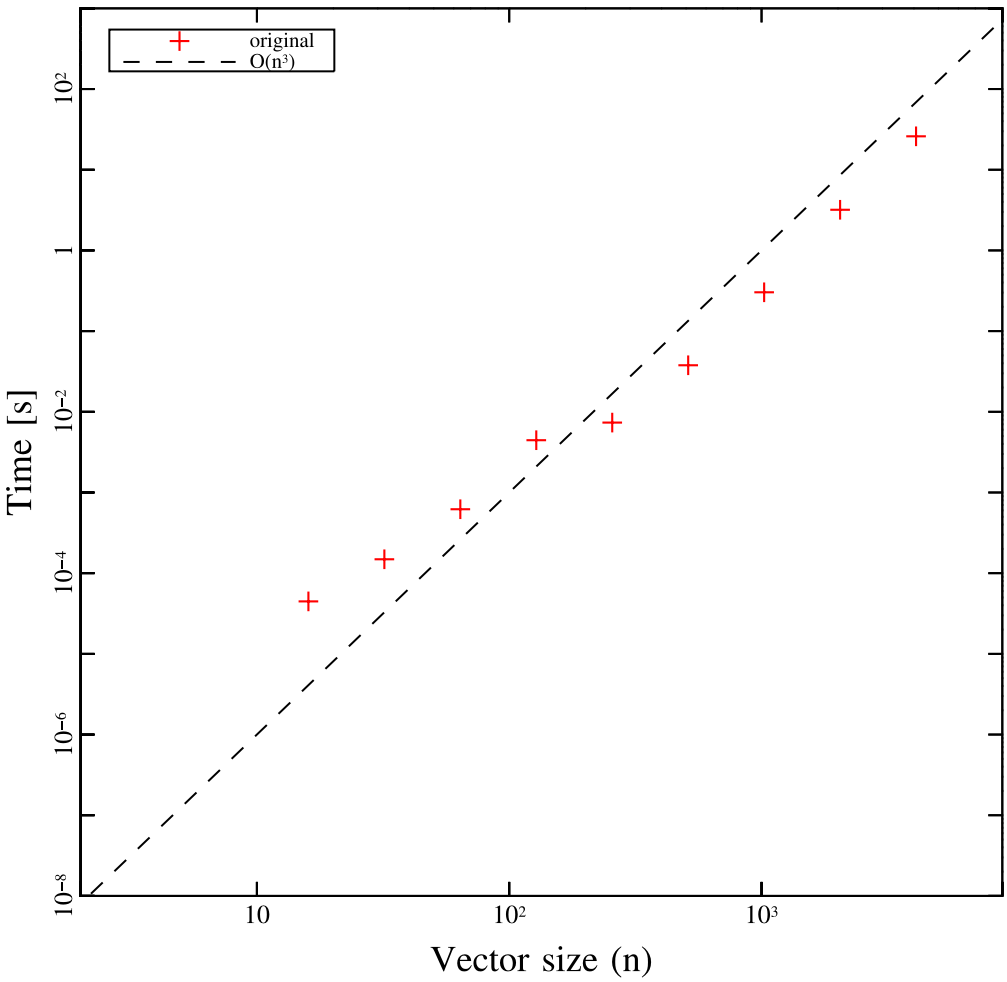


Fig. 7

▲

(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 $\mathbf{A}\mathbf{x} = \mathbf{b}$.

HINT 1 for (3-12.d): Notice the following equality:

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 1 & 1 & \dots & 1 \end{bmatrix} \cdot \begin{bmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & \ddots & \vdots \\ \vdots & & \ddots & 0 \\ 0 & 0 & \dots & a_n \end{bmatrix} = \mathbf{E} \cdot \mathbf{D} \,.$$

Then figure out an efficient way to solve a linear system of equations with coefficient matrix $\mathbf{E}$. ⌟

SOLUTION of (3-12.d):

$$\mathbf{A}^{-1} = \mathbf{D}^{-1} \cdot \mathbf{E}^{-1} = \begin{bmatrix} a_1^{-1} & 0 & 0 & \dots & 0 \\ 0 & a_2^{-1} & 0 & \ddots & \vdots \\ 0 & 0 & a_3^{-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & 0 & a_n^{-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & \dots & 0 \\ -1 & 1 & 0 & \ddots & \vdots \\ 0 & -1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} a_1^{-1} & 0 & 0 & \dots & 0 \\ -a_2^{-1} & a_2^{-1} & 0 & \ddots & \vdots \\ 0 & -a_3^{-1} & a_3^{-1} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & 0 & -a_n^{-1} & a_n^{-1} \end{bmatrix}$$

While the matrix $\mathbf{A}$ is dense, the inverse $\mathbf{A}^{-1}$ is sparse and banded with bandwidth $= 2$!

$$\mathbf{x} = \mathbf{D}^{-1}\mathbf{E}^{-1}\mathbf{b} \Longrightarrow x_j = \begin{cases} b_1/a_1 & j = 1 \\ \frac{b_j - b_{j-1}}{a_j} & j = 2, \dots, n \end{cases}$$

▲

(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 is consists of vectors $\mathbf{a}$ and $\mathbf{b}$.

HINT 1 for (3-12.e): Exploit the solution of (3-12.d). ⌟

SOLUTION of (3-12.e):

C++11-code 3.12.3: Solution of (3-12.e)

```
void solveA_fast(const VectorXd & a, const VectorXd & b, VectorXd & x) {
    // Size of b, which is the size of a
    int n = b.size();
    assert( n == a.size()
            && "Error: size mismatch!");
    x.resize(n);

    x(0) = b(0)/a(0);
    for(int i=1; i<n; i++) {
          x(i) = (b(i) - b(i-1))/a(i);
     }
}
```

Get it on GitLab (structuredLSE.cpp).

▲

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

HINT 1 for (3-12.f): The `main()` code to test your implementation is already provided in the template. ⌟

SOLUTION of (3-12.f):

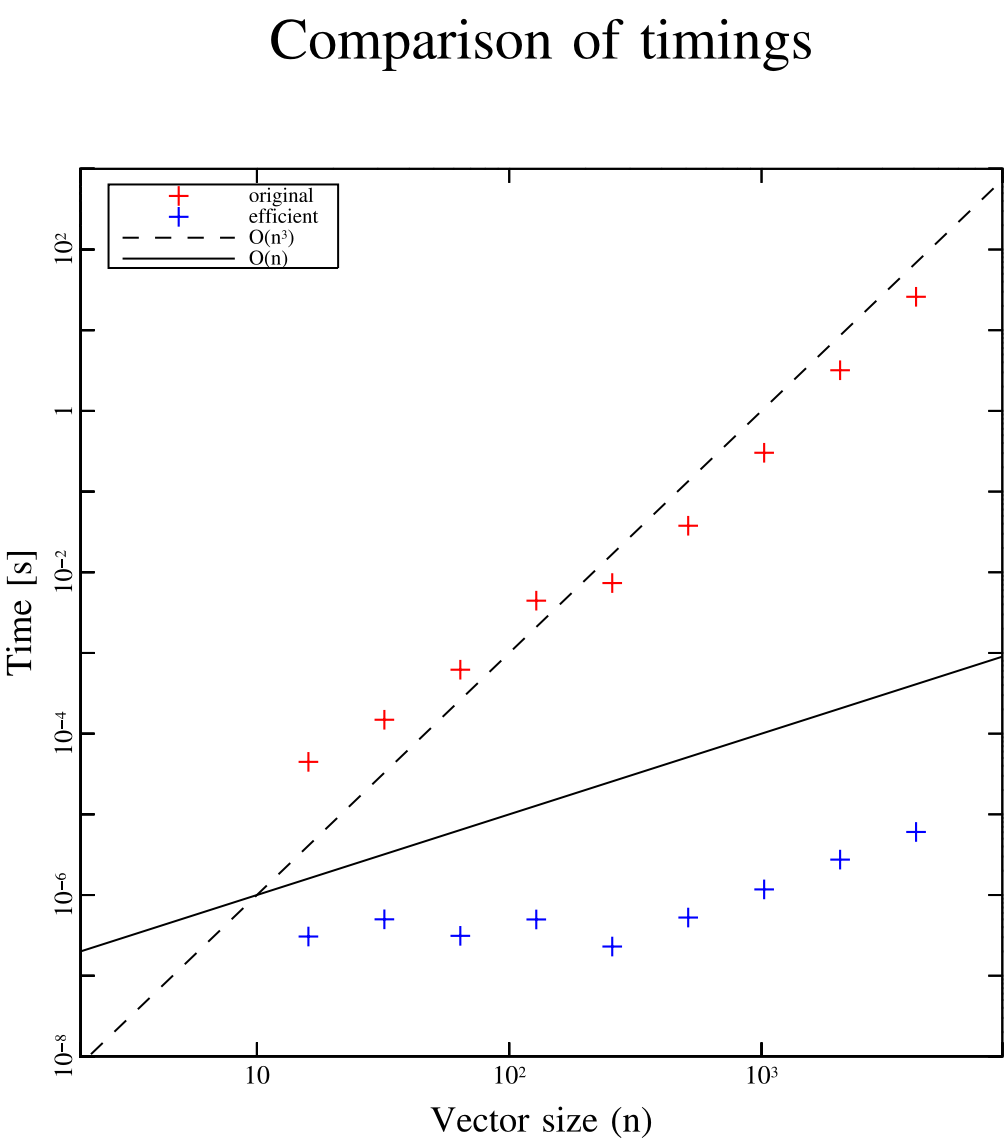


Fig. 8

▲

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**.

HINT 1 for (3-13.a): Use a **std::vector**<your_type> to store the collection of triplets: you can either define an auxiliary **struct** `Triplet` containing (i, j, v) (with the appropriate types) or use `Eigen::Triplet<double>`.

SOLUTION of (3-13.a):

C++11-code 3.13.1: struct `Triplet` for struct `TripletMatrix`

```
template <typename scalar>
struct Triplet {
  // Default constructor
  Triplet()
  : i(0), j(0), v(0) { }

  // Constructor taking indexes i and j and a value v
  Triplet(size_t i_, size_t j_, scalar v_)
```

```
    : i(i_), j(j_), v(v_) { }

    size_t i, j; // Row and col
    scalar v; // Value in (row,col)
};
```

Get it on GitLab (triplettoCRS.cpp).

C++11-code 3.13.2: Solution of (3-13.a)

```
template <typename scalar>
struct TripletMatrix {
  size_t rows, cols; // Sizes:  nrows and ncols
  std::vector<Triplet<scalar> > triplets;

  MatrixXd densify() const;
};
```

Get it on GitLab (triplettoCRS.cpp).

▲

(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**.

HINT 1 for (3-13.b): A CRS matrix is defined rather strictly: the arrays `values`, `col_ind` and `row_ptr` have to be *contiguous* in memory. This means that they should be stored each in a separate array data structure.

To store the data, you can use **std::vector**. Notice that, by using **std::vector**, we are not particularly strict about the "compressed" attribute of this format. In fact, this choice may waste some memory: the **std::vector** container has a padding at the end of its data that allows for `push_back` with amortized $O(1)$ complexity.

SOLUTION of (3-13.b):

C++11-code 3.13.3: Solution of (3-13.b)

```
template <typename scalar>
struct CRSMatrix {
  size_t rows, cols; // Sizes:  nrows and ncols
  std::vector<scalar> val;
  std::vector<size_t> col_ind;
  std::vector<size_t> row_ptr;

  MatrixXd densify() const;
};
```

Get it on GitLab (triplettoCRS.cpp).

▲

(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();
```

HINT 1 for (3-13.c): These methods can be helpful to debug your code, but they are not strictly necessary.

SOLUTION of (3-13.c):

C++11-code 3.13.4: `densify()` for struct `TripletMatrix`

```
template <class scalar>
MatrixXd TripletMatrix<scalar>::densify() const {
  MatrixXd M;
  // Initialization
  M = MatrixXd::Zero(rows, cols);

  for(auto it = triplets.begin(); it != triplets.end(); ++it) {
    M(it->i, it->j) += it->v;
  }

  return M;
}
```

Get it on GitLab (triplettoCRS.cpp).

C++11-code 3.13.5: `densify()` for struct `CRSMatrix`

```
template <typename scalar>
MatrixXd CRSMatrix<scalar>::densify() const {
  MatrixXd M;
// Initialization
  M = MatrixXd::Zero(rows, cols);
  std::vector<size_t> row_ptr_end = row_ptr;
  row_ptr_end.push_back(col_ind.size());

  for(size_t i=0; i<row_ptr.size(); ++i) {
      for(size_t j=row_ptr_end[i]; j<row_ptr_end[i+1]; ++j) {
          M(i, col_ind[j]) = val[j];
      }
  }

  return M;
}
```

Get it on GitLab (triplettoCRS.cpp).

▲

(3-13.d) ⚁ Write a C++ function that converts a matrix $\mathbf{T}$ in COO format to a matrix $\mathbf{C}$ in CRS format:

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

Try to be as efficient as possible.

HINT 1 for (3-13.d): The entries of the vectors of a matrix in CRS format must be ordered and without repetitions. The entries of the column indices vector must each correspond to one cell, starting from cell $(1,1),(1,2),\dots,(m,n)$ (skipping only zero cells). On the other hand, the triplets in the COO format may be in arbitrary ordering and with repetitions. Take care of these aspects in your function definition.

You may consider the idea of converting the vector of triplets to an intermediate CRS-like format. This format should then be easily convertible to the proper CRS format.

If you use an **std::vector** container, have a look at function **std::**`sort` or functions **std::**`lower_bound` and **std::**`insert` (both options lead to an equivalent function with different complexities).

You may also want to sort the vector of triplets depending on one element of the triplets. The standard C++ function **std::**`sort` provides a way to sort a vector of type **std::vector**`<your_type>` by defining the `your_type` member operator **bool** `your_type::`**operator**`<(`**const** `your_type& other)` **const**`;`. This operator returns true if `*`**this** entry is less than another entry in your particular ordering. Sorting is then performed according to this ordering. ⌟

SOLUTION of (3-13.d):

We present two possible solutions. They are both based on first converting the vector of triplets to a vector of vectors (rows of the original matrix), whose entries are **std::**`pairs` made of (col, val). For this purpose, **struct** `ColValPair` is also defined:

C++11-code 3.13.6: struct `ColValPair`

```
template <typename scalar>
struct ColValPair {
  ColValPair(size_t col_, scalar v_)
    : col(col_), v(v_) { }

  /* Comparison operator for std::sort and std::lower\_bound:
     Basic sorting operator < to use with std::functions (i.e. for ordering
        according to first component col).
     We keep the column values sorted, either by sorting after insertion of by
        sorted insertion.
     It returns true if this->col < other.col.
   */
  bool operator<(const ColValPair& other) const {
    return this->col < other.col;
  }

  size_t col; // Col index
  scalar v; // Scalar value at col
};
```

Get it on GitLab (triplettoCRS.cpp).

1. The first solution inserts the pairs (col, val) already sorted in this data structure: Code 3.13.6.

2. The other solution inserts all the triplets via `push_back`, then sorts each row and cumulatively sums all the duplicated column values: Code 3.13.8.

C++11-code 3.13.7: First solution of (3-13.d)

```
template <typename scalar>
void tripletToCRS_insertsort(const TripletMatrix<scalar>& T, CRSMatrix<scalar>& C) {
  // Initialization
  C.rows = T.rows;
  C.cols = T.cols;
  // Temporary "quasi-CRS" format for the conversion
  std::vector< std::vector< ColValPair<scalar> > > preCRS;
  preCRS.resize(C.rows);

  // Loop over all triplets
  for(auto triplet = T.triplets.begin(); triplet != T.triplets.end(); ++triplet) {

    // Store row (containing all ColValPair for this row) inside row_ptr
    std::vector<ColValPair<scalar> >& row = preCRS.at(triplet->i); // Notice the reference!

     // Create a ColValPair that must be inserted somewhere
     ColValPair<scalar> col_val(triplet->j, triplet->v);

    // Find the place (as iterator) where to insert col_val, i.e. the first position lb_it s.t col_val < row
    // WARNING: Costly call, which returns an iterator to an element (ColValPair) in row
    auto lb_it = std::lower_bound(row.begin(), row.end(), col_val);

    // If lower bound has already a col (and is not end()) with some value,
    // just add the value to the column, othervise insert it
    if(lb_it != row.end() && lb_it->col == triplet->j) {
       lb_it->v += triplet->v;
     } else {
       // WARNING: Costly call (loop over all rows in the worst-case scenario)
       row.insert(lb_it, col_val);
     }
  }

  // Convert 'preCRS' to proper CRS format (CRSMatrix)
  properCRS(preCRS, C);

}
```

Get it on GitLab (triplettoCRS.cpp).

C++11-code 3.13.8: Second solution of (3-13.d)

```
template <typename scalar>
void tripletToCRS_sortafter(const TripletMatrix<scalar>& T, CRSMatrix<scalar>& C) {
  // Initialization
  C.rows = T.rows;
  C.cols = T.cols;
  // Temporary "quasi-CRS" format for the conversion
  std::vector< std::vector< ColValPair<scalar> > > preCRS;
  preCRS.resize(C.rows);

```

```
  // Loop over all triplets and push them at the right place (cheap)
  for(auto triplet = T.triplets.begin(); triplet != T.triplets.end(); ++triplet)
      {
      ColValPair<scalar> cp(triplet->j, triplet->v);
      preCRS.at(triplet->i).push_back(cp);
  }
  // Loop over all rows, sort them (according to col) and sum
      duplicated values
  for(auto row_it = preCRS.begin(); row_it != preCRS.end(); ++row_it) {

    // WARNING: costly call in this case
    std::sort(row_it->begin(), row_it->end());

    // Sum duplicated cols
    // NOTE: iterating backwards should be faster
    for(auto col_it = row_it->begin(); col_it != row_it->end(); ++col_it) {

      // Check that next col is not at the end and that has same col
          index
      if((col_it + 1) != row_it->end() && (col_it + 1)->col == col_it->col) {

        // Last col with same value (starting from next col)
        auto last_col_it = col_it + 1;
        while(last_col_it != row_it->end() && last_col_it->col == col_it->col) {
          col_it->v += last_col_it->v;
          ++last_col_it;
        }

        // Remove range from next col to last col with same value
        // WARNING: std::vector keeps the data as c-array, so mildly
            costly call (from col_it to end(), at most M)
        row_it->erase(col_it+1, last_col_it);
      }
    }
  }

  // Convert 'preCRS' to proper CRS format (CRSMatrix)
  properCRS(preCRS, C);

}
```

Get it on GitLab (triplettoCRS.cpp).

The aforementioned intermediate data structure is then converted to the proper CRS format by function `properCRS`:

C++11-code 3.13.9: `properCRS`

```
template <typename scalar>
void properCRS(const std::vector< std::vector< ColValPair<scalar> > >& preCRS,
    CRSMatrix<scalar>& C) {

  C.row_ptr.push_back(0);
  for(auto & row : preCRS) {
    // If one whole row is empty,
    // the same index is stored in 'row_ptr' twice.
    C.row_ptr.push_back(C.row_ptr.back() + row.size());
    for(auto & col_val : row) {
      C.col_ind.push_back(col_val.col);
      C.val.push_back(    col_val.v);
    }
```

```
  }
  C.row_ptr.pop_back();
}
```

Get it on GitLab (triplettoCRS.cpp).

▲

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

HINT 1 for (3-13.e): If you use the C++ standard library functions, look at the documentation. There is written the complexity of basic container operations.

SOLUTION of (3-13.e):

We presented two alternative solutions for (3-13.d): pushing back the column/value pairs to an unsorted vector or inserting into a sorted vector in the right position.

Let k be the number of triplets.

1. The first method has k `push_back` amortized $O(1)$ operations and one **std**`::sort` of a vector of at most k entries (i.e. $O(k \log k)$ complexity).
2. The second method inserts k triplets using two $O(k)$ operation for a complexity of $O(k^2)$.

The first method has an apparent lower complexity. However, if there are many repeated triplets with the same index or if you add triplets to an already defined matrix, the second method could prove itself less expensive.

▲

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

HINT 1 for (3-13.f): To run your simulations, complete the `main()` code which is already provided in the template.

SOLUTION of (3-13.f):

Consider the complete `main()` code provided in the solution.

▲

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 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 of (3-14.a):

C++11-code 3.14.1: Solution of (3-14.a)

```
void CCS(const MatrixXd & A, VectorXd & val, VectorXd & row_ind, VectorXd &
    col_ptr)
{
  // Number of rows and columns
  int m = A.rows();
  int n = A.cols();
  //TODO (3-14.a): Implement a function that stores the matrix A
  // in the vectors (val,row_ind,col_ptr) in CCS format
  // Do not use Eigen methods for Sparse matrices.

  //START
  // Number of nonzero entries
  int nnz = 0;
  for(int i=0; i<m; ++i) { // Row iterator
    for(int j=0; j<n; ++j) { // Col iterator
      if(A(i,j) != 0) {
        ++nnz;
      }
    }
  }

```

```
  // Initialization
  val.resize(nnz);
  row_ind.resize(nnz);
  col_ptr.resize(n);

  // Store A in CCS format
  int index = 0;
  for(int j=0; j<n; ++j) { // Col iterator

    // Update 'col\_ptr'
    col_ptr(j) = index;

    for(int i=0; i<m; ++i) { // Row iterator
      if(A(i,j) != 0) {
        // Record the value to 'val'
        val(index) = A(i,j);
        // Record the row index to 'row\_ind'
        row_ind(index) = i;
        ++index;
      }
    }
  }
  // END
}
```

Get it on GitLab (sparseCCS.cpp).

▲

(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 of (3-14.b):

The algorithm needs to check whether each element $a_{i,j}$ of the $n \times n$ matrix is nonzero. The computational complexity is $O(n^2)$, given the double loop to build `val` and `row_ind`. Forming `col_ptr` is only $O(n)$ and does not therefore change the global complexity $O(n^2)$.

The complexity does not depend on the number nnz of nonzero elements in $\mathbf{A}$, since we have to iterate throughout all elements in any case.

▲

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

```
class EllpackMat {
public:
  EllpackMat(const Triplets &triplets, index_t m, index_t n);
  double operator()(index_t i, index_t j) const;
  void mvmult(const VectorXd &x, VectorXd &y) const;
  void mtvmult(const VectorXd &x, VectorXd &y) const;
private:
  std::vector<double> val; //< Vector containing values
  // corresponding to entries in 'col'
  std::vector<index_t> col; //< Vector containing column
  // indices of the entries in 'val'.
  // The position of a cell in 'val' and 'col'
  // is determined by its row number and original position in 'triplets'
  index_t maxcols; //< Number of non-empty columns
  index_t m, n;    //< Number of rows, number of columns
};
```

Get it on GitLab (ellpack.cpp).

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

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

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

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

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

  for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
    if (col[l] == j)
      return val[l];
  }
  return 0;
}
```

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.

HINT 1 for (3-15.a): An object of `Eigen::Triplets<double>` type is a structure representing a triplet (r,c,v), which are row, column and value of a cell in $\mathbf{A}$. It provides the methods `index_t row()`, `index_t col()` and **double** `value()`, which return r, c and v.

As first thing, you will have to determine the quantity `maxcols` based on `Eigen::Triplets<double> triplets` and the number of rows m. ⌟

SOLUTION of (3-15.a):

C++11-code 3.15.3: Solution of (3-15.a)

```
EllpackMat::EllpackMat(const Triplets &triplets, index_t m, index_t n)
    : m(m), n(n) {
  // TODO (3-15.a): implement the constructor for the class EllpackMat
  // START

  // Find maxcols, number of non-empty columns
  // It is assumed that all triplets uniquely define one entry
  std::vector<unsigned int> counters(m);
  // Fill counters with 0
  std::fill(counters.begin(), counters.end(), 0);
  // 'maxcols' starts with no entry for each row
  maxcols = 0;
  // Loop over each triplet and increment the corresponding counter, updating
  // maxcols if necessary
  for (const Triplet_new &tr : triplets) {
    if (++counters[tr.row()] > maxcols)
      maxcols = counters[tr.row()];
  }
  std::cout << "Maxcols: " << maxcols << std::endl;

  // Reserve space
  col.resize(m * maxcols, -1);
  val.resize(m * maxcols, 0);

  // Loop over each triplet and find a column where to put the value
  for (const Triplet_new &tr : triplets) {
    assert(0 <= tr.row() && tr.row() < m && 0 <= tr.col() && tr.col() < n &&
           "Index out of bounds!");

    index_t l;
    for (l = tr.row() * maxcols; l < (tr.row() + 1) * maxcols; ++l) {
      // Store the current triplet in the first
```

```
      // empty position between 'tr.row()*maxcols'
      // and '(tr.row()+1)*maxcols - 1'
      if (col[l] == -1) {
        col[l] = tr.col();
        val[l] = tr.value();
        break;
      }
    }
    assert(l < (tr.row() + 1) * maxcols &&
           "You did not reserve enough columns!");
  }
  // END
}
```

Get it on GitLab (ellpack.cpp).

▲

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

```
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 $O(\mathrm{nnz}(\mathbf{A}))$.

HINT 1 for (3-15.b): Think about the way the Ellpack format stores the matrix values and how matrix-vector multiplication works. E.g., to obtain y_i, you have to compute the scalar product between row i of $\mathbf{A}$ and vector $\mathbf{x}$, which is the sum of the products between each (nonzero) element of $\mathbf{A}_{i,:}$ and $\mathbf{x}$. Hence, you can do the following:

1. Iterate along the values of $\mathbf{A}_{i,:}$ in the way Ellpack stores them.
2. Multiply each of them by the corresponding element of $\mathbf{x}$, using **std::vector**`<index_t> col`.

SOLUTION of (3-15.b):

C++11-code 3.15.4: Solution of (3-15.b)

```
void EllpackMat::mvmult(const VectorXd &x, VectorXd &y) const {
  assert(x.size() == n && "Incompatible vector x size!");
  assert(y.size() == m && "Incompatible vector y size!");
  // TODO (3-15.b) : implement the multiplication A^T * x using the
  // class EllpackMat, with optimal complexity.

  // START
  for (index_t i = 0; i < m; ++i) {
    y(i) = 0;
    for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
      if (col[l] == -1)
        break;
      y(i) += x(col[l]) * val[l];
    }
  }
```

```
  // END
}
```

Get it on GitLab (ellpack.cpp).

▲

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

HINT 1 for (3-15.c): The solution is very similar to (3-15.b). Think about which vector among $\mathbf{A}_{i,:}$, $\mathbf{x}$ and $\mathbf{y}$ should be reordered using **std::vector**`<index_t> col` to obtain the correct result.

SOLUTION of (3-15.c):

C++11-code 3.15.5: Solution of (3-15.c)

```
void EllpackMat::mtvmult(const VectorXd &x, VectorXd &y) const {
  assert(x.size() == m && "Incompatible vector x size!");
  assert(y.size() == n && "Incompatible vector y size!");
  // TODO (3-15.c) : implement the multiplication A^T * x using the class
  // EllpackMat, with optimal complexity.

  // START
  y = VectorXd::Zero(n);
  for (index_t i = 0; i < m; ++i) {
    for (index_t l = i * maxcols; l < (i + 1) * maxcols; ++l) {
      if (col[l] == -1)
        break;
      y(col[l]) += x(i) * val[l];
    }
  }
  // END
}
```

Get it on GitLab (ellpack.cpp).

▲

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 $\mathbb{R}^{n,m}$:

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

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 of (3-16.a):

It is sufficient to check that, for any $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{n,m}$:

$$L(\mathbf{X} + \mathbf{Y}) = L(\mathbf{X}) + L(\mathbf{Y}) \quad , \quad L(\alpha \mathbf{X}) = \alpha L(\mathbf{X}), \alpha \in \mathbb{R} \, .$$

This is a straightforward computation.

▲

(3-16.b) ⚀ The space $\mathbb{R}^{n,m}$, as a vector-space, is isomorphic to $\mathbb{R}^{n \cdot m}$, via the mapping $\mathbf{X} \mapsto \mathrm{vec}(\mathbf{X})$, 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.

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

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

SOLUTION of (3-16.b):

The matrix $\mathbf{A}$ is given by:

$$\mathbf{A} = \begin{bmatrix} -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -4 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -4 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -4 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -4 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -4 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -4 \end{bmatrix}$$

▲

(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 $f : \mathbb{N}^2 \to \mathbb{R}$, 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:

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

SOLUTION of (3-16.c):

C++11-code 3.16.1: Evaluation function

```
void eval(MatrixXd &X, std::function<double(index_t, index_t)> f) {
  // TO DO (3-16.c): fill the matrix X with values from f.
  // Hint: do not forget the shift of indices in C++
  // START
  for (index_t i = 0; i < (index_t)X.rows(); ++i) {
    for (index_t j = 0; j < (index_t)X.cols(); ++j) {
      // Notice that we shift indices due to C++ indexing.
      X(i, j) = f(i + 1, j + 1);
    }
  }
  // END
}
```

Get it on GitLab (grid_function.cpp).

C++11-code 3.16.2: Definition of f

```
std::function<double(index_t, index_t)> define_f(int n, int m) {
  // TO DO (3-16.c): Define a lambda function f that takes as input
     n,m
  // and outputs a double
  // START
```

```
  auto f = [n, m](index_t i, index_t j) {
    if (i > n / 4 && i < n * 3 / 4 && j > m / 4 && j < m * 3 / 4)
      return 1.;
    else
      return 0.;
  };
  // END
  return f;
}
```

Get it on GitLab (grid_function.cpp).

▲

(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 `Xn`, `Xm`), builds and returns the sparse matrix $\mathbf{A}$ in CRS format. The matrix $\mathbf{A}$ can be built using an intermediate triplet/COO format.

HINT 1 for (3-16.d): To break down the task into smaller pieces, you may implement a function

```
inline index_t to_vector_index(Eigen::Index i, Eigen::Index j,
                               Eigen::Index Xn, Eigen::Index Xm);
```

which, given an index pair $(i,j) \in \mathbb{N}^2$ returns an index $I \in \mathbb{N}$ s.t. $(\mathbf{X})_{i,j} = (\mathrm{vec}(\mathbf{X}))_I$. The arguments `Xn`, `Xm` pass the size of $\mathbf{X} \in \mathbb{R}^{n,m}$. ⌟

SOLUTION of (3-16.d):

C++ code 3.16.3: Auxiliary function `to_vector_index()`

```
inline index_t to_vector_index(index_t i, index_t j, const shape_t &size) {
  // TO DO, hint in (3-16.d) : for a given pair of indices (i,j)
  // return the index of vec(X) corresponding to (X)_{i,j}
  // START
  return size[1] * i + j;
  // END
}
```

C++11-code 3.16.4: Build matrix $\mathbf{A}$

```
SparseMatrix<double> build_matrix(const Matrix3d &S, const shape_t &size) {
  // Will be used to store triplets
  std::vector<Triplet<double>> triplets;
  // The matrix A will have size N × N.
  // N = n*m
  const index_t N = size[0] * size[1];

  // Now, construct A.
```

```
  SparseMatrix<double> A(N, N);

  // TO DO: (3-16.d) store the matrix in COO format in the vector "triplets"
  // then convert triplets to CRS/CCS format using the functions
  // setFromTriplets() and makeCompressed() available in Eigen.
  // Hint: use the inline function to_vector_index to find quickly
  // the indices of each triplet.
  // Hint 2: to improve efficiency, you can allocate first a sufficient
  // number of triplets with the function triplets.reserve()

  // START

  // Used to preallocate size of vector "triplets":
  // predicted maximum number of triplets
  const index_t ntriplets =
      (size[0] - 2) * (size[1] - 2) * 3 *
          3                       // elements not at boundary:
                                  // those have 3*3 entries per element
      + 2 * 3 * (size[0] - 2); // top/bottom boundary rows (minus corners)
  +2 * 3 * (size[1] - 2);      // left/right columns (minus corners)
  +2 * 2 * 4;                  // corners

  // Reserve enough space (more efficient)
  triplets.reserve(ntriplets);

  // Loop over entire grid
  for (index_t i = 0; i < size[0]; ++i) {
    for (index_t j = 0; j < size[1]; ++j) {

      // Loop over the entire stencil matrix S.
      for (index_t k = 0; k < 3; ++k) {
        for (index_t l = 0; l < 3; ++l) {

          if (i + k - 1 >= 0 && i + k - 1 <= size[0] - 1 && j + l - 1 >= 0 &&
              j + l - 1 <= size[1] - 1) {
            // The index of vec(X) w.r.t
            // the indices (i,j) of X
            const index_t I = to_vector_index(i, j, size);
            // Index on vec(X) relative to index
            // (i+k-2,j+l-2)
            // NOTE: C++ indexing starts from 0, so we change the
            // offset.
            const index_t J = to_vector_index(i + k - 1, j + l - 1, size);
            // Add triplet of corresponding stencil matrix entry
            triplets.push_back(Triplet<double>(J, I, S(k, l)));
          }
        }
      }
    }
  }

  // Build from triplets and compress matrix.
  A.setFromTriplets(triplets.begin(), triplets.end());
  A.makeCompressed();

  // END
  return A;
}
```

Get it on GitLab (grid_function.cpp).

▲

(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. $\mathrm{vec}(\mathbf{Y}) := \mathbf{A}\,\mathrm{vec}(\mathbf{X})$. You can use objects of type `Eigen::Map` to "reshape" a matrix into a column vector.

SOLUTION of (3-16.e):

C++11-code 3.16.5: Multiplication with $\mathbf{A}$

```
void mult(const SparseMatrix<double> &A, const MatrixXd &X, MatrixXd &Y) {
  // Size check
  assert(A.rows() == X.rows() * X.cols() && A.cols() == X.rows() * X.cols() &&
         "Inconsistent size of A!");
  // Ensure correct size of Y.
  Y.resizeLike(X);
  // TO DO (3-16.e): compute vec(Y) = A*vec(X)
  // Hint: Use Eigen::Map to reshape X into vec(X).

  // START
  // The second "const" here is needed to "promise" X will not be modified.
  const Map<const VectorXd> x(X.data(), X.size());
  // Notice how "Map" can also be written onto.
  Map<VectorXd>(Y.data(), X.size()) = A * x;

  // END
}
```

Get it on GitLab (grid_function.cpp).

▲

(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. $\mathrm{vec}(\mathbf{Y}) := \mathbf{A}\,\mathrm{vec}(\mathbf{X})$. You can use objects of type `Eigen::Map` to "reshape" a matrix into a column vector. Objects of this type are read and write compatible.

SOLUTION of (3-16.f):

C++11-code 3.16.6: Solving the linear system

```
void solve(const SparseMatrix<double> &A, const MatrixXd &Y, MatrixXd &X) {
  // Size check
  assert(A.rows() == Y.rows() * Y.cols() && A.cols() == Y.rows() * Y.cols() &&
```

```
        "Inconsistent size of A!");
  // Ensure correct size of X.
  X.resizeLike(Y);
  // TO DO (3-16.f): compute X such that vec(Y) = A * vec(X) using a
  // sparse solver in Eigen.
  // START
  // Sparse LU decomposition object
  SparseLU<SparseMatrix<double>> solver;

  // Prepare LU factorization
  solver.analyzePattern(A);
  solver.factorize(A);

  // Check if factorization failed.
  if (solver.info() != Success) {
    std::cerr << "Fatal: LU decomposition of A failed!!!" << std::endl
              << "Is the matrix invertible?" << std::endl;
    return;
  }

  // Solve system after factorization.
  // Notice how you can assign to a Map object.
  Map<VectorXd>(X.data(), Y.size()) =
      solver.solve(Map<const VectorXd>(Y.data(), Y.size()));
  // END
}
```

Get it on GitLab (grid_function.cpp).

▲

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.

HINT 1 for (3-17.a): Think about simple binary matrices, only made of columns, rows or diagonals of ones (no complicated patterns). Do not focus on the size: consider only square matrices and take e.g. $n = 6$. Since we consider only binary matrices, we exclude the case of zeros in the product because of (unlikely) cancellations: zero cells c_{ij} in the product are only due to being $= \sum_{k=1}^{n} a_{ik} b_{kj}$, $\forall k\, a_{ik} = 0 \vee b_{kj} = 0$. ⌟

SOLUTION of (3-17.a):

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

We have that $\mathbf{AB}$ is dense, while $\mathbf{BA}$ is sparse:

$$\mathbf{AB} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \quad \mathbf{BA} = \begin{bmatrix} 6 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Of course, there are many other examples.

▲

(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 of (3-17.b):

C++11-code 3.17.1: Solution of (3-17.b)

```
TripVec Mat2COO(const MatrixXd &A)
{
    // Initialization
    int m = A.rows();
     int n = A.cols();
    TripVec triplets;

  //int k = 0;
     for(int i=0; i<m; ++i) { // Row iterator
          for(int j=0; j<n; ++j) { // Col iterator
               if(A(i,j) != 0) {
                    Trip triplet(i, j, A(i,j));
              //triplets[k] = triplet;
                 //++k;
                    triplets.push_back(triplet);
             }
          }
     }

     return triplets;
}
```

Get it on GitLab (matmatCOO.cpp).

▲

(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.

HINT 1 for (3-17.c): A cell $c_{i,j}$ of the matrix product $\mathbf{C} = \mathbf{AB}$ is $= \sum_{k=1}^{n} a_{ik} b_{kj}$. The output matrix C in COO format can be composed by multiple triplets.

SOLUTION of (3-17.c):

C++11-code 3.17.2: Solution of (3-17.c)

```
TripVec COOprod_naive(const TripVec &A, const TripVec &B)
{
    // Initialization
```

```
    TripVec C;

    for(auto const& a: A) {
        for(auto const& b: B) {
            if(a.col() == b.row()) {
                Trip triplet(a.row(), b.col(), a.value()*b.value());
                C.push_back(triplet);
            }
        }
    }

    return C;
}
```

Get it on GitLab (matmatCOO.cpp).

▲

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

HINT 1 for (3-17.d): Assume that there are no duplicates in the input triplet vectors of A and B.

SOLUTION of (3-17.d):

The double loop iterates throughout all the triplets of A and B. As the COO format only stores nonzero cells, the complexity is $O(\mathrm{nnz}(A) \cdot \mathrm{nnz}(B))$, assuming that there are no duplicates in the input triplet vectors. (More generally, the complexity would be $O(\mathrm{length}(triplets_A) \cdot \mathrm{length}(triplets_B))$.)

This is still better than the standard complexity $O(n^3)$ of matrix-matrix multiplications, as we do not compute the sum $\sum_{k=1}^{n} a_{ik} b_{kj}$ for any cell c_{ij} of the product matrix.

▲

(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)?

HINT 1 for (3-17.e): Is there a way to avoid the double loop of (3-17.c)?

Remember that $c_{i,j} = \sum_{k=1}^{n} a_{ik} b_{kj}$. Think about sorting the triplets of A along the columns and the triplets of B along the rows. For example, first there are the triplets corresponding to cells a_{1k}, $k = 1, \ldots, n$, then the triplets for a_{2k}, and so on.

Once you have accomplished this, you can iterate only along the intersection between the nonempty columns of A and the nonempty rows of B. For each value of k, you can run a small double loop only between the triplets of A and B which already have k in common and obtain all possible triplets $a_{ik} b_{kj}$.

SOLUTION of (3-17.e):

C++11-code 3.17.3: Solution of (3-17.e)

```
TripVec COOprod_effic(TripVec &A, TripVec &B)
{
    // Initialization
    TripVec C;

    std::sort(A.begin(), A.end(), [](const Trip& a1, const Trip& a2) {return
        a1.col() < a2.col();});
    std::sort(B.begin(), B.end(), [](const Trip& b1, const Trip& b2) {return
        b1.row() < b2.row();});

    size_t i_A = 0, i_B = 0;
    std::set<int> intersect;
    while(i_A != A.size() && i_B != B.size())
    {
        if(A[i_A].col() < B[i_B].row()) ++i_A;
        else if(B[i_B].row() < A[i_A].col()) ++i_B;
        else {
          intersect.insert(A[i_A].col()); //
              intersect.insert(B[i_B].row());
          ++i_A; ++i_B;
        }
    }

    TripVec::iterator A_idx = A.begin();
    TripVec::iterator B_idx = B.begin();
    for(auto i: intersect) {

        A_idx = std::find_if(A_idx, A.end(),
                    [i](const Trip& a){return a.col() == i;});
        B_idx = std::find_if(B_idx, B.end(),
                    [i](const Trip& b){return b.row() == i;});

        TripVec::iterator A_it;
        TripVec::iterator B_it;
        for(A_it=A_idx; A_it!=A.end(); ++A_it) {
            if(A_it->col() != i) {
                break;
            } else {
                for(B_it=B_idx; B_it!=B.end(); ++B_it) {
                    if(B_it->row() != i) {
                        break;
                    } else {
                        Trip triplet(A_it->row(), B_it->col(),
                            (A_it->value())*(B_it->value()));
                        C.push_back(triplet);
                    }
                }
            }
        }
        A_idx = A_it;
        B_idx = B_it;
    }

    return C;
}
```

Get it on GitLab (matmatCOO.cpp).

▲

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

HINT 1 for (3-17.f): Assume that there are no duplicates in the input triplet vectors of A and B.

The efficient implementation is composed of three steps. What is the asymptotic complexity of:

1. sorting triplet vectors A and B?
2. finding the interception of common indices k?
3. iterating along the intersection to obtain all possible triplets $a_{ik}b_{kj}$?

Do some of these steps have a negligible complexity? ⌟

SOLUTION of (3-17.f):

We assume that there are no duplicates in the input triplet vectors.

1. Sorting triplet vectors A and B $\implies$ $O(n \cdot \log(n))$.
2. Finding the interception of common indices k $\implies$ $O(n)$ (having already sorted both A and B).
3. Iterating along the intersection to obtain all possible triplets $a_{ik}b_{kj}$ $\implies$ This is proportional to the product between k_{common}, the number of common indices k for matrices A and B, and $\tilde{n}^2$, which we define as the square of the average number of nonzero entries per column of A / row of B.

   k_{common} is certainly greater than the number of nonzero entries in matrix AB. However, $\tilde{n}$ depends on the sparsity pattern of the input matrices. In any case, as O should be an upper bound, the complexity is $O\left(k_{common}n^2\right)$.

The total asymptotic complexity is therefore $O\left(k_{common}n^2\right)$. The complexities of the first two step are in fact smaller than the third.

▲

(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.

HINT 1 for (3-17.g): The `main()` code to run your simulations is already provided in the template. ⌟

SOLUTION of (3-17.g):

Comparison of timings – sparse matrices

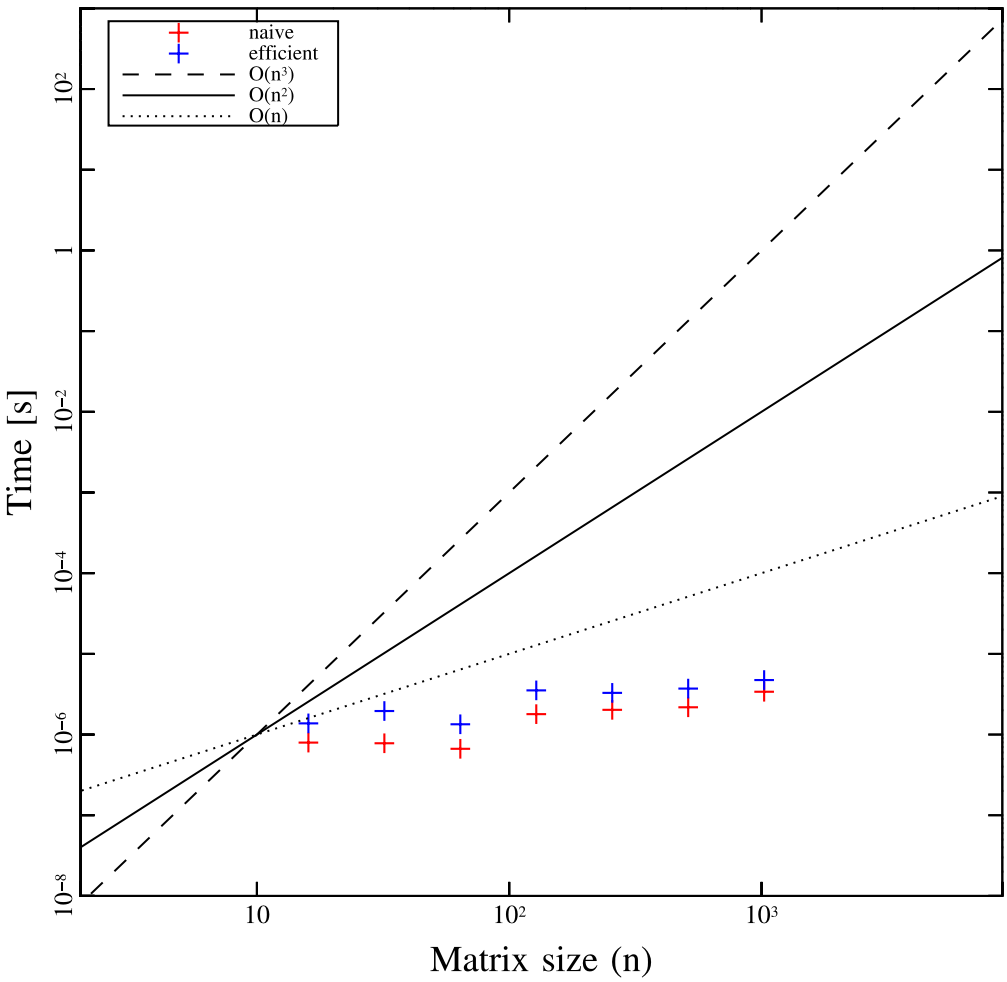


Fig. 9

Comparison of timings – sparse/dense matrices

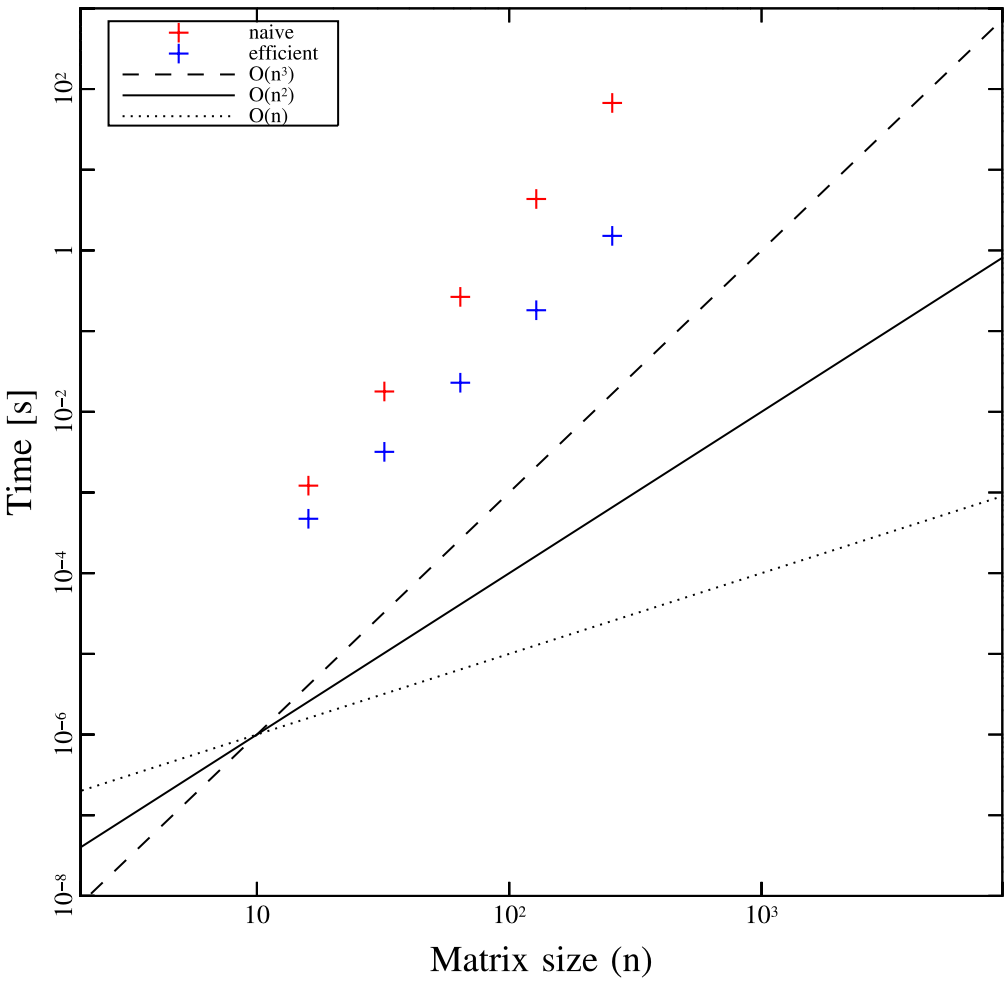


Fig. 10

Both the coded algorithms are faster than the standard matrix-matrix multiplications, as we do not compute the sum $\sum_{k=1}^{n} a_{ik} b_{kj}$ for any cell c_{ij} of the product matrix. The naive algorithm outperforms the efficient algorithm in the special case of sparse matrices: the product of sparse matrices can be dense (and therefore the number of common indices k_{common} can be high), as seen in (3-17.a). However, the efficient algorithm is generally better when considering any kind of matrix.

▲

End Problem 3-17

Problem 3-18: A special Sylvester Equation

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

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

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" $\mathrm{vec}(\mathbf{X}) \in \mathbb{R}^{n^2}$ of a matrix $\mathbf{X} \in \mathbb{R}^{n,n}$ from [Lecture → Rem. 1.2.3.4] and jot down $\mathrm{vec}(\mathbf{X})$ for

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

SOLUTION of (3-18.b):

The vectorized matrix is obtained by stacking the columns of the matrix on top of each other:

$$\mathrm{vec}(\mathbf{X}) = [x_{11}, x_{21}, x_{12}, x_{22}]^\top \in \mathbb{R}^4 .$$

▲

(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

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

that is, express the entries of $\mathbf{C}$ in terms of the entries $(\mathbf{S})_{i,j}$ and $(\mathbf{T})_{i,j}$ of $\mathbf{S}$ and $\mathbf{T}$.

SOLUTION of (3-18.c):

We obtain the k-ht column of $\mathbf{C}$ as

$$\mathbf{C}_{:,k} = \mathrm{vec}(\mathbf{S}\,\mathrm{vec}^{-1}(\mathbf{e}_k)\mathbf{T}) ,$$

where $\mathbf{e}_k \in \mathbb{R}^4$ is the k-th unit vector, $k = 1, 2, 3, 4$. For instance

$$\mathbf{C}_{:,1} = \mathrm{vec}\left(\begin{bmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \right) = \mathrm{vec}\left(\begin{bmatrix} s_{11} & 0 \\ s_{21} & 0 \end{bmatrix} \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \right)$$

$$= \mathrm{vec}\left(\begin{bmatrix} s_{11}t_{11} & s_{11}t_{12} \\ s_{21}t_{11} & s_{21}t_{12} \end{bmatrix} \right) = \begin{bmatrix} s_{11}t_{11} \\ s_{21}t_{11} \\ s_{11}t_{12} \\ s_{21}t_{12} \end{bmatrix} .$$

Thus we find

$$\mathbf{C} = \begin{bmatrix} s_{11}t_{11} & s_{12}t_{12} & s_{11}t_{21} & s_{12}t_{21} \\ s_{21}t_{11} & s_{22}t_{11} & s_{21}t_{21} & s_{22}t_{21} \\ s_{11}t_{12} & s_{12}t_{12} & s_{11}t_{22} & s_{12}t_{22} \\ s_{21}t_{12} & s_{22}t_{12} & s_{21}t_{22} & s_{22}t_{22} \end{bmatrix} = \mathbf{T}^\top \otimes \mathbf{S} \,.$$

▲

(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

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

HINT 1 for (3-18.d): Read **[LAN70]**. ⌟

SOLUTION of (3-18.d):

By the rules of matrix multiplication for $\mathbf{S}, \mathbf{X}, \mathbf{T} \in \mathbb{R}^{n,n}$:

$$(\mathbf{AXT})_{i,j} = \sum_{\ell=1}^{n}\sum_{\nu=1}^{n} s_{i,\ell}x_{\ell,\nu}t_{\nu,j} \,, \tag{3.18.1}$$

where we abbreviated $s_{i,\ell} = (\mathbf{S})_{i,\ell}$, $x_{\ell,\nu} := (\mathbf{X})_{\ell,nu}$, $t_{\nu,1} := (\mathbf{T})_{\nu,1}$.

We can extract a column of $\mathbf{C}$ by multiplying with a unit vector. Hence, we consider $\mathbf{E}_{k,m}$, the all-zero matrix except for an entry $= 1$ at position (k,m). From (3.18.1) we find

$$(\mathbf{SE}_{k,m}\mathbf{T})_{i,j} = s_{i,k}t_{m,j} \,. \tag{3.18.2}$$

Write $P(i,j) \in \{1,\dots,n^2\}$ for the **p**osition of the matrix entry $(\mathbf{Z})_{i,j}$ in the vector $\operatorname{vec}(Z)$, $\mathbf{Z} \in \mathbb{R}^{n,n}$, $i,j \in \{1,\dots,n\}$. Hence, from (3.18.2) we conclude

$$(\mathbf{C})_{P(i,j),P(k,m)} = s_{i,k}t_{m,j} = \left(\mathbf{T}^\top \otimes \mathbf{S}\right)_{P(i,j),P(k,m)} \quad \Leftrightarrow \quad \boxed{\mathbf{C} = \mathbf{T}^\top \otimes \mathbf{S}} \,. \tag{3.18.3}$$

▲

Throughout the remainder of this problem let $\mathbf{A} \in \mathbb{R}^{n,n}$, $n \in \mathbb{N}$, stand for a *symmetric, positive definite* (s.p.d.) matrix, whose rows and columns contain at most five non-zero entries. Hence, for large n the matrix $\mathbf{A}$ can be considered *sparse* in the sense of [Lecture → Notion 2.7.0.1].

Then consider the special Sylvester equation:

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

and the equivalent matrix equation:

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

(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 $\mathrm{vec}(\mathbf{X})$. 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 of (3-18.e):

According to (3.18.3) the coefficient matrices $\mathbf{C} \in \mathbb{R}^{n^2,n^2}$ are ($\mathbf{I} \in \mathbb{R}^{n,n}$ the identity matrix)

$$\text{For (3.18.4):} \quad \mathbf{C}_4 = \mathbf{A}^{-1} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{A} \,, \tag{3.18.6}$$

$$\text{For (3.18.5):} \quad \mathbf{C}_5 = \mathbf{I} \otimes \mathbf{I} + \mathbf{A} \otimes \mathbf{A} \,. \tag{3.18.7}$$

Here we have made use of the symmetry of $\mathbf{A}$.

▲

(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).

HINT 1 for (3-18.f):

Inverses of regular sparse matrices can be densely populated!

$$\begin{bmatrix} 1 & -1 & 0 & \dots & \dots & 0 \\ 0 & 1 & -1 & 0 & & \vdots \\ \vdots & \ddots & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & 0 \\ \vdots & & & 0 & 1 & -1 \\ 0 & \dots & & \dots & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 1 & 1 & 1 & \dots & & \dots & 1 \\ 0 & 1 & 1 & \dots & & \dots & 1 \\ \vdots & \ddots & \ddots & & & & \vdots \\ & & \ddots & \ddots & & & \vdots \\ & & & \ddots & \ddots & & \vdots \\ & & & & \ddots & \ddots & \vdots \\ 0 & \dots & & & & 0 & 1 \end{bmatrix}$$

SOLUTION of (3-18.f):

By the properties of $\mathbf{A}$ we know

$$\mathrm{nnz}(\mathbf{A}) \leq 5n \,.$$

However, as explained in the hint, $\mathrm{nnz}(\mathbf{A}^{-1}) = n^2$ is possible. Moreover, for the Kronecker product we have the simple estimate

$$\mathrm{nnz}(\mathbf{S} \otimes \mathbf{T}) \leq \mathrm{nnz}(\mathbf{S}) \cdot \mathrm{nnz}(\mathbf{T}) \,. \tag{3.18.8}$$

Combining all these estimates gives us

$$\mathrm{nnz}(\mathbf{A}^{-1} \otimes \mathbf{I} + \mathbf{I} \otimes \mathbf{A}) \leq n^3 + 5n^2 \,,$$
$$\mathrm{nnz}(\mathbf{I} \otimes \mathbf{I} + \mathbf{A} \otimes \mathbf{A}) \leq 26n^2 \,.$$

Hence, (3.18.5) features a much sparser coefficient matrix than (3.18.4).

▲

(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

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

HINT 1 for (3-18.g): In linear algebra we learned that symmetric matrices can always be diagonalized. ⌟

HINT 2 for (3-18.g): Remember [Lecture → Lemma 1.1.2.7]. ⌟

SOLUTION of (3-18.g):

By the very definition of the Kronecker product, both matrices $\mathbf{C}_4$ and $\mathbf{C}_5$ from (3.18.6) and (3.18.7), respectively, are symmetric. Since both $\mathbf{A}$ and $\mathbf{A}^{-1}$ are s.p.d. all of their eigenvalues are positive. Hence, by the formula of Thm. 3.18.9 this also holds for $\mathbf{C}_4$ and $\mathbf{C}_5$. Thus, by the third criterion from [Lecture → Lemma 1.1.2.7], these matrices are s.p.d.

▲

(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 of (3-18.h):

If $\mathbf{A}$ is s.p.d. and diagonal with diagonal entries $d_\ell = (\mathbf{A})_{\ell,\ell} > 0$, then (3.18.4) becomes

$$(d_i + d_j^{-1})(\mathbf{X})_{i,j} = \delta_{i,j} \ , \quad i,j \in \{1,\dots,n\} \ .$$

Hence, $\mathbf{X}$ will also be diagonal

$$(\mathbf{X})_{i,j} = \begin{cases} \frac{d_i}{d_i^2+1} & \text{, if } i = j \ , \\ 0 & \text{else.} \end{cases}$$

Implementation is straightforward, but care must be taken to initialize the result matrix efficiently. Reserve space for the single non-zero entry per row/column, see [Lecture → Code 2.7.2.1].

C++ code 3.18.10: Solving a "diagonal" Sylvester equation

```
template <typename Vector>
SparseMatrix<double> solveDiagSylvesterEq(const Vector &diagA) {
  int n = diagA.size();
  SparseMatrix<double> X(n, n);
  // TO DO: Fill in the entries of X.
  // Don't forget to use makeCompressed().
  // START
  // As X is a diagonal matrix, we reserve space
  // for one non-zero entry per column.
  X.reserve(VectorXi::Constant(n, 1));
  for (int j = 0; j < n; j++) {
    // Set diagonal entry X_{j,j} = d_j/(d_j^2+1)
    X.insert(j, j) = diagA[j] / (diagA[j] * diagA[j] + 1.0);
  }
  X.makeCompressed();
  // END
  return X;
}
```

▲

(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 of (3-18.i):

The comments in Code 3.18.11 explain the details of the algorithm.

C++ code 3.18.11: Computing $\mathbf{M} \otimes \mathbf{M}$ for a sparse matrix $\mathbf{M}$

```
SparseMatrix<double> sparseKron(const SparseMatrix<double> &M) {
  int n = M.rows();
  assert(n == M.cols() && "Matrix M must be square.");

  // JM is an array of size n+1, with
  // JM[j] = number of non-zero entries in all columns 0,..., (j-1) of M.
  const int *JM = M.outerIndexPtr();

  // Define nnzColwiseM(j) = nnz( M.col(j) )
  VectorXd nnzColwiseM(n);
  for (int j = 0; j < n; j++) {
    nnzColwiseM(j) = JM[j + 1] - JM[j];
  }

  // B will be the Kronecker product of M with itself.
  SparseMatrix<double> B(n * n, n * n);

  // TO DO: Fill in the entries of B.
  // Hint: You can use nnzColwiseM when reserving space for B.
  // Use M.valuePtr() and M.innerIndexPtr() to define arrays similar
  // to JM. Use those arrays to access the non-zero entries of M.
  // START

  VectorXd nnzColwiseB;
  // We have that nnz(B.col(j*n+l)) = nnz(M.col(j)) * nnz(M.col(l)).
  MatrixXd tmp = nnzColwiseM * nnzColwiseM.transpose(); // O(n^2)
  nnzColwiseB = MatrixXd::Map(tmp.data(), n * n, 1);

  // Reserve sufficient space.
  B.reserve(nnzColwiseB);

  // We have that B.block(i*n,j*n,n,n) = M(i,j) * M,
  // i.e. B( i*n + k, j*n + l ) = M(i,j)*M(k,l).
  // Hence, we loop over i,j,k,l.

  // Row indices of non-zero entries (size = nnz):
  const int *IM = M.innerIndexPtr();
  // Values of non-zero entries (size = nnz):
  const double *valM = M.valuePtr();

  // Assuming nnz in each column is bounded by a constant, the below is O(n^2).
  // Loop over the column index j:
  for (int j = 0; j < n; j++) {
    // Loop over the non-zero entries in column j:
    for (int s = JM[j]; s < JM[j + 1]; s++) {
      int i = IM[s];          // Row index of non-zero entry #s.
      double Mij = valM[s]; // M(i,j) = Mij

      // We now compute Mij * M.
      // Loop over the column index l:
      for (int l = 0; l < n; l++) {
        // Loop over the non-zero entries in column l:
        for (int t = JM[l]; t < JM[l + 1]; t++) {
          int k = IM[t];          // Row index of non-zero entry #t.
          double Mkl = valM[t]; // M(k,l) = Mkl
          // Insert entry (k,l) of block (i,j).
          B.insert(i * n + k, j * n + l) = Mij * Mkl;
        }
      }
```

```
    }
  }
  B.makeCompressed();
  // END
  return B;
}
```

▲

(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 of (3-18.j):

C++ code 3.18.12: Solving Sylvester equation (3.18.4)**.**

```
MatrixXd solveSpecialSylvesterEq(const SparseMatrix<double> &A) {
  const int n = A.rows();
  assert(n == A.cols() && "Matrix A must be square.");
  MatrixXd X(n, n);

  // TO DO: Solve the equation X*A^{-1} + A*X = I .
  // START
  // The equation is equivalent to X + A*X*A = A,
  // which has the system matrix C = kron(I,I) + kron(A,A).
  // Define left hand side of C*Vec(X) = b.
  SparseMatrix<double> C(n * n, n * n);
  C.setIdentity();
  // May involve expensive copying of data?
  C += sparseKron(A);

  // Define right hand side.
  // Do not use a "sparse vector"!
  VectorXd b(n * n);
  b.setZero();
  const int *JA = A.outerIndexPtr();
  const int *IA = A.innerIndexPtr();
  const double *valA = A.valuePtr();
  // Form b by stacking the columns of A.
  // Iterate over columns of A:
  for (int j = 0; j < n; j++) {
    // Loop over the non-zero entries in column j:
    for (int s = JA[j]; s < JA[j + 1]; s++) {
      const int i = IA[s];     // Row index of non-zero entry #s.
      b[j * n + i] = valA[s]; // b(j*n+i) = Aij
    }
  }
  // Call sparse direct solver.
  SparseLU<SparseMatrix<double>> solver;
  // Since C is s.p.d., we could also use
```

```
  // SimplicialLLT<SparseMatrix<double>> solver;
  solver.compute(C);
  // x = Vec(X)
  VectorXd x;
  x = solver.solve(b);
  X = MatrixXd::Map(x.data(), n, n);
  // END
  return X;
}
```

▲

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):

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

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

SOLUTION of (4-1.a):

The resulting system of nonlinear equations is:

$$y_i = f(t_i) = \alpha e^{\beta t_i}\ , \quad i = 1, \dots, n\ .$$

The non-linearity is due to the occurrence of parameter β as exponent of e.

▲

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

HINT 1 for (4-1.b): Use the inverse function of the exponential.

SOLUTION of (4-1.b):

Define $g(t) := \log f(t)$. This is well defined, as $f(t) > 0 \; \forall t$.

With $a := \log \alpha$ we can then obtain:

$$g(t) = \log f(t) = \log \alpha + \log e^{\beta t} = a + \beta t$$

Moreover, we impose $b_i := \log y_i$.

The new problem is therefore expressed by the following overdetermined linear system:

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \quad \text{with } \mathbf{A} = \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_n \end{bmatrix}, \mathbf{x} = \begin{bmatrix} a \\ \beta \end{bmatrix}, \mathbf{b} = \begin{bmatrix} \log y_1 \\ \log y_2 \\ \vdots \\ \log y_n \end{bmatrix}$$

▲

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

HINT 1 for (4-1.c): Apply the definition of normal equations given in [Lecture → Section 3.1.2].

SOLUTION of (4-1.c):

We determine the parameters of the normal equations $\mathbf{A}^\top \mathbf{A}\mathbf{x} = \mathbf{A}^\top \mathbf{b}$. We obtain:

$$\mathbf{A}^\top \mathbf{A} = \begin{bmatrix} n & \sum_{i=1}^n t_i \\ \sum_{i=1}^n t_i & \sum_{i=1}^n t_i^2 \end{bmatrix}, \mathbf{A}^\top \mathbf{b} = \begin{bmatrix} \sum_{i=1}^n b_i \\ \sum_{i=1}^n t_i b_i \end{bmatrix} \tag{4.1.2}$$

Once we have found the least squares solution $\mathbf{x} = \begin{bmatrix} a \\ \beta \end{bmatrix}$, we just need to replace $\alpha = e^a$ and β in function f.

▲

(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 α and β in a vector of length 2.

SOLUTION of (4-1.d):

The solution is a straightforward adaptation of [Lecture → Code 3.2.0.1]. We start with initializing the matrix $\mathbf{A}$. Alternatively, we could just initialize $\mathbf{A}^\top \mathbf{A} \in \mathbb{R}^{2,2}$ as given in (4.1.2).

C++11-code 4.1.3: Solution of (4-1.d)

```
VectorXd linReg(const VectorXd &t, const VectorXd &y) {
  assert( t.size() == y.size() && "t and y must have same size" );
```

```
  // Coefficient matrix of overdetermined linear system.
  MatrixXd A(t.size(),2);
  A.col(0).setOnes();
  A.col(1) = t;

  // Normal equation
  MatrixXd lhs = A.transpose() * A; // Left-hand side
  VectorXd rhs = A.transpose() * y; // Right-hand side

  // Intercept and slope.
  VectorXd x = lhs.lu().solve(rhs);
  return x;
}
```

▲

(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 α and β, given the nonlinear relationship (4.1.1) between $\mathbf{t}$ and $\mathbf{f}$:

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

HINT 1 for (4-1.e): Use the function `linReg` from (4-1.d). Pay attention to give the correct input $\mathbf{f}$ to `linReg` and to extract the right parameters α and β from its output. ⌟

SOLUTION of (4-1.e):

C++ code 4.1.4: Solution of (4-1.e)

```
VectorXd expFit(const VectorXd &t, const VectorXd &y) {
  assert( t.size() == y.size() && "t and y must have same size");

  // Transform to a linear equation
  // b := log(y) = x(0) + x(1)*t
  VectorXd b = y.array().log().matrix();
  // Solve, and transform back to the nonlinear equation
  // y = exp(x(0)) * exp(x(1)*t) = alpha*exp(beta*t)
  VectorXd x = linReg(t, b);
  x(0) = exp(x(0));

  return x;
}
```

▲

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 overdetermined linear system of equations $\mathbf{Ax} = \mathbf{b}$ we minimize the residual norm $\|\mathbf{Ax} - \mathbf{b}\|_2$ w.r.t. $\mathbf{x}$. However, we also face a linear least squares problem when minimizing the residual norm w.r.t. the entries of $\mathbf{A}$. This is the unusual situation considered in this problem.

This is an exercise in converting a problem statement into a proper linear least squares problem in the form of an overdetermined linear system of equations as in [Lecture → Section 3.0.1] and then solving it through the normal equations, see [Lecture → Section 3.2].

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

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

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

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

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

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

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 $m, k \in \mathbb{N}$.

HINT 1 for (4-2.a): Compute the symbolic product $\mathbf{T}_{\alpha,\beta}\mathbf{z}$ using $\mathbf{z} = [z_1, z_2, \dots, z_n]^\top$. Express the resulting vector as the product of a matrix and the vector $\begin{bmatrix}\alpha\\ \beta\end{bmatrix}$. ⌟

SOLUTION of (4-2.a):

Clearly we set $\mathbf{x} = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$ and $\mathbf{b} := \mathbf{c}$ The vector $\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c}$ whose norm has to be minimized in Eq. (4.2.1) can be written as:

$$\mathbf{T}_{\alpha,\beta}\mathbf{z} - \mathbf{c} = \begin{bmatrix} \alpha z_1 + \beta z_2 \\ \alpha z_2 + \beta(z_1 + z_3) \\ \dots \\ \alpha z_{n-1} + \beta(z_{n-2} + z_n) \\ \alpha z_n + \beta z_{n-1} \end{bmatrix} - \mathbf{c} = \begin{bmatrix} z_1 & z_2 \\ z_2 & z_1 + z_3 \\ \vdots & \vdots \\ z_{n-1} & z_{n-2} + z_n \\ z_n & z_{n-1} \end{bmatrix} \begin{bmatrix}\alpha\\ \beta\end{bmatrix} - \mathbf{c} =: \mathbf{Ax} - \mathbf{b} \tag{4.2.4}$$

In this form, we have moved the unknown α and β into vector $\mathbf{x}$, while the data $\mathbf{z}$ and $\mathbf{c}$ are moved into matrix $\mathbf{A}$ and the right-hand side vector $\mathbf{b} \equiv \mathbf{c}$ (which is not affected by the transformation). The number of unknown k is equal to 2 and the number of equations m is equal to n.

▲

(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 α^* and β^* according to Eq. (4.2.1) from data vectors $\mathtt{z}$ and $\mathtt{c}$ (i.e. $\mathbf{z}$ and $\mathbf{c}$ from Eq. (4.2.1)). Use the *method of normal equations*.

HINT 1 for (4-2.b): Similarly to Problem 4-1, $\mathbf{Ax} = \mathbf{b}$ is an overdetermined problem: the number of equations, i.e. the number of rows of $\mathbf{A}$, n, is greater than the number of unknowns, i.e. the number of columns of $\mathbf{A}$, 2. We must therefore solve the normal equations $\mathbf{A}^\top\mathbf{Ax} = \mathbf{A}^\top\mathbf{b}$ with EIGEN. ⌟

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

SOLUTION of (4-2.b):

C++ code 4.2.5: Solution of (4-2.b)

```
VectorXd lsqEst(const VectorXd &z, const VectorXd &c) {
  assert( z.size() == c.size() && "z and c must have same size" );

  // Reformulating T_{α,β}z = c as Ax = b with x = (α,β)^T,
  // we have b = c, and A defined as follows.
  int n = z.size();
  MatrixXd A(n,2);
  // Initialize all entries of A:
  A.col(0) = z;
  A(0,1) = 0;
  A.col(1).tail(n-1) = z.head(n-1);
  // Add remaining terms:
  A.col(1).head(n-1) += z.tail(n-1);

  // Normal equation
  MatrixXd lhs = A.transpose() * A; // Left-hand side
  VectorXd rhs = A.transpose() * c; // Right-hand side

  // Least squares estimate of (alpha,β).
  VectorXd x = lhs.lu().solve(rhs);
  return x;
}
```

Get it on GitLab (tridiagleastsquares.cpp).

▲

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

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

of the functions

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

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

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

(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 γ_i by related to the data (4.3.2) and (4.3.3) by means of solving the *normal equations*.

SOLUTION of (4-3.a):

We have to recast (4.3.3) as a least squares problem arising from an overdetermined linear system of equations $\mathbf{A}\mathbf{x} = \mathbf{b}$. We find

$$\sum_{i=1}^{10} (f(t_i) - f_i)^2 = \sum_{i=1}^{10} \left(\sum_{j=1}^{4} \gamma_j \phi_j(t_i) - f_i \right)^2 ,$$

which means

$$\mathbf{A} = \begin{bmatrix} \phi_1(t_1) & \phi_2(t_1) & \phi_3(t_1) & \phi_4(t_1) \\ \vdots & & & \vdots \\ \phi_1(t_{10}) & \phi_2(t_{10}) & \phi_3(t_{10}) & \phi_4(t_{10}) \end{bmatrix} \in \mathbb{R}^{10,4} \,.$$

First we look at the helper function `make_A`. This only creates the matrix $\mathbf{A}$ by applying the four model functions to the vector t and building up the matrix column wise.

C++-code 4.3.4: Creating the matrix A

```
MatrixXd make_A(const VectorXd &b) {
  size_t n = b.size();
  MatrixXd A(n, 4);
  //TO DO (4-3.a) Build the matrix A
  // Hint: evaluate the functions \phi_j at the time points defined in b
  //START
  for (size_t i = 0; i < n; i++) {
    double t = b(i);
    A(i, 0) = 1.0 / t;
    A(i, 1) = 1.0 / (t*t);
    A(i, 2) = std::exp(-(t-1));
    A(i, 3) = std::exp(-2*(t-1));
  }
  //END
  return A;
}
```

Get it on GitLab (linear_data_fit.cpp).

In `data_fit_normal`, we use `make_A` to create the matrix. For the normal equation

$$\mathbf{A}^T\mathbf{A}\mathbf{x} = \mathbf{A}^T\mathbf{b}\,,$$

we calculate $\mathbf{A}^T\mathbf{A}$ and $\mathbf{A}^T\mathbf{b}$ and then solve the system.

C++-code 4.3.5: Solving the Normal Equations

```
VectorXd data_fit_normal(const VectorXd &t, const VectorXd &b) {
  //TO DO (4-3.a) Solve normal equations to find the coefficients of the
  // linear fitting
  VectorXd gamma(4);
  //START
  MatrixXd A = make_A(t);
  MatrixXd At = A.transpose();
  MatrixXd AtA = At * A;
  VectorXd Atb = At * b;
  gamma =  AtA.ldlt().solve(Atb);
  //END
  return gamma;
}
```

Get it on GitLab (linear_data_fit.cpp).

▲

(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 of (4-3.b):

In `data_fit_qr`, the first few steps are the same as in in `data_fit_normal`. But instead of solving the normal equation, we compute the QR-decomposition and use this to solve the problem, see also [Lecture → Code 3.3.4.2].

C++-code 4.3.6: Solving with QR-Decomposition

```
VectorXd data_fit_qr(const VectorXd &t, const VectorXd &b) {
  //TO DO (4-3.b) Find the coefficients for the linear
  // fitting by means of the QR decomposition of A
  VectorXd gamma(4);
  //START
  MatrixXd A = make_A(t);
  gamma = A.colPivHouseholderQr().solve(b);
  //END
  return gamma;
}
```

Get it on GitLab (linear_data_fit.cpp).

▲

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

Write a C++ function

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

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

SOLUTION of (4-3.c):

We use the following code for this and the next part:

C++-code 4.3.7: `fit_plot()`: computing values of $f(t)$ using the estimated parameters

```
VectorXd fit_plot(const VectorXd &gamma, const VectorXd &t) {
  //TO DO (4-3.c): Define the data for the first plot:
  // evaluate the function f at the grid defined by t

  VectorXd y;
  //START
  MatrixXd A = make_A(t);
  y = A*gamma;
  //END
  return y;
}
```

▲

(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 γ_j-coefficients passed in the `gamma` argument.

SOLUTION of (4-3.d):

C++-code 4.3.8: `error_plot()`: computing the deviation from measured data

```
VectorXd error_plot( const VectorXd &gamma ) {
  //TO DO (4-3.c): Compute the vector of squared errors of your
  // fit at the provided sample points
  VectorXd err;
  //START
  VectorXd t = VectorXd::LinSpaced(10, 0.1, 1.0);
  VectorXd f(10);
  f << 100. , 34. , 17. , 12. , 9. , 6. , 5. , 4. , 4. , 2.;

  err = fit_plot(gamma,t)-f;
  err = err.cwiseProduct(err);
  //END
  return err;
}
```

▲

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. 3.3.3.20].

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

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

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

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

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

HINT 1 for (4-4.a): There is a direct proof just employing the rules for multiplying matrices. First establish the property of $\mathbf{Q} = \mathbf{R}^{-1}\mathbf{A}$ and than that of $\mathbf{R}$. ⌟

SOLUTION of (4-4.a):

We give a direct proof based on the rules of matrix multiplication, assuming that $\mathbf{A}$ is invertible.

First, we rely the observation made in [Lecture → Lemma 1.3.1.9] that the inverse of a regular upper triangular matrix is again upper triangular and apply it to $\mathbf{R}$: $\mathbf{R}^{-1}$ is upper triangular. Thus, writing $a_{i,j} := (\mathbf{A})_{i,j}$ ($\Rightarrow a_{i,j} = 0$, if $|i-j| > 1$) and $s_{i,j} := \left(\mathbf{R}^{-1}\right)_{i,j}$ ($\Rightarrow s_{i,j} = 0$ for $i > j$), we find

$$\begin{aligned}
(\mathbf{Q})_{j,k} &= \sum_{0\le l\le n} a_{j,l}s_{l,k} = \sum_{l\in\{j-1,j,j+1\}} a_{j,l}s_{l,k} \quad \text{, because } \mathbf{A} \text{ is tridiagonal,}\\
&= \sum_{l\in\{j-1,j,j+1\},\, l\le k} a_{j,l}s_{l,k} \quad \text{, because } \mathbf{R}^{-1} \text{ is upper triangular,}\\
&= 0 \quad \text{, if } j > k+1\,, \quad j,k\in\{1,\dots,n\}\,.
\end{aligned}$$

Next, $\mathbf{R} = \mathbf{Q}^\top\mathbf{A}$ is upper triangular by definition and ($q_{i,j} := (\mathbf{Q})_{i,g}$, $q_{i,j} = 0$, if $i > j+1$)

$$\begin{aligned}
(\mathbf{R})_{j,k} &= \sum_{0\le l\le n} \left(\mathbf{Q}^\top\right)_{j,l} a_{l,k} = \sum_{0\le l\le n} q_{l,j}a_{l,k}\\
&= \sum_{l\in\{k-1,k,k+1\}} q_{l,j}a_{l,k} \quad \text{, because } \mathbf{A} \text{ is tridiagonal,}\\
&= \sum_{l\in\{k-1,k,k+1\},\, l\le j+1} q_{l,j}a_{l,k} \quad \text{, because of the pattern of } \mathbf{Q}\text{,}\\
&= 0 \quad \text{, if } j+1 < k-1 \quad \text{, i.e., if } k > j+2\,, \quad j,k\in\{1,\dots,n\}\,.
\end{aligned}$$

Alternatively, the proof that for the upper triangular factor $\mathbf{R}$ only the zeroth, first, and second upper diagonal are occupied can be constructive and employ $n-1$ successive Givens rotations, see [Lecture → Rem. 3.3.3.21]. Applying these Givens rotations to an identity matrix reveals the structure of $\mathbf{Q}^\top$.

▲

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

```
class TriDiagonalQR {
public:
  explicit TriDiagonalQR(const TriDiagonalMatrix &A);

  template <typename VecType>
  Eigen::VectorXd applyQT(const VecType &x) const;
  template <typename VecType>
  Eigen::VectorXd solve(const VecType &b) const;
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

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

that is, $$\mathbf{R} = \begin{bmatrix} x_1 & y_1 & z_1 & & & & \\ & x_2 & y_2 & z_2 & & & \\ & & x_3 & y_3 & z_3 & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & x_{n-2} & y_{n-2} & z_{n-2} \\ & & & & & x_{n-1} & y_{n-1} \\ & & & & & & x_n \end{bmatrix} \quad \Rightarrow \quad \mathbf{B} = \begin{bmatrix} x_1 & y_1 & z_1 \\ \vdots & \vdots & \vdots \\ x_{n-2} & y_{n-2} & z_{n-2} \\ x_{n-1} & y_{n-1} & 0 \\ x_n & 0 & 0 \end{bmatrix}.$$

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 ρ 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 (ρ, γ, σ), 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

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

SOLUTION of (4-4.b):

The first part of the implementation amounts to copying the code of [Lecture → Code 3.3.3.16]. Then use the formulas from [Lecture → Rem. 3.3.3.20] to define the code ρ depending on the entries of the rotation matrix:

$$\mathbf{G} = \begin{bmatrix} \gamma & \sigma \\ -\sigma & \gamma \end{bmatrix}, \quad \gamma^2 + \sigma^2 = 1 \quad \Rightarrow \quad \blacktriangleright \quad \rho := \begin{cases} \operatorname{sign}(\sigma) & \text{, if } \gamma = 0\,, \\ \frac{1}{2}\operatorname{sign}(\gamma)\sigma & \text{, if } |\sigma| < |\gamma|\,, \\ 2\operatorname{sign}(\sigma)/\gamma & \text{, if } |\sigma| \geq |\gamma|\,. \end{cases}$$

C++ code 4.4.3: Computing end econding a Givens rotation

```
std::tuple<double, double, double> compGivensRotation(Eigen::Vector2d a) {
  double rho, gamma, sigma;
  // TO DO (4-4.b): Compute the parameters of a rotation that when
  // applied to a results in the vector (||a||,0)^T or (-||a||,0)^T.
  // Hint: You can use the function sign() above when determining rho
  // from gamma and sigma.
  // START
  if (a(1) != 0.0) {
    double t;
```

```
    // Separate cases to avoid cancellation/overflow
    if (std::abs(a(1)) > std::abs(a(0))) {
      t = -a(0) / a(1);
      sigma = 1.0 / std::sqrt(1.0 + t * t);
      gamma = sigma * t;
    } else {            // Then abs(a(0)) >= abs(a(1)) > 0
      t = -a(1) / a(0); // a(0) != 0
      gamma = 1.0 / std::sqrt(1.0 + t * t);
      sigma = gamma * t;
    }
  } else {
    // No rotation.
    gamma = 1.0;
    sigma = 0.0;
  }
  // Determine rho
  if (gamma == 0.0)
    rho = sign(sigma);
  else if (std::abs(gamma) > std::abs(sigma))
    rho = 0.5 * sign(gamma) * sigma;
  else
    rho = 2.0 * sign(sigma) / gamma;
  // END
  return std::make_tuple(rho, gamma, sigma);
}
```

The function `sign()` just returns the sign of a number.

▲

(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 `A`.

HINT 1 for (4-4.c): First copy the data from `A` to `B` and then perform in-situ (in-place) transformations on `B`. ⌟

HINT 2 for (4-4.c): An implementation, which uses a different data structure, however, can be found in the first part of [Lecture → Code 3.3.4.5]. ⌟

SOLUTION of (4-4.c):

We employ $n-1$ Givens rotations targeting the rows

$$(1,2) \to (2,3) \to \cdots \to (n-1,n)\,.$$

The challenge is to figure out, which entries of $\mathbf{B}$ are affected by which Givens rotations.

Outline of the transformation:

$$\begin{bmatrix} * & * & 0 & 0 & 0 & 0 & 0 & 0 \\ * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{12}} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \end{bmatrix} \xrightarrow{\mathbf{G}_{23}\cdots\mathbf{G}_{n-1,n}} \begin{bmatrix} * & * & * & 0 & 0 & 0 & 0 & 0 \\ 0 & * & * & * & 0 & 0 & 0 & 0 \\ 0 & 0 & * & * & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & * & * & 0 & 0 \\ 0 & 0 & 0 & 0 & * & * & * & 0 \\ 0 & 0 & 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & * & * \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & * \end{bmatrix}$$

$*$ ≙ entry set to zero by Givens rotation, $*$ ≙ new non-zero entry

C++ code 4.4.4: Constructor of TriDiagonalQR

```
TriDiagonalQR::TriDiagonalQR(const TriDiagonalMatrix &A) : n(A.n) {
  rho = Eigen::VectorXd::Zero(n - 1);
  B = Eigen::MatrixXd::Zero(n, 3);
  B.col(0) = A.d;
  B.col(1).tail(n - 1) = A.l;
  B.col(2).head(n - 1) = A.u;
  // Currently we have for k=0,...,n-1:
  // B(k,0) = A(k,k), B(k,1) = A(k,k-1), B(k,2) = A(k,k+1).

  // A bit clumsy, but less error prone than juggling indices
  Eigen::MatrixXd Ablock(2, 3);
  double gamma, sigma;
  for (int k = 0; k < (n - 1); k++) {
    // The block of the current A that will be affected by the next rotation.
    Ablock << B(k, 0), B(k, 2), 0, B(k + 1, 1), B(k + 1, 0), B(k + 1, 2);
    // TO DO (4-4.c): Find the k-th Givens rotation and update rho and B.
    // START
    std::tuple<double, double, double> params =
        compGivensRotation(Ablock.col(0));
    rho[k] = std::get<0>(params); // Store rotation parameter
    gamma = std::get<1>(params);
    sigma = std::get<2>(params);

    // Update B
    Eigen::Matrix2d G;
    G << gamma, -sigma, sigma, gamma;
    Ablock = G * Ablock; // Rotate to eliminate Ablock(1,0)
    B.row(k) = Ablock.row(0);
    B(k + 1, 0) = Ablock(1, 1);
    B(k + 1, 2) = Ablock(1, 2);
    // END
  }
}
```

▲

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

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

SOLUTION of (4-4.d):

The following code reconstructs the rotation matrix from ρ.

C++ code 4.4.5: Decoding compact storage of Givens matrices

```
inline Eigen::Matrix2d Givens(double rho) {
  double gamma, sigma;
  if (std::abs(rho) < 1.0) {
    sigma = 2.0 * rho;
    gamma = std::sqrt(1.0 - sigma * sigma);
  } else if (std::abs(rho) > 1.0) {
    gamma = 2.0 / rho;
    sigma = std::sqrt(1 - gamma * gamma);
  } else {
    gamma = 0;
    sigma = rho;
  }
  Eigen::Matrix2d G;
  G << gamma, sigma, -sigma, gamma;
  return G;
}
```

This code applies the $n-1$ rotations encoded in `rho` to consecutive entries of the vector $\mathbf{x}$.

C++ code 4.4.6: Method `applyQT()` of TriDiagonalQR

```
template <typename VecType>
Eigen::VectorXd TriDiagonalQR::applyQT(const VecType &x) const {
  assert(n == x.size() &&
         "Input must have size equal to the number of rows in Q.");
  Eigen::VectorXd y(n);
  // TO DO (4-4.d): Fill in the entries of y.
  // START
  for (int k = 0; k < n; k++) {
    y[k] = x[k];
  }
  Eigen::Matrix2d G;
  for (int k = 0; k < (n - 1); k++) {
    G = Givens(rho[k]).transpose();
    y.segment(k, 2) = G * y.segment(k, 2);
  }
  // END
  return y;
}
```

▲

(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 $\mathbf{A}\mathbf{x} = \mathbf{b}$, $\mathbf{A} \in \mathbb{R}^{n,n}$ 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.

HINT 1 for (4-4.e): You may get some inspiration from the second part of [Lecture → Code 3.3.4.5].

SOLUTION of (4-4.e):

- First step: $\mathbf{c} = \mathbf{Q}^\top \mathbf{b}$ by `applyQT()`
- Second step: back-substitution for a tridiagonal matrix

C++ code 4.4.7: Method `solve` of TriDiagonalQR

```
template <typename VecType>
Eigen::VectorXd TriDiagonalQR::solve(const VecType &b) const {
  assert(n == b.size() &&
         "Input must have size equal to the number of rows in Q.");
  Eigen::VectorXd x(n);
  // TO DO (4-4.e): Solve the linear system A*x = b if A is invertible,
  // otherwise throw an std::runtime_error.
  // START
  if ((B.col(0).array().abs() < 1E-10).any()) {
    throw std::runtime_error("A is not invertible!");
  }
  // Multiply by Q^T on both sides of Q*R*x=b:
  // R*x = Q^T*b = c
  Eigen::VectorXd c = applyQT(b);

  // Back-substitution
  int k = n - 1;
  x(k) = c(k) / B(k, 0);
  k--;
  x(k) = (c(k) - B(k, 1) * x(k + 1)) / B(k, 0);
  k--;
  while (k >= 0) {
    x(k) = (c(k) - B(k, 1) * x(k + 1) - B(k, 2) * x(k + 2)) / B(k, 0);
    k--;
  }
  // END
  return x;
}
```

▲

(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 $n \to \infty$?

SOLUTION of (4-4.f):

All functions/methods enjoy the asymptotic complexity $O(n)$ for $n \to \infty$ as they all involve a single loop.

▲

(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**.

HINT 1 for (4-4.g): Take into account [Lecture → Rem. 2.5.0.10] for the sake of efficient implementation.

SOLUTION of (4-4.g):

C++ code 4.4.8: Implementation og `invit()` for TriDiagonalMartrix

```
template <>
unsigned int invit<TriDiagonalMatrix>(const TriDiagonalMatrix &A,
                                      Eigen::VectorXd &x, double TOL,
                                      unsigned int maxit) {
  int it = 0;
  // TO DO (4-4.g): Specialize (efficiently) the template function invit()
  // for the TriDiagonalMatrix class.
  // START
  x.normalize();
  Eigen::VectorXd x_old(x.size());
  TriDiagonalQR Aqr(A);
  do {
    x_old = x;
    x = Aqr.solve(x_old);
    x.normalize();
  } while (((x - x_old).norm() > TOL) && (it++ < maxit));
  // END
  return it;
}
```

▲

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

$$\mathbf{x}^* = \operatorname*{argmin}_{\|\mathbf{x}\|_2=1}\{\|\mathbf{A}\mathbf{x}\|_2\} \tag{4.5.1}$$

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

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

Show that the following holds:

$$\|\mathbf{A}\mathbf{v}_n\|_2 = \sigma_n := \boldsymbol{\Sigma}_{n,n}. \tag{4.5.3}$$

HINT 1 for (4-5.a): For your proof, exploit the properties of orthogonal matrices and define $\mathbf{y} := \mathbf{V}^\top\mathbf{x}$. Also remember that, by convention, the diagonal matrix $\boldsymbol{\Sigma}$ of a singular value decomposition presents the diagonal entries sorted in decreasing order: $\boldsymbol{\Sigma}_{1,1} \geq \boldsymbol{\Sigma}_{2,2} \cdots \geq \boldsymbol{\Sigma}_{n,n}$. ⌟

SOLUTION of (4-5.a):

See also [Lecture → Section 3.4.4.1]. For any orthogonal matrix $\mathbf{V}$ and any vector $\mathbf{x}$, the following holds:

$$\|\mathbf{V}\mathbf{x}\|^2 = (\mathbf{V}\mathbf{x})^\top\mathbf{V}\mathbf{x} = \mathbf{x}^\top\mathbf{V}^\top\mathbf{V}\mathbf{x} = \mathbf{x}^\top\mathbf{x} = \|\mathbf{x}\|^2. \tag{4.5.4}$$

Here, we used that $\mathbf{V}$ is orthogonal, which implies that $\mathbf{V}^\top\mathbf{V} = \mathbf{I}$. (The same holds for $\mathbf{V}^\top$ instead of $\mathbf{V}$).

Define $\mathbf{y} := \mathbf{V}^\top\mathbf{x}$. We have:

$$\|\mathbf{A}\mathbf{x}\| = \|\mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top\mathbf{x}\| = \|\mathbf{U}\boldsymbol{\Sigma}\mathbf{y}\| = \|\boldsymbol{\Sigma}\mathbf{y}\| \tag{4.5.5}$$

The first equality uses the singular value decomposition of $\mathbf{A}$. The second equality uses the definition $\mathbf{y} := \mathbf{V}^\top\mathbf{x}$. The last equality holds due to $\mathbf{U}$ being orthogonal.

Now, we show that:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{v}_n\|_2 = \sigma_n. \tag{4.5.6}$$

Firstly, we prove "$\geq$". We have the following:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\| = \min_{\|\mathbf{y}\|=1} \|\boldsymbol{\Sigma}\mathbf{y}\| \tag{4.5.7}$$

$$= \min_{\|\mathbf{y}\|=1} \sqrt{\sigma_1^2 y_1^2 + \ldots + \sigma_n^2 y_n^2} \tag{4.5.8}$$

$$\geq \min_{\|\mathbf{y}\|=1} \sqrt{\sigma_n^2 y_1^2 + \ldots + \sigma_n^2 y_n^2} \tag{4.5.9}$$

$$= \sigma_n \min_{\|\mathbf{y}\|=1} \sqrt{y_1^2 + \ldots + y_n^2} \tag{4.5.10}$$

$$= \sigma_n \min_{\|\mathbf{y}\|=1} \|\mathbf{y}\| = \sigma_n. \tag{4.5.11}$$

Here we first use the previously obtained equation and then compute $\mathbf{\Sigma y}$ using the fact that $\mathbf{\Sigma}$ is a diagonal matrix (when computing the norm).

Secondly, we prove "$\leq$". We can bound the minimum by:

$$\min_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\| = \min_{\|\mathbf{y}\|=1} \|\mathbf{\Sigma y}\| \leq \|\mathbf{\Sigma e}_n\| = \sigma_n$$

We have therefore proven the second equality (4.5.6), i.e. that $\min_{\|\mathbf{x}\|=1} \|\mathbf{Ax}\| = \sigma_n$, and $\mathbf{e}_n = \operatorname{argmin}_{\|\mathbf{y}\|=1} \|\mathbf{\Sigma y}\|$.

Let us reconstruct $\mathbf{x}$ from $\mathbf{y} = \mathbf{V}^\top \mathbf{x}$. Since $\mathbf{V}$ is orthogonal, we have that:

$$\mathbf{e}_n = \mathbf{y} = \mathbf{V}^\top \mathbf{x} \quad \Longrightarrow \quad \mathbf{x} = \mathbf{Vy} = \mathbf{Ve}_n = \mathbf{V}_{:,n} =: \mathbf{v}_n. \tag{4.5.12}$$

▲

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 $\mathbf{A}$:

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

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

$$\mathbf{B} := \operatorname*{argmin}_{\mathbf{X} \in \mathcal{P}(\mathbf{A})} \|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F \,, \tag{4.6.2}$$

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 $\mathbf{B}$.

HINT 1 for (4-6.a): Remember that $\|\mathbf{M}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{M}\mathbf{e}_i\|_2^2$, where the $\mathbf{e}_i$ are the Cartesian unit vectors and the norm is the standard Euclidean norm. ⌟

SOLUTION of (4-6.a):

Let $\mathbf{x}_i, \mathbf{b}_i, \mathbf{a}_i$ denote the i-th columns of the matrices $\mathbf{X}, \mathbf{B}, \mathbf{A}$ in (4.6.2), respectively. Notice, that the values in the i-th column of $\mathbf{I} - \mathbf{A}\mathbf{X}$ depend only on the values in the vector $\mathbf{x}_i$. Hence, in order to minimize the Frobenius norm $\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2$ one needs to (independently) minimize each of the terms $\|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2$:

$$\|\mathbf{I} - \mathbf{A}\mathbf{X}\|_F^2 = \sum_{i=1}^{n} \|\mathbf{I}\mathbf{e}_i - \mathbf{A}\mathbf{X}\mathbf{e}_i\|_2^2 = \sum_{i=1}^{n} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2^2. \tag{4.6.3}$$

The corresponding minimization problems read

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathcal{P}(\mathbf{a}_i)} \|\mathbf{e}_i - \mathbf{A}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \dots, N, \tag{4.6.4}$$

where for a vector $\mathbf{a}$ we define $\mathcal{P}(\mathbf{a}) := \{\mathbf{x} \in \mathbb{R}^N : (\mathbf{a})_i = 0 \ \Rightarrow \ (\mathbf{x})_i = 0\}$.

▲

(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.

HINT 1 for (4-6.b): For an efficient implementation, you can exploit the internal CRS data structure underlying `Eigen::SparseMatrix<double>`. Use

- `SparseMatrix<double>::valuePtr()`,
- `SparseMatrix<double>::innerIndexPtr()` and
- `SparseMatrix<double>::outerIndexPtr()`, which

give the raw CCS format, in which the sparse matrix is stored, see [Lecture → § 2.7.1.4].

Moreover, build the output matrix $\mathbf{B}$ in EIGEN using the intermediate triplet format (`std::vector<Triplet<double>>`), see [Lecture → Section 2.7.2], before converting it to `SparseMatrix<double>`. Use the method `setFromTriplets()`. ⌟

SOLUTION of (4-6.b):

The formula (4.6.4) is not yet in the canonical form. Fix i. Define $m_i := nnz(\mathbf{a}_i)$. Notice that $\mathcal{P}(\mathbf{a}_i) \simeq \mathbb{R}^{m_i}$ via the mapping $\mathbf{x} \mapsto [x_{j_1}, \dots, x_{j_{m_i}}] =: \tilde{\mathbf{x}}$, where j_k, $k = 1, \dots, m_i$ are the indices s.t. $a_{j_k} \neq 0$.

The product $\mathbf{Ax}$ can be expressed, equivalently, with $\mathbf{C}\tilde{\mathbf{x}}$, where $\mathbf{C} \in \mathbb{R}^{N \times m_i}$ is defined as the matrix $\mathbf{A}$, without the columns i, s.t. $a_i = 0$:

$$\mathbf{C} := [\mathbf{A}_{j_1}, \dots, \mathbf{A}_{j_{m_i}}] \,.$$

We obtain:

$$\mathbf{b}_i := \operatorname*{argmin}_{\mathbf{x}_i \in \mathbb{R}^{m_i}} \|\mathbf{e}_i - \mathbf{C}\mathbf{x}_i\|_2, \quad \text{for} \quad i = 1, \dots, N.$$

This canonical form can be solved with the standard normal formula.

See code in Equation 4.6.5.

C++11-code 4.6.5: Computation of $\mathbf{B}$.

```
SparseMatrix<double> spai(SparseMatrix<double> & A) {
    // Size check
    assert(A.rows() == A.cols() &&
    "Matrix must be square!");
    unsigned int N = A.rows();

    // Needed to make sure ...Ptr functions return
    // arrays specified in CRS format
    A.makeCompressed();

    // Obtain pointers to data of A
    double* valPtr = A.valuePtr();
    index_t* innPtr = A.innerIndexPtr();
    index_t* outPtr = A.outerIndexPtr();

    // Create vector for triplets of B and reserve enough space
```

```
    std::vector<Triplet<double>> triplets;
    triplets.reserve(A.nonZeros());

    // Loop over each column of A
    for(unsigned int i = 0; i < N; ++i) {
        // Number of non zeros in current column of A
        index_t nnz_i = outPtr[i+1] - outPtr[i];
        // If column is empty:  skip column (matrix is not invertible)
        if(nnz_i == 0) continue;

        // Temporarily create a (small, dense) matrix on which normal formula
        // will be applied.  We project the space P(A)
        // onto P(a_i)
        SparseMatrix<double> C(N, nnz_i);
        std::vector<Triplet<double>> C_triplets;
        C_triplets.reserve(nnz_i*nnz_i);

        // Need to build matrix C.  To this end we remove all columns
        // from A for which a_i = 0.
        // To this end:  loop over all non-zero entries of the i-th column
        // This loop has length n
        for(unsigned int k = outPtr[i]; k < outPtr[i+1]; ++k) {
            // Row of this non-zero entry
            index_t row_k = innPtr[k];
            // Number of non-zero entries for row_k-th column
            index_t nnz_k = outPtr[row_k+1] - outPtr[row_k];
            // Loop over all non-zeros of row_kth-column
            // This loop has length complexity n
            for(unsigned int l = 0; l < nnz_k; ++l) {
                unsigned int innIdx = outPtr[row_k] + l;
                C_triplets.emplace_back(Triplet<double>(innPtr[innIdx], k - outPtr[i], valPtr[innIdx]));
            }
        }
        C.setFromTriplets(C_triplets.begin(), C_triplets.end());
        C.makeCompressed();
        SparseMatrix<double> S = C.transpose() * C;
        MatrixXd M = MatrixXd(S);
        VectorXd xt = C.row(i).transpose();

        // Compute C^T C and solve normal equation.
        // Complexity of product:
        // Complexity of solve:
        // Size of b is at most n.
        VectorXd b = M.partialPivLu().solve(xt);
        // Loop as length n.
        for(unsigned int k = 0; k < b.size(); ++k) {
            triplets.emplace_back(Triplet<double>(innPtr[outPtr[i] + k], i, b(k)));
        }
    }

    // Build and return SPAI preconditioner
    SparseMatrix<double> B = SparseMatrix<double>(N,N);
    B.setFromTriplets(triplets.begin(), triplets.end());
    B.makeCompressed();
    return B;
}
```

Get it on GitLab (spai.cpp).

▲

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

SOLUTION of (4-6.c):

In the body of the `for` loop (of size N) the matrix $\mathbf{C}$ is assembled with two for loops of length at most n, then the matrix $\mathbf{C}^\top\mathbf{C}$ is assembled with complexity $O(n^3)$ (note: $\mathbf{C}$ is sparse, with only n non-zeros per column), and then the corresponding linear system of size $n \times n$ is solved (complexity $O(n^3)$). Hence, the resulting total complexity is dominated by $O(Nn^3)$.

▲

(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 $\mathbf{A}\mathbf{x} = \mathbf{b}$ 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 $\mathbf{b} = [1, 1, \dots, 1]^\top$. Write a C++ function

```
std::vector<std:pair<unsigned int, unsigned int>>
testSPAIPrecCG(unsigned int L);
```

that returns the number of CG iterations (with and without the SPAI preconditioner) versus the system size $N = n^2$ for $n = 2^{1,\dots,L}$.

SOLUTION of (4-6.d):

▲

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 $n = 8$.

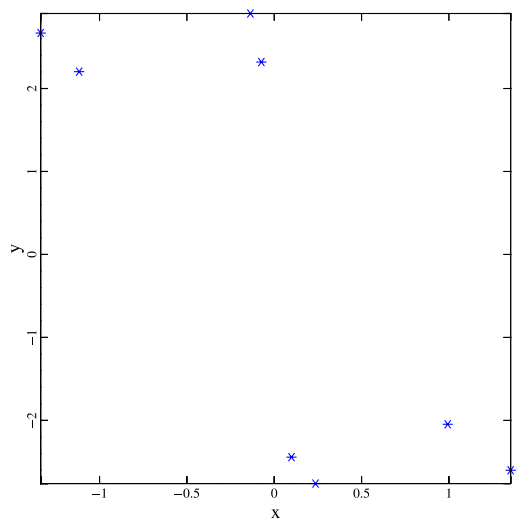


Fig. 11 Example of input points $\mathbf{p}^i$.

The shape of the sign can be represented by a 2×8 matrix with the 8 coordinates in $\mathbb{R}^2$ defining the shape of the sign. We assume that the stop sign resp. the priority road sign are defined by the "model" points (known a priori) $\mathbf{x}^i_{stop} \in \mathbb{R}^2$ resp. $\mathbf{x}^i_{priority} \in \mathbb{R}^2$ for $i = 2, \dots, 15$.

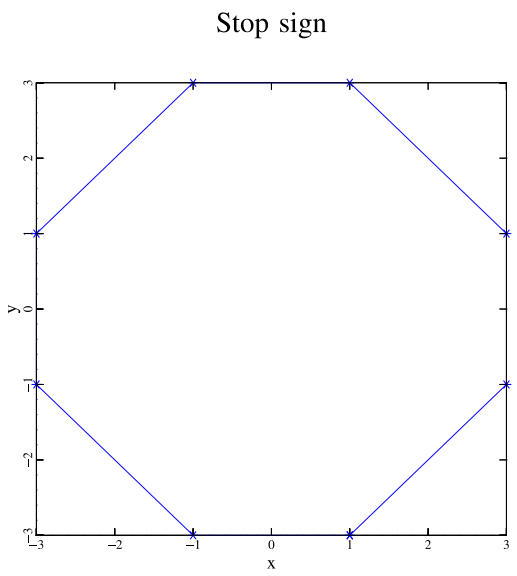


Fig. 12 The 8-points $\mathbf{x}^i_{stop}$ defining the model of a stop sign.

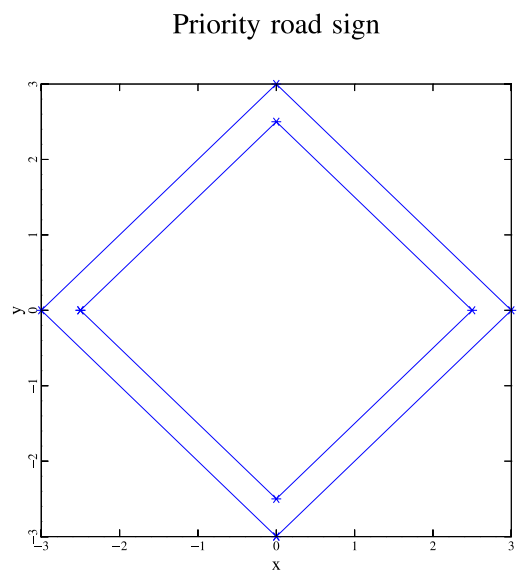


Fig. 13 The 8-points $\mathbf{x}^i_{priority}$ defining the model of a priority road sign.

However, in a real case scenario, one can imagine that the photographed shape is not exactly congruent to the one specified by the "model" points. Instead, one can assume that the points on the photo ($\mathbf{p}^i \in \mathbb{R}^2$) are the result of a *linear transformation* of the original points $\mathbf{x}^i \in \mathbb{R}^2$ for $i = 1, \dots, 15$; i.e. we can assume that there exists a matrix $\mathbf{A} \in \mathbb{R}^{2,2}$, s.t.

$$\mathbf{p}^i = \mathbf{A}\mathbf{x}^i, i = 1, \dots, n \,. \tag{4.7.1}$$

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

Moreover, we have some error in the data, i.e. our points do not represent exactly one of the linearly transformed shapes (it is plausible to imagine that there is some measurement error, and that the photographed shape is not exactly the same as our "model" shape), i.e. (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:

$$\mathbf{w} = \mathbf{B}\mathbf{v},$$

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 of (4-7.a):

Let $\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$. Then, for every $i = 1, \dots, n$:

$$\begin{bmatrix} p^i_1 \\ p^i_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} x^i_1 \\ x^i_2 \end{bmatrix} = \begin{bmatrix} a_{11}x^i_1 + a_{12}x^i_2 \\ a_{21}x^i_1 + a_{22}x^i_2 \end{bmatrix} \tag{4.7.2}$$

Since all entries of $\mathbf{A}$ are unknown, we want to write the above system as $\mathbf{p}^i = \mathbf{X}^i\mathbf{v}$, where $\mathbf{X}^i$ is given and:

$$\mathbf{v} = \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix} \tag{4.7.3}$$

Rewriting (4.7.2):

$$\begin{bmatrix} p_1^i \\ p_2^i \end{bmatrix} = \begin{bmatrix} x_1^i & x_2^i & 0 & 0 \\ 0 & 0 & x_1^i & x_2^i \end{bmatrix} \begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix} \tag{4.7.4}$$

Vertically stacking all the equations, for $i = 1, \dots, n$:

$$\underbrace{\begin{bmatrix} p_1^1 \\ p_2^1 \\ p_1^2 \\ p_2^2 \\ \vdots \\ p_1^n \\ p_2^n \end{bmatrix}}_{\mathbf{w}} = \underbrace{\begin{bmatrix} x_1^1 & x_2^1 & 0 & 0 \\ 0 & 0 & x_1^1 & x_2^1 \\ x_1^2 & x_2^2 & 0 & 0 \\ 0 & 0 & x_1^2 & x_2^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_1^n & x_2^n & 0 & 0 \\ 0 & 0 & x_1^n & x_2^n \end{bmatrix}}_{\mathbf{B}} \underbrace{\begin{bmatrix} a_{11} \\ a_{12} \\ a_{21} \\ a_{22} \end{bmatrix}}_{\mathbf{v}} .$$

The matrix $\mathbf{B}$ is a $2n \times 4$ matrix, $\mathbf{w}$ is a $2n \times 1$ vector and $\mathbf{v}$ a 4×1 vector.

▲

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

SOLUTION of (4-7.b):

The matrix has $\operatorname{rank}(\mathbf{B}) \leq 4$. The matrix has not full rank, if and only if its columns are linearly dependent, which is the case if and only if there exists a $c \in \mathbb{R}$ s.t. $x_2^i = cx_1^i$. This is equivalent to $\mathbf{x}^i = x_1^i[1, c]^\top$, i.e. all points $\mathbf{x}^i$ lie on the same line.

▲

(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 `X`.

SOLUTION of (4-7.c):

C++11-code 4.7.5: Building the system matrix $\mathbf{B}$.

```
MatrixXd shape_ident_matrix(const MatrixXd & X) {
    assert(X.rows() == 2 && "X must have 2 rows!");
    unsigned n = X.cols();

    MatrixXd B = MatrixXd::Zero(2*n, 4);

    for(unsigned int row = 0; row < n; ++row) {
        // Odd row, first two columns
        B(2*row,   0) = X(0,row);
        B(2*row,   1) = X(1,row);
        // Even row, last two columns
        B(2*row+1,2) = X(0,row);
        B(2*row+1,3) = X(1,row);
    }

    return B;
}
```

Get it on GitLab (shape_ident.cpp).

▲

(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 `X` resp. `P`.

SOLUTION of (4-7.d):

The following code demonstrates advanced use of EIGEN's facilities. Of course, everything can be done "in a pedestrian way".

C++11-code 4.7.6: Solving the lest square problem.

```
double solve_lsq(const MatrixXd & X,
                 const MatrixXd & P,
                 MatrixXd & A) {
    assert(X.rows() == 2 && "X must have 2 rows!");
    assert(P.rows() == 2 && "P must have 2 rows!");
    assert(X.cols() == P.cols() && "P and X must have same size!");
    unsigned n = X.cols();
    // Build system matrix
    MatrixXd B = shape_ident_matrix(X);

    // Solve LSQ system using normal equation
    // We need to do some reshaping in order to properly set up the
       system
    A = Map<MatrixXd>(
            MatrixXd(
                // Solve LSQ problem using normal equation
```

```
                    (B.transpose() * B).ldlt()
                                    .solve(B.transpose() *
                                           // Ned to vectorize matrix
                                           Map<const MatrixXd>(P.data(), 2*n, 1)
                                           )
                    // Pass an array to "Map"
                    ).data(),
                // Need to reshape vector to 2x2 matrix
                2, 2).transpose();

    // Residual:  must reshape P
    return (Map<const MatrixXd>(P.data(), 2, n) - A*X).norm();
}
```

Get it on GitLab (shape_ident.cpp).

▲

(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 indentifies the shape (either Stop or Priority sign) of the input points $\mathbf{p}^i$. The function returns an `enum` that classifies the shape of points specified by `P`. Return the linear transformation in the matrix `A`. The "model points" $\mathbf{x}^i_{stop}$ resp. $\mathbf{x}^i_{priority}$ are passed trough `Xstop` resp. `Xpriority`.

SOLUTION of (4-7.e):

We can solve the least square problem for both sets of points $\mathbf{x}^i_{stop}$ and $\mathbf{x}^i_{priority}$. It is reasonable to assume that the solution with the smallest residual error is the correct one. Of course this is just an heuristic argument.

C++11-code 4.7.7: Code for shape identification.

```
Shape identify(const MatrixXd Xstop,
               const MatrixXd Xpriority,
               const MatrixXd & P,
               MatrixXd & A) {
    // Best linear transformation for both "model" shape
    MatrixXd Astop, Apriority;

    // Residuals and linear transform for both "models"
    double res_stop = solve_lsq(Xstop, P, Astop);
    double res_priority = solve_lsq(Xpriority, P, Apriority);

    std::cout << "'Stop' residual norm: " << res_stop << std::endl
              << "'Prioriy' residual norm: " << res_priority << std::endl;

    // If residual with stop Model is bigger than the one with
```

```
        priority sign,
    // probably it it is priority sign, otherwise is a stop sign.
    if(res_stop <= res_priority) {
        std::cout << "Points appear to define a stop sign!" << std::endl;
        A = Astop;
        return Stop;
    }
    else {
        std::cout << "Points appear to define a priority road sign!" << std::endl;
        A = Apriority;
        return Priority;
    }
}
```

Get it on GitLab (shape_ident.cpp).

▲

(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 of (4-7.f):

The points `P1`, `P2`, resp. `P3` define the shape of a priority road sign, a stop sign, and a stop sign respectively.

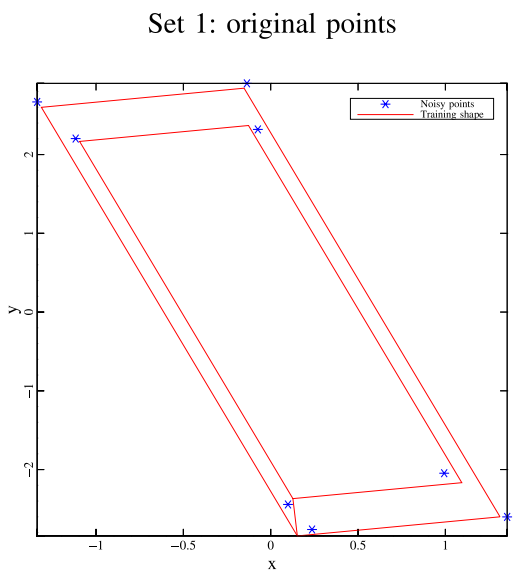


Fig. 14 The points to identify, in their original state.

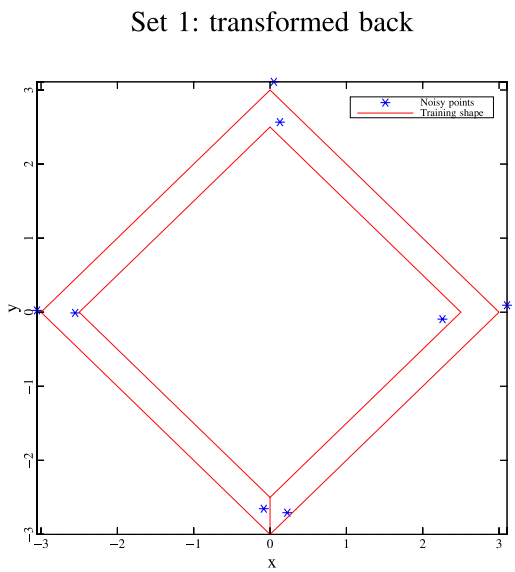


Fig. 15 The points of the "photo" after transformation under the best linear approximation. Notice that the "noise" is still visible.

C++11-code 4.7.8: Code for shape identification.

```
    {
        MatrixXd P1(2, n);
        P1 << 0.23657, 1.35369, -0.13624, -1.33702,
              0.0989619, 0.993235, -0.0735973, -1.11657,
             -2.76114, -2.60103, 2.90403, 2.66831,
```

```
            -2.44302, -2.04656, 2.31922, 2.20296;

        std::cout << "****************** Set 1 ******************"
                  << std::endl;
        MatrixXd A;
        Shape s = identify(Xstop, Xpriority, P1, A);
    }
    {
        MatrixXd P2(2, n);
        P2 << -1.12783, -1.75868, -1.40935, -0.0664574,
               1.09654, 1.75873, 1.5195, -0.0607661,
               1.72169, 0.344036, -0.889686, -1.87847,
              -1.57535, -0.41511, 0.834371, 1.88514;

        std::cout << "****************** Set 2 ******************"
                  << std::endl;
        MatrixXd A;
        Shape s = identify(Xstop, Xpriority, P2, A);
    }
    {
        MatrixXd P3(2, n);
        P3 << -1.23988, -0.731643, 0.00492048, 1.08039,
               1.34128, 0.670982, -0.101797, -1.02859,
               1.53076, 2.02881, 1.36163, -0.340912,
              -1.47697, -1.99975, -1.47947, 0.374859;

        std::cout << "****************** Set 23 ******************"
                  << std::endl;
        MatrixXd A;
        Shape s = identify(Xstop, Xpriority, P3, A);
    }
```

Get it on GitLab (shape_ident.cpp).

▲

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) $\operatorname{rank}(\mathbf{X}) = k$

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

HINT 1 for (4-8.a): Recall [Lecture → Thm. 3.4.1.1] and [Lecture → Lemma 3.4.1.12].

SOLUTION of (4-8.a):

See also [Lecture → Lemma 3.4.4.14].

Denote by σ_i the singular values of the matrix $\mathbf{X}$ [Lecture → Def. 3.4.1.3]. According to [Lecture → Lemma 3.4.1.12] the rank of a matrix $\mathbf{X}$ is given by the number of its nonzero singular values, $k = \operatorname{argmax}_i\{\sigma_i \neq 0\}$, or, equivalently, by the dimension of its range (the subspace of $\mathbb{R}^m$ generated by the columns of $\mathbf{X}$), see [Lecture → Def. 2.2.1.3].

(i)⇒(ii): If $\operatorname{rank}(\mathbf{x}) = k$, then $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top$, with $\boldsymbol{\Sigma} = \operatorname{diag}(\sigma_1,\dots,\sigma_k,0,\dots,0)$, see [Lecture → Eq. (3.4.1.13)].

Thus, we can omit the last rows/columns of the matrices in the decomposition: $\mathbf{X} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^\top = \mathbf{U}_{:,1:k}\boldsymbol{\Sigma}_{1:k,1:k}(\mathbf{V}_{:,1:k})^\top$.

By defining $\left\{\begin{array}{l}\mathbf{A} := \mathbf{U}_{:,1:k}\boldsymbol{\Sigma}_{1:k,1:k} \\ \mathbf{B} := \mathbf{V}_{:,1:k}\end{array}\right\}$, the claim follows.

(ii)⇒(i): The matrix $\mathbf{B}^\top$ corresponds to a linear mapping from $\mathbb{R}^n$ to $\mathbb{R}^k$, with $k \leq n$. Since it has rank k, the range of $\mathbf{B}^\top$ is the whole space $\mathbb{R}^k$.

The matrix $\mathbf{A}$ corresponds to a linear mapping from $\mathbb{R}^k$ to $\mathbb{R}^m$, with $k \leq m$. Since it has rank k, it is injective, i.e. $\dim[\ker(\mathbf{A})] = k - \operatorname{rank}(\mathbf{A}) = 0$, therefore the dimension of its range is k.

Finally, the rank of $\mathbf{X} = \mathbf{A}\mathbf{B}^\top : \mathbb{R}^n \to \mathbb{R}^m$ is equal to the dimension of its range, which is then k.

▲

(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 $\operatorname{rank}(\mathbf{X}) = k$ into $\mathbf{A}\mathbf{B}^\top$, 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 $\operatorname{rank}(\mathbf{X}) = k$ is in doubt.

HINT 1 for (4-8.b): Study [Lecture → Code 3.4.2.1], use EIGEN's facilities for *economical SVD* and pay attention on how deal with the SVD-factors $\mathbf{U}$, $\mathbf{\Sigma}$ and $\mathbf{V}$ properly. ⌟

SOLUTION of (4-8.b):

C++11-code 4.8.1: Solution of (4-8.b)

```
std::pair<MatrixXd, MatrixXd> factorize_X_AB(const MatrixXd &X,
                                             const unsigned int k) {
  size_t m = X.rows(), n = X.cols();

  double tol = 1e-6; // Tolerance for numerical rank: \lref{ex:svdrank}
  // TO DO (4-8.b) Define A,B with k columns such that X = A*B^T
  // Hint: warnings can be displayed with std::cerr, error messages with assert
  // START
  assert(k <= std::min(m, n) && "Rank k cannot be larger than dimensions of X");

  JacobiSVD<MatrixXd> svd(X, ComputeThinU | ComputeThinV);
  // With Eigen::svd you can ask for thin U or V to be computed.
  // In case of a rectangular m × n matrix,
  // with j the smaller value among m and n,
  // there can only be at most j singular values.
  // The remaining columns of U and V do not correspond
  // to actual singular vectors and are not computed in thin format.

  VectorXd s = svd.singularValues();

  if (k + 1 <= std::min(m, n) && s(k) > tol * s(0)) {
    // 1.\ condition checks if there is a singular value after k.
    // (S cannot be larger than the dimensions of X.)
    // 2.\ condition checks if such singular value
    // (by definition non-negative) is "numerically" different from 0.
    std::cerr << "Rank of matrix X is greater than required rank k"
              << std::endl;
  }
  // We know that k <= std::min(m,n) by the assert
  MatrixXd A = svd.matrixU().leftCols(k) * s.asDiagonal();
  MatrixXd B = svd.matrixV().leftCols(k);
  // END
  return std::make_pair(A, B);
}
```

Get it on GitLab (LowRankRep.cpp).

▲

(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 $\mathbf{B}$

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

outline, how the economical singular value decomposition $\mathbf{AB}^\top = \mathbf{U\Sigma V}$ [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 of (4-8.c):

We insert both QR-factorizations:

$$\mathbf{AB}^\top = \mathbf{Q}_A\mathbf{R}_A\mathbf{R}_B^\top\mathbf{Q}_B^\top = \mathbf{Q}_A\mathbf{S}\mathbf{Q}_B \quad \text{with} \quad \mathbf{S} := \mathbf{R}_A\mathbf{R}_B^\top \in \mathbb{R}^{k,k} \,. \tag{4.8.2}$$

Next, we rely on the singular value decomposition of the (small) square matrix $\mathbf{S}$:

$$\mathbf{S} = \mathbf{U\Sigma V}^\top \quad , \quad \mathbf{U},\mathbf{S},\mathbf{H} \in \mathbb{R}^{k,k} \,, \quad \mathbf{U},\mathbf{V} \quad \text{orthogonal} \,, \quad \mathbf{\Sigma} \quad \text{diagonal} \,. \tag{4.8.3}$$

Then, insert (4.8.3) into (4.8.2):

$$\mathbf{AB}^\top = \mathbf{Q}_A \underbrace{\mathbf{R}_A\mathbf{R}_B^\top}_{=\tilde{\mathbf{U}}\mathbf{\Sigma}\tilde{\mathbf{V}}^\top} \mathbf{Q}_B^\top = (\underbrace{\mathbf{Q}_A\tilde{\mathbf{U}}}_{=:\mathbf{U}})\mathbf{\Sigma}(\underbrace{\mathbf{Q}_B\tilde{\mathbf{V}}}_{=:\mathbf{V}})^\top \,.$$

Note that $\mathbf{U} \in \mathbb{R}^{m,k}$, $\mathbf{V} \in \mathbb{R}^{m,k}$, and

$$(\mathbf{Q}_A\mathbf{U})^\top\mathbf{Q}_A\mathbf{U} = \mathbf{U}^\top \underbrace{\mathbf{Q}_A^\top\mathbf{Q}_A}_{=\mathbf{I}} \mathbf{U} = \mathbf{U}^\top\mathbf{U} = \mathbf{I} \,,$$
$$(\mathbf{Q}_B\mathbf{V})^\top\mathbf{Q}_B\mathbf{V} = \mathbf{V}^\top \underbrace{\mathbf{Q}_B^\top\mathbf{Q}_B}_{=\mathbf{I}} \mathbf{V} = \mathbf{V}^\top\mathbf{V} = \mathbf{I} \,.$$

This shows that ((4-8.c)) really provides an economical SVD of $\mathbf{AB}^\top$.

▲

(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 $\mathbf{AB}^\top = \mathbf{U\Sigma V}^\top$, where $\mathbf{U},\mathbf{V} \in \mathbb{R}^{n,k}$ have orthogonal columns and $\mathbf{\Sigma} \in \mathbb{R}^{k,k}$ is a diagonal matrix. The SVD-factors are returned as a triple $(\mathbf{U},\mathbf{\Sigma},\mathbf{V})$.

HINT 1 for (4-8.d): If you directly compute an SVD of $\mathbf{AB}^\top$, you will always have to deal with dimensions m and n, regardless of $k \ll m,n$. As elaborated in Sub-problem (4-8.c) exploit the economical QR decomposition of both $\mathbf{A}$ and $\mathbf{B}$ separately to reduce the problem to the computation of the SVD of a small matrix! ⌟

SOLUTION of (4-8.d):

Of course, (4.8.3) is cheaper than the direct SVD of the $m \times n$ matrix $\mathbf{AB}^\top$, because the SVD has to be computed only for the $k \times k$ matrix $\mathbf{R}_A\mathbf{R}_B^\top$, with $k \ll m,n$.

C++11-code 4.8.4: Solution of (4-8.d)

```
std::tuple<MatrixXd, MatrixXd, MatrixXd> svd_AB(const MatrixXd &A,
                                                const MatrixXd &B) {
```

```
  assert(A.cols() == B.cols() &&
         "Matrices A and B should have the same column number");
  size_t m = A.rows();
  size_t n = B.rows();
  size_t k = A.cols();
  // TO DO (4-8.d): Define the tuple of
  // matrices U,S,V of the svd decomposition of A*B^T
  // Hint: You can define a tuple of objects using std::make_tuple
  // START
  // QA: m x k; QB: n x k; RA,RB k x k
  HouseholderQR<MatrixXd> QRA = A.householderQr();
  MatrixXd QA = QRA.householderQ() * MatrixXd::Identity(m, std::min(m, k));
  MatrixXd RA = MatrixXd::Identity(std::min(m, k), m) *
                QRA.matrixQR().triangularView<Upper>();
  HouseholderQR<MatrixXd> QRB = B.householderQr();
  MatrixXd QB = QRB.householderQ() * MatrixXd::Identity(n, std::min(n, k));
  MatrixXd RB = MatrixXd::Identity(std::min(n, k), n) *
                QRB.matrixQR().triangularView<Upper>();

  // U,V: k x k
  JacobiSVD<MatrixXd> svd(RA * RB.transpose(), ComputeFullU | ComputeFullV);
  // Thin matrices are unnecessary here as RA*RB' is
  // a square k x k matrix!
  // Moreover, if you had computed the thin matrices U and V
  // of the direct SVD of AB',
  // you would still have dealt with the minimum between m and n,
  // without exploiting k << m,n!

  VectorXd s = svd.singularValues();
  MatrixXd U = svd.matrixU();
  MatrixXd V = svd.matrixV();
  // U: m x k; V: n x k
  U = QA * U;
  V = QB * V;

  return std::make_tuple(U, s.asDiagonal(), V);
  // END
}
```

Get it on GitLab (LowRankRep.cpp).

If the matrices $\mathbf{A}$ and $\mathbf{B}$ came from the decomposition of a matrix $\mathbf{X}$ as in the previous subproblems, their columns would be orthogonal (i.e. $\mathbf{A}^\top\mathbf{A}$ and $\mathbf{B}^\top\mathbf{B}$ would be diagonal). Therefore, the triangular part of their QR decompositions would be diagonal, which can be used to speed up Code 4.8.4. However, our code works for every choice of matrices $\mathbf{A}, \mathbf{B}$ with the same number of columns k.

▲

(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 $m = n \to \infty$? Discuss the effort required by the different steps of your algorithm.

SOLUTION of (4-8.e):

The algorithm can be divided into the following steps:

1. $(\mathbf{A},\mathbf{B}) \mapsto (\mathbf{Q}_A,\mathbf{R}_A,\mathbf{Q}_B,\mathbf{R}_B)$, i.e. two economical QR decompositions (k Householder transformations of length $n-1, n-2, \dots, n-k+1$): each is worth $\mathcal{O}(nk^2)$ operations (see [Lecture → Exp. 3.3.3.31]).
2. $(\mathbf{R}_A,\mathbf{R}_B) \mapsto \mathbf{R}_A\mathbf{R}_B^\top$, i.e. a matrix multiplication $\mathcal{O}(k^3)$, independent of n.
3. $\mathbf{R}_A\mathbf{R}_B^\top \mapsto (\tilde{\mathbf{U}},\mathbf{\Sigma},\tilde{\mathbf{V}})$, i.e. an SVD for the $k \times k$ matrix, independent of n.
4. $(\tilde{\mathbf{U}},\tilde{\mathbf{V}},\mathbf{Q}_A,\mathbf{Q}_B) \mapsto (\mathbf{Q}_A\tilde{\mathbf{U}},\mathbf{Q}_B\tilde{\mathbf{V}})$, i.e. two $(n,k) \times (k,k)$ multiplications $\mathcal{O}(nk^2)$.

The total computational effort is linear in n.

▲

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

SOLUTION of (4-8.f):

From $\operatorname{rank}(\mathbf{X}) = k$ it follows that there is a basis for the image of $\mathbf{X}$ with k basis vectors, i.e. each column of $\mathbf{X}$ is a linear combination of these k vectors. The same holds for $\mathbf{Y}$ with other k basis vectors, where some can be obtainable from linearcombinations of the basis vectors of $\mathbf{X}$. Hence, each column of $\mathbf{X}+\mathbf{Y}$ is a linear combination of *at most* $2k$ vectors. Thus, $\operatorname{rank}(\mathbf{X}+\mathbf{Y}) \leq 2k$.

▲

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

HINT 1 for (4-8.g): Study [Lecture → Eq. (3.4.1.8)] in the case $p = 2$. ⌟

SOLUTION of (4-8.g):

We can define $\mathbf{A} := [\mathbf{A}_X\ \mathbf{A}_Y] \in \mathbb{R}^{m,2k}$ and $\mathbf{B} := [\mathbf{B}_X\ \mathbf{B}_Y] \in \mathbb{R}^{n,2k}$.

Hence, we have that $\mathbf{A}\mathbf{B}^\top = [\mathbf{A}_X\ \mathbf{A}_Y]\begin{bmatrix}\mathbf{B}_X^\top \\ \mathbf{B}_Y^\top\end{bmatrix} = \mathbf{A}_X\mathbf{B}_X^\top + \mathbf{A}_Y\mathbf{B}_Y^\top = \mathbf{X}+\mathbf{Y}$.

▲

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

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

Here the matrices $\mathbf{A}_X, \mathbf{A}_Y \in \mathbb{R}^{m,k}$ and $\mathbf{B}_X, \mathbf{B}_Y \in \mathbb{R}^{n,k}$ are given and passed to the function through the arguments `Ax`, `Ay`, `Bx`, and `By`.

HINT 1 for (4-8.h): $\|\cdot\|_F$ stands for the Frobenius norm of a matrix (see [Lecture → Def. 3.4.4.16]). Recall [Lecture → Thm. 3.4.4.18] and apply the function implemented in Sub-problem (4-8.d) to the product representation of $\mathbf{X} + \mathbf{Y}$ derived in Sub-problem (4-8.g). ⌟

SOLUTION of (4-8.h):

According to [Lecture → Thm. 3.4.4.18], the best low rank approximation of a matrix can be obtained by first calculating the SVD of the matrix and then extracting the first k columns of $\mathbf{U}$ and $\mathbf{V}$ and the upper left $k \times k$ submatrix of $\mathbf{\Sigma}$.

C++11-code 4.8.5: Solution of (4-8.h)

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

  assert(Ax.rows() == Ay.rows() && Ax.cols() == Ay.cols() &&
         "Matrices Ax and Ay should have the same dimensions");
  assert(Bx.rows() == By.rows() && Bx.cols() == By.cols() &&
         "Matrices Bx and By should have the same dimensions");
  // TO DO (4-8.h): Use the function svd_AB with the appropriate A,B and
  // return the pair Az, Bz of matrices of rank k with Z = Az*Bz
  // Hint: you can define pairs of objects using std::make_pair
  // Hint: use std::get<i>(my_tuple) to extract the i-th element of a tuple
  // START
  MatrixXd A(Ax.rows(), Ax.cols() + Ay.cols());
  A << Ax, Ay;
  MatrixXd B(Bx.rows(), Bx.cols() + By.cols());
  B << Bx, By;
  // U: m x 2k; S: 2k x 2k; V: n x 2k
  MatrixXd U, S, V;
  std::tuple<MatrixXd, MatrixXd, MatrixXd> svd_tuple = svd_AB(A, B);

  size_t k = Ax.cols();
  MatrixXd Az = std::get<0>(svd_tuple).leftCols(k) * std::get<1>(svd_tuple);
  MatrixXd Bz = std::get<2>(svd_tuple).leftCols(k);

  return std::make_pair(Az, Bz);
  // END
}
```

Get it on GitLab (LowRankRep.cpp).

▲

(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 $m = n \to \infty$?

SOLUTION of (4-8.i):

`svd_AB` has linear complexity in n and m, as derived in (4-8.e). The multiplication of the truncated matrices $\mathbf{U}$ and $\mathbf{S}$ requires $\mathcal{O}(mk)$ operations (since S is diagonal). Hence, the total computational effort is $\mathcal{O}(n+m)$.

▲

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:

$$\text{Given } n \in \mathbb{N}, \mathbf{z} \in \mathbb{R}^n, \mathbf{g} \in \mathbb{R}^n, \quad \text{find} \quad \mathbf{M}^* = \operatorname*{argmin}_{\mathbf{M}\in\mathbb{R}^{n,n},\, \mathbf{M}\mathbf{z}=\mathbf{g}} \|\mathbf{M}\|_F\,, \tag{4.9.1}$$

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)]

$$\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}\in\mathbb{R}^N,\, \mathbf{C}\mathbf{x}=\mathbf{d}} \|\mathbf{A}\mathbf{x}-\mathbf{b}\|_2\,, \tag{4.9.2}$$

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 of (4-9.a):

We denote $\mathbf{m}^* := \mathrm{vec}(\mathbf{M}^{*\top}) \in \mathbb{R}^{n^2}$ the vector obtained by the concatenation of the rows of $\mathbf{M}^*$ [Lecture → Rem. 1.2.3.4]. Then, since $\|\mathbf{M}^*\|_F = \|\mathbf{m}^*\|_2$, the functional to be minimised is just the 2-norm of $\mathbf{m}^*$, i.e., $\mathbf{A} = \mathrm{Id}_{n^2}$ and $\mathbf{b} = \mathbf{0}$. The constraint $\mathbf{M}^*\mathbf{z} = \mathbf{g}$ can be written as $\mathbf{C}\mathbf{m}^* = \mathbf{d}$, where $\mathbf{d} = \mathbf{g}$ and the $n \times n^2$ matrix $\mathbf{C} := \mathrm{Id}_n \otimes \mathbf{z}^\top$ is the Kronecker product of the $n \times n$ identity matrix and $\mathbf{z}^\top$. More explicitly,

$$\mathbf{C} = \begin{bmatrix} \mathbf{z}^\top & \mathbf{0}^\top & \cdots & \mathbf{0}^\top \\ \mathbf{0}^\top & \mathbf{z}^\top & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0}^\top & \cdots & \mathbf{0}^\top & \mathbf{z}^\top \end{bmatrix} \in \mathbb{R}^{n,n^2}\,.$$

▲

(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 of (4-9.b):

$\mathbf{C}$ has full rank, if $\mathbf{z}$ has at least one non-zero component ζ, because in this case $\mathbf{C}$ contains $\zeta\mathbf{I} \in \mathbb{R}^{n,n}$ as a sub-matrix.

▲

(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.

HINT 1 for (4-9.c): The augmented normal equations read:

$$\mathbf{S} := \begin{bmatrix} \mathsf{Id}_{n^2} & \mathbf{C}^\top \\ \mathbf{C} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{m}^* \\ \mathbf{p} \end{bmatrix} = \begin{bmatrix} \mathbf{0} \\ \mathbf{g} \end{bmatrix} =: \mathbf{h}. \tag{4.9.3}$$

Then carry out a block Gaussian elimination of the unknown $\mathbf{m}^*$. Which linear system for $\mathbf{p}$ emerges?

⌟

SOLUTION of (4-9.c):

By block Gaussian elimination of the unknown $\mathbf{m}^*$ from the augmented normal equations (4.9.3) we obtain

$$-\mathbf{C}\mathbf{C}^\top \mathbf{p} = \mathbf{g}\,. \tag{4.9.4}$$

By [Lecture → Thm. 3.1.2.9] the matrix $\mathbf{C}^\top\mathbf{C}$ is regular, if and only if $\mathbf{C} \in \mathbb{R}^{p,n}$ has full rank p, see Sub-problem (4-9.b).

▲

(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 of (4-9.d):

See code in Equation 4.9.5. As pointed out in the solution of Sub-problem (4-9.a) the matrix $\mathbf{C}$ is a Kronecker product.

C++11-code 4.9.5: Solution of minimization problem

```
MatrixXd min_frob(const VectorXd & z, const VectorXd & g) {
  assert(z.size() == g.size() && "Size mismatch!");
  unsigned int n = g.size();
  //TO DO (4-9.d): solve the augmented normal equations
  //Hint: Use Map to reshape a vector into a matrix
  //START
  // Build temporary matrix C.
  MatrixXd C = kroneckerProduct(MatrixXd::Identity(n,n),
                                z.transpose());
  VectorXd minus_p = (C * C.transpose()).ldlt().solve(g);
  VectorXd M = C.transpose() * minus_p;
```

```
  return Map<MatrixXd> (M.data(), n, n).transpose();
  //END
}
```

Get it on GitLab (lsqfrob.cpp).

This code is not optimal in terms of efficiency and memory usage: Of course, exploiting the special structure of the matrix $\mathbf{C}$ it is possible to solve the linear system of equations for efficiently.

▲

(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

$$\mathbf{M} = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2} \tag{4.9.6}$$

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 of (4-9.e):

See code in Code 4.9.7. The norm of the difference between $\mathbf{M}$ and $\mathbf{M}^*$ is within machine precision, suggesting that $\mathbf{M} = \mathbf{M}^*$.

C++11-code 4.9.7: Testing code

```
bool testMformula(unsigned int n) {
  double tol = 1e-10;
  // TO DO (4-9.e): compare the formula: return true if the relative
  // error in Frobenius norm is less than tol.
  //Hint: Define z using LinSpaced and g using replicate()
  //START
  VectorXd z = VectorXd::LinSpaced(n,1,n);
  Vector2d v = {1,-1};
  VectorXd g = v.replicate((int) n/2, 1);

  MatrixXd Mstar = min_frob(z, g);

  MatrixXd M = g*z.transpose() / z.squaredNorm(); // M

  if ((Mstar-M).norm() < tol * M.norm()) {
      return 1;
  }
  else {
    return 0;
  }
  //END
}
```

Get it on GitLab (lsqfrob.cpp).

▲

(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 of (4-9.f):

We show that $\mathbf{M} = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2}$ in vectorized form solves the augmented normal equations. To begin with note that

$$\mathbf{C}\mathbf{C}^\top = \|\mathbf{z}\|_2^2 \mathbf{I} \in \mathbb{R}^{p,p} \,.$$

Hence the solution of (4.9.3) is

$$\mathbf{p} = -\|\mathbf{z}\|_2^{-2}\mathbf{g} \quad , \quad \mathbf{m}^* = -\mathbf{C}^\top \|\mathbf{z}\|_2^{-2}\mathbf{g} = \|\mathbf{z}\|_2^{-2}\mathbf{g} \otimes \mathbf{z} \in \mathbb{R}^{n^2} \,. \tag{4.9.8}$$

Then "un-vectorize" $\mathbf{m}^*$!

▲

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** $L : \mathbb{R}^{n,n} \times \mathbb{R}^n \longrightarrow \mathbb{R}$ such that:

$$\mathbf{M}^* = \operatorname*{argmin}_{\mathbf{M}\in\mathbb{R}^{n,n}} \left\{ \sup_{\mathbf{m}\in\mathbb{R}^n} [L(\mathbf{M},\mathbf{m})] \right\}. \tag{4.9.9}$$

The matrix $\mathbf{M}^*$ should provide the solution of (4.9.1).

HINT 1 for (4-9.g): Study [Lecture → Section 3.6.1] again and try to grasp the heuristic idea behind enforcing constraints by means of Lagrangian multipliers. ⌟

SOLUTION of (4-9.g):

We want to find $L(\mathbf{M},\mathbf{m})$ which:

1. attains its minimum value at $\mathbf{M}^*$ (which minimizes $\|\mathbf{M}\|_F$ and respects the condition $\mathbf{M}\mathbf{z} = \mathbf{g}$) and
2. is maximized by $\mathbf{m}$.

In general, $\mathbf{m}$ can make the functional L go to ∞. To counterbalance its effect, it must be multiplied by a vector which is $\mathbf{0}$ when $\mathbf{M}\mathbf{z} = \mathbf{g}$, i.e. $\mathbf{M}\mathbf{z} - \mathbf{g}$. The functional L is therefore:

$$L(\mathbf{M},\mathbf{m}) = \|\mathbf{M}\|_F^2 + \mathbf{m}^\top(\mathbf{M}\mathbf{z} - \mathbf{g}).$$

▲

(4-9.h) ⚁ (10 min.) Find the derivative $\mathbf{grad}\,\Phi \in \mathbb{R}^{n,n}$ of the function Φ defined as:

$$\Phi : \mathbb{R}^{n,n} \to \mathbb{R}, \quad \Phi(\mathbf{X}) = \|\mathbf{X}\|_F^2 \tag{4.9.10}$$

HINT 1 for (4-9.h): You might be confused that you are asked to compute a gradient in the form of a matrix. However, function Φ takes matrices as arguments: in this subproblem matrices play the role of the usual vector arguments. Indeed, matrices can easily be identified with vectors by e.g. piling their columns one on top of the other.

Hence, you may simply consider matrix argument of Φ as a vector written in a different way, compute the gradient of Φ from this perspective and, at the end, convert the result to the matrix form. ⌟

SOLUTION of (4-9.h):

The simplest approach is by component-wise partial derivatives:

$$\frac{\partial}{\partial x_{ij}}\Phi(\mathbf{X}) = 2(\mathbf{X})_{ij} \quad \Rightarrow \quad \mathbf{grad}\,\Phi(\mathbf{X}) = 2\mathbf{X}\,. \tag{4.9.11}$$

▲

(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 $\mathbf{grad}\,L(\mathbf{M},\mathbf{m}) = 0$, for the L obtained in Sub-problem (4-9.g).

HINT 1 for (4-9.i): What are the necessary and sufficient condition for minimizing (4.9.9)? Remember [Lecture → Eq. (3.6.1.5)] and set $\frac{\partial L}{\partial \mathbf{M}} = 0$ and $\frac{\partial L}{\partial \mathbf{m}} = 0$. ⌟

SOLUTION of (4-9.i):

Using $\mathbf{grad}\,\Phi(\mathbf{X}) = 2\mathbf{X}$, the derivative of L w.r.t. $\mathbf{M}$ is:

$$\frac{\partial L}{\partial \mathbf{M}} = 2\mathbf{M} + \mathbf{m}\mathbf{z}^\top = 0. \tag{4.9.12}$$

The derivative of L w.r.t. $\mathbf{m}$ is:

$$\frac{\partial L}{\partial \mathbf{m}} = \mathbf{M}\mathbf{z} - \mathbf{g} = 0. \tag{4.9.13}$$

▲

(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.

HINT 1 for (4-9.j): What special structure for $\mathbf{M}^*$, solution of (4.9.9), is implied by one of the saddle point equations? ⌟

SOLUTION of (4-9.j):

Plugging (4.9.12) into (4.9.13) we obtain:

$$-\frac{1}{2}\mathbf{m}\|\mathbf{z}\|_2^2 - \mathbf{g} = 0 \quad \Longrightarrow \quad \mathbf{m} = -2\frac{\mathbf{g}}{\|\mathbf{z}\|_2^2} \quad \overset{((4.9.12))}{\Longrightarrow} \quad \mathbf{M}^* = \frac{\mathbf{g}\mathbf{z}^\top}{\|\mathbf{z}\|_2^2}. \tag{4.9.14}$$

The matrix $\mathbf{M}^*$ is the outer product of two vectors.

▲

End Problem 4-9 , 130 min.

Chapter 5

Filtering Algorithms

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. Therfore, we need efficient ways to evaluate the derivatives of interpolants. In this problem we discuss this issue for polynomial interpolation for (i) monomial representation and (ii) "update-friendly" point evaluation. We generalize the Horner scheme [Lecture → 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,

$$p(t) = \mathtt{c[0]}t^k + \mathtt{c[1]}t^{k-1} + \cdots + \mathtt{c[k-1]}t + \mathtt{c[k]}\,, \quad t \in \mathbb{R}\,. \tag{6.1.1}$$

As indicated above, the type **CoeffVec** must provide component access via **operator** `[]` (**int**) **const** and a `size()` method.

SOLUTION of (6-1.a):

The idea is to apply the Horner scheme [Lecture → Eq. (5.2.1.6)] to

$$p'(t) = k\mathtt{c[0]}t^{k-1} + k-1\mathtt{c[1]}t^{k-2} + \cdots + 2\mathtt{c[k-2]}t + \mathtt{c[k-1]}\,, \quad t \in \mathbb{R}\,. \tag{6.1.2}$$

The corresponding extension of [Lecture → Code 5.2.1.7] is straightforward.

C++-code 6.1.3: Implementation of `evaldp()`

```
// START
template <typename CoeffVec>
std::pair<double, double> evaldp(const CoeffVec &c, const double x) {
  double px, dpx;
  int s = c.size(); // degree + 1
  // Standard Horner scheme
  px = c[0]; for (int i = 1; i < s; ++i) px = x * px + c[i];
  // Horner scheme for derivative
  dpx = (s - 1) * c[0];
  for (int i = 1; i < s - 1; ++i) dpx = x * dpx + (s - i - 1) * c[i];

  return { px, dpx };
}
// END
```

▲

(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 of (6-1.b):

C++-code 6.1.4: Solution of (6-1.b): `evaldp_naive()`

```
// START
template <typename CoeffVec>
std::pair<double, double> evaldp_naive(const CoeffVec &c, const double x) {
  std::pair<double, double> p;
  double px, dpx;
  int n = c.size();

  px = c[0] * std::pow(x, n - 1);
  for (int i = 1; i < n; ++i) {
    px = px + c[i] * std::pow(x, n - i - 1);
  }

  dpx = (n - 1) * c[0] * std::pow(x, n - 2);
  for (int i = 1; i < n - 1; ++i) {
    dpx = dpx + (n - i - 1) * c[i] * std::pow(x, n - i - 2);
  }

  p.first = px;
  p.second = dpx;

  return p;
}
// END
```

▲

(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 of (6-1.c):

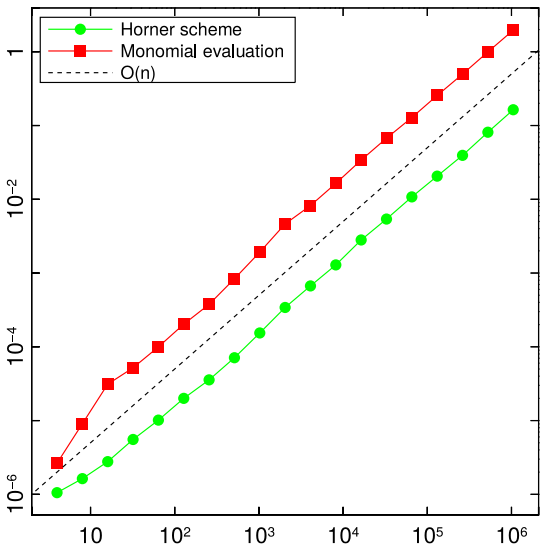


Fig. 16

The answer depends on your implementation. In any case, the code for `evaldp()` from Sub-problem (6-1.a) should require $\approx n$ multiplications and additions, and the asymptotic complexity is therefore $\mathcal{O}(n)$ for $n \to \infty$, where n is the degree of the polynomial.
Also the algorithm from Code 6.1.4 requires $\approx n$ multiplications and additions. However, the implementation also calls the `pow()` function, which may be costly. Beside in Fig. 16 we report comparative runtime measurements for the two implementation. They reveal that **std::**`pow()` probably incurs only constant computational cost.

▲

(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();
```

HINT 1 for (6-1.d): Please recall from [Lecture → Rem. 1.5.3.15] the advice on comparing numerically computed values ⌟

SOLUTION of (6-1.d):

C++-code 6.1.5: Solution of Sub-problem (6-1.d)

```
bool polyTestTime(unsigned int d) {

  bool ret = false;
  double x = .123;
  std::pair<double, double> p, p_naive;
  int repeats = 10;

  std::cout << std::setw(10) << "n" << std::setw(25) << "Horner scheme:"
            << std::setw(25) << "Monomial approach:" << "\n"
            << "
              ==================================================================\n";
  // TO DO (6-1.d): Compare the output of evaldp() and evaldp_naive() for
  // x = .123 and tabulate their runtimes for n=2,4,...,2^d.
  // START
  double TOL = 1E-8;
  int max_n = std::pow(2,d);
  VectorXd c = VectorXd::LinSpaced(max_n+1,1,max_n+1);
  for(int n = 2; n <= max_n; n*=2) {
    Timer tm_slow, tm_fast;

    for(int r=0; r<repeats; r++) {
      tm_fast.start();
      p = evaldp(c.head(n+1),x);
      tm_fast.stop();

      tm_slow.start();
      p_naive = evaldp_naive(c.head(n+1),x);
      tm_slow.stop();
    }

    std::cout << std::setw(10) << n << std::setw(25) << tm_fast.mean()
              << std::setw(25) << tm_slow.mean() << "\n";

    // Check correctness
    double err_p = std::abs(p.first - p_naive.first);
    double err_dp = std::abs(p.second - p_naive.second);
    if ((err_p >= TOL) || (err_dp >= TOL)) {
      std::cout << "Discrepancy at n = " << n << std::endl;
      return false;
    }
  }
  ret = true;
  // END
  return ret;
}
```

▲

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].

HINT 1 for (6-1.e): You may EIGEN's **Array** facilities and componentwise operations to acheive a vectorized implementation. ⌟

SOLUTION of (6-1.e):

Differentiating the recursion formula [Lecture → Eq. (5.2.3.9)] we obtain

$$
\begin{aligned}
p_i(t) \equiv y_i \quad , \quad p_i'(t) \equiv 0, \qquad i = 0,\dots,n\,,\\
p_{i_0,\dots,i_m}(t) &= \frac{(t-t_{i_0})p_{i_1,\dots,i_m}(t) - (t-t_{i_m})p_{i_0,\dots,i_{m-1}}(t)}{t_{i_m}-t_{i_0}},\\
p'_{i_0,\dots,i_m}(t) &= \frac{p_{i_1,\dots,i_m}(t) + (t-t_{i_0})p'_{i_1,\dots,i_m}(t) - p_{i_0,\dots,i_{m-1}}(t) - (t-t_{i_m})p'_{i_0,\dots,i_{m-1}}(t)}{t_{i_m}-t_{i_0}}\,.
\end{aligned}
$$

C++-code 6.1.6: Example code using Aitken-Neville scheme

```
VectorXd dipoleval(const VectorXd &t, const VectorXd &y, const VectorXd &x) {
  assert(t.size() == y.size() && "t and y must have same size!");
  VectorXd ret; ret.resizeLike(x); // return value
  // TO DO (6-1.e): Use the Aitken-Neville algorithm to evaluate at x
  // the derivative of the interpolating polynomial of the data (t,y).
  // START
  // Non-vectorized version: loop over all evaluation points
  for (int i = 0; i < x.size(); ++i) {
    VectorXd p(y);
    VectorXd dP = VectorXd::Zero(y.size());
    // Aitken-Neville outer loop
    for (int im = 1; im < y.size(); ++im) {
      for (int i0 = im - 1; i0 >= 0; --i0) {
        dP(i0) = (p(i0 + 1) + (x(i) - t(i0)) * dP(i0 + 1) - p(i0) -
                  (x(i) - t(im)) * dP(i0)) /
                 (t(im) - t(i0));
        p(i0) = ((x(i) - t(i0)) * p(i0 + 1) - (x(i) - t(im)) * p(i0)) /
                (t(im) - t(i0));
      }
    }
    ret(i) = dP(0);
  }
  // END
  return ret;
}
```

▲

(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 testDipolEval(void);
```

Return **false** in case the test fails, **true** otherwise.

SOLUTION of (6-1.f):

C++-code 6.1.7: Implementation of validation function `dipoleval_alt()`

```
VectorXd dipoleval_alt(const VectorXd &t, const VectorXd &y,
                       const VectorXd &x) {
  assert(t.size() == y.size() && "t and y must have same size!");
  VectorXd ret;// return value
  // TO DO (6-1.f): Use polyfit() and polyval() to evaluate at x
  // the derivative of the interpolating polynomial of the data (t,y).
  // START
  int n = y.size();
  VectorXd P = polyfit(t, y, n - 1).head(n - 1);

  for (int i = 0; i < P.size(); ++i) {
    P(i) *= n - 1 - i;
  }
  ret = polyval(P, x);
  // END
  return ret;
}
```

C++-code 6.1.8: Testing framework

```
bool testDipolEval(void) {
  bool ret = false;
  VectorXd dPx, dPx_alt;
  // TO DO (6-1.f): Compare dipoleval() and dipoleval_alt() for the data
  // given on the problem sheet.
  // START
  double TOL = 1E-8;
  int n = 10; // polynomial degree
  VectorXd x = VectorXd::LinSpaced(100,-1,4);
  VectorXd t = VectorXd::LinSpaced(n+1,0,3);
  VectorXd y = sin(t.array()).matrix();
  dPx = dipoleval(t,y,x);
  dPx_alt = dipoleval_alt(t,y,x);
  double err = (dPx-dPx_alt).norm();
  if (err < TOL) ret = true;
  // END
  return ret;
}
```

▲

(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.

HINT 1 for (6-1.g): Axis labels can be added to the graphics object of MATPLOTLIBCPP by the functions `matplotlibcpp::xlabel()` and `matplotlibcpp::ylabel()`.

SOLUTION of (6-1.g):

C++-code 6.1.9: Plotting of polynomial interpolant

```
void plotPolyInterpolant(const std::string &filename) {
  plt::figure();
  // TO DO (6-1.g): Plot the derivative of the polynomial that
  // interpolates the data from (6-1.f).
  // Hint: matplotlibcpp (here = plt) implements Python's
  // matplotlib functions such as figure(), plot(), xlabel(), title(),...
  // You can use main.cpp of the LinearDataFit problem as a reference.
  // START
  VectorXd dPx;
  int n = 10; // polynomial degree
  VectorXd x = VectorXd::LinSpaced(100,0,3);
  VectorXd t = VectorXd::LinSpaced(n+1,0,3);
  VectorXd y = sin(t.array()).matrix();
  dPx = dipoleval(t,y,x);
  plt::plot(x,dPx);
  plt::title("Derivative of interpolating polynomial");
```

```
  plt::xlabel("t");
  plt::ylabel("p'(t)");
  // END
  plt::savefig("./cx_out/" + filename + ".png");
}
```

▲

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

$$p(x) = \sum_{i=0}^{n} y_i L_i(x) \quad \text{with} \quad L_i(x) := \prod_{\substack{j=0\\ j\neq i}}^{n} \frac{x - t_j}{t_i - t_j} \,. \tag{6.2.1}$$

(6-2.a) ⚀ Show that

$$\sum_{i=0}^{n} L_i(t) = 1 \quad \forall t \in \mathbb{R} \,. \tag{6.2.2}$$

HINT 1 for (6-2.a): Which p will you get in (6.2.1), when setting $y_j = 1$? Recall the uniqueness of Lagrange interpolants asserted in [Lecture → Thm. 5.2.2.7].

SOLUTION of (6-2.a):

Consider the interpolation points $(t_i, 1)_{i=0}^n$. Clearly, an interpolating polynomial for these points is $p \equiv 1$. By uniquess of Lagrange interpolants, we have $1 = p(x) = \sum_{i=0}^n L_i(x)$.

▲

(6-2.b) ⚁ Show that

$$\sum_{i=0}^{n} L_i(0) t_i^j = \begin{cases} 1 & \text{for } j = 0 \,, \\ 0 & \text{for } j = 1,\dots,n \,. \end{cases} \tag{6.2.3}$$

(Read t_i raised to the power j.)

HINT 1 for (6-2.b): Consider $y_i = t_i$ in (6.2.1) for $i = 0,\dots,n$. What will be the Lagrange interpolant p in this case?

SOLUTION of (6-2.b):

For $j = 0$, see the previous problem. Consider $j > 0$ and the interpolation points $(t_i, t_i^j)_{i=0}^n$. Clearly, an interpolating polynomial for these points is $p(x) \equiv x^j$. By uniquess of Lagrange interpolants (and $j <= n$), we have $x^j = p(x) = \sum_{i=0}^n L_i(x) t_i^j$. Evaluating at $x = 0$: $0 = \sum_{i=0}^n L_i(0) t_i^j$.

▲

(6-2.c) ⚁ Show that $p(x)$ can also be written as

$$p(x) = \omega(x) \sum_{i=0}^{n} \frac{y_i}{(x-t_i)\omega'(t_i)} \quad \text{with} \quad \omega(t) := \prod_{j=0}^{n}(t-t_j) \ . \tag{6.2.4}$$

HINT 1 for (6-2.c): Apply the product rule in a nested way to differentiate $\omega(t)$. ⌟

SOLUTION of (6-2.c):

Simply exploiting the product rule for a product of several functions

$$\frac{d}{dx}\left\{ x \mapsto \prod_{j=1}^{n} f_j(x) \right\} = \sum_{i=1}^{n} f_i'(x) \prod_{j\neq i} f_j(x) \ , \tag{6.2.5}$$

we obtain:

$$\omega'(x) = \sum_{i=0}^{n} \prod_{\substack{j=0 \\ j\neq i}}^{n} (x-t_j) \ .$$

Since $(x-t_j) = 0$ when $x = t_j$, it follows that $\omega'(t_i) = \prod_{\substack{j=0 \\ j\neq i}}^{n}(t_i - t_j)$. Therefore:

$$\begin{aligned}
\omega(x) \sum_{i=0}^{n} \frac{y_i}{(x-t_i)\omega'(t_i)} &= \sum_{i=0}^{n} \left[\frac{y_i}{(x-t_i)\prod_{\substack{j=0 \\ j\neq i}}^{n}(t_i-t_j)} \prod_{j=0}^{n}(x-t_j) \right] \\
&= \sum_{i=0}^{n} \left[\frac{y_i}{\prod_{\substack{j=0 \\ j\neq i}}^{n}(t_i-t_j)} \prod_{\substack{j=0 \\ j\neq i}}^{n}(x-t_j) \right] = \sum_{i=0}^{n} \left[y_i \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x-t_j}{t_i-t_j} \right] = p(x) \ .
\end{aligned}$$

▲

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

$$L_i(t_j) = \delta_{ij}\,, \qquad L_i'(t_j) = 0\,, \tag{6.3.1a}$$

$$K_i(t_j) = 0\,, \qquad K_i'(t_j) = \delta_{ij}\,, \tag{6.3.1b}$$

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 $\mathcal{P}_{2n+1}$.

HINT 1 for (6-3.a): Recall how the standard Lagrange polynomial interpolation is linked to the Vandermonde matrix presented in [Lecture → Rem. 5.2.2.10]. ⌟

SOLUTION of (6-3.a):

Let $p(x) = \sum_{i=0}^{2n+1} a_i x^i$ be the interpolation polynomial in monomial form. We seek a linear system of equations $\mathbf{V}\mathbf{a} = \mathbf{b}$, for $\mathbf{a} = [a_0, \dots, a_{2n+1}]^\top \in \mathbb{R}^{2n+2}$. The first $n+1$ equations of the system arise from the interpolation conditions $p(t_i) = y_i$:

$$y_j = p(t_j) = \sum_{i=0}^{2n+1} a_i t_j^i,\ j = 0, \dots, n\,.$$

cf. the Vandermonde matrix. Differentiating the polynomial in monomial representation we get $p'(x) = \sum_{i=1}^{2n+1} i a_i x^{i-1}$. Using the interpolation condition $p'(t_i) = c_i$ for the derivative gives $n+1$ more rows of the linear system:

$$c_j = p'(t_j) = \sum_{i=1}^{2n+1} a_i i t_j^{i-1},\ j = 0, \dots, n\,.$$

Hence, we find $\mathbf{b} = [y_0, \dots, y_n, c_0, \dots, c_n]^\top$ and the system matrix $\mathbf{V}$ turns out as

$$\mathbf{V} = \begin{bmatrix} 1 & t_0 & t_0^2 & \ddots & t_0^{2n+1} \\ \vdots & \vdots & \vdots & \vdots & \\ 1 & t_n & t_n^2 & \dots & t_n^{2n+1} \\ 0 & 1 & 2t_0 & \dots & (2n+1)t_0^{2n} \\ \vdots & \vdots & \vdots & \vdots & \\ 0 & 1 & 2t_n & \dots & (2n+1)t_n^{2n} \end{bmatrix} \in \mathbb{R}^{2n+2,2n+2}\,.$$

▲

(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 $n = 1$, $t_0 = -1$, $t_1 = 1$.

HINT 1 for (6-3.b): Study [Lecture → Ex. 5.2.2.18] and consult [Lecture → Eq. (5.4.1.5)].

,

SOLUTION of (6-3.b):

We can find the generalized Lagrange polynomials in monomial form by solving the linear system derived in Sub-problem (6-3.a). In this special case we have

$$\mathbf{V} = \begin{bmatrix} 1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 \\ 0 & 1 & -2 & 3 \\ 0 & 1 & 2 & 3 \end{bmatrix} \in \mathbb{R}^{4,4} .$$

By (6.3.1) the correspoding right-hand side vectors are just the the Cartesian coordinate vectors:

$$L_0 \leftrightarrow \quad \mathbf{b} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad , \quad L_1 \leftrightarrow \quad \mathbf{b} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad , \quad K_0 \leftrightarrow \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad , \quad K_1 \leftrightarrow \quad \mathbf{b} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} .$$

Thus we obtain the monomial expansion coefficients of L_0, L_1, K_0, K_1 as the columns of $\mathbf{V}^{-1}$:

$$\mathbf{V}^{-1} = \frac{1}{4} \begin{bmatrix} 2 & 2 & 1 & -1 \\ -3 & 3 & -1 & -1 \\ 0 & 0 & -1 & 1 \\ 1 & -1 & 1 & 1 \end{bmatrix} .$$

The resulting polynomials can also we written as

$$L_0(t) = 3\left(\tfrac{1-t}{2}\right)^2 - 2\left(\tfrac{1-t}{2}\right)^3 \quad , \quad L_1(t) = 3\left(\tfrac{1+t}{2}\right)^2 - 2\left(\tfrac{1+t}{2}\right)^3 ,$$
$$K_0(t) = 2\left[\left(\tfrac{1-t}{2}\right)^2 - \left(\tfrac{1-t}{2}\right)^3\right] \quad , \quad K_1(t) = -2\left[\left(\tfrac{1+t}{2}\right)^2 - \left(\tfrac{1+t}{2}\right)^3\right] .$$

▲

(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 of (6-3.c):

1. Any parabola that touches the x-axis in $x = a$ is of the form $x \mapsto c(x-a)^2$ with some $x \in \mathbb{R}$.

2. Every polynomial $p \in \mathcal{P}_n$, $n \geq 2$, with double zero in $t = a$ can be written as

$$p(t) = (t-a)^2 q(t) \quad \text{for some} \quad q \in \mathcal{P}_{n-2} \,.$$

▲

(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 of (6-3.d):

Group the terms of the product accoding to

$$h(t) = g_1(t) \cdot (g_2(t) \cdot \dots \cdot g_m(t)) \,,$$

and apply the standard product rule $(fg)' = f'g + fg'$. Thus, by simple induction with respect to the length m of the product, one shows

$$\boxed{h'(t) = \sum_{j=1}^m \left\{ g_j'(t) \cdot \prod_{\substack{k=1 \\ k \neq j}}^m g_k(t) \right\}} \,. \tag{6.3.2}$$

▲

(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.

HINT 1 for (6-3.e): Of course, the insights obtained in Sub-problem (6-3.c) and Sub-problem (6-3.d) should be used. ⌟

HINT 2 for (6-3.e): Note that the polynomials L_i and K_i have double zeros in t_j, $j \neq i$. According to Sub-problem (6-3.c) they can be written as

$$\begin{aligned} L_i(t) &= (\alpha_i t + \delta_i) \prod_{\substack{j=0 \\ j \neq i}}^n (t - t_j)^2 \,, \\ K_i(t) &= (\gamma_i t + \beta_i) \prod_{\substack{j=0 \\ j \neq i}}^n (t - t_j)^2 \,. \end{aligned} \tag{6.3.3}$$

Then determine α_i, δ_i, γ_i and β_i from (6.3.1) using the interpolation conditions and the result of Sub-problem (6-3.d). ⌟

SOLUTION of (6-3.e):

To begin with we set

$$h_i(t) := \prod_{\substack{j=0,\dots,n \\ j \neq i}} (t - t_j)^2 \,, t \in \mathbb{R} \,,$$

and apply (6.3.2):

$$h_i'(t) = 2\sum_{j\neq i}(t-t_j)\prod_{k\neq i,j}(t-t_k)^2 = 2h_i(t)\cdot\sum_{j\neq i}\frac{1}{t-t_j} \quad \blacktriangleright \quad q_i(t) := \frac{h_i'(t)}{h_i(t)} = 2\sum_{j\neq i}\frac{1}{t-t_j}\,.$$

The interpolation conditions (6.3.1) give us $L_i(t_i) = K_i'(t_i) = 1$ and $K_i(t_i) = L_i'(t_i) = 0$. Using the representations from (6.3.3) we find equations for α_i and δ_i

$$\begin{aligned} 1 &= L_i(t_i) = (\alpha_i t_i + \delta_i)h_i(t_i)\,, \\ 0 &= L_i'(t_i) = \delta_i h_i(t_i) + (\alpha_i + \delta_i t_i)h_i'(t_i)\,, \end{aligned}$$

which leads to the linear system of equations:

$$\begin{bmatrix} h_i(t_i) & t_i h_i(t_i) \\ h_i'(t_i) & h_i(t_i) + t_i h_i'(t_i) \end{bmatrix}\begin{bmatrix}\alpha_i \\ \delta_i\end{bmatrix} = \begin{bmatrix}1\\0\end{bmatrix}\,.$$

We find the solution

$$\begin{bmatrix}\alpha_i \\ \delta_i\end{bmatrix} = \frac{1}{h_i(t_i)^2}\begin{bmatrix} h_i(t_i) + t_i h_i'(t_i) \\ -h_i'(t_i)\end{bmatrix} = \frac{1}{h_i(t_i)}\begin{bmatrix} 1 + t_i q_i(t_i) \\ -q_i(t_i)\end{bmatrix}\,.$$

Let's now compute β_i and γ_i for K_i. Expanding the formula for K_i using the chain rule:

$$0 = (\gamma_i t_i + \beta_i)\prod_{\substack{j=0\\ j\neq i}}^{n}(t_i - t_j)^2\,,$$

$$1 = 2(\gamma_i t_i + \beta_i)\sum_{k=0,k\neq i}^{n}(t_i - t_k)\prod_{\substack{j=0\\ j\neq i,k}}^{n}(t_i - t_j)^2 + \gamma_i\prod_{\substack{j=0\\ j\neq i}}^{n}(t_i - t_j)^2\,.$$

Since $H_i \neq 0$, we have $\gamma_i t_i + \beta_i = 0$. Hence: $\gamma_i = H_i^{-1}$. And, thus $\beta_i = -t_i H_i^{-1}$. We conclude with:

$$K_i(t) = \frac{t - t_i}{H_i}\prod_{\substack{j=0\\ j\neq i}}^{n}(t - t_j)^2\,.$$

▲

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:

```
class LinearInterpolant {
public:

    /*!
     * \brief LinearInterpolant builds interpolant from data.
     * Sort the array for the first time:
     * the data is not assumed to be sorted
     * sorting is necessary for binary search
     * \param TODO
     */
    LinearInterpolant(/* TODO: pass data here */);

    /*!
     * \brief operator () Evaluation operator.
     * Return the value of I at x, i.e. I(x).
     * Performs bound checks (i.e. if x < t_0 or x >= t_n ).
     * \param x Value x ∈ ℝ.
     * \return Value I(x).
     */
    double operator() (double x);
private:
    // TODO: your data there
};
```

Get it on GitLab (linearinterpolant.cpp).

SOLUTION of (6-4.a):

C++11-code 6.4.2:

```
LinearInterpolant::LinearInterpolant(const data & i_points_) {
    assert( i_points_.size() > 1
            && "Must specify at least two nodes");
    // assignment of std::vectors in C++ performs deep copy
    i_points = i_points_;

    // This is needed in std::sort
    auto Ordering = [] (const pair & P, const pair & Q) -> bool {
        return P.first < Q.first;
    };
```

```
    // Sort the array
    std::sort(i_points.begin(), i_points.end(), Ordering);
}
```

Get it on GitLab (linearinterpolant.cpp).

C++11-code 6.4.3:

```
double LinearInterpolant::operator() (double x) {
    // Lambda for compasiron of pair type,
    // we want to compare only the
    // first element of the pair, i.e. t_i
    auto Compare = [] (const pair & P, double V) -> bool { return P.first < V; };

    // Find the place i (as iterator), wehre t_i >= x, notice that i_points
    // *must* be sorted (this is needed in std::lower_bound)
    // IMPORTANT: "lower_bound" performs a binary search
    // on the data, provided the data has a random access iterator
    // http://www.cplusplus.com/reference/iterator/RandomAccessIterator/
    // this is crucial here, since evaluation operators must be fast
    // this allows to reduce the complexity,
    // from linear complexity to a logarithmic
    auto it = std::lower_bound(i_points.begin(), i_points.end(), x, Compare);

    // Bound checks, if i = 0 and t_0 != x (we are before the first node)
    // or if x > t_n (we are after the last node).
    // In such cases return 0 (*it and *(it-1) would be undefined in such cases)
    if( ( it == i_points.begin() && it->first != x )
            || it == i_points.end() )
        return 0;

    // Actually compute the interpolated value, dist_rato
    // contains the distance from t_{i-1} to x (as a ratio)
    double dist_ratio = (x - (it-1)->first) / (it->first - (it-1)->first);
    return (it-1)->second * (1 - dist_ratio) + it->second * dist_ratio;
}
```

Get it on GitLab (linearinterpolant.cpp).

▲

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

$$\mathcal{B}_{\mathrm{Re}} := \Big\{ C_0 := t \mapsto 1, S_1 := t \mapsto \sin(2\pi t), C_1 := t \mapsto \cos(2\pi t), S_2 := t \mapsto \sin(4\pi t), \\ C_2 t \mapsto \cos(4\pi t), \dots, S_n : t \mapsto \sin(2n\pi t), C_n := t \mapsto \cos(2n\pi t) \Big\} . \tag{6.5.1}$$

Another (ordered) basis of $\mathcal{P}_{2n}^T$ is

$$\mathcal{B}_{\exp} := \{ B_k := t \mapsto \exp(2\pi k \imath t) : k = -n, \dots, n \} . \tag{6.5.2}$$

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 $\mathbf{S} \in \mathbb{C}^{2n+1,2n+1}$ that effects the transformation of a coefficient representation with respect to $\mathcal{B}_{\mathrm{Re}}$ into a coefficient representation with respect to $\mathcal{B}_{\exp}$.

HINT 1 for (6-5.a): Recall Euler's formula expressing trigonometric functions in terms of the complex exponential function.

SOLUTION of (6-5.a):

Let $x_k(t) := \sin(2\pi kt), y_k(t) := \cos(2\pi kt)$ and $B_k(t) := \exp(2\pi \imath kt)$. Notice $B_0 = C_0 = 1$. Then, using trigonometric identities, $x_k(t) = \frac{1}{2\imath} B_{-k}(t) - \frac{1}{2\imath} B_k$, $y_k(t) = \frac{1}{2} B_{-k}(t) + \frac{1}{2} B_k$. We get the matrix entries by writing the basis elements of $\mathcal{B}_{\mathrm{Re}}$ in terms of the B_k, using Euler's formula.

$$\begin{aligned}
y_0(t) &= B_0(t) , \\
x_1(t) &= \tfrac{1}{2\imath}(\exp(2\pi\imath t) - \exp(-2\pi\imath t)) = \tfrac{1}{2\imath}(B_1(t) - B_{-1}(t)) , \\
y_1(t) &= \tfrac{1}{2}(\exp(2\pi\imath t) + \exp(-2\pi\imath t)) = \tfrac{1}{2}(B_1(t) + B_{-1}(t)) , \\
&\vdots \\
x_n(t) &= \tfrac{1}{2\imath}(\exp(2\pi\imath nt) - \exp(-2\pi\imath nt)) = \tfrac{1}{2\imath}(B_n(t) - B_{-n}(t)) , \\
y_n(t) &= \tfrac{1}{2}(\exp(2\pi\imath nt) + \exp(-2\pi\imath nt)) = \tfrac{1}{2}(B_n(t) + B_{-n}(t)) ,
\end{aligned}$$

Pay attention to the indices! The basis function C_0 is associated with matrix column 1, S_k, with matrix column $2k$, C_k with matrix column $2k+1$, $k = 1, \dots, n$. Further B_{k-n} is associated with matrix row $k+1$, $k = 0, \dots, 2n$.

$$\mathbf{C} = \begin{bmatrix}
0 & 0 & 0 & \dots & 0 & -\frac{1}{2} \\
0 & 0 & 0 & \dots & \frac{\imath}{2} & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & \frac{\imath}{2} & -\frac{1}{2} & \dots & 0 & 0 \\
1 & 0 & 0 & \dots & 0 & 0 \\
0 & -\frac{\imath}{2} & \frac{1}{2} & \dots & 0 & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \dots & -\frac{\imath}{2} & 0 \\
0 & 0 & 0 & \dots & 0 & \frac{1}{2}
\end{bmatrix}$$

▲

(6-5.b) ⚁ The shift operator $\mathsf{S}_\tau : C^0(\mathbb{R}) \to C^0(\mathbb{R})$ acts on a function according to $\mathsf{S}_\tau f(t) = f(t-\tau)$, for $\tau \in \mathbb{R}$. If $\mathbf{c} \in \mathbb{C}^{2n+1}$ is the coefficient vector of $p \in \mathcal{P}_{2n}^T$ with respect to $\mathcal{B}_{\exp}$, what is the coefficient vector $\tilde{\mathbf{c}} \in \mathbb{C}^{2n+1}$ of $\mathsf{S}_\tau p$?

HINT 1 for (6-5.b): All you need to do is to study the action of S_τ on basis functions. ⌟

SOLUTION of (6-5.b):

Consider $S_\tau B_k = \exp(2\pi\imath kt - 2\pi\imath k\tau) = B_k \exp(-2\pi\imath k\tau)$. If $p = \sum_{k=-n}^{n} c_k B_k(t)$ (expressed in the basis $\mathcal{B}_{exp}$), then

$$S_\tau p = \sum_{k=-n}^{n} c_k S_\tau B_k(t) = \sum_{k=-n}^{n} c_k \exp(-2\pi\imath k\tau) B_k(t).$$

The new coefficient vector is therefore $\tilde{\mathbf{c}} = [c_k \exp(2\pi\imath k\tau)]_{k=-n}^{n}$.

▲

Now we consider the set of interpolation nodes

$$\mathcal{T}_n := \left\{ \frac{j+1/2}{2n+1} : \; j = 0, \dots, 2n \right\} . \tag{6.5.3}$$

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

$$b_j(t_k) = \delta_{kj} \quad \text{[Kronecker symbol]} , \quad j,k = 0, \dots, 2n \tag{6.5.4}$$

(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 of (6-5.c):

We have to feed unit vectors as `y`-argument to `trigpolyvalequid` and `M=1000`. Note that the function uses the equidistant points $t_j = \frac{j}{2n+1}$, $j = 0, \dots, 2n$, which does not match the node set $\mathcal{T}_n$ from (6.5.3). Fortunately this can be compensated by reading the returned values as values for evaluation points $x_k = \frac{k}{M} + \frac{1}{4N+2}$ and taking into account 1-periodicity.

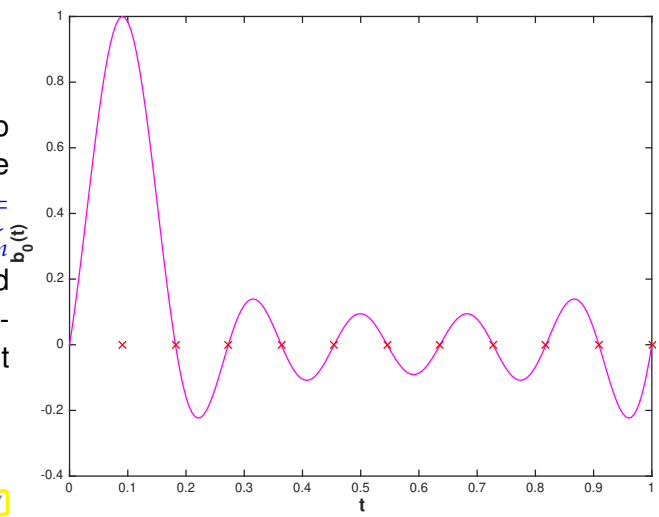


Fig. 17

C++11-code 6.5.5: Code.

```
// Basis vector e_1
ArrayXd e = ArrayXd::Zero(2*n+1);
e(0) = 1;
VectorXd y;
trigpolyvalequid(e, 1e3, y);

ArrayXd t = ArrayXd::LinSpaced(M, 0, 1);

// Shift function right a bit
ArrayXd y_shift(M);
unsigned int h = M / (2*n+1);
y_shift << y.tail(h), y.head(M - h);

mgl::Figure fig;
fig.title("b_0(t)");
fig.xlabel("t");
fig.ylabel("y");
fig.plot(t, y_shift, "r").label("b_0(t)");
fig.legend();
fig.save("b0_n");
```

▲

(6-5.d) ⚁ Show that

$$b_{k+1} = \mathsf{S}_\delta b_k \quad \text{with} \quad \delta := \frac{1}{2n+1}\,. \tag{6.5.6}$$

HINT 1 for (6-5.d): Verify (6.5.4) for $\mathsf{S}_\delta b_k$ ⌟

SOLUTION of (6-5.d):

We have (use periodicity and equidistance of the points):

$$\mathsf{S}_\delta b_k(t_j) = b_k(t_j - \frac{1}{2n+1}) = b_k(\frac{(j-1)+1/2}{2n+1}) = b_k(t_{j-1}) = \delta_{k,j-1} = \delta_{k+1,j} = b_{k+1}(t_j).$$

The claim follows from uniqueness of interpolating trigonometric polynomials.

▲

(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 $\mathcal{B}_{\exp}$ of $\mathcal{P}_{2n}^T$.

HINT 1 for (6-5.e): Remember the Fourier matrix [Lecture → Eq. (4.2.0.13)]. ⌟

HINT 2 for (6-5.e): Examine [Lecture → Eq. (5.6.3.2)]. However, take into account that the interpolation nodes for this problem are slightly different from those used in those formulas. ⌟

SOLUTION of (6-5.e):

The entries are

$$
\begin{aligned}
(\mathbf{A})_{i,j} &= B_{j-n-1}(t_i) \\
&= \exp(2\pi\imath(j-n-1)\frac{i+1/2}{2n+1}) \\
&= \exp(2\pi\imath i\frac{j-n-1}{2n+1})\exp(\pi\imath\frac{j-n-1}{2n+1}), \\
&= \begin{cases} (\mathbf{F}_{2n+1})_{i,j+n} & j \le n \\ (\mathbf{F}_{2n+1})_{i,j-n} & j > n \end{cases} \exp(\pi\imath\frac{j-n-1}{2n+1}).
\end{aligned}
$$

▲

(6-5.f) ⚁ What is the inverse of the interpolation matrix found in the previous problem?

HINT 1 for (6-5.f): Recall [Lecture → Lemma 4.2.0.14].

SOLUTION of (6-5.f):

We want to write $\mathbf{A}$ in terms of $\mathbf{F}_{2n+1}$, since $\mathbf{F}_{2n+1}^{-1} = \frac{1}{2n+1}\mathbf{F}_{2n+1}^H$ is easy to compute. Notice that $\mathbf{A}$ is almost $\mathbf{F}_{2n+1}$ expect fow a column shift (expressed by multiplication of a permuation matrix $\mathbf{P}$) and rescaling (expressed by multiplication with diagonal matrix $\mathbf{d}$. (Notice that $\mathbf{P}$ is orthogonal and $|d_j| = 1$)

$$\mathbf{A} = \mathbf{F}_{2n+1} \cdot \mathbf{P} \cdot \operatorname{diag}(\mathbf{d}).$$

Hence:

$$
\begin{aligned}
\mathbf{A}^{-1} &= \operatorname{diag}(\mathbf{d})^{-1}\mathbf{P}^{-1}\frac{1}{2n+1}\mathbf{F}_{2n+1}^H \\
\mathbf{A}^H &= \operatorname{diag}(\mathbf{d})^H\mathbf{P}^H\mathbf{F}_{2n+1}^H \\
\operatorname{diag}(\mathbf{d}^H)^{-1}\mathbf{P}^{-1^H}\mathbf{A}^H &= \mathbf{F}_{2n+1}^H \\
\mathbf{A}^{-1} &= \frac{1}{2n+1}(\operatorname{diag}(\mathbf{d})\operatorname{diag}(\mathbf{d}^*))^{-1}\mathbf{P}^{-1}(\mathbf{P}^{-1})^T\mathbf{A}^H = \frac{1}{2n+1}\mathbf{A}^H.
\end{aligned}
$$

▲

(6-5.g) ⚂ Give a closed-form expression for the cardinal basis functions b_k defined in (6.5.4).

HINT 1 for (6-5.g): Use that b_k has the coefficient vector $\mathbf{X}\mathbf{e}_{k+1}$ with respect to $\mathcal{B}_{\exp}$, where $\mathbf{e}_j \in \mathbb{C}^{2n+1}$, $j = 1,\dots,2n+1$, is the j-th unit vector, $\mathbf{X}$ the matrix found in (6-5.f).

SOLUTION of (6-5.g):

See P. D. MORRIS AND E. W. CHENEY, *Stability properties of trigonometric interpolating operators*, Math. Z., 131 (1973), pp. 153–164.

Let $\mathbf{c}$ be the coefficient vector of b_k w.r.t. $\mathcal{B}_{exp}$. We have

$$\mathbf{c} = \mathbf{A}^{-1}\mathbf{e}_{k+1} = \frac{1}{2n+1}\mathbf{A}^H\mathbf{e}_{k+1} = \frac{1}{2n+1}\left(\exp(-2\pi\imath(k+1)\frac{j}{2n+1})\exp(-\pi\imath\frac{j}{2n+1})\right)_{j=-n}^{n}.$$

Expressing b_k with the basis:

$$b_k(t) = \sum_{j=-n}^{n} c_j B_j(t) = \sum_{j=-n}^{n} \frac{1}{2n+1}(\exp(2\pi\imath(k+1)\frac{j}{2n+1})\exp(-\pi\imath\frac{j}{2n+1})\exp(-2\pi j\imath t)$$
$$= \frac{1}{2n+1}\sum_{j=-n}^{n}(\exp(2\pi\imath(k+1/2)\frac{j}{2n+1} - 2\pi j\imath t)$$

Using Euler's formula

$$(n+\tfrac{1}{2})b_k(t) = \tfrac{1}{2} + \sum_{j=1}^{n}\cos(2\pi j(t-t_k))\,, \quad t \in \mathbb{R}\,. \tag{6.5.7}$$

This can be simplified further using the geometric sum formula and a few manipulations:

$$\begin{aligned}
&\tfrac{1}{2} + \sum_{j=1}^{n}\cos(2\pi j(t-t_k)) \\
=&Re(\tfrac{1}{2} + \sum_{j=1}^{n}\exp(2\pi\imath j(t-t_k))) \\
=&Re(-\tfrac{1}{2} + \frac{1-\exp(2\pi\imath(n+1)(t-t_k))}{1-\exp(2\pi\imath(t-t_k))}) \\
=&Re(-\tfrac{1}{2} + \frac{\exp(-\pi\imath(t-t_k)) - \exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))}) \\
=&Re(-\tfrac{1}{2}\frac{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k)) - 2\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))} - \frac{\exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-} \\
=&Re(\tfrac{1}{2}\frac{\exp(-\pi\imath(t-t_k)) + \exp(\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))} - \frac{\exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\exp(-\pi\imath(t-t_k)) - \exp(\pi\imath(t-t_k))}) \\
=&\tfrac{1}{2}Re(\frac{\cos(\pi\imath(t-t_k))}{-\imath\sin(\pi\imath(t-t_k))}) - Re(\frac{\exp(2\pi\imath(n+1)(t-t_k))\exp(-\pi\imath(t-t_k))}{\imath\sin(\pi\imath(t-t_k))}) \\
=&\frac{\sin(2\pi\imath(n+1)(t-t_k) - \pi\imath(t-t_k))}{\sin(\pi\imath(t-t_k))} \\
=&\frac{\sin(2\pi\imath(n+1/2)(t-t_k))}{\sin(\pi\imath(t-t_k))}
\end{aligned}$$

Hence:

$$(n+\tfrac{1}{2})b_k(t) = \frac{\sin(2\pi(n+\frac{1}{2})t)}{2\sin(\pi t - t_k)}\,, \quad t \in \mathbb{R}\,. \tag{6.5.8}$$

▲

(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

$$\lambda(n) = \sup_{\mathbf{y}\in\mathbb{C}^{2n+1}\setminus\{\mathbf{0}\}} \frac{\|\mathsf{I}_n\mathbf{y}\|_{L^\infty([0,1])}}{\|\mathbf{y}\|_\infty} = \sup_{t\in[0,1]}\sum_{k=0}^{2n}|b_k(t)|\,, \tag{6.5.9}$$

where $\mathsf{I}_n : \mathbb{C}^{2n+1} \to C^0(\mathbb{R})$ is the trigonometric interpolation operator.

Use sampling in 10^4 equidistant points to obtain a good guess for the supremum norm of a function on $[0,1]$. Maximum efficiency of the implementation need not be a concern. Use the formula involving sum of cosines.

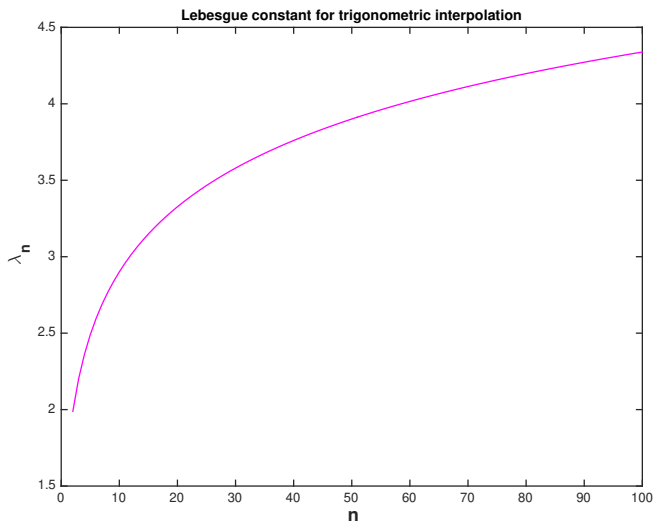
SOLUTION of (6-5.h):

C++11-code 6.5.10: Implementation of function λ.

```
double trigIpL(std::size_t n) {
    double ret = 0;

    // Nodes where to evaluate
    ArrayXd t = ArrayXd::LinSpaced(1e4, 0, 1);
    // Will contain value of funtion at all nodes
    ArrayXd s = ArrayXd::Zero(t.size());
    // Sum over all basis functions
    for(unsigned int k = 0; k <= 2*n; ++k) {
        // Compute sum of cosines
        ArrayXd b = ArrayXd::Constant(t.size(), 0.5);
        for(unsigned int j = 1; j <= n; ++j) {
            double tk = (k + 0.5) / (2*n+1.);
            b += cos(2*M_PI*j*(t-tk));
        }
        s += b.cwiseAbs();
    }
    // Find max and rescale
    return s.maxCoeff() / (n + 1/2.);
}
```

Graph of $n \mapsto \lambda(n)$



(6-5.i) The following table gives the values $\lambda(n)$ for $n = 2^k$, $k = 2, 3, \ldots, 8$.

$^k k$	$\lambda(2^k)$
4	2.7
8	3.07
16	3.46
32	3.89
64	4.32
128	4.75
256	5.18
512	5.62

From these numbers predict the asymptotic behavior of $\lambda(n)$ for $n \to \infty$. Is it $O(\log n)$, $O(n)$, $O(n \log n)$, or $O(n^2)$?

SOLUTION of (6-5.i):

The asymptotic behavior of $\lambda(n)$ is $O(logn)$, see E. W. CHENEY AND T. J. RIVLIN, *A note on some Lebesgue constants*, Rocky Mountain J. Math., 6 (1976), pp. 435–439.

▲

End Problem 6-5

Problem 6-6: Quadratic Splines

[Lecture → Def. 5.5.0.1] 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 5.5.1] 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 5.5.1] is required. This exercise involves substantial implementation in C++ and EIGEN.

Consider a 1-periodic function $f : \mathbb{R} \to \mathbb{R}$, that is, $f(t+1) = f(t)$ for all $t \in \mathbb{R}$, and a set of knots

$$\mathcal{M} := \{0 = t_0 < t_1 < t_2 < \cdots < t_{n-1} < t_n = 1\} \subset [0,1] \,.$$

We want to approximate f using a *1-periodic* quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$, which interpolates f *in the midpoints* of the intervals $[t_{j-1}, t_j]$, $j = 0, \dots, n$.

In analogy to the local representation of a cubic spline function according to [Lecture → Eq. (5.5.1.4)], we parametrize a quadratic spline function $s \in \mathcal{S}_{2,\mathcal{M}}$ according to

$$s|_{[t_{j-1},t_j]}(t) = d_j\,\tau^2 + c_j\,4\,\tau(1-\tau) + d_{j-1}\,(1-\tau)^2 \,, \quad \tau := \frac{t - t_{j-1}}{t_j - t_{j-1}} \,, \quad j = 1, \dots, n \,, \tag{6.6.1}$$

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 $t = t_{j-1}, t_j, \frac{1}{2}(t_{j-1} + t_j)$?

HINT 1 for (6-6.a): The coefficients d_k "are associated with" the nodes t_k while the c_j are associated with the middle points of the intervals (t_{j-1}, t_j). ⌟

SOLUTION of (6-6.a):

By straightforward computations we find:

$$s(t_j) = d_j \quad , \quad s(t_{j-1}) = d_{j-1} \quad , \quad s(\tfrac{1}{2}(t_{j-1} + t_j)) = c_j + \tfrac{1}{4}(d_{j-1} + d_j) \,.$$

Hence the coefficients d_j agree with the (unique!) value of the quadratic spline s at the knots t_j.

▲

(6-6.b) ⊡ (5 min.) What is the dimension of the subspace of *1-periodic* spline functions in $\mathcal{S}_{2,\mathcal{M}}$?

HINT 1 for (6-6.b): 1-periodicity means that $s(0) = s(1)$ and $s'(0) = s'(1)$. ⌟

SOLUTION of (6-6.b):

We have $n+1$ unknowns d_k and n unknowns c_j, the constraint $d_0 = s(t_0) = s(t_0 + 1) = s(t_n) = d_n$ leaves us with a total of $2n$ unknowns. The continuity of the derivatives in the n nodes impose the same number of constraints, therefore the total dimension of the spline space is $2n - n = n$.

▲

(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 of (6-6.c):

We observe that $s(t_j^-) = s|_{[t_{j-1},t_j]}(t_j) = d_j = s|_{[t_j,t_{j+1}]}(t_j) = s(t_j^+)$, thus we get continuity for free by fixing the coefficients d_j. However the derivatives do not necessarily match, which means that the representation (6.6.1) does not guarantee the continuity of the derivative.

▲

(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 $y_j := f(\frac{1}{2}(t_{j-1} + t_j))$, $j = 1, \dots, n$.

HINT 1 for (6-6.d): By [Lecture → Def. 5.5.0.1] we know $\mathcal{S}_{2,\mathcal{M}} \subset C^1([0,1])$, which provides linear constraints at the nodes, analogous to [Lecture → Eq. (5.5.1.5)] for cubic splines. Also take into account 1-periodicity. ⌟

SOLUTION of (6-6.d):

We can plug $t = \frac{1}{2}(t_j + t_{j-1})$ into (6.6.1) and set the values equal to y_j. We obtain $\tau = 1/2$ and the following conditions:

$$\frac{1}{4}d_j + c_j + \frac{1}{4}d_{j-1} = y_j, \quad j = 1, ..., n. \tag{6.6.2}$$

We obtain conditions on d_j by matching the derivatives at the interfaces. The derivative of the quadratic spline can be computed from (6.6.1), after defining $\Delta_j = t_j - t_{j-1}$:

$$\begin{aligned} s'|_{[t_{j-1},t_j]}(t) &= \Delta_j^{-1}\frac{\partial}{\partial\tau}\Big(\tau^2 d_j + 4\tau(1-\tau)c_j + (1-\tau)^2 d_{j-1}\Big) \\ &= \Delta_j^{-1}\Big(2\tau d_j + 4(1-2\tau)c_j - 2(1-\tau)d_{j-1}\Big). \end{aligned}$$

Setting $\tau = 1$ in $[t_{j-1}, t_j]$ and $\tau = 0$ in $[t_j, t_{j+1}]$, the continuity of the derivative in the node t_j enforces the condition

$$\frac{2d_j - 4c_j}{\Delta_j} = s'|_{[t_{j-1},t_j]}(t_j) = s'(t_j^-) = s'(t_j^+) = s'|_{[t_{j-1},t_j]}(t_j) = \frac{4c_{j+1} - 2d_j}{\Delta_{j+1}};$$

(this formula holds for $j = 1, \dots, n$, if we define $t_{n+1} = t_1 + 1$ and $c_{n+1} = c_1$). Simplifying for d_j we obtain:

$$d_j = \frac{2\frac{c_j}{\Delta_j} + 2\frac{c_{j+1}}{\Delta_{j+1}}}{\frac{1}{\Delta_j} + \frac{1}{\Delta_{j+1}}} = 2\frac{c_j\Delta_{j+1} + c_{j+1}\Delta_j}{\Delta_j + \Delta_{j+1}} = 2\frac{c_j(t_{j+1} - t_j) + c_{j+1}(t_j - t_{j-1})}{t_{j+1} - t_{j-1}}, \tag{6.6.3}$$

for $j = 1, \dots, n$. Plugging this expression into (6.6.2), we get the following system of equations:

$$\frac{1}{2}\frac{c_{j+1}(t_j - t_{j-1}) + c_j(t_{j+1} - t_j)}{t_{j+1} - t_{j-1}} + c_j + \frac{1}{2}\frac{c_j(t_{j-1} - t_{j-2}) + c_{j-1}(t_j - t_{j-1})}{t_j - t_{j-2}} = y_j, \qquad j = 1, \dots, n, \tag{6.6.4}$$

with the periodic definitions $t_{-1} = t_{n-1} - 1$, $c_0 = c_n$, $c_{-1} = c_{n-1}$. We collect the coefficients and finally we obtain

$$\underbrace{\left\{\frac{1}{2}\frac{t_j - t_{j-1}}{t_j - t_{j-2}}\right\}}_{A_{j-1}} c_{j-1} + \underbrace{\left\{\frac{1}{2}\frac{t_{j+1} - t_j}{t_{j+1} - t_{j-1}} + 1 + \frac{1}{2}\frac{t_{j-1} - t_{j-2}}{t_j - t_{j-2}}\right\}}_{B_j} c_j + \underbrace{\left\{\frac{1}{2}\frac{t_j - t_{j-1}}{t_{j+1} - t_{j-1}}\right\}}_{C_{j+1}} c_{j+1} = y_j \,, \tag{6.6.5}$$

for $j = 1, \dots, n$. If we have an equidistant knot set with $t_j - t_{j-1} = 1/n$, this simplifies to

$$\frac{1}{4}c_{j-1} + \frac{3}{2}c_j + \frac{1}{4}c_{j+1} = y_j \,, \quad j = 1, \dots, n. \tag{6.6.6}$$

The system of equations in matrix form looks as follows:

$$\begin{bmatrix} B_1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & A_0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ C_{n+1} & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ y_{n-1} \\ y_n \end{bmatrix}. \tag{6.6.7}$$

▲

(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 of (6-6.e):

Denote by $\mathbf{M}$ the system matrix from (6.6.7). Then

$$\mathbf{M} = \begin{bmatrix} B_1 - 1 & C_2 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ A_1 & B_2 & C_3 & 0 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ 0 & A_2 & B_3 & C_4 & 0 & \cdots\cdots & 0 & 0 & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & & \ddots & \ddots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & A_{n-2} & B_{n-1} & C_n \\ 0 & 0 & 0 & 0 & 0 & \cdots\cdots & 0 & 0 & A_{n-1} & B_n - A_0 C_{n+1} \end{bmatrix} + \mathbf{u}\mathbf{v}^\top \,,$$

$$\text{with} \qquad \mathbf{u} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ C_{n+1} \end{bmatrix} \in \mathbb{R}^n \quad , \quad \mathbf{v} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ \vdots \\ 0 \\ A_0 \end{bmatrix} \in \mathbb{R}^n \, .$$

▲

(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 $n \to \infty$

SOLUTION of (6-6.f):

The linear system to be solved features a system matrix which is a rank-1 perturnation of a *tridiagonal* coefficient matrix $\mathbf{M} \in \mathbb{R}^{n,n}$. For $\mathbf{M}$ we can compute a QR-decomposition $\mathbf{M} = \mathbf{QR}$ with asymptotic effort $O(n)$ as explained in [Lecture → Rem. 3.3.3.21]. Applying the (transposed) Givens rotations to the right-hand side vector incurs $O(n)$ cost and so does back-substitution with the tridiagonal $\mathbf{R}$-factor. Summing up, solving a linear system with coefficient matrix $\mathbf{M}$ takes $O(n)$ operations.

Instead of a QR-decomposition you may also propose the simpler LU-decomposition, $\mathbf{M} = \mathbf{LU}$ which can be computed without pivoting (→ [Lecture → Lemma 2.8.0.9]) and with optimal asymptotic complexity $O(n)$ for $n \to \infty$, because $\mathbf{M}$ is tridiagonal.

Finally we use the formula [Lecture → Eq. (2.6.0.22)] with $\mathbf{A} \leftarrow \mathbf{M}$ and the vectors $\mathbf{u}$ and $\mathbf{v}$ found in Sub-problem (6-6.e). Besides computing $\mathbf{M}^{-1}\mathbf{y}$ and $\mathbf{M}^{-1}\mathbf{u}$ with effort $O(n)$ it merely involves cheap vector operations.

▲

(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 $[c_1, \dots, c_n]^\top \in \mathbb{R}^n$ when supplied a sorted knot vector `t` (of length $n-1$, because $t_0 = 0$ and $t_n = 1$ will be taken for granted), an n-vector `y` of data values y_j, $j = 1, \dots, n$, 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 of (6-6.g):

Use EIGEN's built-in *sparse solver* to solve (6.6.7). It will achieve optimal asymptotic computational complexity $O(n)$ for $n \to \infty$. The use of a sparse solver is mandatory.

C++ code 6.6.8: Implementation of `compute_c()`

```
VectorXd compute_c( const VectorXd &t, const VectorXd &y) {
  const unsigned int n = y.size(); //number of intervals
  VectorXd c(n);
  // TO DO (6-6.g) : Build the (sparse) matrix for spline interpolation
  // at midpoints. Then compute the coefficients c for the data y.

  //START
  assert(t.size() == n-1 && "number of intervals mismatch");
  assert(t.minCoeff() > 0 && t.maxCoeff() < 1
                 && "mesh nodes out of range");

  std::pair<VectorXd, VectorXd> p = increments(t);
  VectorXd dt = p.first;
  VectorXd ddt = p.second;

  //Note: A_0 = A_n and C_1 = C_{n+1}
  VectorXd A = dt.tail(n).cwiseQuotient(2 * ddt.tail(n));
  VectorXd C = dt.head(n).cwiseQuotient(2 * ddt.head(n));

  std::vector< Triplet<double> > triplets(3*n);

  //Build matrix as triplets
  for (unsigned int j = 0; j < n - 1 ; ++j ) {
    triplets.push_back(Triplet<double>(j, j, A(j) + C(j) + 1 ));
    triplets.push_back(Triplet<double>(j , j + 1, C(j+1) ));
    triplets.push_back(Triplet<double>(j + 1, j, A(j)  ));
  }
  triplets.push_back(Triplet<double>(n - 1, n - 1, A(n - 1) + C(n - 1) + 1 ));
  triplets.push_back(Triplet<double>(n - 1, 0, C(0) ));
  triplets.push_back(Triplet<double>(0, n - 1, A(n - 1)));

  SparseMatrix<double> M(n,n);
  M.setFromTriplets(triplets.begin(), triplets.end());
  M.makeCompressed();
  //solve linear system
  SparseLU<SparseMatrix<double>> solver;
  solver.compute(M);
  c = solver.solve(y);
  //END
  return c;
}
```

You need not use an intermediate COO-format, because you know in advance that the system matrix sports at most three non-zero entries per row. This paves the way for efficient initialization after a `reserve()` call, see [Lecture → Code 2.7.2.1].

▲

(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 of (6-6.h):

We can simply rely on the formula (6.6.3).

C++ code 6.6.9: Implementation of `compute_d()`

```
VectorXd compute_d ( const VectorXd &c, const VectorXd &t ) {
  const unsigned int n = c.size(); //number of intervals
  VectorXd d_ext(n + 1);
  //TO DO (): compute coefficients d_j for j = 0...n
  //Hint: periodic conditions give d_0 = d_n
  //START
  std::pair<VectorXd, VectorXd> p = increments(t);
  VectorXd dt = p.first;
  VectorXd ddt = p.second;

  //extend c periodically for vectorization
  VectorXd c_ext(c.size() + 1);
  c_ext << c , c(0);

  //coefficients d_j for j = 1...n
  VectorXd d = c.cwiseProduct(dt.tail(n)) +
                c_ext.tail(n).cwiseProduct(dt.segment(1,n));
  d = 2 * d.cwiseQuotient(ddt.tail(n));
  //extend d by periodic condition
  d_ext << d(n - 1), d;
  //END
  return d_ext;
}
```

▲

(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 of (6-6.i):

We use an auxiliary function that,among others, computes the length of knot intervals:

C++ code 6.6.10: Implementation of `increments()`

```
std::pair<VectorXd, VectorXd> increments(const VectorXd &t) {
  // TO DO (6-6.g) : compute the increments t_j - t_{j-1}
  // for j = 0,...,n+1 and t_{j+1} - t_{j-1} for j = 0,...,n
  // Assume that t is sorted and that t does not include the endpoints.
  // Hint: use periodic conditions to define t_{-1}, t_{n+1} for vectorization.

  //START
  // Create extended vector using the periodicity:
  const unsigned int n = t.size() + 1; //number of intervals
  VectorXd ext_t(n+3);
  ext_t << t(n-2) - 1, 0, t, 1, 1 + t(0) ;
  //Increments in t
  VectorXd dt = ext_t.tail(n+2).array() - ext_t.head(n+2).array();
  VectorXd ddt = ext_t.tail(n+1).array() - ext_t.head(n+1).array();
  //END

  return std::make_pair(dt,ddt);
}
```

Thanks to the ordering of the knots and of the evaluation points, we just need to process both arrays sequentially. Once we have found the know interval in which an evaluation point is located, we can simply use the representation (6.6.1).

C++ code 6.6.11: Implementation of `quadspline()`

```
VectorXd quadspline(const VectorXd &t, const VectorXd &y, const VectorXd &x) {
  VectorXd fval(x.size());

  //TO DO (6-6.h): evaluate the spline at the points defined in x.
  // Assume that x is sorted.
  assert(x.minCoeff() >= 0 && x.maxCoeff() <= 1
                && "evaluation samples out of range");

  //START
  VectorXd c = compute_c(t, y);
  const unsigned int n = y.size(); //number of intervals

  std::pair<VectorXd,VectorXd> p = increments(t);
  VectorXd dt = p.first;
  VectorXd ddt = p.second;

  VectorXd d_ext = compute_d(c, t);
  //extend t for vectorization
  VectorXd t_ext(n + 1);
  t_ext << 0, t, 1;

  unsigned int i = 0;
  //loop over intervals
  for (unsigned int j = 1; j <= n; ++j) {
    //loop over sample points in the interval
    while ( i < x.size() && x(i) <= t_ext(j) ) {
      double tau = (x(i) - t_ext(j - 1)) / dt(j);
      double uat = 1 - tau;
      fval(i) = d_ext(j) * tau * tau + 4 * c(j - 1) * tau * uat
                        + d_ext(j - 1) * uat * uat;
      ++i;
    }
```

```
    if (i == x.size()) {break;}
  }
  //END
  return fval;
}
```

▲

(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 $s(\frac{1}{2}(t_j + t_{j-1})) = f(\frac{1}{2}(t_j + t_{j-1})$, $j = 1, \dots, 10$. 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 of (6-6.j):

C++ code 6.6.12: Implementation of `plotquadspline()`

```
void plotquadspline( const std::string &filename ) {

  VectorXd mesh = VectorXd::LinSpaced(9,.1,.9);
  auto f = [] (VectorXd t) {
    return (2 * M_PI * t).array().sin().exp();
  };
  // TO DO (6-6.i) : plot the quadratic spline for the function f based on
  // the intervals defined in t. Plot also the data (interpolation) points.

  //START
  plt::figure();
  // The interpolation points are the midpoints of the intervals
  VectorXd t = VectorXd::LinSpaced(10, .05, .95) ;
  VectorXd y = f(t);
  // plot data points
  plt::plot(t, y, "o",{{"label","data"}} );

  // std::cout << y <<std::endl;
  VectorXd x = VectorXd::LinSpaced(200,0,1);
  VectorXd spline_val = quadspline(mesh, y, x);
  // std::cout << spline_val <<std::endl;

  plt::plot(x, spline_val, {{"label","spline"}} );
  plt::title("Quadratic spline");
  plt::xlabel("t");
  plt::ylabel("s(t)");

  plt::savefig("./cx_out/" + filename + ".png");
  //END
}
```

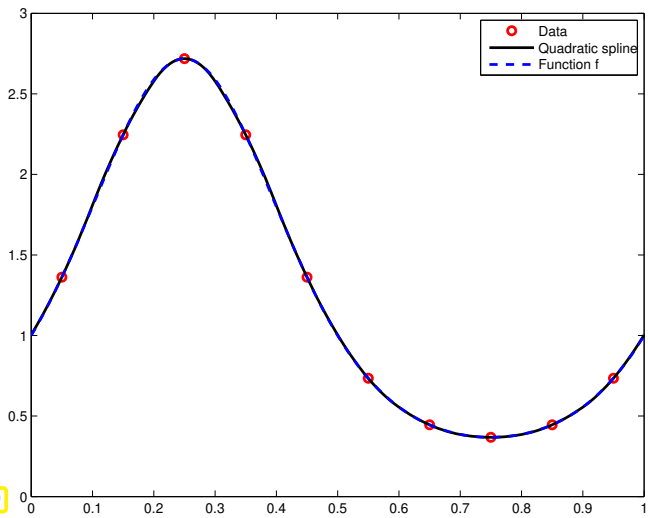


Fig. 19

◁ plot of the quadratic spline s.

▲

(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 $n \to \infty$.

SOLUTION of (6-6.k):

n	E_n
2	0.606776
4	0.434326
8	0.0402719
16	0.00253016
32	0.000261929
64	3.13621e-05
128	3.87463e-06
256	4.8303e-07

C++ code 6.6.13: Implementation of `qsq_error()`

```
std::vector<double> qsp_error(unsigned int q) {

  std::vector<double> Err;
  unsigned int N = 10000;
```

```
  VectorXd x = VectorXd::LinSpaced(N, 0, 1);
  auto f = [] (VectorXd t) {
    return (2 * M_PI * t).array().sin().exp();
  };
  //TO DO (6-6.j) : compute L^infty errors for all n = 2, 4, ..., 2^q
  //START
  unsigned int n_max = std::pow(2,q);

  for(unsigned int n = 2; n <= n_max; n *= 2 ) {
    VectorXd t = VectorXd::LinSpaced(n - 1, 1.0/n, 1.0 - 1.0/n);
    // The interpolation points are the midpoints of the intervals
    VectorXd interp = VectorXd::LinSpaced(n, .5/n, 1.0 - .5/n);
    VectorXd y = f(interp);
    VectorXd f_spline = quadspline(t, y, x);
    VectorXd f_exact = f(x);
    // compute max error
    Err.push_back( (f_spline - f_exact).cwiseAbs().maxCoeff() );
  }
  //END
  return Err;
}
```

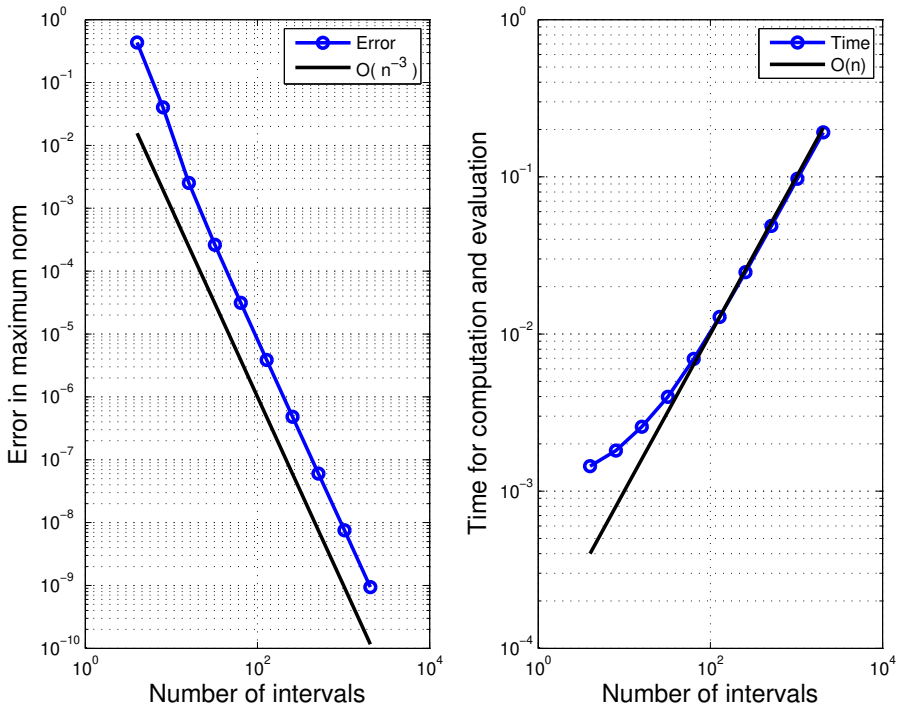


Fig. 20

◁ plot of errors E_n and runtimes as a function of the number n of knots.

In this *doubly logarithmic* plot the data points (n, E_n) are neatly aligned. This indicates asymptotic algebraic convergence $O(n^{-3})$, the rate can be obtained from the slopes of the lines.

▲

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 5.4.2], [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 5.4.2].

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 $I_{\mathcal{T}} : \mathbb{R}^{n+1} \to C^0([t_0, t_n])$, 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 $y_j \le y_{j+1}$, $\forall\, j = 0, \dots, n-1$, 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:

$$\big(I_{\mathcal{T}}(\mathbf{y})\big)'(t_j) = 0\,, \qquad \forall j \in \{1, \dots, n-1\}\,, \quad \forall \mathbf{y} \in \mathbb{R}^{n+1}\,. \tag{6.7.2}$$

(6-7.a) ⚁ (15 min.) Show that the assertion (6.7.2) holds for the special data vectors

$$\mathbf{y} = \mathbf{y}^{(j)} := [\underbrace{0, \dots, 0}_{j \text{ times}}, 1, \dots, 1]^\top\,, \quad j = 0, \dots, n\,.$$

HINT 1 for (6-7.a): Recall the considerations in the beginning of [Lecture → Section 5.4.2].

SOLUTION of (6-7.a):

- The data $\mathbf{y}^{(j)}$ are increasing and, owing to monotonicity preservation, so must be the interpolant $I_{\mathcal{T}}(\mathbf{y}^{(j)})$.
- We have $\left(\mathbf{y}^{(j)}\right)_1 = \dots = \left(\mathbf{y}^{(j)}\right)_j = 0$ and $\left(\mathbf{y}^{(j)}\right)_{j+1} = \dots = \left(\mathbf{y}^{(j)}\right)_{n=1} = 1$. Thus, the function $I_{\mathcal{T}}(\mathbf{y}^{(j)})$ is constant in $[t_0, t_{j-1}]$ and in $[t_j, t_n]$. Since it is also C^1-smooth, its derivative is identically zero in the two intervals (every time the interval itself does not degenerate into a point).
- All the points t_k, with $k = 1, \dots, n-1$, belong to one of these two intervals, therefore $\big(I_{\mathcal{T}}(\mathbf{y}^{(j)})\big)'(t_k) = 0$.

The derivative of $I_{\mathcal{T}}(\mathbf{y}^{(1)})$ is not necessarily zero in t_0 as well as the derivative of $I_{\mathcal{T}}(\mathbf{y}^{(n)})$ in t_n.

▲

(6-7.b) ⚁ (10 min.) Prove that $\operatorname{span}\{\mathbf{y}^{(j)},\ j=0,\dots,n\} = \mathbb{R}^{n+1}$.

SOLUTION of (6-7.b):

Every Cartesian coordinate vector of $\mathbb{R}^{n+1}$ can be written as a linear combination of the $\mathbf{y}^{(j)}$s:

$$\mathbf{e}^{(j)} := (\underbrace{0,\dots,0}_{j},1,0,\dots,0)^T = \mathbf{y}^{(j)} - \mathbf{y}^{(j+1)}, \quad \forall\, j=0,\dots,n-1, \quad \mathbf{e}^{(n)} := (\underbrace{0,\dots,0}_{n},1)^T = \mathbf{y}^{(n)}\,,$$

$$\Rightarrow \quad \mathbb{R}^{n+1} = \operatorname{span}\{\mathbf{e}^{(j)},\ j=0,\dots,n\} \subseteq \operatorname{span}\{\mathbf{y}^{(j)},\ j=0,\dots,n\} \subseteq \mathbb{R}^{n+1}\,,$$

$$\Rightarrow \quad \operatorname{span}\{\mathbf{y}^{(j)},\ j=0,\dots,n\} = \mathbb{R}^{n+1}\,.$$

▲

(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 of (6-7.c):

Thanks to the result of Sub-problem (6-7.b), every $\mathbf{y} \in \mathbb{R}^{n+1}$ can be decomposed as $\mathbf{y} = \sum_{j=0}^{n} c_j \mathbf{y}^{(j)}$. Therefore, the derivatives in the nodes are

$$\big(I_{\mathcal{T}}(\mathbf{y})\big)'(t_k) \overset{\text{linearity of } I_{\mathcal{T}}}{=} \sum_{j=0}^{n} c_j \big(I_{\mathcal{T}}(\mathbf{y}^{(j)})\big)'(t_k) \overset{\text{Sub-problem (6-7.a)}}{=} 0\,, \quad k=1,\dots,n-1\,.$$

▲

End Problem 6-7 , 40 min.

Problem 6-8: Approximating π by extrapolation

In [Lecture → Section 5.2.3.3] we learned about the approximation of a limit $\lim_{h\to 0}\psi(h)$ based on the evaluation of the function ψ "far away" from 0. In this exercise we will practice the underlying technique of "Lagrangian polynomial extrapolation" for a geometric approximation problem.

Study [Lecture → Section 5.2.3.3] and, in particular [Lecture → Code 5.2.3.20], 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

$$x^* = \lim_{n\to\infty} T(n) \ ,$$

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

$$g : (0, 1/n_{\min}] \mapsto \mathbb{R}, \quad \text{for which, obviously} \quad x^* = \lim_{h\to 0} g(h) \ .$$

Secondly, the function g is approximated by an interpolating polynomial $p_k \in \mathcal{P}_k$ with $p_k(n_j^{-1}) = T(n_j)$, $j = 0, \dots, k$. In many cases we can expect that $p_k(0)$ will provide a good approximation for x^*.

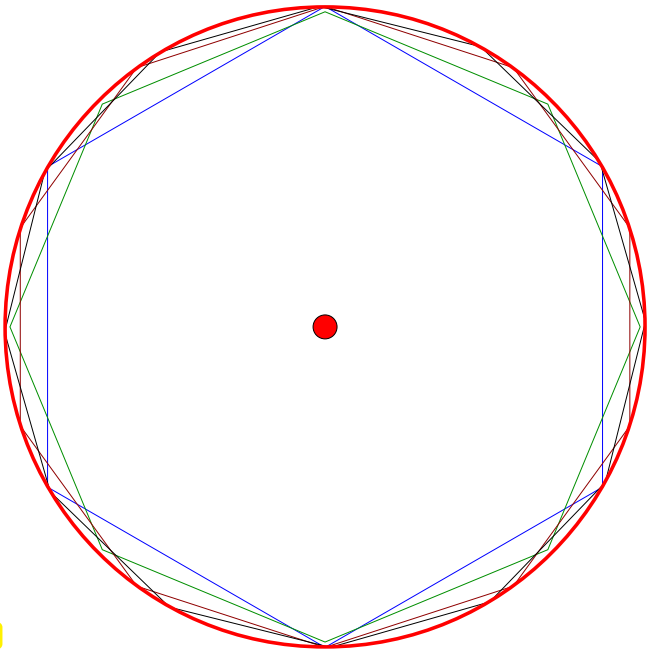


Fig. 21

The unit circle can be approximated by inscribed regular polygons with k edges. Thus, the length of half circumference (i.e., π) can be approximated by the half the length of the perimeters s_k of such polygons. These values $s_k/2$ can be calculated by elementary trigonometry and a closed form formula is

$$\tfrac{1}{2}s_k = k\sin(\pi/k) \ . \tag{6.8.1}$$

(6-8.a) ⚁ (10 min.) Show that for any $L \in \mathbb{N}$ and for $k \to \infty$

$$\tfrac{1}{2}s_k = \pi + \sum_{\ell=1}^{L} c_\ell k^{-2\ell} + O(k^{-2(L+1)}) \quad \text{for some} \quad c_\ell \in \mathbb{R} \quad \text{independent of} \quad L \ . \tag{6.8.2}$$

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}.

HINT 1 for (6-8.a): Remember the power series for sin:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots = \sum_{\ell=0}^{\infty} \frac{(-1)^\ell x^{2\ell+1}}{(2\ell+1)!}\,, \quad x \in \mathbb{R}\,. \tag{6.8.3}$$

⌟

SOLUTION of (6-8.a):

Using the power series for sin (6.8.3) we obtain

$$k\sin(\pi/k) = \pi + \sum_{\ell=1}^{\infty} \frac{(-1)^\ell \pi^{2\ell+1} k^{-2\ell}}{(2\ell+1)!}\,.$$

Since this series converges absolutely for all k, we have

$$\sum_{\ell=L+1}^{\infty} \frac{(-1)^\ell \pi^{2\ell+1} k^{-2\ell}}{(2\ell+1)!} \le C k^{-2(L+1)} \quad \text{for} \quad C > 0 \quad \text{independent of} \quad k\,.$$

▲

(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 π 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 of (6-8.b):

C++ code 6.8.4: Implementation of `extrapolate_to_pi()`

```
double extrapolate_to_pi(unsigned int k) {
  double pi=0;
  // TO DO (6-8.b): Use the Aitken-Neville scheme to approximate
  // pi by extrapolation.
  // Hint: You can use the constant M_PI and Eigen's sin(),
  // when calculating s_j for j = 2,...,k.
  // START

  // Data for interpolation
  ArrayXd J = ArrayXd::LinSpaced(k-1,2,k);
  ArrayXd T = 1.0/J; // t_j = 1/j
  ArrayXd S = J*sin(M_PI*T);

  // Aitken-Neville scheme for evaluating the interpolating
  // polynomial of (t_j,s_j) at x = 0.
  for (int l = 0; l < S.size(); l++) {
    for (int m = l-1; m >= 0; m--) {
      S(m) = S(m+1) + (S(m+1) - S(m))*(0.0 - T(l))/(T(l) - T(m));
    }
  }
  // Now S(0) has contributions from all data points.
  pi = S(0);
  // END
```

```
  return pi;
}
```

▲

(6-8.c) ⚀ (20 min.) Write a C++ function

```
void plotExtrapolationError(void);
```

that generates a *linear-logarithmic* plot of the extrapolation errors

$$\mathrm{err}(k) := |\pi - p_k(0)| \,, \quad k = 2, \ldots, 10 \,,$$

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 of (6-8.c):

C++ code 6.8.5: Implementation of `extrapolate_to_pi()`

```
void plotExtrapolationError(void) {
  plt::figure();
  // TO DO (6-8.c): Plot the error made by extrapolate_to_pi(k) for
  // k = 2,3,...,10. Use a log-scale for the y-axis, and a linear x-axis.
  // Also, tabulate the results an errors of extrapolate_to_pi().
  // Hint: Use the constant M_PI to compute the error made by extrapolate_to_pi().
  // Hint: matplotlibcpp (here = plt) implements Python's
  // matplotlib functions such as figure(), plot(), xlabel(), title(),...
  // You can use main.cpp of the LinearDataFit problem as a reference.
  // START
  int kmax = 10;
  VectorXd K = VectorXd::LinSpaced(kmax-1,2,kmax);
  VectorXd Err(K.size());
  std::cout << std::setw(5) << " k" << std::setw(15) << "approx. pi" <<
      std::setw(15) << "error" << std::endl;
  for(int l=0; l < K.size(); l++) {
    double pi = extrapolate_to_pi( K(l) );
    Err(l) = std::abs(pi - M_PI);
    std::cout << std::setw(5) << K(l) << std::setw(15) << pi << std::setw(15) <<
        Err(l) << std::endl;
  }
  // Number of data points is k-1, so polynomial degree is k-2.
  VectorXd Deg = K - 2*VectorXd::Ones(kmax-1);
  plt::semilogy(Deg, Err);
  plt::title("Approximation of pi");
  plt::xlabel("Polynomial degree");
  plt::ylabel("Error");
  plt::grid();
  // END
  plt::savefig("./cx_out/pi_error.png");
}
```

The tabulated errors should be:

```
 num. data points, approximation of pi, error:
1                 2.00000000          1.1415926
2                 3.79422863          0.6526359
3                 3.24473110          0.1031384
4                 3.12532464          0.0162680
5                 3.14048823          0.0011044
6                 3.14167629          0.0000836
7                 3.14159488          0.0000022
```

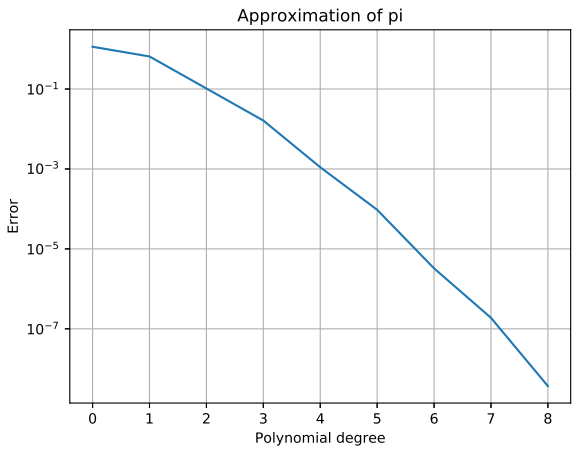


Fig. 22

Linear-logarithmic plot of error in the approximation of π by extrapolation.

▲

(6-8.d) ⚁ (10 min.) [depends on Sub-problem (6-8.c)]

Which kind of convergence of $\mathrm{err}(k) \to 0$ for $k \to \infty$ can you conclude from the data generated in Sub-problem (6-8.c)? The main types of asymptotic convergence to 0 are introduced in [Lecture → Def. 6.1.2.7].

HINT 1 for (6-8.d): Study [Lecture → Rem. 6.1.2.9] to learn how to determine the type of convergence from error data. ⌟

SOLUTION of (6-8.d):

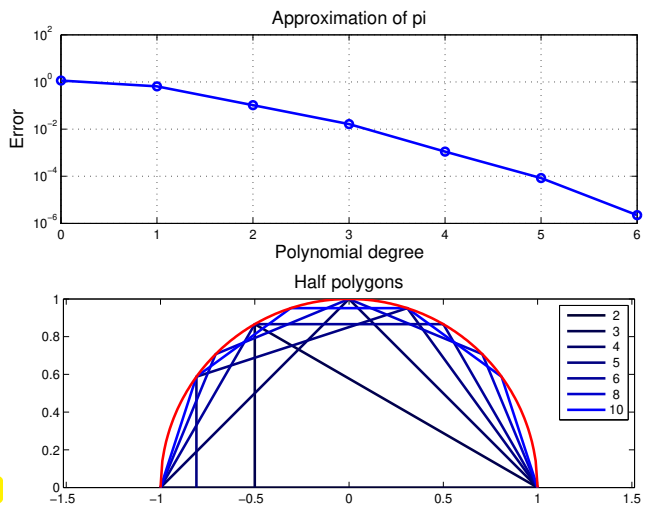


Fig. 23

The convergence is exponential with respect to the degree k of the polynomial used for extrapolation. We deduce this from the observation that the points in the error plot are nearly located on a line.
At second glance, the error points are located on a slightly downward-bending curve, which even hints at superexponential convergence, that is, faster than exponential.

▲

(6-8.e) ⚁ Modify your implementation of `plotExtrapolationError()` from Sub-problem (6-8.c) such that it plots and tabulates err_k for k up to 30. Describe and explain your observations.

SOLUTION of (6-8.e):

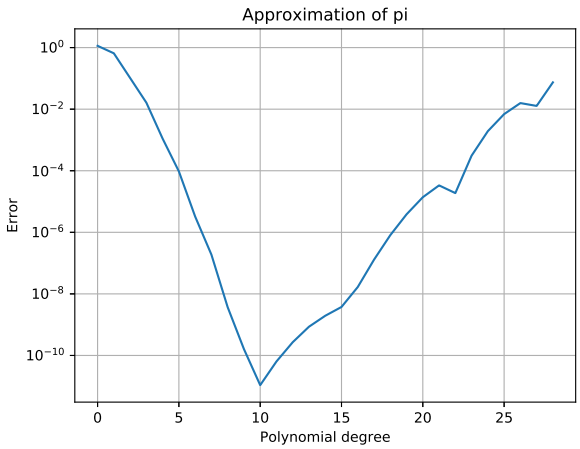


Fig. 24

For large k the error starts increasing again!
Two possible explanations:

1. High-degree polynomial interpolants display strong oscillations between the data points, which affect $p(0)$.
2. The Aitken-Neville algorithm is based on the computation of differences of similar quantities, therefore, ultimately, it will be vulnerable to cancellation errors.

Both effects may play a role here.

▲

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 5.5].

For a mesh $\mathcal{M} := \{a = t_0 < t_1 < \ldots < t_{n-1} < t_n = b\}$ we consider the following three spaces of piecewise polynomials:

$$V_{-1} := \left\{ v : [a,b] \to \mathbb{R} : \; v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\, j = 1,\ldots,n \right\} , \tag{6.9.1}$$

$$V_0 := \left\{ v \in C^0([a,b]) : \; v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\, j = 1,\ldots,n \right\} , \tag{6.9.2}$$

$$V_1 := \left\{ v \in C^1([a,b]) : \; v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\, j = 1,\ldots,n \right\} , \tag{6.9.3}$$

$$V_2 := \left\{ v \in C^2([a,b]) : \; v|_{[t_{j-1},t_j[} \in \mathcal{P}_2,\, j = 1,\ldots,n \right\} . \tag{6.9.4}$$

Here, $\mathcal{P}_d$ stands for the space of polynomials $\mathbb{R} \to \mathbb{R}$ of degree $\leq d$.

(6-9.a) ⚀ (8 min.) Determine the dimensions of the four spaces defined above:

$$\dim V_{-1} = \boxed{} ,$$
$$\dim V_0 = \boxed{} ,$$
$$\dim V_1 = \boxed{} ,$$
$$\dim V_2 = \boxed{} .$$

SOLUTION of (6-9.a):

- The space V_{-1}has no continuity constraints at the nodes of the mesh. Thus, the dimensions of the local polynomial spaces on the mesh cells just add up:

$$\dim V_{-1} = n \cdot \dim \mathcal{P}_2 = 3n .$$

- Functions in the space V_1 have to satisfy a single continuity condition on every interface between two cells of the mesh, which means $n-1$ constraints:

$$\blacktriangleright \qquad \dim V_0 = \dim V_{-1} - (n-1) = 2n+1 .$$

- Another continuity condition for the first derivative is imposed in the case of the space V_1. This introduces another $n-1$ constraints.

$$\blacktriangleright \qquad \dim V_1 = \dim V_0 - (n-1) = n+2 .$$

- Further $n-1$ continuity conditions have to take into account in the case of V_2:

$$\blacktriangleright \qquad \dim V_2 = \dim V_1 - (n-1) = 3 = \dim \mathcal{P}_2 .$$

  In fact, we have $V_2 = \mathcal{P}_2$!

▲

(6-9.b) ⚁ (3 min.) Which of the spaces V_k, $k = -1, 0, 1$, coincides with one of the spline spaces $\mathcal{S}_{d,\mathcal{M}}$ introduced in [Lecture → Def. 5.5.0.1] in class?

$$V_{\boxed{}} = \mathcal{S}_{\boxed{},\mathcal{M}} \,.$$

SOLUTION of (6-9.b):

Spline spaces according to [Lecture → Def. 5.5.0.1] are piecewise polynomial spaces with maximal smoothness that do not collapse to spaces of simple polynomials:

$$\blacktriangleright \qquad V_1 = \mathcal{S}_{2,\mathcal{M}} \,.$$

▲

(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 $j = 1, \dots, n$

○ necessary ○ sufficient ○ neither

(ii) $v'(x) \geq 0$ for all $x \in [a, b]$

○ necessary ○ sufficient ○ neither

(iii) $v'(\frac{1}{2}(t_{j-1} + t_j)) \geq 0$ for all $j \in \{1, \dots, n\}$

○ necessary ○ sufficient ○ neither

(iv) $v'(t_j) \geq 0$ for all $j \in \{0, \dots, n\}$

○ necessary ○ sufficient ○ neither

SOLUTION of (6-9.c):

- Obviously all the conditions are necessary, since $v \in C^1([a, b])$.
- (i) is not sufficient; $M = \{0, 1\}$ and $v(x) = x(x - 1)$ provides a counterexample
- (ii) is sufficient of course, because $v \in C^1([a, b])$. This is theorem from calculus.
- (iii) is not sufficient; $M = \{0, 1\}$ and $v(x) = x(2x - 1)$ provides a counterexample.
- (iv) is sufficient, because v' is continuous and piecewise linear. If such a function is ≥ 0 at all nodes, it will be ≥ 0 everywhere, which yields condition (ii).

▲

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 $v \in V_1$.

$$a = \boxed{} \quad , \quad b = \boxed{} \,.$$

SOLUTION of (6-9.d):

- Since $v \in C^0([0,2])$ (continuity at $x = 1$): $a \cdot 1 + b = 0$
- From $v \in C^1([0,2])$: $a = -2$

Hence $a = -2$ and $b = 2$.

▲

(6-9.e) ⚁ (4 min.) Write down a $v \in V_1$ such that

$$v(0) = 0 \quad , \quad v(1) = 1 \quad , \quad v(2) = 0 \,.$$

$$v(x) = \begin{cases} \boxed{} & \text{for } 0 \leq x < 1 \,, \\ \boxed{} & \text{for } 1 \leq x \leq 2 \,. \end{cases}$$

SOLUTION of (6-9.e):

A possible choice is

$$v(x) = x(2 - x) \,.$$

In this case that function can be written in both boxes.

Note that this v is not the only solution. In fact, the space of possible solutions is one-dimensional: 3 conditions imposed on a functions from a 4-dimensional space.

▲

End Problem 6-9 , 25 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 $\mathcal{M} = \{0 = t_0 < t_1 < \cdots < t_{n-1} < t_n = 1\}$ on the unit interval $I = [0,1]$, $n \in \mathbb{N}$, we define the *piecewise linear interpolant*:

$$\mathsf{I}_{\mathcal{M}} : C^0(I) \to \mathcal{P}_{1,\mathcal{M}} = \{s \in C^0(I),\ s_{|[t_{j-1},t_j]} \in \mathcal{P}_1\ \forall\, j\}, \quad \text{s.t.} \quad (\mathsf{I}_{\mathcal{M}} f)(t_j) = f(t_j), \quad j = 0,\dots,n \tag{7.1.1}$$

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 $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ when $n \to \infty$?

HINT 1 for (7-1.a): Look for a suitable estimate in [Lecture → Section 6.5.1]. ⌟

SOLUTION of (7-1.a):

Equation [Lecture → Eq. (6.5.1.9)] says (we write $f'' := f^{(2)}$ for the second derivative):

$$\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} \le \frac{1}{2n^2} \|f''\|_{L^\infty(I)}\,, \tag{7.1.2}$$

because the meshwidth is $h = 1/n$. Thus, the convergence is quadratic, i.e. algebraic with order 2:

$$\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} = O(n^{-2}) \quad \text{for} \quad n \to \infty\,. \tag{7.1.3}$$

▲

(7-1.b) (5 min.) What is the smoothness class of the function $f : I := [0,1] \to \mathbb{R},\ f(t) = t^\alpha,\ 0 < \alpha < 2$? In other words, for which $k \in \mathbb{N}$ do we have $f \in C^k(I)$?

HINT 1 for (7-1.b): Notice that I is a closed interval and check the continuity of the derivatives in the endpoints of I.

SOLUTION of (7-1.b):

If $\alpha = 1$, $f(t) = t$ clearly belongs to $C^\infty(I)$. If $0 < \alpha < 1$, $f'(t) = \alpha t^{\alpha-1}$ blows up to infinity for t going to 0, therefore $f \in C^0(I) \setminus C^1(I)$. If $1 < \alpha < 2$, f' is continuous but $f''(t) = \alpha(\alpha-1)t^{\alpha-2}$ blows up to infinity for t going to 0, therefore $f \in C^1(I) \setminus C^2(I)$.

More generally, for $\alpha \in \mathbb{N}$ we have $f(t) = t^\alpha \in C^\infty(I)$; on the other hand, if $\alpha > 0$ is not an integer, $f \in C^{\lfloor\alpha\rfloor}(I)$, where $\lfloor\alpha\rfloor = \mathrm{floor}(\alpha)$ is the largest integer not larger than α.

▲

(7-1.c) (30 min.) Implement a templated C++ function

```
template <typename FUNCTION>
double pwlintpMaxError(FUNCTION &&f, const Eigen::VectorXd &t);
```

that computes $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ when f is passed through the functor `f` (with evaluation operator **double operator** `()` `(double)` **const**) and the mesh $\mathcal{M}$ through its *sorted* vector $[t_0, t_1, \dots, t_n]^\top \in \mathbb{R}^{n+1}$, $n \in \mathbb{N}$, of node positions, which also defines the interval $I := [t_0, t_n]$.

The maximum norm should be approximated by sampling in 10^5 equidistant points $in I$, see, e.g., [Lecture → Ex. 6.5.1.7].

SOLUTION of (7-1.c):

For efficient implementation it is essential to make use of the fact that the the nodes passed in `t` are sorted.

C++ code 7.1.4: Implementation of `pwlintpMaxError()`

```
template <typename FUNCTION>
double pwlintpMaxError(FUNCTION &&f, const Eigen::VectorXd &t) {
  int n = t.size() - 1;
  int N = 1e5;          // Sampling resolution for approximating L-infinty norm.
  double maxerr = 0.0; // L-infinity error so far.

  // TO DO (7-1.c): Approximate the maximum norm of (f - If) on [0,1],
  // where If is the piecewise linear interpolation of f on the mesh t.
  // START
  // We iterate over the sampling points x = 0, 1/N, 2/N, ... 1.
  // For each x we need to interpolate f in the correct mesh interval.
  // We are currently in the interval [tj, tj1] = [0,t(1)].
  int j = 0;
  double tj = t[j], tj1 = t[j + 1];
  double fj = f(tj), fj1 = f(tj1);
  for (int i = 0; i <= N; i++) {
    // We are currently in the mesh interval [tj, tj1].
    double x = (1.0 * i) / N;
    // If x is outside the current mesh interval,
```

```
    // then find the interval containing x.
    while (x > tj1) { // x >= tj always satisfied.
      tj = tj1;
      fj = fj1;
      j++;
      tj1 = t[j + 1];
      fj1 = f(tj1);
    }
    // Now we have tj <= x <= tj1.

    // Evaluate linear interpolant [tj,tj1] at x.
    double lfx = fj + (fj1 - fj) / (tj1 - tj) * (x - tj);
    // Update L-infinity error on [0,x]
    double err = std::abs(f(x) - lfx);
    if (err > maxerr) {
      maxerr = err;
    }
  }
  // END
  return maxerr;
}
```

▲

(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 the error norms $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ as a function of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *equidistant meshes*. Use meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$, and values for α passed in `alpha`. Do not forget axis labels, plot title, and a meaningful legend. Save the plot in `cvgplotequidistand.png`.

Describe your observations qualitatively.

SOLUTION of (7-1.d):

C++ code 7.1.5: Implementation of `cvgplotEquidistantMesh()`

```
void cvgplotEquidistantMesh(const Eigen::VectorXd &alpha) {
  int n_alphas = alpha.size();
  plt::figure();
  // TO DO (7-1.d): Create log-log plots of the maximum norm
  // errors (obtained by pwlintMaxError()) with
  // number of mesh intervals 32, 64, ..., 4096.
  // START
  int kmin = 5, kmax = 12;
  int n_meshes = kmax - kmin + 1;
  // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
  ArrayXd N = Eigen::pow(2, ArrayXd::LinSpaced(n_meshes, kmin, kmax));

  // Errors(i,j) = L-infinity error of interpolation of t^alpha(j)
  // using a mesh M with #M = N(i)+1.
```

```
  MatrixXd Errors(n_meshes, n_alphas);
  for (int i = 0; i < n_meshes; i++) {
    int n = N(i); // The mesh has #M = (n+1) = (N(i)+1) nodes.
    VectorXd mesh = VectorXd::LinSpaced(n + 1, 0.0, 1.0);
    for (int j = 0; j < n_alphas; j++) {
      // Compute L-infinity error for f_j(t) = t^alpha(j).
      double alphaj = alpha(j);
      auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
      Errors(i, j) = pwlintpMaxError<std::function<double(double)>>(fj, mesh);
    }
  }

  // Define a few line-styles to distinguish plots.
  // Colours change automatically.
  int n_styles = 4;
  std::vector<std::string> style;
  style.push_back("o--");
  style.push_back("v-.");
  style.push_back("s:");
  style.push_back("D-");
  // Plot errors
  for (int j = 0; j < n_alphas; j++) {
    char lab[20];
    sprintf(lab,"alpha = %.2f",alpha(j));
    plt::loglog(N, Errors.col(j), style[j%n_styles], {{"label", lab}});
  }
  plt::title("Error of pw. lin. intp. of f(t)=t^alpha on uniform meshes");
  plt::ylabel("Error");
  plt::xlabel("Number of mesh intervals");
  plt::legend("lower left", std::vector<double>());
  // END
  plt::savefig("./cx_out/cvgplotEquidistant.png");
}
```

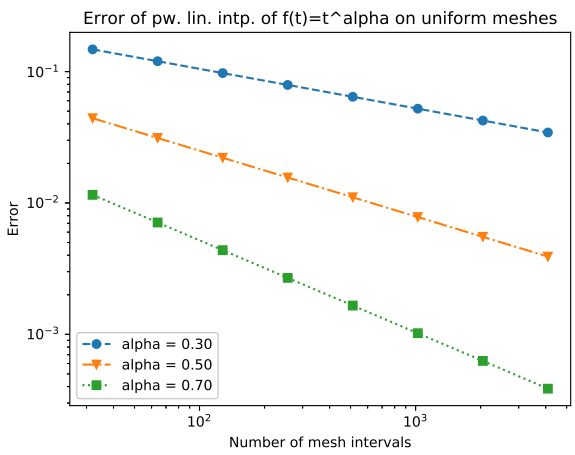


Fig. 25

◁ The data points $(n, \mathrm{errornorm(n)})$ almost exactly lie on straight lines in the doubly logaritmic plot, *cf.* [Lecture → Rem. 6.1.2.9].

▶ algebraic convergence in n^{-1}

We conjecture an asymptotic behavior $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)} = O(n^{-\beta})$ with $\beta = \beta(\alpha)$ an increasing function of α, because β can be read off the slop of the lines in the plot.

▲

(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 $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$ in terms of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *equidistant meshes* for the passed values of the parameter α and returns the estimates. These regression-based estimates should rely on meshes with $n = 2^k$, $k = 5, 6, \ldots, 12$.

HINT 1 for (7-1.e): Determine the rate of algebraic convergence using 1D linear regression, see which is implemented, as a special case, in the auxiliary function `polyfit`, see [Lecture → Code 5.7.0.21].

SOLUTION of (7-1.e):

C++ code 7.1.6: Implementation of `cvgrateEquidistantMesh()`

```
Eigen::VectorXd cvgrateEquidistantMesh(const Eigen::VectorXd &alpha) {
  int n_alphas = alpha.size();
  VectorXd Rates(n_alphas);

  // TO DO (7-1.e): Fill in the entries of Rates.
  // Hint: For each alpha(j), use polyfit() to estimate convergence
  // rates of the maximum norm (obtained by pwlintMaxError()) with
  // number of mesh intervals 32, 64, ..., 4096.
  // START
  int kmin = 5, kmax = 12;
  int n_meshes = kmax - kmin + 1;
  // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
  ArrayXd N = Eigen::pow(2, ArrayXd::LinSpaced(n_meshes, kmin, kmax));

  // Errors(i,j) = L-infinity error of interpolation of t^alpha(j)
  // using a mesh M with #M = N(i)+1.
  MatrixXd Errors(n_meshes, n_alphas);
  for (int i = 0; i < n_meshes; i++) {
    int n = N(i); // The mesh has #M = (n+1) = (N(i)+1) nodes.
    VectorXd mesh = VectorXd::LinSpaced(n + 1, 0.0, 1.0);
    for (int j = 0; j < n_alphas; j++) {
      // Compute L-infinity error for f_j(t) = t^alpha(j).
      double alphaj = alpha(j);
      auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
      Errors(i, j) = pwlintpMaxError<std::function<double(double)>>(fj, mesh);
    }
  }

  for (int j = 0; j < n_alphas; j++) {
    // Estimate convergence rate for f_j(t) = t^alpha(j) by the slope
    // of line fitted through the errors on a log-log scale.
    VectorXd coeff = polyfit(N.array().log(), Errors.col(j).array().log(), 1);
    Rates(j) = -coeff[1];
  }
  // END
  return Rates;
}
```

▲

(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 $\alpha = 0.05, 0.15, 0.25, \ldots, 2.95$. Save the plot in the file `cvgRateEquidistant.png`.

SOLUTION of (7-1.f):

C++ code 7.1.7: Implementation of `testcvgEquidistantMesh()`

```
void testcvgEquidistantMesh(void) {
  // TO DO (7-1.f): Plot and tabulate the convergence rates of pw.
  // lin. intp. of t^α using equidistant meshes on [0,1],
  // for alpha = 0.05, 0.15, 0.25 ..., 2.95.
  // START
  int n_alphas = 30;
  VectorXd alpha = VectorXd::LinSpaced(n_alphas, 0.05, 2.95);

  VectorXd ConvRates = cvgrateEquidistantMesh(alpha);
  std::cout << std::setw(10) << "alpha" << std::setw(20) << "Convergence rate"
            << std::endl;
  for (int j = 0; j < n_alphas; j++) {
    std::cout << std::setw(10) << alpha(j) << std::setw(20) << ConvRates(j)
              << std::endl;
  }

  plt::figure();
  plt::plot(alpha, ConvRates);
  plt::xlabel("alpha");
  plt::ylabel("conv. rate");
  plt::title("Cvg. rates of pw. lin. intp. of f(t)=t^alpha on uniform meshes");
  plt::savefig("./cx_out/cvgrateEquidistant.png");
  // END
}
```

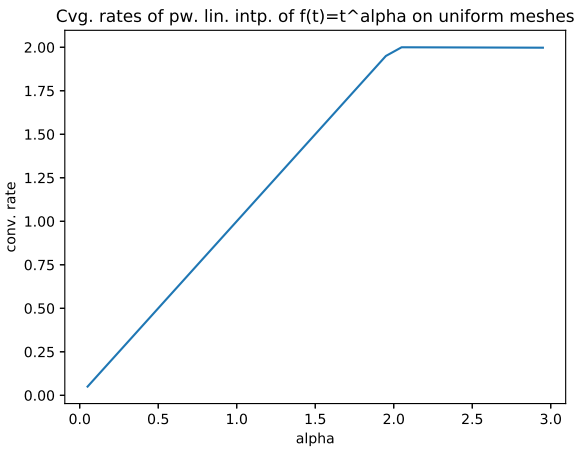


Fig. 26

The empiric rate of algebraic convergence is equal to α, if α is smaller than 2, and equal to 2 otherwise. In brief, we can say that the rate is $\min\{\alpha, 2\}$.
Be careful with the case $\alpha = 1$: here the interpolant exactly agrees with the solution on any mesh. This is why we avoided integers when sampling values for α.

▲

(7-1.g) ⚁ (5 min.) In which mesh interval do you expect $|f - \mathsf{I}_{\mathcal{M}} f|$ to attain its maximum?

HINT 1 for (7-1.g): Recall [Lecture → Thm. 6.1.2.13] and apply it in the piecewise linear setting. ⌟

SOLUTION of (7-1.g):

The error representation [Lecture → Eq. (6.1.2.14)] in the linear case ($n = 1$) for some $\tau_t \in (t_j, t_{j+1})$ reads as:

$$\forall\, t \in (t_j, t_{j+1}) \qquad \left| f(t) - \left(\mathsf{I}_{\mathcal{M}} f\right)(t) \right| = \frac{1}{2} |f''(\tau_t)\ (t - t_j)(t - t_{j+1})|$$
$$\leq \frac{1}{8} |f''(\tau_t)|\ (t_{j+1} - t_j)^2 = \frac{1}{8n^2} |f''(\tau_t)|$$

Therefore the error can be large only in the subintervals where the second derivative of f is large. $|f''(t)| = \left|\alpha(\alpha - 1)t^{\alpha-2}\right|$ is monotonically decreasing for $0 < \alpha < 2$: therefore, we can expect a large error in the first subinterval, the one that is closest to 0.

▲

(7-1.h) ⚁ (20 min.) [depends on Sub-problem (7-1.g)]

Compute ("on paper") the exact value of $\|f - \mathsf{I}_{\mathcal{M}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$, $0 < \alpha < 2$. Compare the order of convergence obtained with the one observed numerically in (7-1.b).

HINT 1 for (7-1.h): Use the insight gained in Sub-problem (7-1.g) to simplify the problem. ⌟

SOLUTION of (7-1.h):

From (7-1.g) we expect that the maximum appears in the first subinterval. Then, for $t \in]0, 1/n[$ and $0 < \alpha < 2$ (with $\alpha \neq 1$), let us find this maximum by considering the error function φ:

$$\varphi(t) = f(t) - \left(\mathsf{I}_{\mathcal{M}} f\right)(t) = t^\alpha - t\,\frac{1}{n^{\alpha-1}} \quad , \quad (\varphi(0) = \varphi(1/n) = 0)\,,$$
$$\varphi'(t) = \alpha t^{\alpha-1} - \frac{1}{n^{\alpha-1}}\,,$$
$$\varphi'(t^*) = 0 \quad \text{if} \quad t^* = \frac{1}{n}\alpha^{-1/(1-\alpha)}\,.$$

This means

$$\max_{t\in(0,1/n)} |\varphi(t)| = |\varphi(t^*)| = \left| \frac{\alpha^{-\alpha/(1-\alpha)}}{n^\alpha} - \frac{\alpha^{-1/(1-\alpha)}}{n^\alpha} \right|$$
$$= \frac{1}{n^\alpha} \left| \alpha^{-\alpha/(1-\alpha)} - \alpha^{-1/(1-\alpha)} \right| = \mathcal{O}(n^{-\alpha}) = \mathcal{O}(h^\alpha)\,.$$

The order of convergence in $h = 1/n$ is equal to the parameter α, as we could have guessed from Fig. 26.

▲

(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.

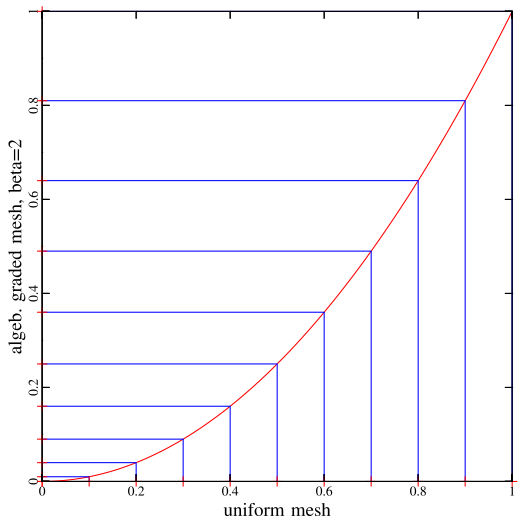


Fig. 27

A suitable choice is an **algebraically graded mesh**, defined as

$$\mathcal{G} = \left\{ t_j = \left(\frac{j}{n}\right)^{\beta}, j = 0, \dots, n \right\} \tag{7.1.8}$$

for the **grading parameter** $\beta > 1$.

◁ Algebraically graded mesh for $\beta = 2$.

Realize a C++ function

```
Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
                                  const Eigen::VectorXd &beta);
```

that estimates rates of asymptotic algebraic convergence (→ [Lecture → Rem. 6.1.2.9]) of $\|f - \mathsf{I}_{\mathcal{G}} f\|_{L^\infty(I)}$ for $f(t) := t^\alpha$ in terms of $n := \sharp\mathcal{M} - 1 \to \infty$ for families of *graded meshes* for the passed values of the parameter α (argument `alpha`) and of the grading parameter $\beta > 0$ (argument `beta`). The function should return the measured rates as entries of a matrix, a column for each α-value, a row for each β-value. The regression-based estimates should rely on meshes with $n = 2^k$, $k = 5, 6, \dots, 12$.

SOLUTION of (7-1.i):

C++ code 7.1.9: Implementation of `cvgrateGradedMesh()`

```
Eigen::MatrixXd cvgrateGradedMesh(const Eigen::VectorXd &alpha,
                                  const Eigen::VectorXd &beta) {
  int n_alphas = alpha.size();
  int n_betas = beta.size();
  MatrixXd Rates(n_betas, n_alphas);

  // TO DO (7-1.i): Fill in the entries of Rates.
  // START
  int kmin = 5, kmax = 12;
  int n_meshes = kmax - kmin + 1;
  // n=2^kmin, 2^(kmin+1), ..., 2^kmax.
  ArrayXd N = Eigen::pow(2, ArrayXd::LinSpaced(n_meshes, kmin, kmax));

  // Errors(i,k*n_alphas+j) = L-infinity error of interpolation of t^alpha(j)
  // using a mesh M with #M = N(i)+1 and grading parameter beta(k).
  MatrixXd Errors(n_meshes, n_alphas * n_betas);
  for (int i = 0; i < n_meshes; i++) {
    int n = N(i); // The mesh has #M = (n+1) = (N(i)+1) nodes.
    VectorXd mesh = VectorXd::LinSpaced(n + 1, 0.0, 1.0);
    for (int k = 0; k < n_betas; k++) {
      // Transform the mesh using the grading parameter beta.
      VectorXd gradedMesh = pow(mesh.array(), beta(k)).matrix();
```

```
      for (int j = 0; j < n_alphas; j++) {
        // Compute L-infinity error for f_j(t) = t^alpha(j).
        double alphaj = alpha(j);
        auto fj = [alphaj](double t) { return std::pow(t, alphaj); };
        int ind = k * n_alphas + j;
        Errors(i, ind) =
            pwlintpMaxError<std::function<double(double)>>(fj, gradedMesh);
      }
    }
  }

  for (int k = 0; k < n_betas; k++) {
    for (int j = 0; j < n_alphas; j++) {
      // Estimate convergence rate for f_j(t) = t^alpha(j) by the slope
      // of line fitted through the errors on a log-log scale.
      int ind = k * n_alphas + j;
      VectorXd coeff =
          polyfit(N.array().log(), Errors.col(ind).array().log(), 1);
      Rates(k, j) = -coeff[1];
    }
  }
  // END
  return Rates;
}
```

▲

(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 α in order to recover an asymptotic algebraic convergence like $O(n^{-2})$ of $\|f - \mathsf{I}_{\mathcal{G}} f\|_{L^\infty(I)}$ for $\sharp\mathcal{G} \sim n \to \infty$?

To arrive at a conjecture rely on the function `cvgrateGradedMesh()` from Sub-problem (7-1.i) and write another function

```
void testcvgGradedMesh(void);
```

that calls `cvgrateGradedMesh()` for

$$\alpha \in \{0.05, 0.15, 0.25, \dots, 2.95\}\,,$$
$$\beta \in \{0.1, 0.2, 0.3, \dots, 2.0\}\,,$$

and creates a useful plot of the data (unsing MATPLOTLIBCPP) and stores it in the file `alphabeta.png`.

SOLUTION of (7-1.j):

C++ code 7.1.10: Implementation of `testcvgGradedMesh()`

```
void testcvgGradedMesh(void) {
  // TO DO (7-1.j): Plot the convergence rates from
  // cvgrateGradedMesh() using alpha = 0.05, 0.15, ..., 2.95
  // and beta = 0.1, 0.2, ..., 2.0.
```

```
  // Note: Running this code may take a while, so start by
  // using fewer values for alpha and beta.
  // Hint: You can use plt::plot_surface(X,Y,Z) where X, Y, Z
  // are matrices that all have the same dimensions.
  // You can use x.replicate() to create X and Y
  // for an appropriate vector x.
  // START
  int n_alphas = 30;
  VectorXd alpha = VectorXd::LinSpaced(n_alphas, 0.05, 2.95);
  int n_betas = 20;
  VectorXd beta = VectorXd::LinSpaced(n_betas, 0.1, 2.0);

  // ConvRates(i,j) = convergence rate using beta(i) and alpha(j).
  MatrixXd ConvRates = cvgrateGradedMesh(alpha, beta);
  // Create meshgrid of alpha and beta values for plotting,
  // such that alphaGrid(i,j) = alpha(j)
  // and betaGrid(i,j) = beta(i).
  MatrixXd alphaGrid = alpha.replicate(1,n_betas).transpose();
  MatrixXd betaGrid = beta.replicate(1,n_alphas);

  plt::figure();
  plt::plot_surface(alphaGrid,betaGrid,ConvRates);
  plt::xlabel("alpha");
  plt::ylabel("beta");
  plt::title("Cvg. rates of pw. lin. intp. of f(t)=t^alpha on graded meshes");
  plt::savefig("./cx_out/alphabeta.png");
  // END
}
```

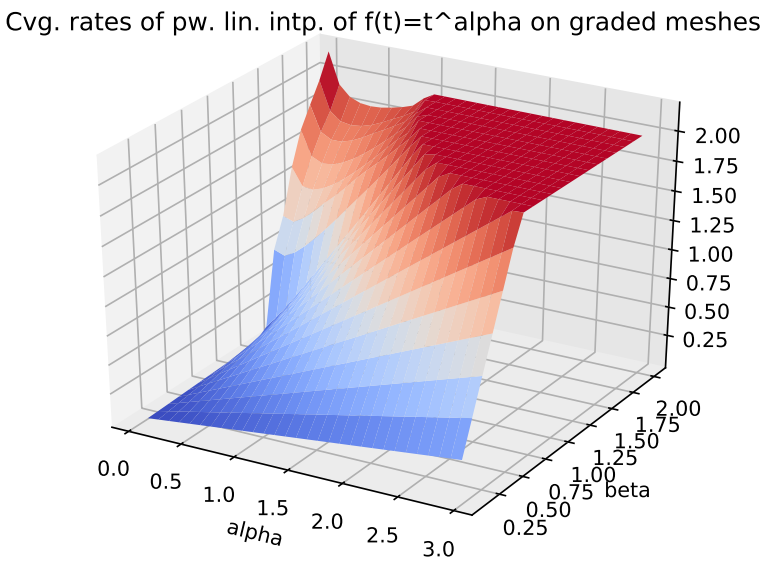


Fig. 28

The comparison of this plot with the analogous ones for different values of α suggests that the choice of $\beta = 2/\alpha$ guarantees quadratic convergence in n^{-1}.

Proceeding as in (7-1.g), we can see that the maximal error in the first subinterval $(0, t_1) = (0, 1/n^\beta)$ is equal to $1/n^{\alpha\beta}\,(\alpha^{-\alpha/(1-\alpha)} - \alpha^{-1/(1-\alpha)}) = \mathcal{O}(n^{-\alpha\beta})$ (replace the interval size $1/n$ with $1/n^\beta$ in those equations). This implies that a necessary condition to have quadratic convergence is $\beta \geq 2/\alpha$. In order to find an upper bound on the optimal β, we should control the error committed in every subinterval. Here the exact computation of $\varphi(t^*)$ becomes very long and complicated.

▲

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 $x_0 < x_1 < x_2 < \cdots < x_n$,

(II) the (interpolation) nodes $t_0 < t_1 < \cdots < t_n$, $n \in \mathbb{N}$, satisfying $x_0 \leq t_j \leq x_n$, $j = 0, \ldots, n$.

The space of piecewise linear continuous functions with respect to the knot set $\mathcal{N} := \{x_j\}_{j=0}^n$ is defined as:

$$\mathcal{S}_{1,\mathcal{N}} := \{s \in C^0([x_0, x_n]) : \; s(t) = \gamma_j t + \beta_j \text{ for } t \in]x_{j-1}, x_j], \; \gamma_j, \beta_j \in \mathbb{R} \; i = 1, \ldots, n\} \; . \tag{7.2.1}$$

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 $s(t_i) = y_i$, $i = 0, \ldots, n$, for given values $y_i \in \mathbb{R}$.

This fits the general framework of [Lecture → § 5.1.0.15].

Below we set $I_0 := [x_0, x_1]$, $I_j :=]x_{j-1}, x_{j+1}]$, $j = 1, \ldots, n-1$, $I_n := [x_{n-1}, x_n]$.

(7-2.a) ⊡ Show that the interpolation problem may not have a solution for some values y_j if a single interval $[x_j, x_{j+1}]$, $j = 0, \ldots, n-1$, contains more than 2 nodes t_i.

SOLUTION of (7-2.a):

In general a linear function cannot interpolate three data points.

▲

(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, $j = 0, \ldots, n$, contains at least one node.

HINT 1 for (7-2.b): You may appeal to the linear system of equations that arises from the above interpolation problem when using the "tent basis function" basis of $\mathcal{S}_{1,\mathcal{N}}$ from [Lecture → Fig. 139]. ⌟

SOLUTION of (7-2.b):

If the condition is violated, there will be a "tent basis function" whose support does not contain a node. The corresponding column in the system matrix of the LSE [Lecture → Eq. (5.1.0.16)] will vanish and the matrix cannot be invertible.

▲

(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.

HINT 1 for (7-2.c): Since we have to distribute $n+1$ nodes on n intervals, there will be a single interval that contains 2 nodes. ⌟

SOLUTION of (7-2.c):

The interpolant is fixed in that interval that contains two nodes. Thus, its value will be known at the endpoints of that interval, which, due to continuity, also fixes the values at the endpoints of adjacent intervals. Since they contain one node each, the interpolant is also fixed there. Continue this argument until you reach the endpoints x_0 and x_n.

▲

(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 satisified. You must not take for granted that knots or nodes are sorted already.

HINT 1 for (7-2.d): While you check the condition of Sub-problem (7-2.c), you can also find the interval $]x_j, x_{j+i}[$ which contains 2 nodes. There you can compute γ_j and β_j of (7.2.1). Given that $s(t) = \gamma_j t + \beta_j$, you can compute $s(x_j)$ and $s(x_{j+i})$. You are then able to compute γ_j and β_j in all the other intervals. ⌟

SOLUTION of (7-2.d):

C++11-code 7.2.2: Solution of (7-2.d)

```
VectorXd PwLinIP::tentBasCoeff(const VectorXd &x, const VectorXd &t,
                               const VectorXd &y) const

{
    // Initialization
    size_t n = t.size();
    auto x_indices = order(x);
    auto t_indices = order(t);
    // You can also implement a solution which does not need
   // sorted vectors and e.g. for each knot x_j looks
    // for the closest node t_{i1} and the next closest node t_{i2}.
    // However, such solution will not become more efficient
    // if you give as input already sorted vectors:  for each knot x_j
    // you will always have to iterate along the sorted vector t
    // to find the included node t_i.

    VectorXd s = VectorXd::Zero(n);

    // Check condition of subproblem 5.5.c
```

```
    size_t i = 0;
    size_t k = 0;
    for(size_t j=0; j<(n-1); ++j) {

      bool intervalOK = false;
        while(i < n) {

          bool inInterval = (x(x_indices[j]) < t(t_indices[i])) &&
                            (t(t_indices[i]) < x(x_indices[j+1]));

            if(inInterval) {
             intervalOK = true;
                if(i == j) { // Index of interval which contains 2
                   nodes
                              // t_i and t_{i+1}.  After that, we have
                              // i > j...
                    k = j;
                }
                break;
            } else {
                ++i;
            }
        }
      if(!intervalOK) {
          std::exit(EXIT_FAILURE);
      }
    }

    // 1.  Find slope γ and intercept β
    // in interval with 2 nodes k
    // 2.  Find s_k and s_{k+1}
    double gamma = (y(t_indices[k+1]) - y(t_indices[k])) /
                   (t(t_indices[k+1]) - t(t_indices[k]));
    double beta = y(t_indices[k]) - gamma * t(t_indices[k]);

    s(x_indices[k])   = gamma * x(x_indices[k])   + beta;
    s(x_indices[k+1]) = gamma * x(x_indices[k+1]) + beta;

    // Find intercept, slope and value s at the lower bound x
    // for all intervals on the left of interval with 2 nodes k
    for(int j=k-1; j>=0; --j) {
        gamma = (s(x_indices[j+1]) - y(t_indices[j])) /
                (x(x_indices[j+1]) - t(t_indices[j]));
        beta = y(t_indices[j]) - gamma * t(t_indices[j]);

        s(x_indices[j]) = gamma * x(x_indices[j]) + beta;
    }
    // Find intercept, slope and value s at the upper bound x
    // for all intervals on the right of interval with 2 nodes k
    for(int j=k+2; j<n; ++j) {
        gamma = (y(t_indices[j]) - s(x_indices[j-1])) /
                (t(t_indices[j]) - x(x_indices[j-1]));
        beta = s(x_indices[j-1]) - gamma * x(x_indices[j-1]);

        s(x_indices[j]) = gamma * x(x_indices[j]) + beta;
    }

    return s;
}
```

Get it on GitLab (interleaveLIP.cpp).

▲

(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!

HINT 1 for (7-2.e): Sorting of nodes and knots should be performed in the constructor using **std::**`sort`. To sort the data points, either wrap them into suitable objects for which you supply a comparison operator or exploit function `order` supplied in the code, which returns the indices of **std::**`sort`. Sorting is essential for the efficient implementation of the evaluation operator in the case $n \gg 1$. ⌟

HINT 2 for (7-2.e): The function `tentBasCoeff` can be reused as a private method of the class. ⌟

SOLUTION of (7-2.e):

C++11-code 7.2.3: Full declaration of class `PwLinIP`

```
class PwLinIP {
public:
    /*!
     * \brief
     * \param[in] x Vector of knots
     * \param[in] t Vector of nodes
     * \param[in] y Vector of values of interpolant in nodes
     */
     PwLinIP(const VectorXd &x, const VectorXd &t, const VectorXd &y);

    /*!
     * \brief operator() evaluate interpolant at arg.
     */
     double operator()(double arg) const;
private:
     VectorXd x_;
     VectorXd t_;
     VectorXd y_;
     VectorXd s_;
    /*!
     * \brief Compute values of interpolant in knots x from (t_i, y_i)
     * \param[in] x Vector of knots
     * \param[in] t Vector of nodes
     * \param[in] y Vector of values of interpolant in nodes t
     * \param[out] s Vector of values of interpolant in knots x
     */
     VectorXd tentBasCoeff(const VectorXd &x, const VectorXd &t,
                           const VectorXd &y) const;
```

```
};
```

Get it on GitLab (interleaveLIP.cpp).

C++11-code 7.2.4: Implementation of `PwLinIP::PwLinIP` (constructor)

```
PwLinIP::PwLinIP(const VectorXd &x, const VectorXd &t,
                 const VectorXd &y)
{
    assert(t.size() == y.size() && t.size() == x.size() &&
           "x, t, y must have same size!");

    size_t n = t.size();
    x_.resize(n);
    t_.resize(n);
    y_.resize(n);

    auto x_indices = order(x);
    for(size_t i=0; i<n; ++i) {
        x_(i) = x[x_indices[i]];
    }

    auto t_indices = order(t);
    for(size_t i=0; i<n; ++i) {
        t_(i) = t[t_indices[i]];
        y_(i) = y[t_indices[i]];
    }

    s_ = tentBasCoeff(x_, t_, y_);
}
```

Get it on GitLab (interleaveLIP.cpp).

C++11-code 7.2.5: Implementation of `PwLinIP::operator()`

```
double PwLinIP::operator()(double arg) const
{
    if(arg < x_(0) || arg > x_(x_.size()-1)) {

        return 0;
    } else {
      std::vector<double> x(x_.size()-1); // Exclude x_0 from search of interval including arg
      VectorXd::Map(&x.front(), x_.size()-1) = x_.tail(x_.size()-1);
      size_t j = std::lower_bound(x.begin(), x.end(), arg) - x.begin() + 1; // Binary search
      // '+1' is needed to restore indexing from x_0

      double gamma = (s_(j) - s_(j-1)) / (x_(j) - x_(j-1));
      double beta = s_(j-1) - gamma * x_(j-1);

      return gamma * arg + beta;
    }
}
```

Get it on GitLab (interleaveLIP.cpp).

▲

(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:

$$b_k(t_j) = \delta_{kj}\,, \quad k,j = 0,\dots,10 \tag{7.2.6}$$

HINT 1 for (7-2.f): You should use the **Figure** class for plotting. ⌟

SOLUTION of (7-2.f):

C++11-code 7.2.7: Solution of (7-2.f)

```
int main() {
    // Initialization
    size_t n = 11;

   // Nodes
    VectorXd x = VectorXd::LinSpaced(n,0,10);
    VectorXd t(n);
    t(0) = 0; t.tail(n-1).setLinSpaced(n-1,0.5,9.5);

   mgl::Figure fig;
   fig.xlabel("t");
   fig.ylabel("y");

   // Plot colors
   const unsigned int M = 3;
   std::vector<std::string> C = {"b", "r", "m"};

   // Loop over basis functions
   for(unsigned int i = 0; i < M; ++i) {
       // Basis function e_i
       VectorXd y = VectorXd::Zero(n);
       y(i) = 1;

       // Create interpoland with values y
       PwLinIP cardinalBasis(x, t, y);

       // Plotting values
       unsigned int N = 1000;
       VectorXd xval = VectorXd::LinSpaced(N,0,10);
       VectorXd s(N);
       for(size_t j=0; j<N; ++j) {
           s(j) = cardinalBasis(xval(j));
       }

       std::stringstream ss;
       ss << "Basis k = " << i;
       fig.plot(xval, s, C[i].c_str()).label(ss.str().c_str());
       fig.legend();

       fig.title("Tent basis functions");
       fig.save("tent_basis_functions.eps");
   }

```

```
}
```

Get it on GitLab (interleaveLIP.cpp).

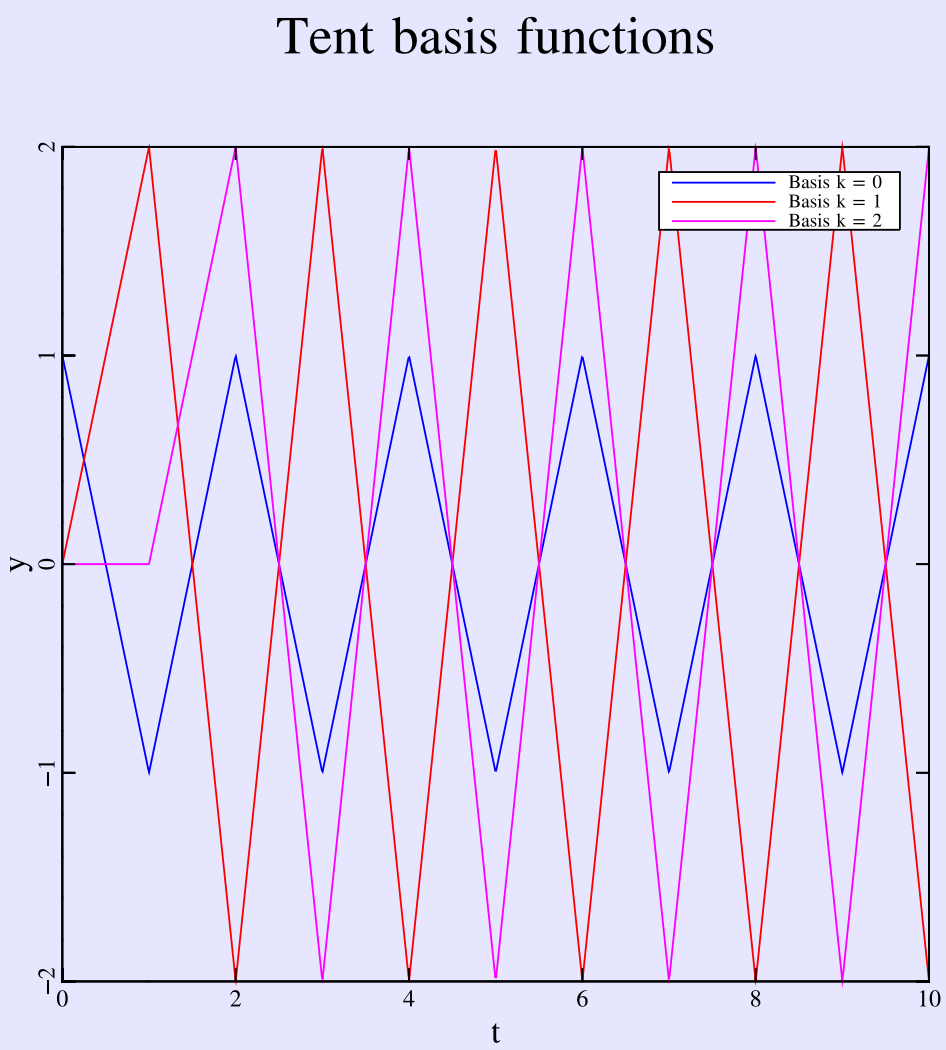


Fig. 29

▲

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 prominence for a wide range of approximation problems, see [**TEM08**].

Considerable implementation in C++ is requested in this problem.

A description of the greedy algorithm for adaptive polynomial interpolation is as follows:

Given a function $f : [a,b] \mapsto \mathbb{R}$ one starts $\mathcal{T}_0 := \{\frac{1}{2}(b+a)\}$. Based on a fixed finite set $\mathcal{S} \subset [a,b]$ of *sampling points* one augments the set of nodes according to

$$\mathcal{T}_{n+1} = \mathcal{T}_n \cup \left\{ \operatorname*{argmax}_{t\in\mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}_n}(t)| \right\} , \tag{7.3.1}$$

where $\mathsf{I}_{\mathcal{T}_n}$ is the polynomial interpolation operator (→ [Lecture → Cor. 5.2.2.8]) for the node set $\mathcal{T}_n$, until the relative interpolation error (in maximum norm) drops below a prescribed tolerance $\texttt{tol} > 0$:

$$\max_{t\in\mathcal{S}} |f(t) - \mathsf{I}_{\mathcal{T}_n}(t)| \leq \texttt{tol} \cdot \max_{t\in\mathcal{S}} |f(t)| \, . \tag{7.3.2}$$

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()`

```
VectorXd intpolyval(const VectorXd &t, const VectorXd &y, const VectorXd &x) {
  // no.  of interpolation nodes = deg.  of polynomial -1
  const unsigned int n = t.size();
  // no.  of evaluation points
  const unsigned int N = x.size();
  // For accumulation of point values of the interpolating polynomial
  VectorXd p = VectorXd(N);
  // Evaluation of barycentric interpolation formula, see [Lecture →
      Code 5.2.3.7]
  // Precompute the weights λ_i with effort O(n^2)
  VectorXd lambda(n);
  for (unsigned k = 0; k < n; ++k) {
    // little workaround:  cannot subtract a vector from a scalar
    // -> multiply scalar by vector of ones
```

```
    lambda(k) =
        1. / ((t(k) * VectorXd::Ones(k) - t.head(k)).prod() *
              (t(k) * VectorXd::Ones(n - k - 1) - t.tail(n - k - 1)).prod());
  }
  // Compute quotient of weighted sums of λ_i/(t-t_i),
  // effort O(n)
  for (unsigned i = 0; i < N; ++i) {
    VectorXd z = (x(i) * VectorXd::Ones(n) - t);

    // check if we want to evaluate at a node <-> avoid division by zero
    double *ptr = std::find(z.data(), z.data() + n, 0.0);
    if (ptr != z.data() + n) {  // if ptr = z.data + n = z.end no zero was found
      p(i) = y(ptr - z.data()); // ptr - z.data gives the position of the zero
    } else {
      VectorXd mu = lambda.cwiseQuotient(z);
      p(i) = (mu.cwiseProduct(y)).sum() / mu.sum();
    }
  }
  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

$$\mathcal{S} := \{a + (b-a)j/N : j = 0, \dots, N\}\,, \tag{7.3.4}$$

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.

HINT 1 for (7-3.a): You might need to convert between the types **std::vector** and **Eigen::VectorXd**. This can be done using EIGEN's **Map** class and the method `data()` of **Eigen::VectorXd**. See also EIGEN documentation.

SOLUTION of (7-3.a):

See Code 7.3.6 below. For this part of the exercise you can ignore the argument `err_vs_step_no`.

▲

(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

$$\epsilon_n := \max_{t\in\mathcal{S}} |f(t) - \mathsf{T}_{\mathcal{T}_n}(t)| \tag{7.3.5}$$

for each intermediate node set $\mathcal{T}_n$.

SOLUTION of (7-3.b):

C++ code 7.3.6: Implementation of function `adaptivepolyintp()`

```
template <class Function>

VectorXd adaptivepolyintp( Function&& f, double a, double b,
                           double tol, int N,
                           /* Ignore the following line for part a) */
                           std::vector<double> *errortab = nullptr ) {
  // TO DO (7-3.a) : implement the greedy algorithm for adaptive interpolation

  // Generate sampling points and evaluate f there
  VectorXd sampling_points = VectorXd::LinSpaced(N, a, b),
                 fvals_at_sampling_points = sampling_points.unaryExpr(f);
  //START
  // Approximate max|f(x)|
  double maxf = fvals_at_sampling_points.cwiseAbs().maxCoeff();

  // Adaptive mesh (initial node)
  std::vector<double> t { (a+b)/2. }, // Set of interpolation nodes
                           y { static_cast<double>(f((a+b)/2.)) }, // Values at nodes
                           errors;          // Error at nodes

  for (int i = 0; i < N; ++i) {
    // *** Step 1: interpolate with current nodes
    //   need to convert std::vector to
    //   Eigen::VectorXd to use the function intpolyval
    Map<VectorXd> te(t.data(), t.size());
    Map<VectorXd> ye(y.data(), y.size());
    VectorXd intpolyvals_at_sampling_points
                 = intpolyval(te, ye, sampling_points);

    // *** Step 2: find node where error is the largest
    VectorXd err =
            (fvals_at_sampling_points - intpolyvals_at_sampling_points)
            .cwiseAbs();
    // We use an Eigen "Visitor"
    //
       https://eigen.tuxfamily.org/dox/group__TutorialReductionsVisitorsBroadcasting.html
    double max = 0; int idx = 0;
    max = err.maxCoeff(&idx);

    // Step 3: check termination criteria
    if (max < tol * maxf) {
```

```
        return te;
    }
    // Step 4: add this node to our set of nodes
    t.push_back(sampling_points(idx));
    y.push_back(fvals_at_sampling_points(idx));
    // TO DO (7-3.b): save error
    if (errortab != nullptr) {
      errortab->push_back(max);
    }
  }
  std::cerr << "Desired accuracy could not be reached."
            << std::endl;
  //END
  return sampling_points; // return all sampling points
}
```

▲

(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 ϵ_n versus n (the number of interpolation nodes). Choose axis scales that reveal the qualitative decay (types of convergence as given in [Lecture → Def. 6.1.2.7]) of this error norm as the number of interpolation nodes is increased. Use interval $[a,b] = [0,1]$, `N=1000` sampling points, tolerance `tol = 1e-6`.

SOLUTION of (7-3.c):

C++ code 7.3.7: Function `plotInterpolationError()`

```
void plotInterpolationError(void) {
  // Declare test functions
  auto f1 = [](double t) { return std::sin(std::exp(2*t)); };
  auto f2 = [](double t) { return std::sqrt(t)/(1 + 16*t*t); };
  // TO DO (7-3.c): generate the plots of error vs number of nodes for f1, f2

  //START
  // Test interval
  const double a = 0, b = 1;
  // Get interpolation nodes
  const unsigned N = 1000; // no. of sampling points
  const double tol = 1e-6; // tolerance

  VectorXd tf1, tf2; // nodes for f1 resp. f2
  std::vector<double> ef1, ef2; // errors for f1 resp. f2

  tf1 = adaptivepolyintp(f1, a, b, tol, N, &ef1);
  tf2 = adaptivepolyintp(f2, a, b, tol, N, &ef2);
  VectorXd n1 = VectorXd::LinSpaced(ef1.size(),1,ef1.size());
  VectorXd n2 = VectorXd::LinSpaced(ef2.size(),1,ef2.size());
  // Plot
  plt::figure();
```

```
  plt::title("Error VS step");
  plt::xlabel("No. of interpolation nodes");
  plt::ylabel("max |f(t) - I_Tf(t)|");
  plt::semilogy(n1,ef1, "ro", {{"label","f_1(t) = sin(e^{2t})"}});
  plt::semilogy(n2,ef2, "bo", {{"label","f_2(t) = sqrtt/(1+16t^2)"}});
  plt::legend();
  plt::savefig("./cx_out/intperrplot.png");
  //END
}
```

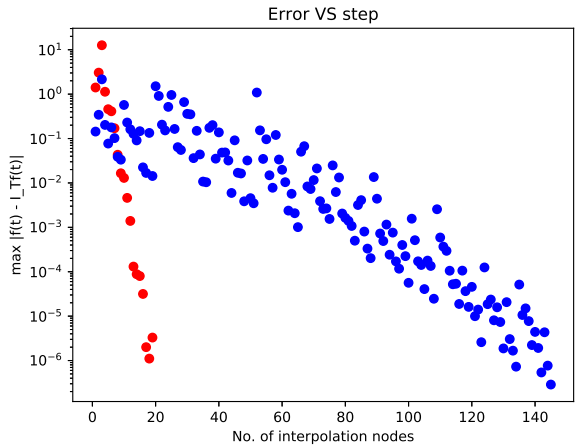


Fig. 30

- $\hat{=}$ $f_1(t) := \sin(e^{2t})$
- $\hat{=}$ $f_2(t) = \frac{\sqrt{t}}{1+16t^2}$

The error data points are approximately located on straight lines in the linear-logaritmic error plots.

Exponential convergence

▲

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:

$$\xi_j^{(n)} = \cos\left(\frac{2j+1}{2n}\pi\right), \quad j = 0, \dots, n-1 \tag{7.4.1}$$

We define the family of discrete L^2 semi-inner products (i.e. not conjugate symmetric), *cf.* [Lecture → Eq. (6.2.2.3)]:

$$(f,g)_n := \sum_{j=0}^{n-1} f(\xi_j^{(n)})g(\xi_j^{(n)}), \quad f,g \in C^0([-1,1]) \tag{7.4.2}$$

We also define the special weighted L^2 semi-inner product:

$$(f,g)_w := \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} f(t)g(t)\,dt \quad f,g \in C^0([-1,1]) \tag{7.4.3}$$

(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 $k \neq l$.

HINT 1 for (7-4.a): Recall the trigonometric identity $2\cos(x)\cos(y) = \cos(x+y) + \cos(x-y)$. ⌟

SOLUTION of (7-4.a):

For $k, l = 0, \dots, n$ with $k \neq l$, by using the substitution $s = \arccos t$ ($ds = -\frac{1}{\sqrt{1-t^2}}\,dt$) and simple trigonometric identities we readily compute:

$$\begin{aligned}
(T_k, T_l)_w &= \int_{-1}^{1} \frac{1}{\sqrt{1-t^2}} \cos(k \arccos t)\cos(l \arccos t)\,dt \\
&= \int_0^{\pi} \cos(ks)\cos(ls)\,ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((k+l)s) + \cos((k-l)s)\,ds \\
&= \frac{1}{2}\{[\sin((k+l)s)/(k+l)]_0^{\pi} + [\sin((k-l)s)/(k-l)]_0^{\pi}\} \\
&= 0
\end{aligned}$$

since $k+l \neq 0$ and $k-l \neq 0$.

▲

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.

HINT 1 for (7-4.b): [Lecture → Code 6.1.3.6] from the lecture notes demonstrates the efficient evaluation of Chebychev polynomials based on their 3-term recurrence formula from [Lecture → Thm. 6.1.3.4].

C++11-code 7.4.5: Code [Lecture → Code 6.1.3.6]

```
vector<double> chebpolmult(const int &n,const double &x)
{
    vector<double> V={1,x};
    for (int k=1; k<n; k++)
        V.push_back(2*x*V[k]-V[k-1]);
    return V;
}
```

SOLUTION of (7-4.b):

As a consequence of [Lecture → Thm. 6.1.3.4], we already know that $\{T_0, \dots, T_n\}$ is a basis for $\mathcal{P}_n$. We check the orthogonality with the C++ code given in Code 7.4.5.

C++11-code 7.4.6: Solution of (7-4.b)

```
// Check the orthogonality of Chebyshev polynomials
    n=10;
    vector<double> V;
    Eigen::MatrixXd scal(n+1,n+1);
    for (int j=0; j<n+1; j++) {
        V=chebpolmult(n,cos(M_PI*(2*j+1)/2/(n+1)));
        for (int k=0; k<n+1; k++) scal(j,k)=V[k];
    }
    cout<<"Scalar products: "<<endl;
    for (int k=0; k<n+1; k++)
        for (int l=k+1; l<n+1; l++)
            cout<<scal.col(k).dot(scal.col(l))<<endl;
```

Get it on GitLab (chebpolyproperties.cpp).

▲

(7-4.c) ⚃ Prove Thm. 7.4.4.

HINT 1 for (7-4.c): Use the relationship of trigonometric functions and the complex exponential together with the summation formula for geometric sums.

SOLUTION of (7-4.c):

It remains to check the orthogonality condition, namely that $(T_k, T_l)_n = 0$ for $k \neq l$. For $k, l = 0, \dots, n+1$ with $k \neq l$, by Eq. (7.4.1) we have:

$$(T_k, T_l)_{n+1} = \sum_{j=0}^{n} T_k\left(\xi_j^{(n)}\right) T_l\left(\xi_j^{(n)}\right) \tag{7.4.7}$$

$$= \sum_{j=0}^{n} \cos\left(k\frac{2j+1}{2(n+1)}\pi\right) \cos\left(l\frac{2j+1}{2(n+1)}\pi\right) \tag{7.4.8}$$

$$= \frac{1}{2}\sum_{j=0}^{n}\left[\cos\left((k+l)\frac{2j+1}{2(n+1)}\pi\right) + \cos\left((k-l)\frac{2j+1}{2(n+1)}\pi\right)\right] \tag{7.4.9}$$

$$\tag{7.4.10}$$

It is now enough to show that:

$$\sum_{j=0}^{n} \cos\left(m\frac{2j+1}{2(n+1)}\pi\right) = 0, \qquad m \in \mathbb{Z}^* \tag{7.4.11}$$

In order to verify this, observe that:

$$\sum_{j=0}^{n} \cos\left(m\frac{2j+1}{2(n+1)}\pi\right) = \mathrm{Re}\left(\sum_{j=0}^{n} e^{im\frac{2j+1}{2(n+1)}\pi}\right) = \mathrm{Re}\left(e^{im\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi}\right)$$

Finally, by the standard formula for the geometric sum we have:

$$e^{im\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi} = e^{im\frac{1}{2(n+1)}\pi}\frac{1-e^{im\frac{n+1}{n+1}\pi}}{1-e^{im\frac{1}{n+1}\pi}}$$

$$= \frac{1-e^{im\pi}}{e^{-im\frac{1}{2(n+1)}\pi} - e^{im\frac{1}{2(n+1)}\pi}} = -\frac{1-e^{im\pi}}{2}\frac{1}{i\sin\left(m\frac{1}{2(n+1)}\pi\right)} \in i\cdot\mathbb{R}$$

This implies that $\mathrm{Re}(e^{im\frac{1}{2(n+1)}\pi}\sum_{j=0}^{n} e^{im\frac{j}{n+1}\pi}) = 0$ as desired.

▲

(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:

$$q_n = \operatorname*{argmin}_{p\in\mathcal{P}_n} |f-p|_{n+1}\,,$$

$|\ |_{n+1}$ is the norm induced by the scalar product $(\ ,\ )_{n+1}$. You should represent q_n through an expansion in Chebychev polynomials of the form:

$$q_n = \sum_{j=0}^{n} \alpha_j T_j\,, \tag{7.4.12}$$

for suitable coefficients $\alpha_j \in \mathbb{R}$, see also [Lecture → Eq. (6.1.3.27)].

HINT 1 for (7-4.d): The task boils down to determining the coefficients α_j. Use Thm. 7.4.4 and a slight extension of [Lecture → Cor. 6.2.1.15]. ⌟

SOLUTION of (7-4.d):

In view of the theorem, the family $\{T_0,\dots,T_n\}$ is an orthogonal basis of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$. By Eq. (7.4.7) and Eq. (7.4.11) we have:

$$\lambda_k^2 := |T_k|_{n+1}^2 = (T_k,T_k)_{n+1} = \begin{cases} \frac{1}{2}\sum_{j=0}^n(\cos(0)+\cos(0)) = n+1 & \text{if } k=0 \\ \frac{1}{2}\sum_{j=0}^n \cos(0) = (n+1)/2 & \text{otherwise} \end{cases} \tag{7.4.13}$$

The family $\{T_k/\lambda_k : k=0,\dots,n\}$ is an ONB of $\mathcal{P}_n$ with respect to the inner product $(\ ,\ )_{n+1}$. Hence, by [Lecture → Cor. 6.2.1.15] we have that:

$$q_n = \sum_{j=0}^n (f,T_j/\lambda_j)_{n+1}\frac{T_j}{\lambda_j} = \sum_{j=0}^n \alpha_j T_j, \quad \alpha_j = \frac{1}{n+1}\begin{cases} (f,T_j)_{n+1} & \text{if } k=0 \\ 2(f,T_j)_{n+1} & \text{otherwise} \end{cases}$$

▲

(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 of (7-4.e):

C++11-code 7.4.14: Solution of (7-4.e)

```
template <typename Function>
void bestpolchebnodes(const Function &f, Eigen::VectorXd &alpha) {
    int n=alpha.size()-1;
    Eigen::VectorXd fn(n+1);
    for (int k=0; k<n+1; k++) {
        double temp=cos(M_PI*(2*k+1)/2/(n+1));
        fn(k)=f(temp);
    }

    vector<double> V;
    Eigen::MatrixXd scal(n+1,n+1);
    for (int j=0; j<n+1; j++) {
        V=chebpolmult(n,cos(M_PI*(2*j+1)/2/(n+1)));
        for (int k=0; k<n+1; k++) scal(j,k)=V[k];
    }

    for (int k=0; k<n+1; k++) {
        alpha(k)=0;
        for (int j=0; j<n+1; j++) {
            alpha(k)+=2*fn(j)*scal(j,k)/(n+1);
        }
    }
    alpha(0)=alpha(0)/2;
}
```

Get it on GitLab (chebpolyproperties.cpp).

▲

(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.

HINT 1 for (7-4.f): Again, [Lecture → Code 6.1.3.6] is useful for evaluating Chebychev polynomials.

SOLUTION of (7-4.f):

C++11-code 7.4.15: Solution of (7-4.f)

```
// Test the implementation
    auto f = [] (double & x) {return 1/(pow(5*x,2)+1);};
    n=20;
    Eigen::VectorXd alpha(n+1);
    bestpolchebnodes(f, alpha);

    // Compute the error
    Eigen::VectorXd X = Eigen::VectorXd::LinSpaced(1e6,-1,1);
    auto qn = [&alpha,&n] (double & x) {
        double temp=0;
        vector<double> V=chebpolmult(n,x);
        for (int k=0; k<n+1; k++) temp+=alpha(k)*V[k];
        return temp;
    };
    double err_max=abs(f(X(0))-qn(X(0)));
    for (int i=1; i<1e6; i++) err_max=std::max(err_max,abs(f(X(i))-qn(X(i))));
    cout<<"Error: "<< err_max <<endl;
```

Get it on GitLab (chebpolyproperties.cpp).

The output (plotted in MATLAB) is shown in Fig. 31.

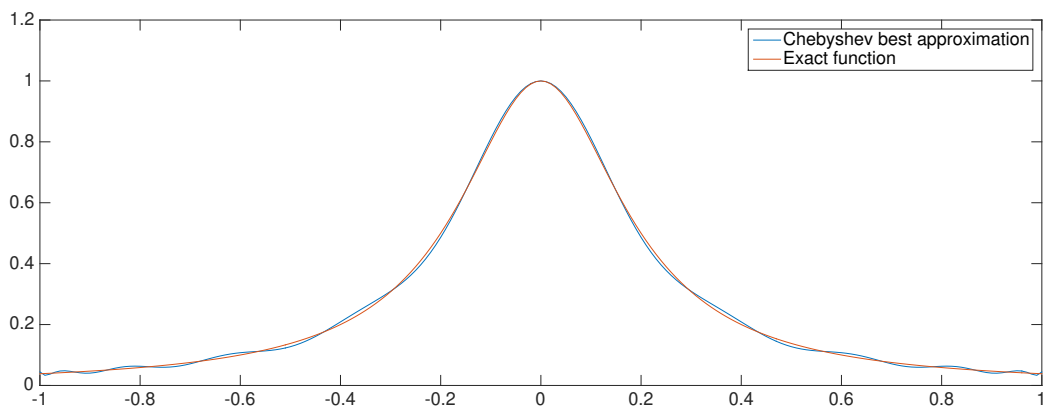


Fig. 31

▲

(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:

$$L_j = \frac{1}{n+1} + \frac{2}{n+1}\sum_{l=1}^{n} T_l(\xi_j^{(n+1)})T_l$$

HINT 1 for (7-4.g): Again, use the above theorem to express the coefficients of a Chebychev expansion of L_j. ⌟

SOLUTION of (7-4.g):

We have already seen that $\{T_l/\lambda_l : l = 0,\dots,n\}$ is an ONB of $\mathcal{P}_n$. Thus we can write:

$$L_j = \sum_{l=0}^{n} \frac{(L_j, T_l)_{n+1}}{\lambda_l^2} T_l = \sum_{l=0}^{n}\sum_{k=0}^{n} L_j(\xi_k^{(n+1)})T_l(\xi_k^{(n+1)})\frac{T_l}{\lambda_l^2}$$

By definition of Lagrange polynomials we have $L_j(\xi_k^{(n+1)}) = \delta_{jk}$, whence:

$$L_j = \sum_{l=0}^{n} T_l(\xi_j^{(n+1)})\frac{T_l}{\lambda_l^2}$$

Finally, the conclusion immediately follows from Eq. (7.4.13).

▲

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 → Thm. 6.1.2.13] and [Lecture → Lemma 6.1.2.17] for *analytic* interpolands. The bound tells us that for all $t \in [a,b]$

$$|f(t) - \mathsf{L}_{\mathcal{T}} f(t)| \leq \left| \frac{w(x)}{2\pi i} \int_\gamma \frac{f(z)}{(z-t)w(z)} dz \right| \leq \frac{|\gamma|}{2\pi} \frac{\max_{a\leq\tau\leq b}|w(\tau)|}{\min_{z\in\gamma}|w(z)|} \frac{\max_{z\in\gamma}|f(z)|}{d([a,b],\gamma)} ,$$

where $d([a,b],\gamma)$ is the geometric distance of the integration contour $\gamma \subset \mathbb{C}$ from the interval $[a,b] \subset \mathbb{C}$ in the complex plane. The contour γ must be contractible in the domain D of analyticity of f and must wind around $[a,b]$ exactly once, see [Lecture → Fig. 185].

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. 199],

$$\gamma_\rho(\theta) := \cos(\theta - i\log(\rho)) , \quad \forall 0 \leq \theta \leq 2\pi , \quad \rho > 1 . \tag{7.5.1}$$

(7-5.a) ⚀ Find an upper bound for the length $|\gamma_\rho|$ of the contour γ_ρ.

HINT 1 for (7-5.a): You may use the arc-length formula for a curve $\gamma : I \to \mathbb{R}^2$:

$$|\gamma| = \int_I \|\dot{\gamma}(\tau)\| d\tau, \tag{7.5.2}$$

where $\dot{\gamma}$ is the derivative of γ w.r.t the parameter τ. Recall that the "length" of a complex number z viewed as a vector in $\mathbb{R}^2$ is just its modulus. ⌟

SOLUTION of (7-5.a):

$\frac{\partial \gamma_\rho(\theta)}{\partial \theta} := -\sin(\theta - i\log(\rho))$, therefore:

$$|\gamma_\rho| = \int_{[0,2\pi]} |\sin(\tau - i\log(\rho))| d\tau \tag{7.5.3}$$

$$= \frac{1}{2\rho} \int_{[0,2\pi]} \sqrt{\sin^2(\tau)(1+\rho^2)^2 + \cos^2(\tau)(1-\rho^2)^2} d\tau \tag{7.5.4}$$

$$\leq \frac{1}{2\rho} \int_{[0,2\pi]} \sqrt{2+2\rho^4} d\tau \tag{7.5.5}$$

$$\leq \frac{1}{\rho} \pi \sqrt{2(1+\rho^4)} \tag{7.5.6}$$

▲

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 $\mathbb{C}$.

HINT 1 for (7-5.b): Consult [Lecture → Rem. 6.1.2.41]. ⌟

SOLUTION of (7-5.b):

f is analytic in $D := \mathbb{C} \setminus \{\frac{2}{3}\pi i c - \frac{1}{3}\pi i \mid c \in \mathbb{Z}\}$.

In fact, $g(t) := 1 + \exp(-3t)$ is an entire function, whereas $h(x) := \frac{1}{x}$ is analytic in $\mathbb{C} \setminus \{0\}$. Therefore, using [Lecture → Thm. 6.1.2.42], f is analytic in $\mathbb{C} \setminus \{z \in \mathbb{C} \mid g(z) = 0\} =: \mathbb{C} \setminus S$. Let $z := a + ib$, $a, b \in \mathbb{R}$. Since:

$$-1 = \exp(z) = \exp(a+ib) = \exp(a)\exp(ib) \Leftrightarrow a = 0,\ b \in 2\pi\mathbb{Z} + \pi \tag{7.5.7}$$

$$-1 = \exp(z) \Leftrightarrow z \in i(2\pi\mathbb{Z} + \pi) \tag{7.5.8}$$

$$-1 = \exp(-3z) \Leftrightarrow z \in \frac{i(2\pi\mathbb{Z} - \pi)}{3} \tag{7.5.9}$$

▲

(7-5.c) ⚁ Write a C++ function that computes an approximation M of:

$$\min_{\rho>1} \frac{\max_{z\in\gamma_\rho}|f(z)|}{d([-1,1],\gamma_\rho)}, \tag{7.5.10}$$

by sampling, where the distance of $[a,b]$ from γ_ρ is formally defined as

$$d([a,b],\gamma) := \inf\{|z-t| \mid z \in \gamma, t \in [a,b]\}. \tag{7.5.11}$$

HINT 1 for (7-5.c): The result of (7-5.b), together with the knowledge that γ_ρ describes an ellipsis, tells you the maximal range $(1, \rho_{max})$ of ρ. Sample this interval with 1000 equidistant steps. ⌟

HINT 2 for (7-5.c): Apply geometric reasoning to establish that the distance of γ_ρ and $[-1,1]$ is $\frac{1}{2}(\rho + \rho^{-1}) - 1$. ⌟

HINT 3 for (7-5.c): If you cannot find ρ_{max} use $\rho_{max} = 2.4$. ⌟

HINT 4 for (7-5.c): You can exploit the properties of $\cos$ and the hyperbolic trigonometric functions $\cosh$ and $\sinh$. ⌟

SOLUTION of (7-5.c):

The ellipse must be restricted such that the minor axis has length $\leq 2\pi/3$ (2 times the smallest point, in absolute value, where f is not-analytic). Since this corresponds to the imaginary part of $\gamma_\delta(\theta)$, when $\theta = \pi/2$, we find:

$$\cos(\pi/2 - i\log(\rho_{max})) = 1/3\pi i \Leftrightarrow \sinh(\log(\rho_{max})) = \pi/3 \Leftrightarrow \rho_{max} = \exp(\sinh^{-1}(\pi/3)).$$

See `chebyApprox.hpp` and `chebyApproxDriver.cpp` for the C++ code:

C++ code 7.5.12: Implementation of `chebyApprox()`

```
#include <Eigen/Dense>
#include <cmath>
#include <complex>
#include <iostream>
#include <limits>

using Eigen::VectorXcd;
using Eigen::VectorXd;
typedef std::complex<double> complex;

// Returns an approximation of
// min_ρ max_{z∈γρ} |f(z)| / d([−1,1],γ)
// and the ρ for which the minimum is attained
// IN : maxRho = maximal value of tho, compute min in (1, maxRho)
// N = number of intervals to discretize
// f, gamma = function handles for f and gamma
std::pair<double, double> chebyApprox(const double maxRho, const unsigned N) {
  // define f and gamma
  const complex i(0, 1); // imaginary unit
  auto f = [](complex z) { return 1. / (1. + std::exp(-3. * z)); };
  auto gamma = [i](double rho, double theta) {
    return std::cos(theta - i * std::log(rho));
  };

  double m = std::numeric_limits<double>::max(); // save min in here
  // discretize rhos and remove first and last value (open interval!)
  VectorXd rhosClosedIntervall = VectorXd::LinSpaced(N, 1, maxRho);
  VectorXd rhos = rhosClosedIntervall.segment(1, N - 2);
  double r = rhos(0); // save argmin here

  // discretize the ellipse
  VectorXd thetas = VectorXd::LinSpaced(N, 0, 2 * M_PI);

  // loop over all rhos and get compute expression
  for (int k = 0; k < rhos.size(); ++k) {
    const double rho = rhos(k); // current rho
    // get values on the ellipse
    VectorXcd z(thetas.size());
    for (int j = 0; j < thetas.size(); ++j) {
      z(j) = gamma(rho, thetas(j));
    }

    // sup of f of γ_ρ
    const VectorXcd fz = z.unaryExpr(f);
    const double M = fz.cwiseAbs().maxCoeff(); // nominator of expression

    // Distance is the one between -1 and the minimal real part of the ellipse
    // (i.e. semi-major-axis - 1):
    const double d = (rho + 1. / rho) / 2. - 1; // denominator of expression

    if (M / d < m) {
      r = rho; // update rho
    }
    m = std::min(m, M / d); // update minimum
  }
  return std::make_pair(m, r);
}
```

▲

(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 of (7-5.d):

In problem (7-5.a) we computed an upper bound for the length of the curve for a given rho. Let M be the approximation of (7-5.c). Then

$$\|f - L_n f\|_{L^\infty([-1,1])} \lesssim \frac{M\sqrt{2(\rho^{-2}+\rho^2)}}{\rho^{n+1}-1}.$$

▲

(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]).

HINT 1 for (7-5.e): The function `intpolyval` has the signature:

```
void intpolyval(const VectorXd& t, const VectorXd& y, const
    VectorXd& x, VectorXd& p);
```

Where (t,y) is the interpolation data and p is used to save the values of the interpolant evaluated in x.

HINT 2 for (7-5.e): Use the semi-logarithmic scale for your plot.

SOLUTION of (7-5.e):

C++ code 7.5.13: Main driver code

```
#include "../../../../../Utils/polyfit.hpp"
#include "../../../../../Utils/polyval.hpp"
#include "./intpolyval.hpp"
#include "chebyApprox.hpp"
#include <Eigen/Dense>
#include <figure/figure.hpp>
#include <vector>

using Eigen::ArrayXd;
using Eigen::VectorXd;

int main() {
  const unsigned N = 1000;
  const double maxRho = std::exp(asinh(M_PI / 3));

  std::pair<double, double> tmp = chebyApprox(maxRho, N);
  const double m = tmp.first, rho = tmp.second;
```

```
  // Test for degrees 1 to 20
  VectorXd degrees = VectorXd::LinSpaced(20, 1, 20), upperBound(20);

  for (int j = 0; j < degrees.size(); ++j) {
    upperBound(j) = m *
                    std::sqrt(2 * (1. / (maxRho * maxRho) + maxRho * maxRho)) /
                    (std::pow(rho, degrees(j) + 1) - 1);
  }

  // Compute actual error for any degree
  auto f = [](double x) { return 1. / (1 + std::exp(-3 * x)); };
  std::vector<double> errInf;

  for (int l = 0; l < degrees.size(); ++l) {
    const int n = degrees(l);
    // Cheby nodes and value of f at the nodes
    ArrayXd k = ArrayXd::LinSpaced(n + 1, 0, n);
    VectorXd t = ((2 * k + 1) / (2 * (n + 1)) * M_PI).cos().matrix(),
             y = t.unaryExpr(f);
    // Evaluate error at a couple of points
    VectorXd x = VectorXd::LinSpaced(500, -1, 1), F = x.unaryExpr(f);

    // Use barycentric formula (in intpolyval.hpp)
    VectorXd LnF;
    intpolyval(t, y, x, LnF);
    errInf.push_back((F - LnF).cwiseAbs().maxCoeff());
  }

  mgl::Figure fig;
  fig.setlog(false, true);
  fig.plot(degrees, errInf, " +r").label("Computed error");
  fig.plot(degrees, upperBound, " ^b").label("Upper bound");
  fig.xlabel("degree n");
  fig.ylabel("L^\\infty error");
  fig.legend(0, 0);
  fig.save("error1.eps");
  return 0;
}
```

▲

(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 of (7-5.f):

The rescaled function $\Phi^* f$ will have a different domain of analyticity and a different growth behavior in the complex plane. The larger a, the closer the pole of $\Phi^* f$ will move to $[-1,1]$, the more the choice of the ellipses is restricted (i.e. ρ_{max} becomes smaller). This will result in a larger bound.

Using [Lecture → Eq. (6.1.3.34)], it follows immediately that the asymptotic behaviour of the interpolation does not change after rescaling of the interval. In fact, if Φ^* is the affine pullback from $[-a,a]$ to

$[-1,1]$, then:

$$\|f - (\hat{L_n})f\|_{L^\infty([-a,a])} = \|\Phi^* f - L_n \Phi^* f\|_{L^\infty([-1,1])},$$

where $(\hat{L_n})$ is the polynomial interpolation operator on $[-a,a]$.

▲

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.

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{\boxed{}} \leq \left(1+\frac{\pi^2}{2}\right)^{\boxed{}} \frac{(\boxed{}-\boxed{})!}{\boxed{}!} \left\|f^{(\boxed{})}\right\|_{\boxed{}}$$

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 of (7-6.a):

Jackson's theorem reads:

Theorem 7.6.1. L^∞ polynomial best approximation estimate

If $f \in C^r([-1,1])$ (r times continuously differentiable), $r \in \mathbb{N}$, then, for any polynomial degree $n \geq r$,

$$\inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \leq (1+\pi^2/2)^r \frac{(n-r)!}{n!} \left\|f^{(r)}\right\|_{L^\infty([-1,1])},$$

where

$$\left\|f^{(r)}\right\|_{L^\infty([-1,1])} := \max_{x\in[-1,1]} |f^{(r)}(x)|.$$

To decide where to put n and r reason, for which choices you obtain a bound that is decreasing in n.

Note that the L^∞-norm/maximum norm can also be indicated by the notations

$$\|\cdot\|_{\infty,-1,1} \quad \text{or} \quad \|\cdot\|_{\infty,[-1,1]} \quad \text{or} \quad \|\cdot\|_{\infty,]-1,1[}\,.$$

▲

(7-6.b) (8 min.) For $n \in \mathbb{N}$ we abbreviate

$$E_n(f) := \inf_{p\in\mathcal{P}_n} \|f-p\|_{L^\infty([-1,1])} \quad \text{for} \quad f \in C^0([-1,1])\,. \tag{7.6.2}$$

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)	$E_n(f+g) \leq E_n(f) + E_n(g)$	○	○
(ii)	$E_n(\lambda f) = \lambda E_n(f)$ for all $\lambda \in \mathbb{R}$	○	○
(iii)	E_n is a norm on $C^0([-1,1])$	○	○
(iv)	$E_n(f) - E_n(g) \leq E_n(f-g)$	○	○

HINT 1 for (7-6.b):

Definition [Lecture → Def. 1.5.5.4]. Norm

X = vector space over field $\mathbb{K}$, $\mathbb{K} = \mathbb{C}, \mathbb{R}$. A map $\|\cdot\| : X \mapsto \mathbb{R}_0^+$ is a norm on X, if it satisfies

(i) $\forall \mathbf{x} \in X$: $\mathbf{x} \neq 0 \quad \Leftrightarrow \quad \|\mathbf{x}\| > 0$ (definite),

(ii) $\|\lambda \mathbf{x}\| = |\lambda| \|\mathbf{x}\| \quad \forall \mathbf{x} \in X, \lambda \in \mathbb{K}$ (homogeneous),

(iii) $\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\| \quad \forall \mathbf{x}, \mathbf{y} \in X$ (triangle inequality).

⌟

SOLUTION of (7-6.b):

- (i) is **correct**: Denote by $p_f \in \mathcal{P}_n$ and $p_g \in \mathcal{P}_n$ two polynomials that realize the $\inf$ in (7.6.2) for f and g, respectively. Then, by the triangle inequality,

$$\begin{aligned} E_n(f+g) &\leq \|f + g - (p_f + p_g)\|_{L^\infty([-1,1])} \\ &\leq \|f - p_f\|_{L^\infty([-1,1])} + \|g - p_g\|_{L^\infty([-1,1])} = E_n(f) + E_n(g) \,. \end{aligned}$$

- (ii) is **wrong**, because for $\lambda = -1$ the assertion cannot hold, since a norm can never be negative.

  This assertion is "almost right", because a slight correction produces the correct assertion

$$E_n(\lambda f) = |\lambda| E_n(f) \quad \forall \lambda \in \mathbb{R} \,. \tag{7.6.3}$$

- (iii) is **wrong**, because $E_n(1) = 0$, since a constant can be exactly represented in any $\mathcal{P}_n$. However, a norm must not vanish for a non-zero element of the vector space.
- (iv) is **correct**, because it is a simple consequence of (i):

$$E_n(f) = E_n(f - g + g) \overset{\text{(i)}}{\leq} E_n(f-g) + E_n(g) \,.$$

▲

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 $f : [-1,1] \to \mathbb{R}$, consider the integral

$$I := \int_{-1}^{1} \arcsin(t)\, f(t)\, \mathrm{d}t. \tag{8.1.1}$$

We want to approximate this integral using global Gauss quadrature. The nodes (vector `x`) and the weights (vector `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 `w` and nodes `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`.

HINT 1 for (8-1.a): Use as exact value of the integral $I \approx 0.870267525725852642$.

SOLUTION of (8-1.a):

In our implementation we defined an auxiliary function that computes the integral for a given function and quadrature rule.

Gauss quadrature convergence

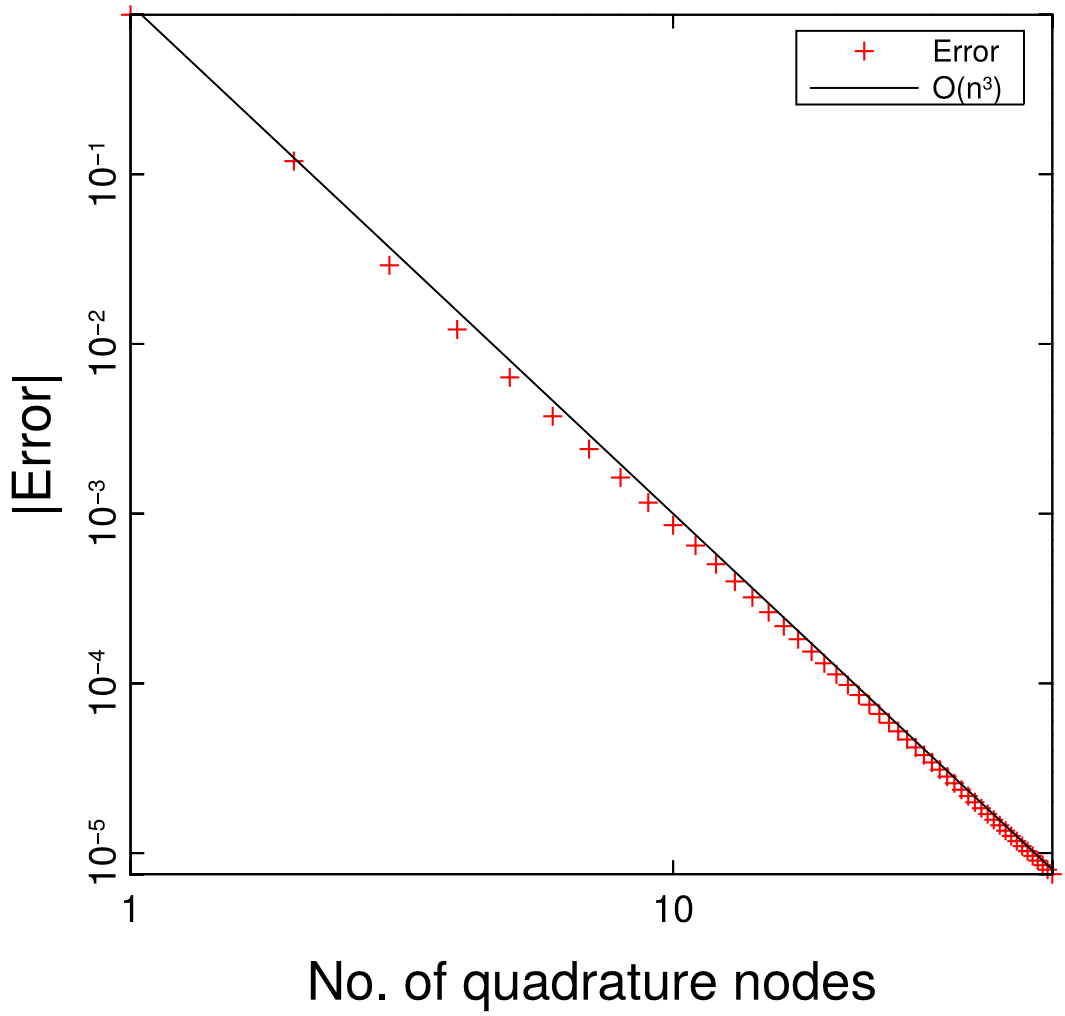


Fig. 32

C++-code 8.1.2: gaussConv.cpp

```
template <class Function>
void gaussConv(const Function& fh, const double I_ex) {

    std::vector<double> evals, // used to save no. of quad nodes
            error; // used to save the error
    // Build integrand
    auto f = [fh](double x) {
        return std::asin(x)*fh(x);
    };

    const unsigned N = 50; // Max. no. of nodes

    for (unsigned n = 1; n <= N; ++n) {
        QuadRule qr;
        gaussquad(n, qr); // Create quadrature rule

```

```
        double I = integrate(qr, f); // Compute integral

        evals.push_back(n); // Save no.  of quadrature nodes
        error.push_back(std::abs(I - I_ex)); // Save error
    }

    // Create convergence plots
    mgl::Figure fig;
    fig.title("Gauss quadrature convergence");
    fig.setlog(true, true); // log-log scaling
    fig.plot(evals, error, " +r").label("Error"); // plot error
    fig.fplot("x^(-3)", "k--").label("O(n^{-3})"); // reference line
    fig.xlabel("No. of quadrature nodes");
    fig.ylabel("|Error|");
    fig.legend();
    fig.save("GaussConv");
}
```

Get it on GitLab (gaussConv.cpp).

▲

(8-1.b) Describe qualitatively and quantitatively the asymptotic (for $n \to \infty$) convergence of the quadrature error you observe in (8-1.a).

HINT 1 for (8-1.b): Types of convergence have been introduced in [Lecture → Def. 6.1.2.7] for approximation errors. They apply to quadrature errors as well.

SOLUTION of (8-1.b):

The plot indicates algebraic convergence (being a straight line in log-log mode), of order approximately $O(n^{-3})$.

▲

(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 of (8-1.c):

With the change of variable $t = \sin(x)$, $\mathrm{d}t = \cos x \mathrm{d}x$ we obtain

$$I = \int_{-1}^{1} \arcsin(t)\, f(t)\, \mathrm{d}t = \int_{-\pi/2}^{\pi/2} x\, f(\sin(x)) \cos(x)\, \mathrm{d}x. \tag{8.1.3}$$

(The change of variable has to provide a smooth integrand on the integration interval.) The integrand is smooth, provided f is smooth, whence exponential convergence follows.

▲

(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 of (8-1.d):

The Gauss quadrature nodes and weights are defined on the interval $[-1, 1]$. We need to transform them to the interval $[-\frac{\pi}{2}, \frac{\pi}{2}]$.

Consider the following general affine transformation:

$$\begin{cases} \Phi : [-1,1] & \to [a,b] \\ \qquad \hat{x} & \to a + \frac{b-a}{2}(\hat{x}+1) = \frac{b-a}{2}\hat{x} + \frac{a+b}{2} \end{cases} \tag{8.1.4}$$

Then we can transform $\int_a^b g(x)\,dx$, the integral which needs to be computed over the domain $[a, b]$, as the following:

$$\int_a^b g(x)\,dx = \frac{b-a}{2}\int_{-1}^{1} g\left(\frac{b-a}{2}\hat{x} + \frac{a+b}{2}\right) d\hat{x} \approx \frac{b-a}{2}\sum_{i=1}^{n} \hat{w}_i f\left(\frac{b-a}{2}\hat{x} + \frac{a+b}{2}\right) \tag{8.1.5}$$

The nodes on $[a, b]$, $\{x_i\}_{i=1}^n$, can be directly obtained from the affine transformation above.

The weights on $[a, b]$, $\{w_i\}_{i=1}^n$, can be obtained by:

$$w_i = \frac{b-a}{2}\hat{w}_i \tag{8.1.6}$$

In our example, this means that $w_i = \frac{\pi}{2}\hat{w}_i$.

Gauss quadrature convergence

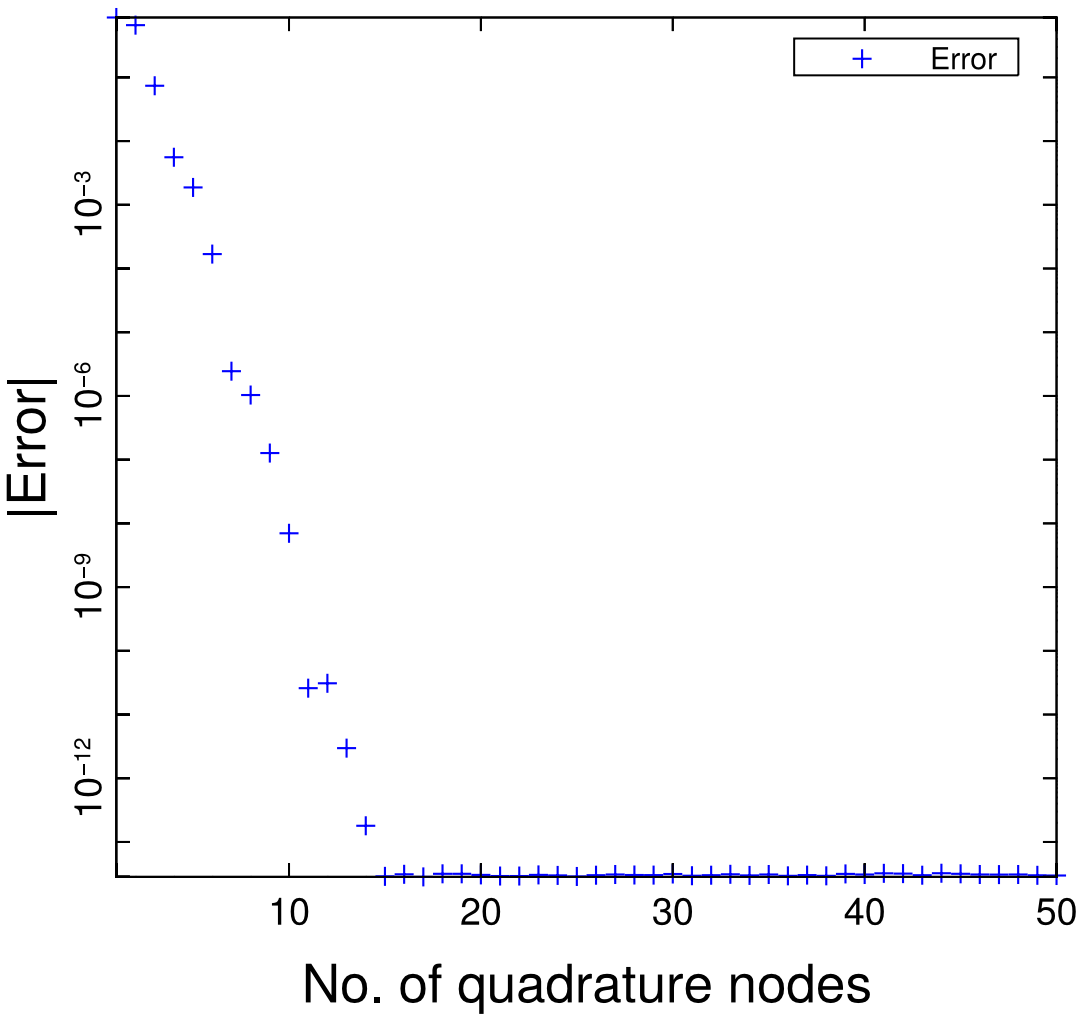


Fig. 33

C++-code 8.1.7: `GaussConvCV` solution.

```
template <class Function>
void gaussConvCV(const Function& f, const double I_ex) {
    std::vector<double> evals, // Used to save no. of quad nodes
                        error; // Used to save the error
    // Transform integrand
    auto g = [f] (double x) {
        return x*f(std::sin(x))*std::cos(x);
    };

    const unsigned N = 50; // Max. no. of nodes

    for (unsigned n = 1; n <= N; ++n) {
        QuadRule qr;
        gaussquad(n, qr); // Obtain quadrature rule

        // Transform nodes and weights to new interval
        Eigen::VectorXd w = qr.weights * M_PI/2;
        Eigen::VectorXd c = ( -M_PI/2 + M_PI/2*(qr.nodes.array() + 1) ).matrix();

```

```
        // Evaluate g at quadrature nodes c
        Eigen::VectorXd gc = c.unaryExpr(g);

        // Same as I = Σ_{i=1}^n w_i g(c_i)
        double I = w.dot(gc);

        evals.push_back(n); // Save no. of quadrature nodes
        error.push_back(std::abs(I - I_ex)); // Save error

    }

    // create convergence plots
    mgl::Figure fig;
    fig.title("Gauss quadrature convergence");
    fig.setlog(false, true); // lin-log scaling
    fig.plot(evals, error, " +b").label("Error"); // plot error
    fig.xlabel("No. of quadrature nodes");
    fig.ylabel("|Error|");
    fig.legend();
    fig.save("GaussConvCV");
}
```

Get it on GitLab (gaussConv.cpp).

▲

(8-1.e) Similar question as in Sub-problem (8-1.b): describe qualitatively the asymptotic (for $n \to \infty$) convergence of the quadrature error you observe in Sub-problem (8-1.d).

SOLUTION of (8-1.e):

The plot indicates exponential convergence (being a straight line in lin-log scale).

▲

(8-1.f) Explain the difference between the results obtained in Sub-problem (8-1.a) and Sub-problem (8-1.d).

SOLUTION of (8-1.f):

The convergence is now exponential. The integrand of the original integral belongs to $C^0([-1,1])$ but not to $C^1([-1,1])$, because the derivative of the $\arcsin$ function blows up in ± 1. The change of variable provides a perfectly smooth (analytic) integrand: $x\cos(x)\sinh(\sin x)$. Gauss quadrature enjoys exponential convergence only if the integrand is (at least) $\in C^\infty$ on the closed integration interval. Otherwise, one can only expect algebraic convergence.

▲

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

$$\int_0^1 f(x)dx \approx Af(0) + Bf'(0) + Cf(x_1) \tag{8.2.1}$$

is exact for polynomials of highest possible degree.

SOLUTION of (8-2.a):

The quadrature is exact for every polynomial $p(x) \in \mathcal{P}^n$ if and only if it is exact for $1, x, x^2, \dots, x^n$. If we apply the quadrature to the first monomials:

$$\begin{aligned}
1 &= \int_0^1 1dx = A \cdot 1 + B \cdot 0 + C \cdot 1 = A + C \\
\frac{1}{2} &= \int_0^1 xdx = A \cdot 0 + B \cdot 1 + C \cdot x_1 = B + Cx_1 \\
\frac{1}{3} &= \int_0^1 x^2dx = A \cdot 0 + B \cdot 0 + C \cdot x_1^2 = Cx_1^2 \\
\frac{1}{4} &= \int_0^1 x^3dx = A \cdot 0 + B \cdot 0 + C \cdot x_1^3 = Cx_1^3
\end{aligned}$$

Thus:

$$x_1 = \frac{3}{4},\ C = \frac{16}{27},\ B = \frac{1}{18},\ A = \frac{11}{27}$$

On the other hand, we have that:

$$\frac{1}{5} = \int_0^1 x^4dx \neq A \cdot 0 + B \cdot 0 + C \cdot x_1^4 = C \cdot x_1^4 = \frac{16}{27}\frac{81}{256}$$

Hence, the quadrature is exact for polynomials up to degree 3.

▲

(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

$$z'(t) = \frac{t}{1+t^2} \quad , \quad z(1) = 1\,. \tag{8.2.2}$$

HINT 1 for (8-2.b): Use the fundamental theorem of calculus and your result from the previous exercise. ⌟

SOLUTION of (8-2.b):

We know that

$$z(2) - z(1) = \int_1^2 z'(x)dx, \tag{8.2.3}$$

hence, applying the quadrature rule and the transformation $x \mapsto x+1$, we obtain:

$$z(2) = \int_0^1 z'(x+1)dx + z(1) \approx \frac{11}{27} \cdot z'(1) + \frac{1}{18} \cdot z''(1) + \frac{16}{27} \cdot z'\left(\frac{7}{4}\right) + z(1). \tag{8.2.4}$$

With $z''(x) = \frac{1-x^2}{(1+x^2)^2}$ and:

$$\begin{aligned} z(1) &= 1, \\ z'(1) &= \frac{1}{1+1^2} = \frac{1}{2}, \\ z''(1) &= \frac{1-1^2}{(1+1^2)^2} = 0, \\ z'\left(\frac{7}{4}\right) &= \frac{\frac{7}{4}}{1+\left(\frac{7}{4}\right)^2} = \frac{28}{65}, \end{aligned}$$

we obtain

$$z(2) = \int_0^1 z'(x+1)dx + z(1) \approx \frac{11}{27} \cdot \frac{1}{2} + \frac{1}{18} \cdot 0 + \frac{16}{27} \cdot \frac{28}{65} + 1 = 1.45897\ldots$$

For the sake of completeness, using the antiderivative of z':

$$z(2) = \int_1^2 z'(x)dx + z(1) = \frac{1}{2}\log(x^2+1)|_1^2 + 1 = 1.45815\ldots$$

▲

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 $f : \mathbb{R} \to \mathbb{R}$ that decay sufficiently fast for $|t| \to \infty$ (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:

$$E_T := \left| \int_{-\infty}^{\infty} g(t)dt - \int_{-b}^{b} g(t)dt \right| \leq 10^{-6} \,. \tag{8.3.1}$$

SOLUTION of (8-3.a):

An antiderivative of g is atan. Since the function g is even, we have

$$E_T = 2\int_{b}^{\infty} g(t)dt = \lim_{x\to\infty} 2\operatorname{atan}(x) - 2\operatorname{atan}(b) = \pi - 2\operatorname{atan}(b) \overset{!}{<} 10^{-6} \,, \tag{8.3.2}$$

i.e. $b > \tan\big((\pi - 10^{-6})/2\big) = \cot(10^{-6}/2) \approx 2 \cdot 10^{6}$.

▲

(8-3.b) ⚀ (5 min.) What is the algorithmic difficulty faced in the implementation of the truncation approach for a generic integrand?

SOLUTION of (8-3.b):

A good choice of b requires a detailed knowledge about the decay of f, which may not be available for f defined implicitly.

▲

A second option is the transformation of the improper integral to a bounded domain by substitution. For instance, we may use the map $t = \cot(s) := \frac{\cos s}{\sin s}$, $\cot :]0, \pi[\to]-\infty, \infty[$.

(8-3.c) ⚀ (10 min.) Into which integral does the substitution $t = \cot(s)$ convert $\int_{-\infty}^{\infty} f(t)dt$?

SOLUTION of (8-3.c):

We have:

$$\frac{dt}{ds} = -\left(1+\cot^2(s)\right) = -\left(1+t^2\right) \tag{8.3.3}$$

$$\int_{-\infty}^{\infty} f(t)\,dt = -\int_{\pi}^{0} f(\cot(s))\left(1+\cot^2(s)\right)ds = \int_0^{\pi} \frac{f(\cot(s))}{\sin^2(s)}\,ds\,, \tag{8.3.4}$$

because $\sin^2(\theta) = \frac{1}{1+\cot^2(\theta)}$.

▲

(8-3.d) (10 min.) Write down the transformed integral explicitly for the integrand $g(t) := \frac{1}{1+t^2}$. Simplify the integrand.

HINT 1 for (8-3.d): Use identities involving squares of trigonometric functions, see, for instance the Pythagorean trigonometric identities online.

SOLUTION of (8-3.d):

Replacing the function $g(t)$, we get:

$$\int_0^{\pi} \frac{1}{1+\cot^2(s)}\frac{1}{\sin^2(s)}ds = \int_0^{\pi} \frac{1}{\sin^2(s)+\cos^2(s)}ds = \int_0^{\pi} ds = \pi$$

▲

(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].

HINT 1 for (8-3.e): A lambda function with signature `(double) -> double` automatically satisfies the requirement for f.

SOLUTION of (8-3.e):

C++-code 8.3.5: Implementation of transformed quadrature rule

```
template <class Function>
double quad(Function &&f, const Eigen::VectorXd &w,
            const Eigen::VectorXd &x, double a, double b) {
  double I = 0;
  for (int i = 0; i < w.size(); ++i) {
    I += f((x(i) + 1) * (b - a) / 2 + a) * w(i);
```

```
  }
  return l * (b - a) / 2.;
}
```

C++-code 8.3.6: Implementation of `quadinf()`

```
template <class Function> double quadinf(const int n, Function &&f) {
  Eigen::VectorXd w, x;
  // Compute nodes and weights of Gauss quadrature rule
  // using Golub-Welsh algorithm

  golubwelsh(n, w, x);
  //!  NOTE: no function cot available in c++, need to resort to trigonometric
  //!  identities.  Both lines below are valid, the first computes three
  //!  trigonometric functions, but the second is prone to cancellation
  auto ftilde = [&f](double x) {
    return f(std::cos(x) / std::sin(x)) / pow(std::sin(x), 2);
  };
  /* auto ftilde = [&f] (double x) { double cot = std::tan(PI_HALF - x); return
   * f(cot) * (1. + pow(cot,2)); }; */
  return quad(ftilde, w, x, 0, PI);
}
```

▲

(8-3.f) (15 min.) Implement a C++ function

```
void cvgQuadInf(void);
```

in order to study the convergence for $n \to \infty$ of the quadrature method implemented in the previous subproblem for the integrand $h(t) := \exp\left(-(t-1)^2\right)$ (shifted Gaussian). To compute the error you can use that $\int_{-\infty}^{\infty} h(t)dt = \sqrt{\pi}$.

Use MATPLOTLIBCPP to create a suitable plot of the quadrature error versus the number of quadrature nodes. What kind of convergence do you observe?

SOLUTION of (8-3.f):

C++-code 8.3.7: Implementation of `cvgQuadInf()`

```
void cvgQuadInf(void) {
  // Number of max Gauss pts.
  const int N = 100;
  plt::figure();
  // TO DO (8-3.f): plot convergence errors against number of integration nodes
  // START
  // variables for the plots
  std::vector<int> num_nots;
  std::vector<double> err;
  std::vector<double> cvg_rate;

```

```
  // Integrand and exact integral
  auto f = [](double t) { return std::exp(-std::pow((t - 1), 2)); };
  double I_ex = std::sqrt(PI);

  for (int n = 5; n <= N; n += 5) {
    // Approximated value of integral
    double QS = quadinf(n, f);
    // plot data
    num_nots.push_back(n);
    err.push_back(std::abs(QS - I_ex));
    //'straight' line in plot
    cvg_rate.push_back(std::pow(0.73, n));
  }
  plt::semilogy(num_nots, err, "+", {{"label", "Error"}});
  plt::semilogy(num_nots, cvg_rate, "--", {{"label", "O(0.73^n)"}});
  plt::legend("best");
  plt::xlabel("No. of quadrature nots");
  plt::ylabel("|Error|");

  plt::grid("True");
  // END
  plt::savefig("./cx_out/convergence.png");
}
```

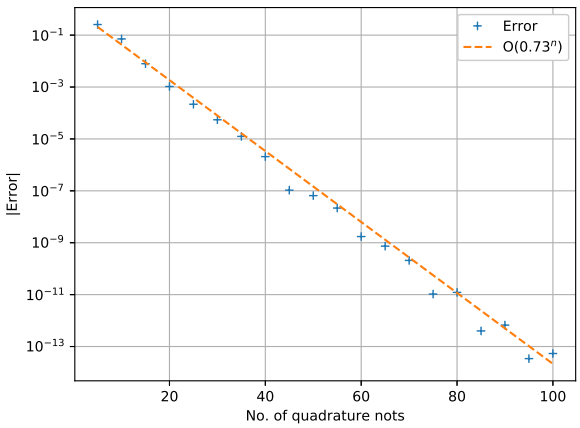


Fig. 34

According to [Lecture → Rem. 6.1.2.9] we observe exponential convergence: The graph of the quadrature error vs. the number of quadrature nodes in lin-log scale is approximately a line with negative slope.

▲

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 conmnected with [Lecture → Chapter 7].

A laser beam has intensity

$$I(x,y) := \exp(-\alpha((x-p)^2 + (y-q)^2)) \,, \quad x,y \in \mathbb{R} \tag{8.4.1}$$

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 of (8-4.a):

The radiant power absorbed by Δ can be written as:

$$\int_\Delta I(x,y)\, dx\, dy = \int_0^1 \int_0^{1-y} I(x,y)\, dx\, dy \,.$$

This is only one way to express the power. You may also change the order of integration or perform some other transformation of the integral.

▲

(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 of (8-4.b):

We transform the quadrature rule according to the formulas derived in [Lecture → Rem. 7.1.0.4].

C++-code 8.4.2: Implementation of `evalquad()`.

```
template <class Function>
double evalquad(const double a, const double b, Function &&f,
                const QuadRule &Q) {
  double I;
  // TO DO: (8-4.b) Use Q to approximate the integral of f over [a,b]
  // START
  I = 0;
  // Loop over all node/weight pairs
  for (int i = 0; i < Q.weights.size(); ++i) {
    I += f((Q.nodes(i) + 1) * (b - a) / 2 + a) * Q.weights(i);
  }
  I = I * (b - a) / 2.;
  // END
  return I;
}
```

▲

(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

$$\int_{\Delta} f(x,y)\,dx\,dy \tag{8.4.3}$$

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 $[-1,1]$.

HINT 1 for (8-4.c): Write (8.4.3) explicitly as a double integral as you were asked to do in Sub-problem (8-4.a). Take special care to correctly find the intervals of integration. ⌟

HINT 2 for (8-4.c): C++ lambda functions are very useful for this kind of implementation, see [Lecture → Section 0.2.3]. ⌟

SOLUTION of (8-4.c):

As we have seen in Sub-problem (8-4.a), the integral can be written as

$$\int_{\Delta} f(x,y)\,dx\,dy = \int_0^1 \int_0^{1-y} f(x,y)\,dx\,dy$$

In the C++ implementation, we define the auxiliary (lambda) function f_y:

$$\forall y \in [0,1],\ f_y : [0,1-y] \to \mathbb{R},\ x \mapsto f_y(x) := f(x,y)$$

We also define the (lambda) approximated integrand:

$$g(y) := \int_0^{1-y} f_y(x)dx \approx \frac{1-y}{2}\sum_{i=0}^{n} w_i f_y\left(\frac{1-y}{2}(x_i+1)\right) =: \mathcal{I}(y), \tag{8.4.4}$$

the integral of which can be approximated, using a nested Gauss quadrature:

$$\int_\Delta f(x,y)\,dx\,dy = \int_0^1 \int_0^{1-y} f_y(x)\,dx\,dy = \int_0^1 g(y)dy \approx \frac{1}{2}\sum_{j=1}^{n} w_j \mathcal{I}\left(\frac{y_j+1}{2}\right). \tag{8.4.5}$$

C++-code 8.4.6: Implementation of `gaussquadtriangle()`.

```
template <class Function>
double gaussquadtriangle(const Function &f, const unsigned N) {
  double I;
  // TO DO: (8-4.c) Use N-node gaussquad() to integrate f over
  // the triangle 0<=x,y, x+y<=1.
  // START
  // Get nodes/weights for integral over dx and dy
  QuadRule Q;
  gaussquad(N, Q);

  // We define the function g as the function of y
  // g(y) := ∫_0^{1-y} f(x,y)dx.
  auto g = [&f, &Q](double y) {
    return evalquad(
        0, 1 - y, [&f, &y](double x) { return f(x, y); }, Q);
  };
  // We integrate the function g over y from 0 to 1 to get
  // ∫_0^1 g(y)dy := ∫_0^1 ∫_0^{1-y} f(x,y)dxdy.
  I = evalquad(0, 1, g, Q);
  // END
  return I;
}
```

This problem can also be solved using loops (so no nested lambda function). This more straightforward approach is listed next.

C++-code 8.4.7: Loop-based implementation of `gaussquadtriangle()`.

```
// EQUIVALENT: Loop based, copy-and-paste implementation
template <class Function>
double gaussquadtriangle_loop(const Function& f, const unsigned N) {
  // Get nodes/weights for integral over dx and dy
  QuadRule Q;
  gaussquad(N, Q);
  // Integration over y from 0 to 1 of g(y) := ∫_0^{1-y} I(x,y)dx
  double I = 0;
  double a = 0., b  = 1.;
  for(int i = 0; i < Q.weights.size(); ++i) {
    // Find out the y at which we are
    double y = (Q.nodes(i) + 1) * (b - a) / 2 + a;
    // Define f_y(x) (y is fixed and f_y is a function of x)
    auto f_y = [&f, &y] (double x) { return f(x,y); };
    // Compute g(y) as ∫_0^{1-y} I(x,y)dx
    I += evalquad(0, 1-y, f_y, Q) * Q.weights(i);
  }
  // Rescale interval
  return I * (b - a) / 2.;
}
```

If you choose that implementation, also have a look at the nested-lambda version listed in Code 8.4.6 and try to understand how it works.

▲

(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 $\int_\Delta I(x,y)\,\mathrm{d}x\mathrm{d}y$ using the parameters $\alpha = 1,\ p = 0,\ q = 0$. Tabulate and plot (using MATPLOTLIBCPP, storing the figure in `convergence.png`) the quadrature error w.r.t. to the number of nodes $n = 1,2,3,\dots,20$ using the "exact" value of the integral $I \approx 0.366046550000405$. What kind of convergence do you observe and why?

HINT 1 for (8-4.d): [Lecture → Rem. 6.1.2.9] explains how to distinguish different kinds of convergence from error plots. ⌟

SOLUTION of (8-4.d):

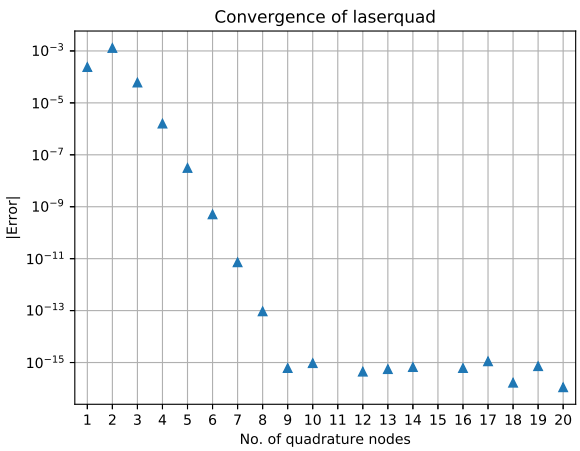


Fig. 35

We observe that the error points approximately lie on a straight line in the linear-logarithmic error plot, before the impact of roundoff errors becomes visible at very low error levels.
Thus we conclude that the implemented quadrature enjoys exponential convergence, recall [Lecture → Def. 6.1.2.7] and [Lecture → Rem. 6.1.2.9].
We expect from theoretical considerations that the convergence is exponential, because we deal with an perfectly smooth (analytic) integrand, see [Lecture → § 7.3.0.39] and [Lecture → Ex. 7.3.0.46].

C++-code 8.4.8: Implementation of `convtest2DQuad()`.

```
void convtest2DQuad(unsigned int nmax = 20) {
  // "Exact" integral
  const double I_ex = 0.366046550000405;

  // TO DO: (8-4.d) Tabulate the error of gaussquadtriangle() for
  // a laser beam intensity, using n=1,2,3,...,nmax nodes.
  // START
  // Parameters
  const double alpha = 1, p = 0, q = 0;
  // Laser beam intensity
  auto I = [alpha, p, q](double x, double y) {
    return std::exp(-alpha * ((x - p) * (x - p) + (y - q) * (y - q)));
  };

  // memory allocation for plot
  std::vector<double> err(nmax);
  std::vector<int> num_pts(nmax);

  std::cout << std::setw(3) << "N" << std::setw(15) << "I_approx"
```

```
            << std::setw(15) << "error" << std::endl;
  for (unsigned n = 1; n <= nmax; n++) {
    double I_approx = gaussquadtriangle(I, n);

    err[n-1] = std::abs(I_ex - I_approx);
    num_pts[n-1] = n;

    std::cout << std::setw(3) << n << std::setw(15) << I_approx << std::setw(15)
              << err[n-1] << std::endl;
  }
  // Error plot rendered by matplotlibcpp
  plt::figure();
  plt::semilogy(num_pts, err, "^", {{"label", "error"}});
  plt::xlim(0.5,nmax+0.5);
  plt::xticks(num_pts);
  plt::xlabel("No. of quadrature nodes");
  plt::ylabel("|Error|");
  plt::title("Convergence of laserquad");
  plt::grid("True");
  plt::savefig("./cx_out/convergence.png");
  // END
}
```

▲

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:

$$\begin{aligned} &\text{function A:} \quad f_A \in C^\infty(I)\,, \quad f_A \notin \mathcal{P}_k \;\forall\, k \in \mathbb{N}\,; \\ &\text{function B:} \quad f_B \in C^0(I)\,, \quad f_B \notin C^1(I)\,; \\ &\text{function C:} \quad f_C \in \mathcal{P}_{12}\,, \end{aligned}$$

where $\mathcal{P}_k$ is the space of the polynomials of degree at most k defined on I. The following quadrature rules are applied to these functions:

- quadrature rule A is a global Gauss quadrature;
- quadrature rule B is a composite trapezoidal rule;
- quadrature rule C is a composite 2-point Gauss quadrature.

The corresponding absolute values of the quadrature errors are plotted against the number of function evaluations in the figure below. Notice that only the quadrature errors obtained with an even number of function evaluations are shown.

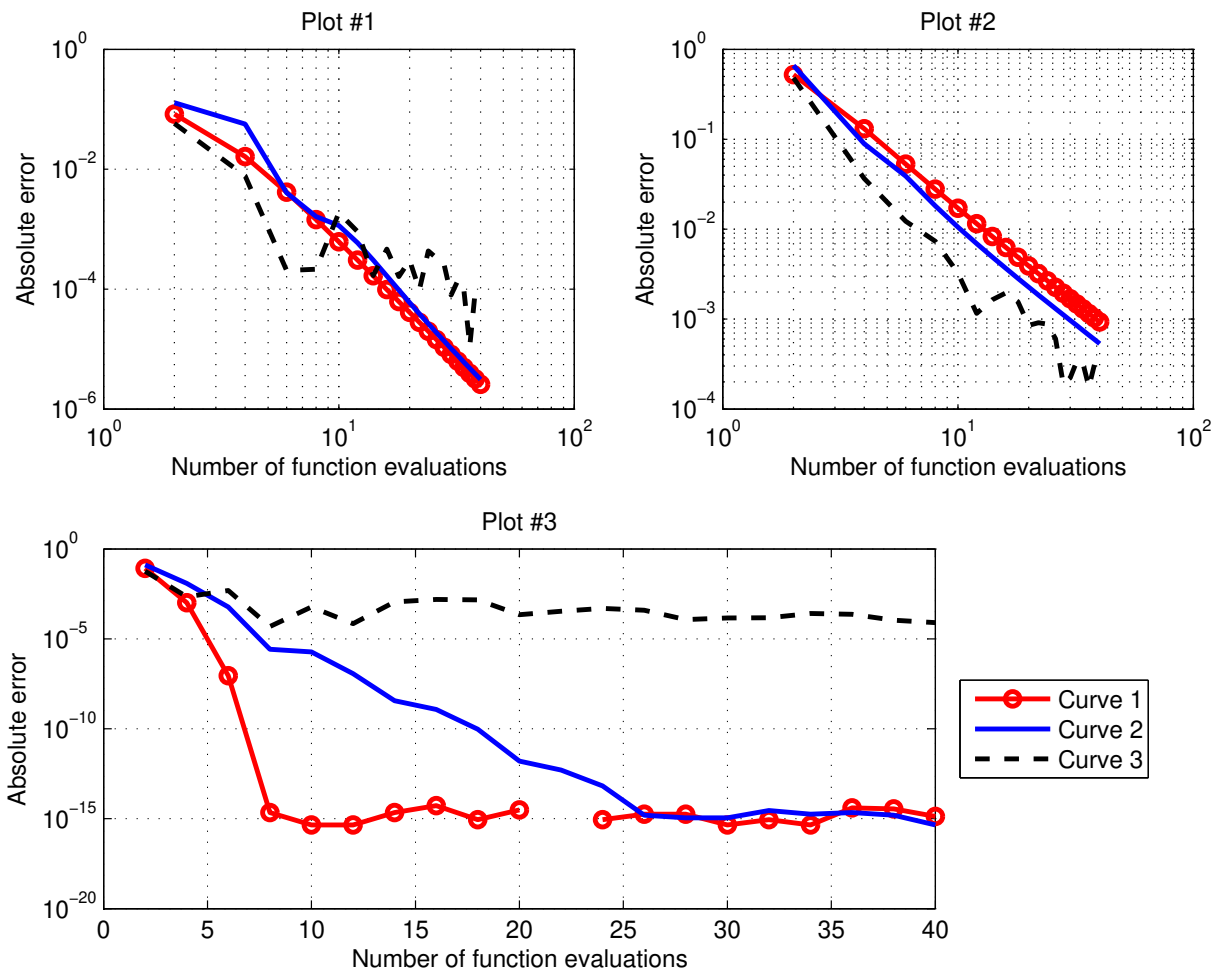


Fig. 36

(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.

HINT 1 for (8-5.a): Notice the different axis scales in the plots. ⌟

SOLUTION of (8-5.a):

Plot #1 — Quadrature rule C, Composite 2-point Gauss:
algebraic convergence for every function, about 4th order for two functions.

Plot #2 — Quadrature rule B, Composite trapezoidal:
algebraic convergence for every function, about 2nd order.

Plot #3 — Quadrature rule A, Global Gauss:
algebraic convergence for one function, exponential for another one, exact integration with 8 evaluations for the third one.

▲

(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 of (8-5.b):

Curve 1 red line and small circles — f_C polynomial of degree 12:
integrated exactly with 8 evaluations with global Gauss quadrature.

Curve 2 blue continuous line only — f_A analytic function:
exponential convergence with global Gauss quadrature.

Curve 3 black dashed line — f_B non smooth function:
algebraic convergence with global Gauss quadrature.

▲

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 Chebyshev polynomials of the second kind U_n, defined as

$$U_n(t) = \frac{\sin((n+1)\arccos t)}{\sin(\arccos t)}, \qquad n \in \mathbb{N}.$$

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

$$U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t), \qquad U_0(t) = 1, \qquad U_1(t) = 2t,$$

for every $n \geq 1$.

SOLUTION of (8-6.b):

$\mathbf{n = 0}$:

This case is trivial, since $U_0(t) = \frac{\sin(\arccos t)}{\sin(\arccos t)} = 1$, as desired.

$\mathbf{n = 1}$:

Using the trigonometric identity $\sin 2x = \sin x \cos x$, we have

$$U_1(t) = \frac{2\sin(\arccos t)}{\sin(\arccos t)} = 2\cos\arccos t = 2t.$$

$\mathbf{n \geq 2}$:

Using the identity $\sin(x+y) = \sin x \cos y + \sin y \cos x$, we obtain:

$$\begin{aligned} U_{n+1}(t) &= \frac{\sin((n+1)\arccos t)t + \cos((n+1)\arccos t)\sin(\arccos t)}{\sin(\arccos t)} \\ &= U_n(t)t + \cos((n+1)\arccos t). \end{aligned}$$

Similarly, we have

$$\begin{aligned} U_{n-1}(t) &= \frac{\sin((n+1-1)\arccos t)}{\sin(\arccos t)} \\ &= \frac{\sin((n+1)\arccos t)t - \cos((n+1)\arccos t)\sin(\arccos t)}{\sin(\arccos t)} \\ &= U_n(t)t - \cos((n+1)\arccos t). \end{aligned}$$

Combining the last two equalities we obtain the desired 3-term recursion.

▲

(8-6.c) ⚀ (5 min.) Show that $U_n \in \mathcal{P}_n$ and has leading coefficient 2^n.

SOLUTION of (8-6.c):

Let us prove the claim by induction.

$\mathbf{n = 0}$: This case is trivial, since $U_0(t) = 1$.

Induction hypothesis: Let us now assume that the statement is true for every $k = 0, \dots, n$ and let us prove it for $n + 1$.

In view of $U_{n+1}(t) = 2tU_n(t) - U_{n-1}(t)$, since by inductive hypothesis $U_n \in \mathcal{P}_n$ and $U_{n-1} \in \mathcal{P}_{n-1}$, we have that $U_{n+1} \in \mathcal{P}_{n+1}$. Moreover, the leading coefficient will be 2 times the leading order coefficient of U_n, namely 2^{n+1}, as desired. □

▲

(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 of (8-6.d):

With the substitution $t = \cos s$ we obtain

$$\begin{aligned}
\int_{-1}^{1} \sqrt{1-t^2} U_m(t) U_n(t)\, dt &= \int_{-1}^{1} \sqrt{1-t^2} \frac{\sin((m+1)\arccos t)\sin((n+1)\arccos t)}{\sin^2(\arccos t)}\, dt \\
&= \int_0^{\pi} \sin s \frac{\sin((m+1)s)\sin((n+1)s)}{\sin^2 s} \sin s\, ds \\
&= \int_0^{\pi} \sin((m+1)s)\sin((n+1)s)\, ds \\
&= \frac{1}{2}\int_0^{\pi} \cos((m-n)s) - \cos((m+n+2)s)\, ds.
\end{aligned}$$

▲

(8-6.e) ⚁ (10 min.) What are the zeros ζ_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 of (8-6.e):

From the definition of U_n we immediately find that the zeros are given by

$$\zeta_j^n = \cos\left(\frac{j}{n+1}\pi\right), \qquad j = 1, \dots, n. \tag{8.6.1}$$

▲

(8-6.f) ⚃ (30 min.) [depends on Sub-problem (8-6.d)]

Show that the choice of weights

$$w_j = \frac{\pi}{n+1}\sin^2\left(\frac{j}{n+1}\pi\right), \qquad j = 1,\dots,n,$$

ensures that the quadrature formula

$$Q_n^U(f) = \sum_{j=1}^{n} w_j f(\xi_j^n) \tag{8.6.2}$$

provides the exact value of the integral

$$I := \int_{-1}^{1} \sqrt{1-t^2} f(t)\,\mathrm{d}t$$

for $f \in \mathcal{P}_{n-1}$ (assuming exact arithmetic).

SOLUTION of (8-6.f):

Since U_k is a polynomial of degree exactly k, the set $\{U_k : k = 0,\dots,n-1\}$ is a basis of $\mathcal{P}_{n-1}$. Therefore, by linearity it suffices to prove the above identity for $f = U_k$ for every k. Fix $k = 0,\dots,n-1$. Setting $x = \pi/(n+1)$, from (**??**) we readily derive

$$\begin{aligned}
\sum_{j=1}^{n} w_j U_k(\xi_j^n) &= \sum_{j=1}^{n} \frac{\pi}{n+1}\sin^2\left(\frac{j}{n+1}\pi\right)\frac{\sin((k+1)\arccos\xi_j^n)}{\sin(\arccos\xi_j^n)} \\
&= x\sum_{j=1}^{n}\sin(jx)\sin((k+1)jx) \\
&= \frac{x}{2}\sum_{j=1}^{n}(\cos((k+1-1)jx) - \cos((k+1+1)jx)) \\
&= \frac{x}{2}\operatorname{Re}\sum_{j=0}^{n}\left(e^{ikxj} - e^{i(k+2)xj}\right) \\
&= \frac{x}{2}\operatorname{Re}\left(\sum_{j=0}^{n} e^{ikxj} - \frac{1-e^{i\pi(k+2)}}{1-e^{i(k+2)x}}\right).
\end{aligned}$$

Thus, for $k = 0$ we have

$$\sum_{j=1}^{n} w_j U_0(\xi_j^n) = \frac{x}{2}\operatorname{Re}\left(\sum_{j=0}^{n} 1 - \frac{1-e^{2\pi i}}{1-e^{2xi}}\right) = \frac{x}{2}\operatorname{Re}((n+1)-0) = \frac{\pi}{2}.$$

On the other hand, if $k = 1,\dots,n-1$ we obtain

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{x}{2}\operatorname{Re}\left(\frac{1-e^{i\pi k}}{1-e^{ikx}} - \frac{1-e^{i\pi(k+2)}}{1-e^{i(k+2)x}}\right) = \frac{(1-(-1)^k)x}{2}\operatorname{Re}\left(\frac{1}{1-e^{ikx}} - \frac{1}{1-e^{i(k+2)x}}\right).$$

In view of the elementary equality $(a+ib)(a-ib) = a^2+b^2$ we have $\mathrm{Re}(1/(a+ib)) = a/(a^2+b^2)$. Thus

$$\mathrm{Re}\left(\frac{1}{1-e^{ikx}}\right) = \mathrm{Re}\left(\frac{1}{1-\cos(kx)-i\sin(kx)}\right) = \frac{1-\cos(kx)}{(1-\cos(kx))^2+\sin(kx)^2} = \frac{1}{2}$$

Arguing in a similar way we have $\mathrm{Re}\,(1-e^{i(k+2)x})^{-1} = 1/2$. Therefore for $k = 1,\dots,n-1$ we have

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{(1-(-1)^k)x}{2}\left(\frac{1}{2}-\frac{1}{2}\right) = 0.$$

To summarise, we have proved that

$$\sum_{j=1}^{n} w_j U_k(\xi_j^n) = \frac{\pi}{2}\delta_{k0}, \qquad k = 0,\dots,n-1.$$

Finally, the claim follows from Sub-problem (8-6.d), since $U_0(t) = 1$.

▲

(8-6.g) ⚀ Show that the quadrature formula (8.6.2) gives the exact value of (**??**) even for every $f \in \mathcal{P}_{2n-1}$.

HINT 1 for (8-6.g): Refer to [Lecture → Thm. 7.3.0.23]. ⌟

SOLUTION of (8-6.g):

The conclusion follows by applying the same argument given in [Lecture → Thm. 7.3.0.23] with the weighted L^2 scalar product with weight w defined above.

▲

(8-6.h) ⚁ Show that the quadrature error

$$|Q_n^U(f) - W(f)|$$

decays to 0 exponentially as $n \to \infty$ for every $f \in C^\infty([-1,1])$ that admits an analytic extension to an open subset of the complex plane.

HINT 1 for (8-6.h): See [Lecture → § 7.3.0.39]. ⌟

SOLUTION of (8-6.h):

By definition, the weights defined above are positive, and the quadrature rule is exact for polynomials up to order $2n-1$. Therefore, arguing as in [Lecture → § 7.3.0.39], we obtain the exponential decay, as desired.

▲

(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 of (8-6.i):

The full code can be found in `quadU.cpp`.

C++ code 8.6.3: Solution for `quadU`.

```
template<typename Function>
double quadU(const Function& f, const unsigned n) {
    // Value of integral
    double I = 0;
    for (unsigned j = 0; j < n; j++) {

        // Weight
        double w = M_PI/(n+1) * std::pow(std::sin((j+1)*M_PI/(n+1)), 2);

        // Node
        double xi = std::cos( (j+1.)/(n+1)*M_PI );

        I += w*f(xi);
    }
    return I;
}
```

▲

(8-6.j) ⚀ Devise a C++ function

```
void testQuadU(unsigned int nmax = 20);
```

that, for the function $f(t) = 1/(2+e^{3t})$ and $n = 1,\dots,n_{\max}$ 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 $0 \le q < 1$ in the asymptotic decay law $E_n(f) \approx Cq^n$ characterising (sharp) exponential convergence, see [Lecture → Def. 6.1.2.7].

SOLUTION of (8-6.j):

The full code can be found in `quadU.cpp`.

C++ code 8.6.4: Computation of convergence of `quadU`.

```
    // Integrand
    auto f = [](double t) { return 1. / (2+std::exp(3*t)); };

    // "Exact" value of integral
    const double exact = 0.483296828976607;

```

```
    // Store error
    std::vector<double> e(25);
    // Header
    std::cout << std::setw(5) << "n"
              << std::setw(20) << "Error"
              << std::setw(20) << "q"
              << std::endl;
    for (unsigned int n = 0; n < 25; n++) {
        // Compute error
        e[n] = std::abs(quadU(f,n+1)-exact);
        if (n > 1)
            // Print table
            std::cout << std::setw(5) << n
                      << std::setw(20) << e[n]
                      << std::setw(20) << e[n]/e[n-1]
                      << std::endl;
    }
    // After the 18th iteration only roundoff error is present.
```

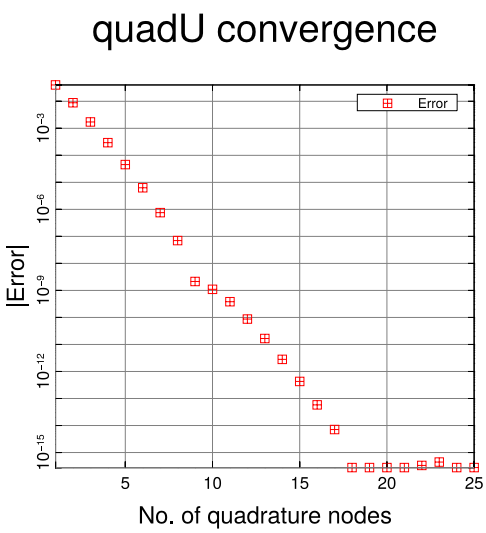


Fig. 37

An approximation of q is given by $E_n(f)/E_{n-1}(f)$:

n	Error	Approximated q
2	0.00171981	0.194545
3	0.000294532	0.171259
4	4.54948e-05	0.154465
5	6.30403e-06	0.138566
6	7.57006e-07	0.120083
7	6.99966e-08	0.0924651
8	2.14329e-09	0.0306199
9	1.09843e-09	0.512495
10	3.82291e-10	0.348035
11	8.71921e-11	0.228078
12	1.66109e-11	0.190509
13	2.80703e-12	0.168988
14	4.28158e-13	0.15253
15	5.80647e-14	0.135615
16	7.10543e-15	0.122371
17	2.77556e-16	0.0390625
18	2.77556e-16	1
19	2.77556e-16	1

▲

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*

$$I(f) := \int_0^2 \sqrt{\frac{2-t}{t}} f(t)\,\mathrm{d}t\,. \tag{8.7.1}$$

(8-7.a) ⚁ Transform $I(f)$ into an integral over $[-1,1]$, whose integrand is $\in C^\infty([-1,1])$ provided that $f \in C^\infty([0,2])$.

HINT 1 for (8-7.a): First split the integral into integrals over $[0,1]$ and $[1,2]$ in order to isolate the two different singularities of the integrand. Apply different transformations to the two parts. ⌟

HINT 2 for (8-7.a): Use the substitution $s = \sqrt{t}$ to deal with the singularity in $t = 0$ and $s = \sqrt{2-t}$ to get rid of the singularity in $t = 2$. ⌟

SOLUTION of (8-7.a):

We wish to transform the integral in $I(f)$ into an integral of a smooth function over a bounded interval. In order to this, write

$$I(f) = \int_0^1 \frac{\sqrt{2-t}}{\sqrt{t}} f(t)\,dt + \int_1^2 \frac{\sqrt{2-t}}{\sqrt{t}} f(t)\,dt.$$

Performing the substitutions $s = \sqrt{t}$ in the first integral and $s = \sqrt{2-t}$ in the second integral we obtain

$$I(f) = 2\int_0^1 \sqrt{2-s^2} f(s^2)\,ds + 2\int_0^1 \frac{s^2}{\sqrt{2-s^2}} f(2-s^2)\,ds = 2\int_0^1 f_1(s)\,ds + 2\int_0^1 f_2(s)\,ds$$

where $f_1(s) = \sqrt{2-s^2} f(s^2)$ and $f_2(s) = \frac{s^2}{\sqrt{2-s^2}} f(2-s^2)$. With the change of variables $t = 2s-1$ we finally obtain

$$I(f) = \int_{-1}^1 f_1((t+1)/2)\,ds + \int_{-1}^1 f_2((t+1)/2)\,ds. \tag{8.7.2}$$

Observe that we never get close to the singularity of the root terms.

▲

(8-7.b) ⚁ (20 min.) For $n \in \mathbb{N}^*$, derive a family of $2n$-point quadrature formulas

$$Q_n(f) = \sum_{j=1}^{2n} w_j^n f(c_j^n), \qquad f \in C([0,2])$$

for which $Q_n(f)$ converges to $I(f)$ exponentially as $n \to \infty$, if $f \in C^\infty([0,2])$.

Remark. A more precise statement of the problem would be ", if f has an analytic extension to a complex neighborhood of $[0,2]$". This is, how the statement $f \in C^\infty([0,2])$ above should be read.

SOLUTION of (8-7.b):

Applying Gauss quadrature to these integrals in (8.7.2), we define

$$Q_n(f) = \sum_{j=1}^{n} \tilde{w}_j^n f_1((\xi_j^n + 1)/2) + \sum_{j=1}^{n} \tilde{w}_j^n f_2((\xi_j^n + 1)/2),$$

where ξ_j^n and $\tilde{w}_j^n$ are nodes and weights of the Gauss quadrature of order n defined on $[-1,1]$. (Alternatively, it is also possible to use the Clenshaw-Curtis quadrature rule.) By construction, we have that $Q_n(f) \to I(f)$ exponentially as $n \to \infty$ if f has an analytic extension to a neighborhood of $[1,2]$, since this implies that the integrands have an analytic extension to a neighborhood of $[-1,1]$.

Define now the nodes

$$c_j^n = \begin{cases} ((\xi_j^n + 1)/2)^2 & \text{if } j = 1,\dots,n, \\ 2 - ((\xi_{j-n}^n + 1)/2)^2 & \text{if } j = n+1,\dots,2n. \end{cases}$$

Therefore, the above expression can be written as

$$Q_n(f) = \sum_{j=1}^{n} \tilde{w}_j^n \sqrt{2 - c_j^n} f(c_j^n) + \sum_{j=1}^{n} \tilde{w}_j^n \frac{2 - c_{j+n}^n}{\sqrt{c_{j+n}^n}} f(c_{j+n}^n) = \sum_{j=1}^{2n} w_j^n f(c_j^n),$$

where the weights are given by

$$w_j^n = \begin{cases} \tilde{w}_j^n \sqrt{2 - c_j^n} & \text{if } j = 1,\dots,n, \\ \tilde{w}_{j-n}^n \frac{2 - c_j^n}{\sqrt{c_j^n}} & \text{if } j = n+1,\dots,2n. \end{cases}$$

▲

(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 of (8-7.c):

The above expression is never exact for polynomials f, because of the presence of the square roots. In other words, $d = 0$, for which $\mathcal{P}_{-1} = \emptyset$.

▲

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

$$(\mathsf{T}f)(x) = \int_I k(x,y)f(y)\,\mathrm{d}y\,, \quad x \in I \subset \mathbb{R}\,, \tag{8.8.1}$$

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 of (8-8.a):

Applying Gauss quadrature to the expression for $g(\xi_l^n)$ we obtain $g_n(\xi_l^n) = \sum_{j=1}^n w_j^n e^{|\xi_l^n - \xi_j^n|} f(\xi_j^n)$, whence $M_{lj} = w_j^n e^{|\xi_l^n - \xi_j^n|}$ for $l,j = 1,\dots,n$.

▲

(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 weigths `w` and the ordered nodes `xi` relative to the `n`-point Gauss quadrature on the interval $[-1,1]$.

SOLUTION of (8-8.b):

In order to write a function with optimal complexity, we need to use the structure of the matrix M. Write

$$g_n(\xi_l^n) = \sum_{j=1}^{n} w_j^n e^{|\xi_l^n - \xi_j^n|} f(\xi_j^n) = \sum_{j=1}^{l-1} w_j^n e^{\xi_l^n} e^{-\xi_j^n} f(\xi_j^n) + \sum_{j=l}^{n} w_j^n e^{-\xi_l^n} e^{\xi_j^n} f(\xi_j^n).$$

Setting $p_l^n = \sum_{j=1}^{l-1} w_j^n e^{-\xi_j^n} f(\xi_j^n)$ and $q_l^n = \sum_{j=l}^{n} w_j^n e^{\xi_j^n} f(\xi_j^n)$, this identity can be rewritten as

$$g_n(\xi_l^n) = e^{\xi_l^n} p_l^n + e^{-\xi_l^n} q_l^n.$$

This expression for $l = 1, \dots, n$ can be computed with $O(n)$ operations. It remains to compute p_l^n and q_l^n in $O(n)$ operations. This can be done by using these recursive formulas, that immediately follow from the definition: for every $l = 1, \dots, n-1$ we have

$$\begin{aligned} p_1^n &= 0, & q_n^n &= w_n^n e^{\xi_n^n} f(\xi_n^n), \\ p_{l+1}^n &= p_l^n + w_l^n e^{-\xi_l^n} f(\xi_l^n), & q_l^n &= q_{l+1}^n + w_l^n e^{\xi_l^n} f(\xi_l^n). \end{aligned}$$

C++-code 8.8.2: Implementation of `comp_g_gausspts`.

```
template<typename Function>
Eigen::VectorXd comp_g_gausspts(Function f, unsigned int n) {

    Eigen::VectorXd g(n);
    Eigen::VectorXd w(n), xi(n);
    gaussrule(n, w, xi);  // Compute Gauss nodes and weights relative to [-1,1]

    Eigen::VectorXd p(n), q(n), expxi(n), fxi(n);
    w = w/2.;  // Rescale the weights to [0,1]
    xi = (xi + Eigen::MatrixXd::Ones(n,1))/2.;  // Rescale the nodes to [0,1]

    for (int l=0; l<n; l++) {
        fxi(l) = f(xi(l));
        expxi(l) = exp(xi(l));
    }

    // Initialize the Eigen::VectorXds p and q
    p(0) = 0;
    q(n-1) = w(n-1) * expxi(n-1) * fxi(n-1);

    // Fill in the Eigen::VectorXds p and q (O(n) complexity)
    for (int l=0; l<n-1; l++) {
        p(l+1) = p(l) + w(l)/expxi(l)*fxi(l);
        q(n-2-l) = q(n-1-l) + w(n-2-l)*expxi(n-2-l)*fxi(n-2-l);
    }

    // Finally, constructing the output (O(n) complexity)
    g = (p.array() * expxi.array() + q.array() / expxi.array()).matrix();
    return g;
}
```

Get it on GitLab (gauss_pts.cpp).

▲

(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 of (8-8.c):

For reasons of symmetry, we have $\xi_{11}^{21} = 1/2$, and so $g_{21}(\xi_{11}^{21})$ should be equal to 1 for $f(y) = e^{-|0.5-y|}$, since Gauss quadrature is exact for constant functions.

C++-code 8.8.3: Testing code.

```
int n = 21;

auto f = [] (double y) {return std::exp(-std::abs(.5-y)); };

Eigen::VectorXd g = comp_g_gausspts(f,n);
std::cout << "g(xi^" << n << "_" << (n+1)/2 << ") = " << g((n+1)/2-1) <<
    std::endl;
```

Get it on GitLab (gauss_pts.cpp).

▲

End Problem 8-8

Problem 8-9: Efficient quadrature of singular integrands

We know that smoothness of the integrand is essential for fast covergence of high-order numerical quadrature like Gauss quadrature rules. In some cases smart transformation can convert the integral into another with a smooth integrand. This problem uses this trick for the sake of efficient numerical quadrature of non-smooth integrands with a special structure.

Before you tackle this problem, read about regularization of integrands by transformation, cf. [Lecture → Rem. 7.3.0.47]. 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:

$$W(f) := \int_{-1}^{1} \sqrt{1-t^2}\, f(t)dt, \tag{8.9.1}$$

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 of (8-9.b):

If $f \in C^r([a,b])$ then we get algebraic convergence with rate r.

If f has an analytic extension to a complex neighbourhood of $[a,b]$ we get exponential convergence. Almost all integrands $\in C^\infty([a,b])$ will allow such an extension.

▲

(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 $n \to \infty$, the error of your implementation has asymptotic exponential convergence to zero.

HINT 1 for (8-9.c): A C++ lambda function provides such operator, e.g. `auto f = [](double t){ return t*t; };` for $f(t) = t^2$. ⌟

HINT 2 for (8-9.c): You may use $\sqrt{1-t^2} = \sqrt{1-t}\sqrt{1+t}$. ⌟

SOLUTION of (8-9.c):

Exploiting the hint, we see that the integrand is non-smooth in ± 1.

There are two possible solutions to the problem.

Method 1: we partition the integration domain $[-1,1] = [0,1] \cap [-1,0]$. Applying the substitution $s = \sqrt{1 \pm t}$ (sign depending on which part of the integrals considered), $t = \pm(s^2 - 1)$:

$$\begin{aligned} \frac{dt}{ds} &= \pm 2s \\ W(f) &:= \int_{-1}^{1} \sqrt{1-t^2} f(t) dt = \int_{-1}^{0} \sqrt{1-t^2} f(t) dt + \int_{0}^{1} \sqrt{1-t^2} f(t) dt \\ &= \int_{0}^{1} 2 \cdot s^2 \sqrt{2-s^2} f(-s^2+1) ds + \int_{0}^{1} 2 \cdot s^2 \sqrt{2-s^2} f(s^2-1) ds. \end{aligned}$$

Notice how the resulting integrand is analytic in a neighbourhood of the domain of integration because, for instance, $t \mapsto \sqrt{1+t}$ is analytic in a neighborhood of $[0,1]$.

Method 2: one may use the trigonometric substitution $t = \sin s$, obtaining

$$\begin{aligned} \frac{dt}{ds} &= \cos s \\ W(f) &:= \int_{-1}^{1} \sqrt{1-t^2} f(t) dt \\ &= \int_{-\pi/2}^{\pi/2} \sqrt{1-\sin^2 s}\, f(\sin s) \cos s \, ds = \int_{-\pi/2}^{\pi/2} \cos^2 s \, f(\sin s) ds. \end{aligned}$$

This integrand is also analytic.

C++-code 8.9.2: Implementation of `quadsingint`.

```
template <class Function>
double quadsingint(const Function& f, const unsigned n) {
    double I = 0.;
    // Method selects one of the two possible solutions
#if METHOD == 1 // s = √(1 ± t)
    // Will use same node twice
    QuadRule Q = gauleg(n);

    for(unsigned i = 0; i < n; ++i) {
        // Transform nodes
        double x = (Q.nodes(i) + 1.) / 2.;
        // Weights
        double w = Q.weights(i) * x * x * std::sqrt(2. - x * x);
        // Symmetric summation
        I += w * (f(x*x - 1) + f(-x*x + 1));
    }
#elif METHOD == 2 // t = cos(s)
    // We are actually using twice the number of nodes than Method 1
    QuadRule Q = gauleg(2*n);

    for(unsigned i = 0; i < 2*n; ++i) {
        // Evualuate transformation
        double x = sin(Q.nodes(i) * M_PI_2);
        // Weights
        double w = Q.weights(i) * cos(Q.nodes(i) * M_PI_2) * cos(Q.nodes(i) *
```

```
                M_PI_2);
            I += w * f(x) * M_PI_2;
        }
#endif
    return I;
}
```

Get it on GitLab (quadsingint.cpp).

▲

(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 of (8-9.d):

Using Method 1:

Let (x_j, w_j), $j = 1,\dots,n$ be the Gauss nodes and weights relative to the Gauss quadrature of order n in the interval $[0,1]$. The nodes are mapped from x_j in $[0,1]$ to c_l for $l \in 1,\dots,2n$ in $[-1,1]$ as follows:

$$c_{2j-i} = (-1)^i(1-x_j^2), \qquad j = 1,\dots,n,\ i = 0,1.$$

The weights $\tilde{w}_l, l = 1,\dots,2n$, become:

$$\tilde{w}_{2j-i} = 2w_j x_j^2\sqrt{2-x_j^2}, \qquad j = 1,\dots,n,\ i = 0,1.$$

Using Method 2:

Let (x_j, w_j), $j = 1,\dots,n$ be the Gauss nodes and weights relative to the Gauss quadrature of order n in the interval $[-1,1]$. The nodes are mapped from x_j to c_j as follows:

$$c_j = \sin(x_j\pi/2), \qquad j = 1,\dots,n$$

The weights $\tilde{w}_j, j = 1,\dots,n$, become:

$$\tilde{w}_j = w_j\cos^2(x_j\pi/2)\pi/2.$$

▲

(8-9.e) 2 Implement a C++ function

```
void tabQuadErr(void)
```

that tabulates the quadrature error:

$$\epsilon_i := |W(f) - \texttt{quadsingint(f,n)}|$$

for $f(t) := \dfrac{1}{2+\exp(3t)}$ and $n = 1,2,...,25$. Then describe the type of convergence observed, see [Lecture → Def. 6.1.2.7].

SOLUTION of (8-9.e):

The convergence is exponential with both methods.

N	quadsingint(f,n)	ϵ_i
1	0.37174	0.111556
2	0.490984	0.00768718
3	0.484771	0.0014743
4	0.483206	9.06151e-05
5	0.483276	2.11519e-05
6	0.483298	1.6518e-06
7	0.483297	2.42568e-07
8	0.483297	2.41621e-08
9	0.483297	1.89626e-09
10	0.483297	3.19954e-10
11	0.483297	4.24399e-12
12	0.483297	3.67528e-12
13	0.483297	2.03004e-13
14	0.483297	3.41949e-14
15	0.483297	4.88498e-15
16	0.483297	1.16573e-15
17	0.483297	3.88578e-16
18	0.483297	3.88578e-16
19	0.483297	6.66134e-16
20	0.483297	4.996e-16
21	0.483297	3.88578e-16
22	0.483297	4.996e-16
23	0.483297	5.55112e-17
24	0.483297	1.11022e-16

C++-code 8.9.3: Implementation of convergence test.

```
int main(int argc, char ** argv) {

    // Max num of Gauss points to use (in each direction)
    const unsigned max_N = 25;
    // "Exact" integral
    double I_ex = 0.483296828976607;

    // If you want to test monomials as f, use this data and set
        MONOMIAL_TEST to true
#if MONOMIAL_TEST
    std::vector<double> ex = { M_PI_2, 0., M_PI_2 / 4, 0, M_PI_2 / 8, 0, M_PI_2 *
        5 / 64};
    assert(argc > 1);
    int n = std::atoi(argv[1]);
    std::cout << "TEST: Monomial of degree:" << n << std::endl;
    auto f = [&n] (double t) { return std::pow(t, n); };
    I_ex = ex[n];
#else // END MONOMIAL_TEST
    // Test function
    auto f = [] (double t) { return 1. / (2. + std::exp(3*t) ); };
#endif

    std::cout << std::setw(3) << "N"
```

```
              << std::setw(15) << "I_approx"
              << std::setw(15) << "error"
              << std::endl;
    double errold, errnew;
    // Observed:  exponential convergence (as expected)
    for(unsigned int N = 1; N < max_N; ++N) {
        double I_approx = quadsingint(f, N);

        errold = errnew;
        errnew = std::abs(I_ex - I_approx);

        // Interpolating points
        std::cout << std::setw(3) << N
                     // Value of integral
                  << std::setw(15) << I_approx
                     // Error
                  << std::setw(15) << errnew
                  << std::endl;
    }
    return 0;
}
```

Get it on GitLab (quadsingint.cpp).

▲

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 ξ_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:

$$-1 < \xi_j^n < \xi_j^{n-1} < \xi_{j+1}^n < 1, \qquad j = 1, \dots, n-1 \,.$$

HINT 1 for (8-10.a): You may follow these steps:

1. Understand that it is enough to show that every pair of zeros $\left(\xi_l^n, \xi_{l+1}^n\right)$ of P_n is separated by a zero of P_{n-1}.
2. Argue by contradiction.
3. By considering the auxiliary polynomial $\prod_{j \neq l, l+1}\left(t - \xi_j^n\right)$ and the fact that the Gauss quadrature is exact on $\mathcal{P}_{2n-1}$, prove that $P_{n-1}(\xi_l^n) = P_{n-1}(\xi_{l+1}^n) = 0$.
4. Choose $s \in \mathcal{P}_{n-2}$ such that $s(\xi_j^n) = P_{n-1}(\xi_j^n)$ for every $j \neq l, l+1$, and obtain a contradiction by using again the fact that the Gauss quadrature is exact on $\mathcal{P}_{2n-1}$.

⌟

SOLUTION of (8-10.a):

By [Lecture → Lemma 7.3.0.29], P_n has exactly n distinct zeros in $]-1,1[$. Therefore, it is enough to prove that every pair of zeros of P_n is separated by one zero of P_{n-1}. By contradiction, assume that there exists $l = 1, \dots, n-1$ such that $]\xi_l^n, \xi_{l+1}^n[$ does not contain any zeros of P_{n-1}. As a consequence, $P_{n-1}(\xi_l^n)$ and $P_{n-1}(\xi_{l+1}^n)$ have the same sign, namely:

$$P_{n-1}(\xi_l^n) P_{n-1}(\xi_{l+1}^n) \geq 0 \,. \tag{8.10.1}$$

Recall that, by construction, we have the following orthogonality property:

$$\int_{-1}^{1} P_{n-1} q \, dt = 0, \qquad q \in \mathcal{P}_{n-2} \,. \tag{8.10.2}$$

Consider the auxiliary polynomial:

$$q(t) = \prod_{j \neq l, l+1} (t - \xi_j^n) \,.$$

By construction, $q \in \mathcal{P}_{n-2}$, whence $P_{n-1}q \in \mathcal{P}_{2n-3}$. Thus, using (8.10.2) and the fact that Gaussian quadrature is exact on P_{2n-1} we obtain:

$$0 = \int_{-1}^{1} P_{n-1}q\,dt \quad = \sum_{j=1}^{n} w_j P_{n-1}(\xi_j^n)q(\xi_j^n) = w_l P_{n-1}(\xi_l^n)q(\xi_l^n) + w_{l+1}P_{n-1}(\xi_{l+1}^n)q(\xi_{l+1}^n)\,.$$

Hence, both terms of the right hand side must have different sign, namely:

$$\left[w_l P_{n-1}(\xi_l^n)q(\xi_l^n)\right]\left[w_{l+1}P_{n-1}(\xi_{l+1}^n)q(\xi_{l+1}^n)\right] \leq 0\,.$$

On the other hand, by [Lecture → Lemma 7.3.0.31] we have $w_l, w_{l+1} > 0$. Moreover, by construction of q we have that $q(\xi_l^n)q(\xi_{l+1}^n) > 0$. Combining these properties with (8.10.1), we obtain that:

$$P_{n-1}(\xi_l^n) = P_{n-1}(\xi_{l+1}^n) = 0\,.$$

Choose now $s \in \mathcal{P}_{n-2}$ such that $s(\xi_j^n) = P_{n-1}(\xi_j^n)$ for every $j \neq l, l+1$. Using again (8.10.2) and the fact that Gaussian quadrature is exact on P_{2n-3} we obtain:

$$0 = \int_{-1}^{1} P_{n-1}s\,dt = \sum_{j=1}^{n} w_j P_{n-1}(\xi_j^n)s(\xi_j^n) = \sum_{j\neq l,l+1} w_j (P_{n-1}(\xi_j^n))^2\,.$$

Since P_{n-1} has only $n-1$ zeros, two zeros of P_{n-1} have already been "found" and the weights w_j are all positive, the right hand side of this equality is strictly positive, thereby contradicting the equality itself.

Alternative solution. There is a shorter proof of this fact based on the recursion formula [Lecture → Eq. (7.3.0.34)]. We sketch the main steps.

We will prove the statement by induction. If $n = 1$ there is nothing to prove. Suppose now that the statement is true for n. By [Lecture → Eq. (7.3.0.34)] we have $P_{n+1}(\xi_j^n) = -\frac{n}{n+1}P_{n-1}(\xi_j^n)$ for every $j = 1,\dots,n$. Further, since the statement is true for n we have $(-1)^{n-j}P_{n-1}(\xi_j^n) > 0$ (given the interleaving property valid for n, each pair $P_{n-1}(\xi_l^n)$, $P_{n-1}(\xi_{l+1}^n)$ must have opposite signs).

Therefore:

$$(-1)^{n+1-j}P_{n+1}(\xi_j^n) > 0, \qquad j = 1,\dots,n\,. \tag{8.10.3}$$

Since the leading coefficient of P_{n+1} is positive, we have $P_{n+1}(x) > 0$ for all $x > \xi_{n+1}^{n+1}$ and $(-1)^{n+1}P_{n+1}(x) > 0$ for all $x < \xi_1^{n+1}$. Combining these two inequalities with (8.10.3) yields the result for $n+1$.

▲

(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 of (8-10.b):

Differentiating [Lecture → Eq. (7.3.0.34)] immediately gives:

$$P_{n+1}'(t) = \frac{2n+1}{n+1}P_n(t) + \frac{2n+1}{n+1}tP_n'(t) - \frac{n}{n+1}P_{n-1}'(t), \quad P_0' = 0, \quad P_1' = 1$$

This equation, combined with [Lecture → Eq. (7.3.0.34)], gives the desired recursions.

▲

(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 of (8-10.c):

C++ code 8.10.4: Suggested solution for `legvals()`

```
void legvals(const VectorXd& x, MatrixXd& Lx, MatrixXd& DLx) {
  const int n = Lx.cols()-1;
  const int N = x.size();
  for (int j = 0; j < N; ++j) {
    Lx(j,0) = 1.;
    Lx(j,1) = x(j);
    DLx(j,0) = 0;
    DLx(j,1) = 1.;
    for (int k = 2; k < n+1; ++k) {
      Lx(j,k) = (2*k-1.)/k*x(j)*Lx(j,k-1) - (k-1.)/k*Lx(j,k-2);
      DLx(j,k) = (2*k-1.)/k*Lx(j,k-1) + (2*k-1.)/k*x(j)*DLx(j,k-1) -
                    (k-1.)/k*DLx(j,k-2);
    }
  }
}
```

Get it on GitLab (legendre.cpp).

▲

(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.

HINT 1 for (8-10.d): For simplicity, you may want to write a C++ function **double** `Pkx(double x, int k)` that computes $P_k(x)$ for a scalar x. Reuse parts of the function `legvals()`. ⌟

SOLUTION of (8-10.d):

The proposed implementation uses the function `Pkx`, which you can find below in Code 8.10.6.

C++-code 8.10.5: Suggested solution for `gaussPts`

```
MatrixXd gaussPts(const int n, const double rtol=1e-10,
                                  const double atol=1e-12) {
  MatrixXd zeros(n,n);
  double x0, x1, f0, f1, s;
  for (int k = 1; k < n+1; ++k) {
    for (int j = 1; j < k+1; ++j) {
      // Initial guesses
      if (j == 1) x0 = -1.;
      else        x0 = zeros(j-2, k-2);
      if (j == k) x1 = 1.;
      else        x1 = zeros(j-1, k-2);

      // Secant method
      f0 = Pnx(x0, k);
      for (int i = 0; i < 1e4; ++i) {
        f1 = Pnx(x1, k);
        s = f1*(x1-x0)/(f1-f0);
        x0 = x1; f0 = f1;
        x1 -= s;
        if ( (std::abs(s) < std::max(atol, rtol*std::min(std::abs(x0),
              std::abs(x1)))) ) {
          zeros(j-1, k-1) = x1;
          break;
        }
      }
    }
  }
  return zeros;
}
```

Get it on GitLab (legendre.cpp).

C++-code 8.10.6: Implementation of `Pkx`

```
double Pnx(const double x, const int n) {
  VectorXd Px(n+1);
  Px(0) = 1.; Px(1) = x;
  for (int k=2; k<n+1; k++){
    Px(k) = (2*k-1.)/k*x*Px(k-1)-(k-1.)/k*Px(k-2);
  }
  return Px(n);
}
```

Get it on GitLab (legendre.cpp).

▲

(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.

HINT 1 for (8-10.e): Fig. 38 shows P_7 and P_8 on a part of $[-1,1]$. The secant method fails to find the zeros of P_8 (blue curve) when started with the zeros of P_7 (red curve).

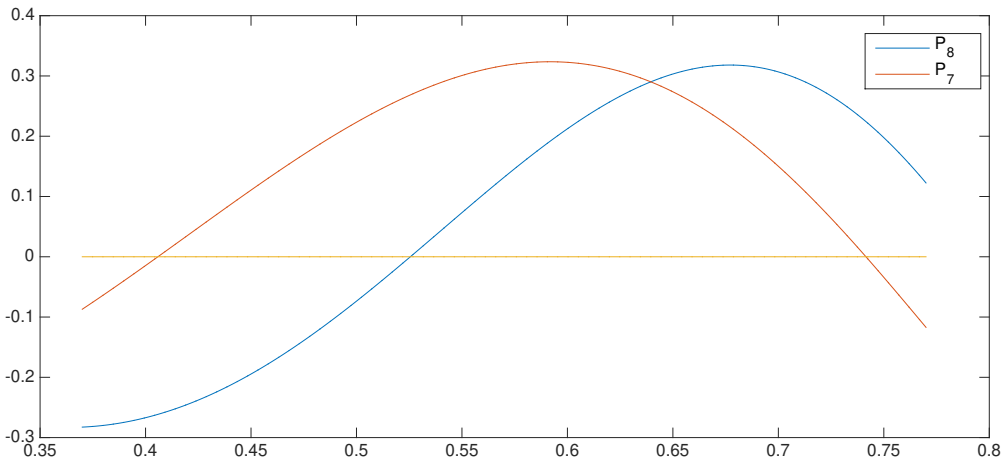


Fig. 38

SOLUTION of (8-10.e):

See file `legendre.cpp` for the implementation. The zeros of P_k, $k = 1, \dots, 7$ are all correctly computed, but the zero $\xi_6^8 \approx 0.525532$ of P_8 is not correct. This is due to the fact that in this case the initial guesses for the secant method are not close enough to the zero (remember, only local convergence is guaranteed). Indeed, taking $x^{(0)}$ and $x^{(1)}$ as the two consecutive zeros of P_7 in Fig. 38, the third iterate $x^{(2)}$ will satisfy $P_8(x^{(1)})P_8(x^{(2)}) > 0$ and the forth iterate will be very far from the desired zero.

▲

(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 $\operatorname{sign}\left[f\left(x^{(k+1)}\right)\right] \neq \operatorname{sign}[f(z)]$. This ensures that f has always a different sign in the last two iterates.

The regula falsi variation of the secant method can be easily implemented with a little modification of the code.

SOLUTION of (8-10.f):

You can find the complete implementation in the file `legendre.cpp`.

C++-code 8.10.7: Suggested solution for `gausZerosLegendrets_regulaFalsi` (corrected version)

```
MatrixXd gaussPts_regulaFalsi(const int n, const double rtol=1e-10,
                                                const double atol=1e-12) {
  MatrixXd zeros(n,n);
  double x0, x1, f0, f1, s;
  for (int k = 1; k < n+1; ++k) {
    for (int j = 1; j < k+1; ++j) {
      // Initial guesses
      if (j == 1) x0 = -1;
      else        x0 = zeros(j-2, k-2);
      if (j == k) x1 = +1;
```

```
      else      x1 = zeros(j-1, k-2);

      // Secant method
      f0 = Pnx(x0, k);
      for (int i = 0; i < 1e4; ++i) {
        f1 = Pnx(x1, k);
        s = f1*(x1-x0)/(f1-f0);
        if (Pnx(x1-s, k)*f1 < 0) // NEW LINE: regula falsi
          x0 = x1; f0 = f1;
        x1 -= s;
        if ( (std::abs(s) < std::max(atol, rtol*std::min(std::abs(x0),
                std::abs(x1)))) ) {
          zeros(j-1, k-1) = x1;
          break;
        }
      }
    }
  }
  return zeros;
}
```

Get it on GitLab (legendre.cpp).

▲

End Problem 8-10

Problem 8-11: Quadrature rules and quadrature errors

When sequences of quadrature rules are applied to evaluate $\int_a^b f(t)\,\mathrm{d}t$, then the asymptotic convergence of quadrature error in terms of $n \to \infty$, $n \hat{=}$ the number of quadrature nodes, is determined both by

(i) the smoothness of the integrand f,

(ii) family of quadrature rules used.

This problems asks you to guess the family of quadrature rules from error curves.

To prepare for this problem study [Lecture → Section 7.3], 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) $\hat{=}$ Equidistant trapezoidal rule, see [Lecture → Eq. (7.4.0.17)],

(SR) $\hat{=}$ Equidistant Simpson rule, see [Lecture → Eq. (7.4.0.5)],

(GR) $\hat{=}$ 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 $\int_a^b f(t)\,\mathrm{d}t$. The integrand is passed through the functor `f`.

```
template <typename Functor>
double equidTrapezoidalRule(Functor &&f, double a, double b, unsigned int n) {
  assert(n>=2);
  const double h = (b - a)/(n-1);
  double t = a + h, s = 0.0;
  for (unsigned int i = 1; i < n-1; t += h, ++i) { s += f(t); }
  return ([          ] * f(a) + [          ] * f(b) + [          ] * s);
}
```

Supplement the missing C++ code in the boxes.

SOLUTION of (8-11.a):

We have to implement the formula from [Lecture → Eq. (7.4.0.17)]:

$$\int_a^b f(t)\,\mathrm{d}t \approx T_n(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{n-2} f(kh) + \tfrac{1}{2}f(b)\right), \quad h := \frac{b-a}{n-1},$$

where the sum is empty for $n = 2$.

- left box: `0.5 * h`,
- middle box: `0.5 * h`,
- right box: `h`.

▲

(8-11.b) ⚁ (3 min.) The following plots display the quadrature errors for the approximation of the integral

$$\int_0^1 f_1(t)\,\mathrm{d}t \quad , \quad f_1(t) := |\sin(5t)| \,,$$

by the three families of quadrature rules and numbers $n = 4, \dots, 100$ of quadrature points.

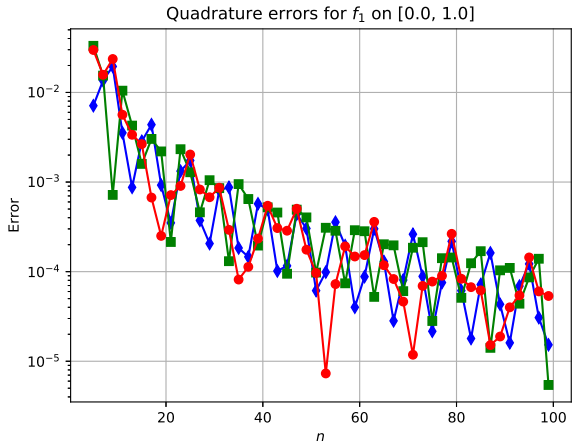


Fig. 39

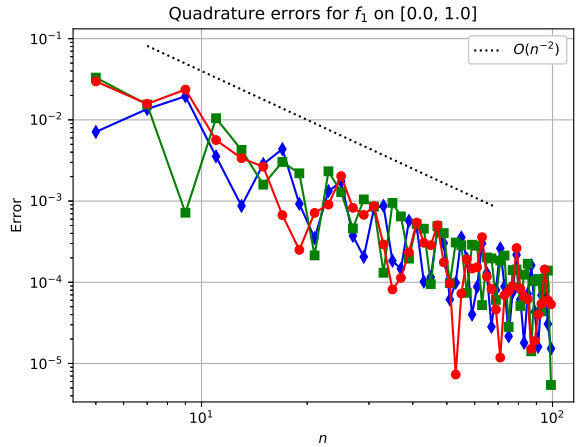


Fig. 40

Tick the families of quadrature rules, if the curve correctly represents the expected behavior of its quadrature error as a function of n.

◆ :	○ TR	○ SR	○ GR
■ :	○ TR	○ SR	○ GR
● :	○ TR	○ SR	○ GR

SOLUTION of (8-11.b):

The integrand f_1 is merely continuous and only piecewise smooth, because its graph has kinks at the sites $\in \mathbb{Z}\frac{\pi}{5}$.

This non-smoothness severely limits the speed of convergence of all families of quadrature rules and for all of them we merely observe algebraic convergence with rate ≈ 2 for $n \to \infty$. Every curve can in principle belong to any family of quadrature rules.

◆ :	✔ TR	✔ SR	✔ GR
■ :	✔ TR	✔ SR	✔ GR
● :	✔ TR	✔ SR	✔ GR

The actual association of rules and error curves as as follows:

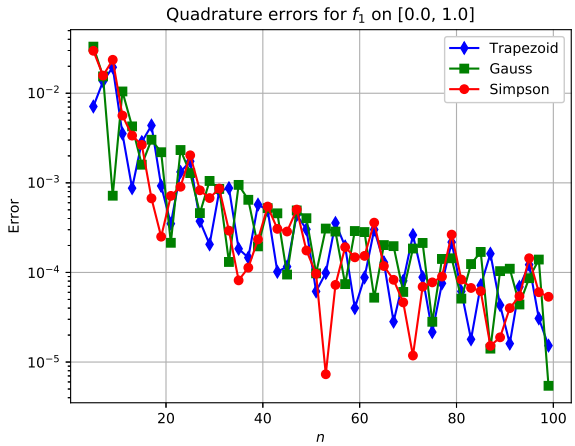


Fig. 41

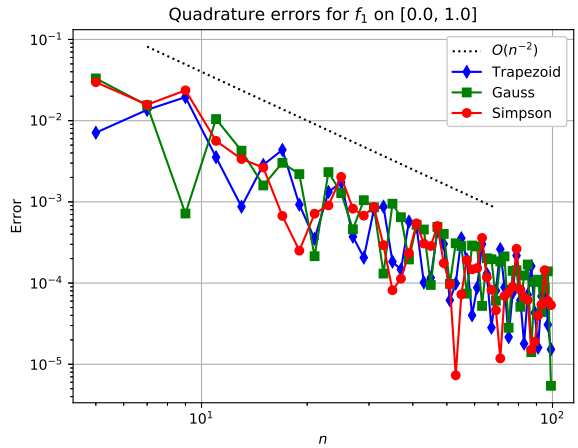


Fig. 42

▲

(8-11.c) (3 min.) Below we have plotted the quadrature errors for the approximation of the integral

$$\int_0^1 f_2(t)\,\mathrm{d}t \quad , \quad f_2(t) := |\sin(5t)|^2 \,,$$

for the three families of quadrature rules and numbers $n = 4, \ldots, 100$ of quadrature points.

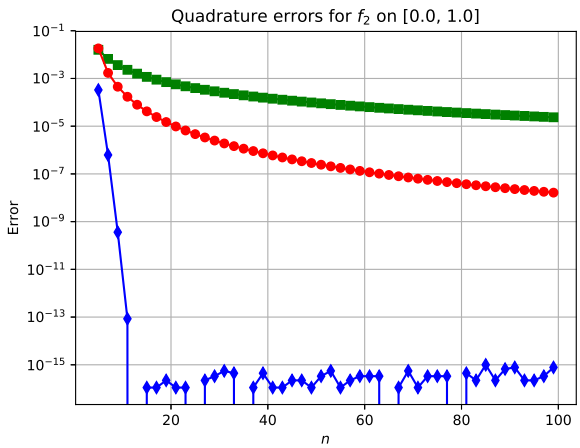


Fig. 43

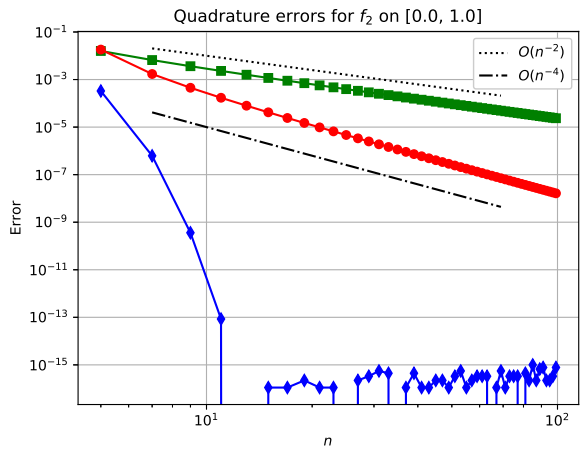


Fig. 44

Select the quadrature rule, if the curve correctly represents the expected behavior of its quadrature error as a function of n.

◆ :	○ TR	○ SR	○ GR
■ :	○ TR	○ SR	○ GR
● :	○ TR	○ SR	○ GR

SOLUTION of (8-11.c):

The integrand belongs to $C^\infty(\mathbb{R})$, it is even analytic.

Therefore the the composite quadrature rules can achieve their maximal rate of (algebraic) convergence, 2 for the equidistant trapezoidal rules, and 4 for the equidistant Simpson rules, see [Lecture → Exp. 7.4.0.12].

Gauss quadrature will display exponential convergence as in [Lecture → Ex. 7.3.0.46], evident from aligned error points in a lin-log scale.

◆:	○ TR	○ SR	✔ GR
■:	✔ TR	○ SR	○ GR
●:	○ TR	✔ SR	○ GR

The actual association of rules and error curves as as follows:

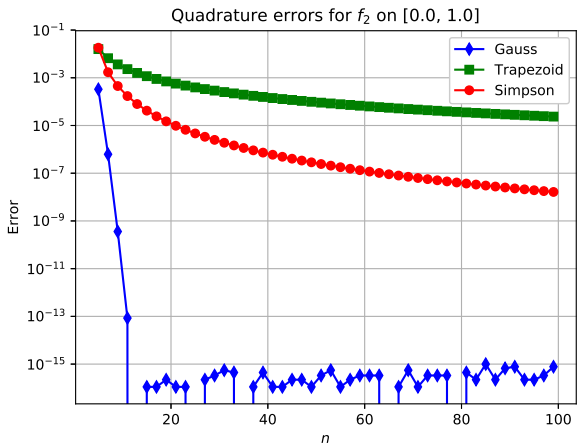


Fig. 45

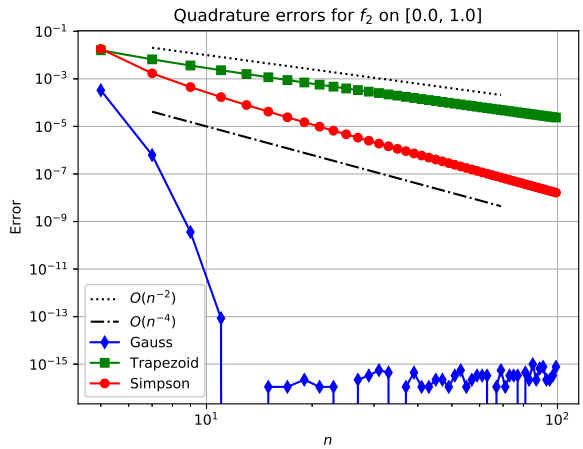


Fig. 46

▲

(8-11.d) ⚁ (3 min.) The plots below display the quadrature errors for

$$\int_0^1 f_3(t)\,\mathrm{d}t\,, \quad f_3(t) := |\sin(5t)|^5\,,$$

and for equidistant trapezoidal rules (TR), equidistant Simpson rules (SR), and Gauss-Legendre rules (GR) as a function of the number n of quadrature points, $n = 4, \dots, 100$.

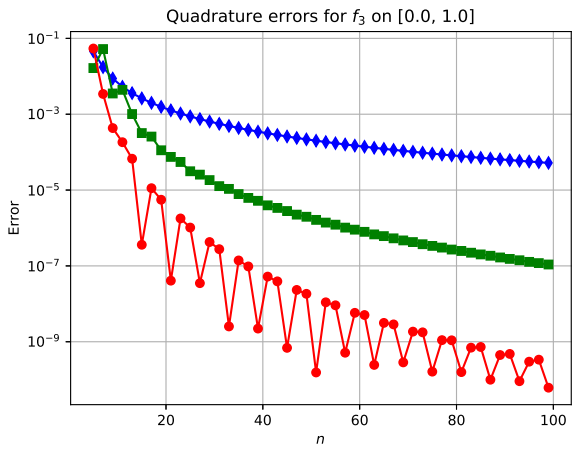


Fig. 47

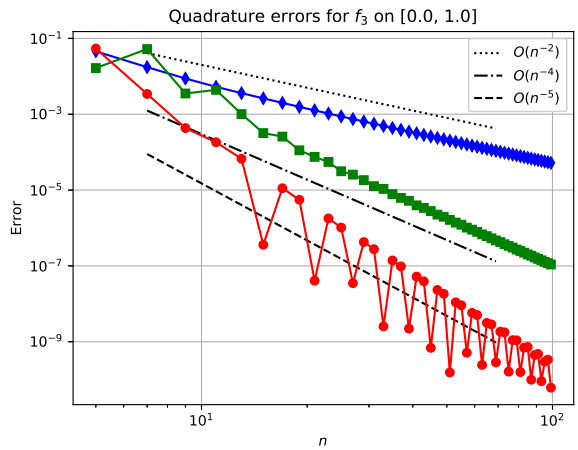


Fig. 48

Select those quadrature rules, for which the curve correctly represents the expected behavior of its quadrature error as a function of n.

◆	○ TR	○ SR	○ GR
■	○ TR	○ SR	○ GR
●	○ TR	○ SR	○ GR

SOLUTION of (8-11.d):

The integrand is not C^∞, but only $\in C^4$ and piecewise smooth.

This is still smooth enough to admit the maximal rate of algebraic convergence for the composite quadrature rules (2 for the equidistant trapezoidal rules and 4 for the equidistant Simpson rules), but it will no longer allow the Gauss-Legendre rules to converge exponentially. They can achieve only algebraic convergence, which reaches the rate 5 in this experiment which cannot be achieved by any of the composite rules.

◆:	✔ TR	○ SR	○ GR
■:	○ TR	✔ SR	○ GR
●:	○ TR	○ SR	✔ GR

The true association of rules and error curves as as follows:

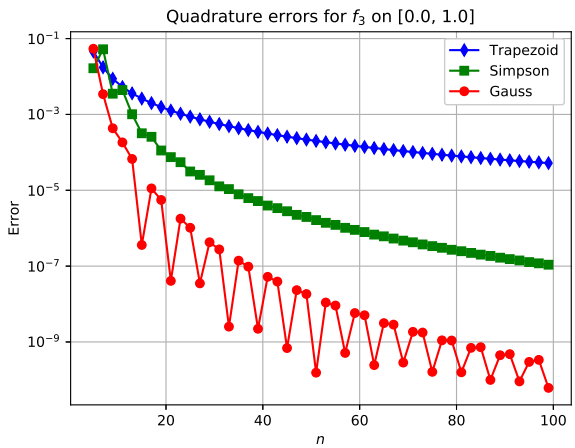


Fig. 49

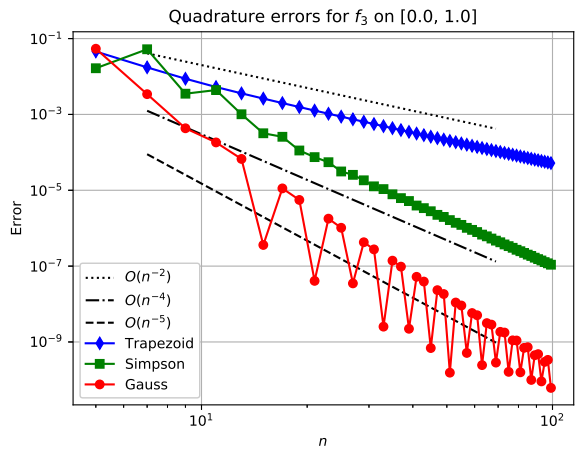


Fig. 50

▲

(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 $\int_0^1 |sin(\pi t)|\,\mathrm{d}t$.

○ true ○ false

SOLUTION of (8-11.e):

(i) **true**, see [Lecture → Thm. 7.3.0.6].

(ii) **false**, because there are n-point quadrature rules of order n with *negative weights*, e.g., the Newton-Cotes rules for larger n, *cf.* [Lecture → Section 7.2].

(iii) **true**, because the proof of [Lecture → Lemma 7.3.0.31] still applies to n-point quadrature rules of order $2n - 1$.

(iv) **false**, because the periodic extension of the integrand is not smooth, since it has kinks in the points $\mathbb{Z}$.

▲

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. 146]: 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 of (9-1.):

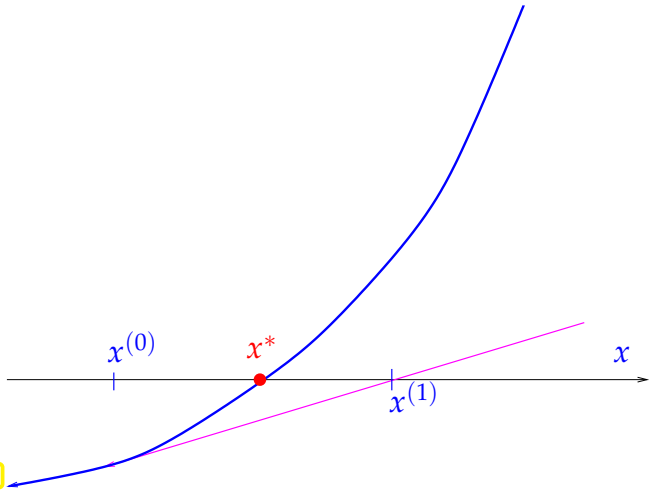


Fig. 51

If $x^{(0)} < x^*$, where $F(x^*) = 0$, then $x^{(1)} > x^*$ and $F(x^{(1)}) > 0$.
Therefore, we may assume $x^{(1)} > x^*$ from now.

Assume $x^{(k)} > x^*$, which means $F(x^{(k)}) > 0$ thanks to the monotonicity of F.

- Since f is *convex* the tangent at the graph of f in $x^{(k)}$ is below the graph. This implies that its intersection with the x-axis lies to the right of x^*. We conclude $x^{(k+1)} > x^*$
- Since F is monotone, F' is positive. Hence

$$x^{(k+1)} := x^{(k)} - \frac{F(x^{(k)})}{F'(x^{(k)})} < x^{(k)} .$$

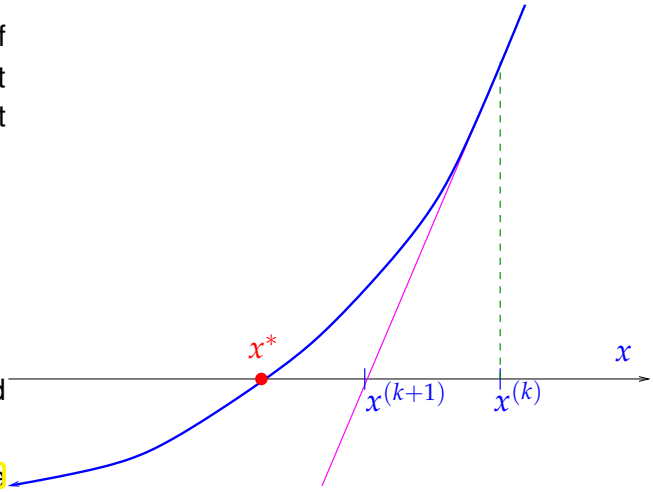


Fig. 52

Therefore, the sequence $\left(x^{(k)}\right)_k$ is decreasing and bounded from below by x^*.
By the completeness of $\mathbb{R}$ such a sequence will have a limit $x^+ := \lim_{k\to\infty} x^{(k)}$.

By the continuity of both F and F', we can simply plug the limit x^+ into the Newton iteration:

$$x^+ = x^+ - \frac{F(x^+)}{F'(x^+)} \quad \blacktriangleright \quad F(x^+) = 0 .$$

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 }
```

▲

HINT 1 for (9-2.a): First try to understand what is accomplished in Line 5 and Line 6 of this code. Recall the properties of logarithms. ⌟

HINT 2 for (9-2.a): Line 9–11 realize the Newton iteration for finding a zero of a real valued function. ⌟

SOLUTION of (9-2.a):

The C++ code computes $y = \log(x)$, for a given x. The program can be regarded as Newton iterations for finding the zero of

$$f(z) = e^z - x\,, \tag{9.2.2}$$

where z is unknown and x is given.

C++11-code 9.2.3: Undocumented function.

```
double myfunction(double x) {
    // We require that x > 0.
    // First we factorize x as x = 2^{n+r} = 2^n * x_new
    // with integer n and r ∈ [-0.5,0.5].
    // Set y = n log(2) and z = log(x_new),
    // then log(x) = y + z.
    double log2=0.693147180559945;
    double y=0;
    // If x x = 2^{n+r} with positive n:
    while(x>std::sqrt(2)){x/=2; y+=log2;} //
    // If x x = 2^{n+r} with negative n:
    while(x<1./std::sqrt(2)){x*=2; y-=log2;} //
    // Now, we have found y, and we use Newton iteration for the
    // function f(z) = exp(z) - x_new to find z = log(x_new).
```

```
    // Initial guess:
    double z=x-1; //
    // The update is dz = -f(z)/f'(z)
    double dz=x*std::exp(-z)-1.0;
    while(std::abs(dz/z)>std::numeric_limits<double>::epsilon()) {
        z+=dz; dz=x*std::exp(-z)-1.0;
    }
    return y+z+dz; //
}
```

(9-2.b) (10 min.) Explain the rationale behind the first two `while` loops in the code Code 9.9.8, Lines 5 & 6, preceding the main iteration.

SOLUTION of (9-2.b):

The algorithm relies on the identities

$$\log(x) = \log(2^{x/2}) = \log 2 + \log(x/2) \quad , \quad \log(x) = \log(\tfrac{1}{2} \cdot 2x) = -\log 2 + \log(2x) \; .$$

to reduce the problem to the computation of $\log(x)$ with $x \in [2^{-1/2}, 2^{1/2}]$.

The rationale for this can be seen by looking at the graph of f as defined in (9.2.2). An initial guess $z^{(0)} := x - 1$ for x in that interval will already be close to the zero of f, so that we can expect local quadratic convergence of Newton's method.

▲

(9-2.c) (10 min.) Explain what the last loop body does.

SOLUTION of (9-2.c):

The `while`-loop computes the zero of the function $z \mapsto f(z)$ from (9.2.2) using Newton iterations:

$$\begin{cases} z^{(0)} = x - 1, \\ z^{(n+1)} = z^{(n)} - \frac{e^{z^{(n)}} - x}{e^{z^{(n)}}}, \quad n \geq 1 \end{cases} .$$

▲

(9-2.d) (5 min.) Explain the conditional expression in the last `while` loop.

SOLUTION of (9-2.d):

This is a correction based termination criterium, see [Lecture → § 8.1.2.4].

▲

(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!

HINT 1 for (9-2.e): Using Taylor expansions, establish a recursion for the iteration error. In that recursion, drop all powers of the iteration error with an exponent > 2, because you may assume that the iteration error is always small for a good initial guess. ⌟

SOLUTION of (9-2.e):

Denote the zero of $f(z)$ with z^*, and $e^{(n)} = z^{(n)} - z^*$. Use Taylor expansion of $f(z)$, $f'(z)$:

$$f(z^{(n)}) = e^{z^*}e^{(n)} + \frac{1}{2}e^{z^*}(e^{(n)})^2 + O((e^{(n)})^3) = xe^{(n)} + \frac{1}{2}x(e^{(n)})^2 + O((e^{(n)})^3),$$
$$f'(z^{(n)}) = e^{z^*} + e^{z^*}e^{(n)} + O((e^{(n)})^2) = x + xe^{(n)} + O((e^{(n)})^2)\ .$$

Appealing to the Newton iteration $z^{(n+1)} = z^{(n)} - \frac{f(z^{(n)})}{f'(z^{(n)})}$, we find the approximate error recursion ($\doteq$ means that we ignore higher powers of the iteration error, which is justified if the initial guess is already close to the solution.)

$$e^{(n+1)} = e^{(n)} - \frac{f(z^{(n)}}{f'(z^{(n)})} = e^{(n)} - \frac{xe^{(n)} + \frac{1}{2}x(e^{(n)})^2 + O((e^{(n)})^3)}{x + xe^{(n)} + O((e^{(n)})^2)}$$
$$\doteq \frac{1}{2}(e^{(n)})^2 = \cdots \doteq (\frac{1}{2})^{1+2+\cdots+2^n}(e^0)^{2^{n+1}} = 2\cdot(\frac{1}{2})^{2^{n+1}}(e^0)^{2^{n+1}} = 2\cdot(\frac{1}{2}e^0)^{2^{n+1}}\ ,$$

where $e^(0) = z^{(0)} - z^* = x - 1 - \log(x)$. So it is enough for us to determine the number n of iteration steps by $|\frac{e^{(n+1)}}{\log x}| = \mathrm{eps}$. Thus the number of required steps is

$$n = \frac{\log(\log(2) - \log(\log(x)) - \log(\mathrm{eps})) - \log(\log(2) - \log(|e^0|))}{\log(2)} - 1$$
$$\approx \frac{\log(-\log(\mathrm{eps})) - \log(-\log(|e^0|))}{\log(2)} - 1$$

The code in Code 9.2.4 is for your reference.

C++11-code 9.2.4: Modified `myfunction`.

```
double myfunction_modified(double x) {
    double log2 = 0.693147180559945;
    double y = 0;
    while(x > std::sqrt(2)) { x/=2; y+=log2; }
    while(x < 1./std::sqrt(2)) { x*=2; y-=log2; }
    double z=x-1;
    double dz=x*std::exp(-z)-1.;
    // TO DO: (9-2.e) Write a for-loop that achieves the same accuracy
    // as the third while-loop of myfunction(). I.e. fix the number of
    // iterations before looping.
    // START
    double e0 = z - std::log(x);
    double eps = std::numeric_limits<double>::epsilon();
    double k=
        (std::log(-std::log(eps))-std::log(-std::log(std::abs(e0))))/std::log(2);
    for (int i=1; i<k; ++i) {
```

```
        z = z+dz;
        dz = x * std::exp(-z) - 1;
    }
    // END
    return y+z+dz;
}
```

▲

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 $F(x) = \arctan x$.

HINT 1 for (9-3.a): Refer to [Lecture → Fig. 270] and find out for which initial guesses the Newton method oscillates between two values.

HINT 2 for (9-3.a):

Clearly the solution is $x^* = 0$.
Graphical considerations may help you to find the solution, see Fig. 53. If $x^{(0)}$ is close to zero, the the Newton iteration will converge, for $|x^{(0)}|$ large it will diverge.
However, there are two initial guesses $\pm x^{(0)}$, for which $x^{(1)} = -x^{(0)}$ and those mark the boundary between the two sets.
The condition $x^{(1)} = -x^{(0)}$ gives you a non-linear equation that you can solve for $x^{(0)}$.

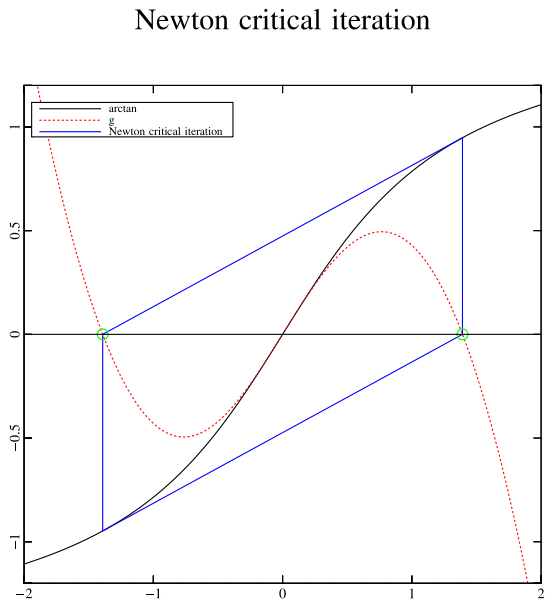


Fig. 53

SOLUTION of (9-3.a):

The function $\arctan(x)$ is **positive, increasing and concave** for positive x, therefore the first iterations of Newton's method with initial points $0 < x^{(0)} < y^{(0)}$ satisfy $y^{(1)} < x^{(1)} < 0$ (draw a sketch to see it). The function is **odd**, i.e., $\arctan(-x) = -\arctan(x)$ for every $x \in \mathbb{R}$, therefore the analogous holds for initial negative values ($y^{(0)} < x^{(0)} < 0$ gives $0 < x^{(1)} < y^{(1)}$). Moreover, opposite initial values give opposite iterations: if $y^{(0)} = -x^{(0)}$ then $y^{(n)} = -x^{(n)}$ for every $n \in \mathbb{N}$.

All these facts imply that, if $|x^{(1)}| < |x^{(0)}|$, then the absolute values of the following iterations will converge monotonically to zero. Vice versa, if $|x^{(1)}| > |x^{(0)}|$, then the absolute values of the Newton's iterations will diverge monotonically. Moreover, the iterations change sign at each step, i.e., $x^{(n)} \cdot x^{(n+1)} < 0$.

It follows that the smallest positive initial guess $x^{(0)}$ for which Newton's method does not converge

satisfies $x^{(1)} = -x^{(0)}$. This can be written as

$$x^{(1)} = x^{(0)} - \frac{f(x^{(0)})}{f'(x^{(0)})} = x^{(0)} - (1 + (x^{(0)})^2) \arctan x^{(0)} = -x^{(0)} .$$

Therefore, $x^{(0)}$ is a zero of the function

$$g(x) := 2x - (1 + x^2) \arctan x \quad \text{with} \quad g'(x) = 1 - 2x \arctan x .$$

▲

(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 of (9-3.b):

Newton's iteration to find the smallest positive initial guess reads

$$x^{(n+1)} = x^{(n)} - \frac{2x^{(n)} - (1 + (x^{(n)})^2) \arctan x^{(n)}}{1 - 2x^{(n)} \arctan x^{(n)}} = \frac{-x^{(n)} + (1 - (x^{(n)})^2) \arctan x^{(n)}}{1 - 2x^{(n)} \arctan x^{(n)}} .$$

C++-code 9.3.1: Solution of Sub-problem (9-3.b)

```
double newton_arctan(double x0_ = 2) {
  // TO DO (9-3.c): define a suitable Newton iteration to find the
  // critical value x0 for F(x) = arctan(x).
  double x0 = x0_;
  // START
  double upd = 1;
  double eps = std::numeric_limits<double>::epsilon();

  while (upd > eps) {
    // Newton iteration for g(x)
    double x1 =
        (-x0 + (1 - x0 * x0) * std::atan(x0)) / (1 - 2 * x0 * std::atan(x0));
    // Relative size of the Newton update
    upd = std::abs((x1 - x0) / x1);
    x0 = x1;
  }
  // END
  return x0;
}
```

▲

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

$$x^{(n+1)} := x^{(n)} - \frac{f(x^{(n)})}{g(x^{(n)})}\,, \quad \text{where} \quad g(x) = \frac{f(x + f(x)) - f(x)}{f(x)}\,. \tag{9.4.1}$$

(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 of (9-4.a):

The implementation of (9.4.1) is straightforward. The finite-precision-based stopping rule can be implemented relying on the actual machine precision of by comparing two successive iterates.

C++-code 9.4.2: Solution of Sub-problem (9-4.a)

```
template <class Function> double steffensen(Function &&f, double x0) {
  double x = x0;
  // TO DO (9-4.a): implement the Steffensen's method for a function f
  // START
  double upd = 1;
  double eps = std::numeric_limits<double>::epsilon();
  // Iterate until machine precision is reached
  while (std::abs(upd) > eps * std::abs(x)) {
    double fx = f(x); // Only 2 evaluations of f at each step
    if (fx != 0) {
      upd = fx * fx / (f(x + fx) - fx);
      x -= upd;
    } else {
      upd = 0;
    }
  }
  // END
  return x;
}
```

▲

(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 of (9-4.b):

C++ code 9.4.3: Testing function for Steffensen's method

```
void testSteffensen(void) {
  // TO DO (9-4.b): write a test of your implementation, that prints
  // an estimate of the zero of f(x) = xe^x - 1
  // START
  double x = steffensen([](double x) { return x * std::exp(x) - 1; }, 1.0);
  std::cout << "The iterative method converges to " << x << std::endl;
  // END
}
```

▲

(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 of the order of Steffensen's method when applied to the test case of Sub-problem (9-4.b).

HINT 1 for (9-4.c): [Lecture → Def. 8.1.1.10] defines the order of convergence of an iteration for solving $f(x) = 0$, while [Lecture → Rem. 8.1.1.12] explains how to extract an estimate for the order from measured errors. ⌟

SOLUTION of (9-4.c):

C++ code 9.4.4: Logger class for keeping track of iteration errors

```
template <class T> class Logger {
  std::vector<T> info;

public:
  // Hint: there is no need to implement a constructor
  // since info is initialized as empty vector by default.

  // TO DO: overload the operator ( ) so that it adds its argument to
  // the member "info".
  void operator()(T val) {
```

```
    // START
    info.push_back(val);
    // END
  }
  // TO DO: Define a member function that fetches the data contained in "info".
  // START
  std::vector<T> getInfo(void) { return info; }
  // END

  void print_log() {
    for (auto v : info) {
      std::cout << v << std::endl;
    }
  }
};
```

C++ code 9.4.5: Steffensen's method with logging of iterates

```
template <class Function>
double steffensen_log(Function &&f, double x0,
                      Logger<double> *logger_p = nullptr) {

  double x = x0;
  // TO DO (9-4.c): Modify the function steffensen: use the class Logger to
  // save all the iterations x of the Steffensen's method.
  // START
  bool log_enabled = false;
  if (logger_p != nullptr) {
    log_enabled = true;
    (*logger_p)(x);
  }
  double upd = 1;
  double eps = std::numeric_limits<double>::epsilon();

  while (std::abs(upd) > eps) {
    double fx = f(x); // Only 2 evaluations of f at each step
    if (fx != 0) {
      upd = fx * fx / (f(x + fx) - fx);
      x -= upd;
      if (log_enabled) { (*logger_p)(x); }
    }
  }
  else {
    upd = 0;
  }
}
// END
return x;
}
```

C++ code 9.4.6: Empiric testing of the order of Steffensen's method

```
void orderSteffensen(void) {
  auto f = [](double x) { return x * std::exp(x) - 1; };
  double x_star = 0.567143290409784; // use as exact value
  // TO DO (9-4.c): tabulate values of
```

```
  // Hint:  to approximate the convergence rate, use the formula
  // (log(e_i) - log(e_{i-1}))/(log(e_{i-1}) - log(e_{i-2}))
  // START
  Logger<double> logger;

  steffensen_log(f, 1.0, &logger);
  std::vector<double> myData = logger.getInfo();
  unsigned n = myData.size();

  VectorXd errs(n), log_errs(n);
  for (unsigned i = 0; i < n; ++i) {
    errs(i) = std::abs(myData[i] - x_star);
    log_errs(i) = std::log(errs(i));
  }
  VectorXd ratios = VectorXd::Zero(n);
  for (unsigned i = 2; i < n; ++i) {
    ratios(i) =
        (log_errs(i) - log_errs(i - 1)) / (log_errs(i - 1) - log_errs(i - 2));
  }
  // Print output
  std::cout.precision(10);
  std::cout << std::setw(20) << "x" << std::setw(20) << "errors"
            << std::setw(20) << "ratios" << std::endl;
  std::cout << "        ____________________________________________________"
            << std::endl;
  for (unsigned i = 0; i < n; ++i) {
    std::cout << std::setw(20) << myData[i] << std::setw(20) << errs(i)
              << std::setw(20) << ratios(i) << std::endl;
  }
  // END
}
```

The output is:

x	error e_n	$\frac{\log(e_n)-\log(e_{n-1})}{\log(e_{n-1})-\log(e_{n-2})}$
1.000000000000000	0.432856709590216	
0.923262600967822	0.356119310558038	
0.830705934728425	0.263562644318641	1.542345498206531
0.727518499997190	0.160375209587406	1.650553641703975
0.633710518522047	0.066567228112263	1.770024323911885
0.579846053882820	0.012702763473036	1.883754995643305
0.567633791946526	0.000490501536742	1.964598248590593
0.567144031581974	0.000000741172191	1.995899954235929
0.567143290411477	0.000000000001693	1.999927865685712
0.567143290409784	0.000000000000000	0.741551601040667

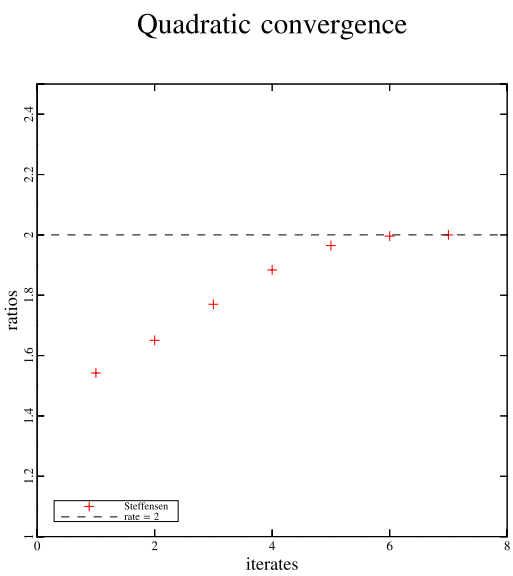


Fig. 54

◁ quotients: $\dfrac{\log(e_n) - \log(e_{n-1})}{\log(e_{n-1}) - \log(e_{n-2})}$

The convergence is obviously quadratic, as can also be seen in Fig. 54. As in the experiments shown in class, roundoff affects the estimated order, once the iteration error approaches the machine precision.

▲

(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?

HINT 1 for (9-4.d): If f is a function and $h : [a,b] \to \mathbb{R}$ with $h(x) \neq 0,\ \forall x \in [a,b]$, then $(fh)(x) = 0 \quad \Leftrightarrow \quad f(x) = 0$. ⌟

SOLUTION of (9-4.d):

The choice $\tilde{f}(x) := (hf)(x) = e^{-x}f(x) = x - e^{-x}$ prevents the blow up of the function g and allows to use a larger set of positive initial points. Of course, $\tilde{f}(x) = 0$ exactly when $f(x) = 0$.

▲

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 $F(\mathbf{x}) = 0$ 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]:

$$\exists C > 0: \quad \|\mathbf{x}^{(k+1)} - \mathbf{x}^*\| \leq C\|\mathbf{x}^{(k)} - \mathbf{x}^*\|^p \qquad \forall k \quad \text{and} \quad p > 1\,. \tag{9.5.1}$$

(9-5.a) (20 min.) Determine $\epsilon_0 > 0$ as large as possible such that

$$\|\mathbf{x}^{(0)} - \mathbf{x}^*\| \leq \epsilon_0 \qquad \Longrightarrow \qquad \lim_{k\to\infty} \mathbf{x}^{(k)} = \mathbf{x}^*\,. \tag{9.5.2}$$

In other words, ϵ_0 tells us which distance of the initial guess from $\mathbf{x}^*$ still guarantees local convergence.

SOLUTION of (9-5.a):

$$\lim_{k\to\infty} x^{(k)} = x^* \iff \lim_{k\to\infty} \|x^{(k)} - x^*\| = 0.$$

Thus, we seek an upper bound $B(k)$ for $\|x^{(k)} - x^*\|$ and claim that $\lim_{k\to\infty} B(k) = 0$.

$$\begin{aligned}
\|x^{(k)} - x^*\| &\leq C\|x^{(k-1)} - x^*\|^p \\
&\leq C \cdot C^p \|x^{(k-2)} - x^*\|^{p^2} \\
&\leq C \cdot C^p \cdot C^{p^2} \|x^{(k-3)} - x^*\|^{p^3} \\
&\vdots \\
&\leq C \cdots C^{p^{k-1}} \|x^{(0)} - x^*\|^{p^k} \\
&= C^{\sum_{i=0}^{k-1} p^i} \|x^{(0)} - x^*\|^{p^k} \\
&\overset{\text{geom. series}}{=} C^{\frac{p^k-1}{p-1}} \|x^{(0)} - x^*\|^{p^k} \\
&\leq C^{\frac{p^k-1}{p-1}} \epsilon_0^{p^k} = \underbrace{C^{\frac{1}{1-p}}}_{\text{const.}} \cdot \left(C^{\frac{1}{p-1}} \epsilon_0\right)^{p^k} = B(k)\,.
\end{aligned}$$

Finally, $\lim_{k\to\infty} B(k) = 0 \quad \iff \quad C^{\frac{1}{p-1}}\epsilon_0 < 1 \quad \Longrightarrow \quad 0 < \epsilon_0 < C^{\frac{1}{1-p}}$

▲

(9-5.b) (15 min.) Provided that $\|\mathbf{x}^{(0)} - \mathbf{x}^*\| < \epsilon_0$ is satisfied with ϵ_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 of (9-5.b):

Using the previous upper bound and the condition τ, we obtain: $\|x^{(k)} - x^*\| \leq C^{\frac{1}{1-p}} \cdot \left(C^{\frac{1}{p-1}}\epsilon_0\right)^{p^k} < \tau$. Solving for the minimal k (and calling the solution $k_{\min}$), with the additional requirement that $k \in \mathbb{N}$, we obtain:

$$\ln\left(C^{\frac{1}{1-p}}\right) + p^k \cdot \underbrace{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}_{<0} < \ln\tau$$

$$k > \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}\right) \cdot \frac{1}{\ln(p)}, \qquad k_{\min} \in \mathbb{N}$$

$$k_{min} = \left\lceil \ln\left(\frac{\ln(\tau) + \frac{1}{p-1}\ln(C)}{\ln\left(C^{\frac{1}{p-1}}\epsilon_0\right)}\right) \cdot \frac{1}{\ln(p)} \right\rceil$$

▲

(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 $(\epsilon_0, \tau) \in \texttt{linspace}\left(\left[0, C^{\frac{1}{1-p}}\right]^2\right)$.

SOLUTION of (9-5.c):

The minimum number of iterations given ϵ_0 and τ is shown in Fig. 55, which was produced with the MATLAB code `k_min_plot.m`.

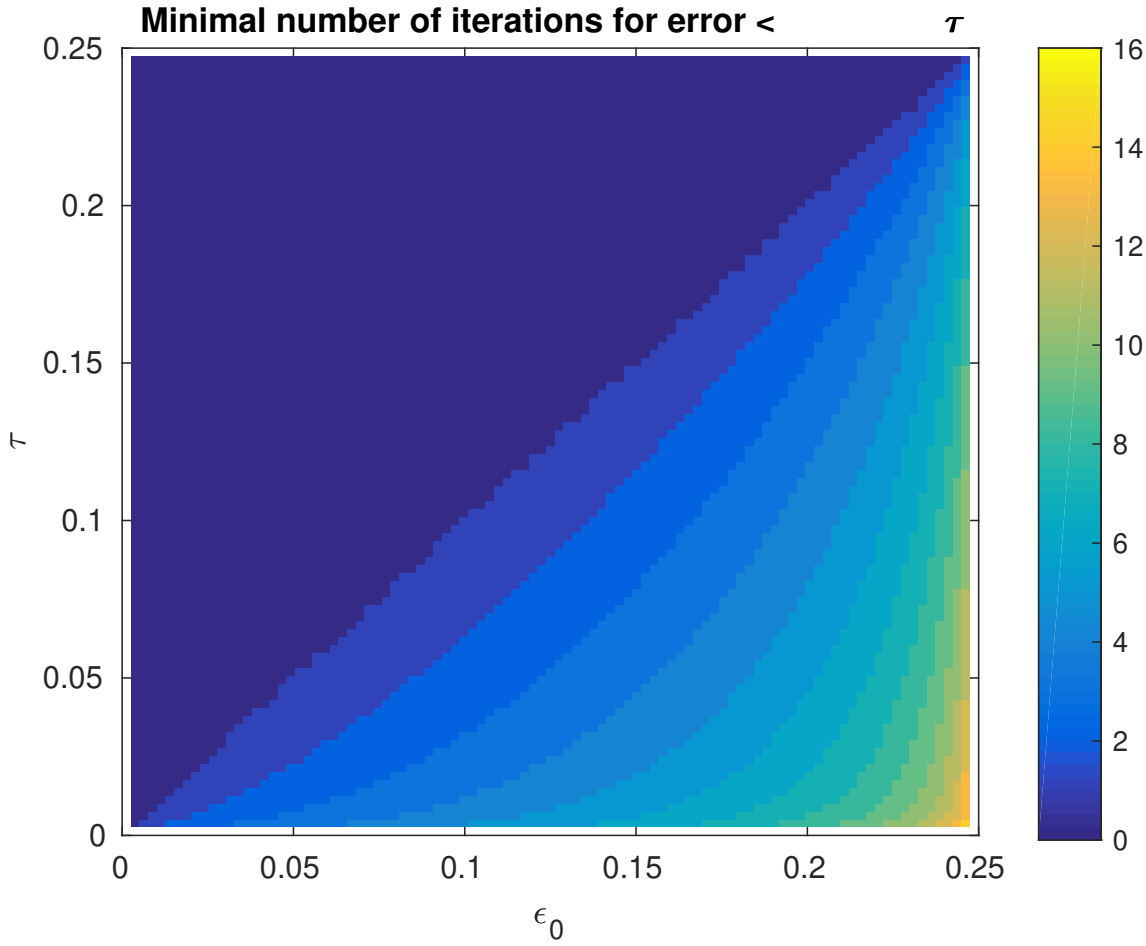


Fig. 55

▲

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 $\left(x^{(n)}\right)_{n\in\mathbb{N}}$, $x^{(n)} \in \mathbb{R}$ we suppose a recursive bound for the norms of the iteration errors of the form

$$\|e^{(n+1)}\| \leq \|e^{(n)}\| \sqrt{\|e^{(n-1)}\|}\,, \tag{9.6.1}$$

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.

HINT 1 for (9-6.a): Consult [Lecture → Rem. 8.1.1.12] to learn how to estimate orders of convergence from sequences of error norms.

SOLUTION of (9-6.a):

To estimate the order of convergence according to [Lecture → Def. 8.1.1.10] we have to track the quotients

$$\frac{\log e^{(k+1)} - \log e^{(k)}}{\log e^{(k)} - \log e^{(k-1)}}\,.$$

C++11-code 9.6.2: Measuring order of convergence

```
int n = 10; // compute ten error terms
std::vector<double> e(n), loge(n);
e[0] = 1; // initial error
loge[0] = std::log(e[0]);
e[1] = 0.8;
loge[1] = std::log(e[1]);
for (int k = 1; k < n; ++k) {
  e[k + 1] = e[k] * std::sqrt(e[k - 1]); // the error recursion
  loge[k + 1] = std::log(e[k + 1]); // logarith of the error
  std::cout << (loge[k + 1] - loge[k]) / (loge[k] - loge[k - 1]) << std::endl;
}
```

The order appear to be arrproximatively 1.36585.

▲

(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].

HINT 1 for (9-6.b): First of all note that we may assume equality in both the error recursion and the bound $\|e^{(n+1)}\| \leq C\|e^{(n)}\|^p$ that defines convergence of order $p > 1$, because in both cases equality corresponds to a worst case scenario. ⌟

HINT 2 for (9-6.b): Study how the order p of convergence was derived from the error recurrence [Lecture → Eq. (8.3.2.30)] in [Lecture → Rem. 8.3.2.26]. ⌟

HINT 3 for (9-6.b): Assuming sharpness of the bounds (worst case scenario) now plug the error recursion and the bound $\|e^{(n+1)}\| \leq C\|e^{(n)}\|^p$ into each other and obtain an equation of the type $\ldots = 1$, where the left hand side involves an error norm that can become arbitrarily small. This implies a condition on p and allows to determine $C > 0$. A formal proof by induction (not required) can finally establish that these values provide a correct choice. ⌟

SOLUTION of (9-6.b):

$\|e^{(n)}\| = C\|e^{(n-1)}\|^p$ (p is the largest convergence order and C is some constant). Then

$$\|e^{(n+1)}\| = C\|e^{(n)}\|^p = C(C\|e^{(n-1)}\|^p)^p = C^{p+1}\|e^{(n-1)}\|^{p^2} , \tag{9.6.3}$$

We assume equality in the error recusion, because this is the worst case. Thus,

$$\|e^{(n+1)}\| = \|e^{(n)}\| \cdot \|e^{(n-1)}\|^{\frac{1}{2}} = C\|e^{(n-1)}\|^{p+\frac{1}{2}} . \tag{9.6.4}$$

Combining the previous equations:

$$C^{p+1}\|e^{(n-1)}\|^{p^2} = C\|e^{(n-1)}\|^{p+\frac{1}{2}} ,$$

i.e.

$$C^p\|e^{(n-1)}\|^{p^2-p-\frac{1}{2}} = 1 . \tag{9.6.5}$$

Since this equation holds for each $n \geq 1$, we have

$$p^2 - p - \frac{1}{2} = 0$$

$$p = \frac{1+\sqrt{3}}{2} \quad \text{or} \quad p = \frac{1-\sqrt{3}}{2} \quad \text{(dropped)} .$$

For C we find the maximal value 1.

▲

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 α, β and the thermal voltage U_T.
Now we consider the circuit depicted in Fig. 56 and assume that all resistors have resistance $R = 1$.

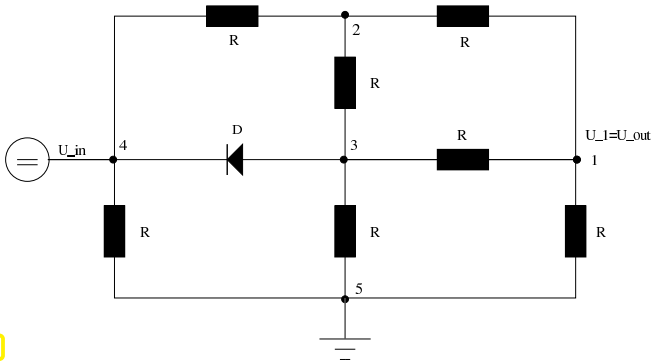


Fig. 56

(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 of (9-7.a):

We consider Kirchhoff's law $\sum_{k,j} I_{kj} = 0$ for every node k. I_{kj} contributes to the sum if node j is connected to node k trough one element (R,C,L,D).

(1) $3U_1 - 0 - U_2 - U_3 = 0$

(2) $3U_2 - U_1 - U_3 - U = 0$

(3) $3U_3 - 0 - U_1 - U_2 + \alpha\left(e^{\beta\left(\frac{U_3-U}{U_T}\right)} - 1\right) = 0$

Thus the nonlinear system of equations reads

$$\mathbf{F}(\mathbf{u}) = \begin{pmatrix} 3U_1 - U_2 - U_3 \\ 3U_2 - U_1 - U_3 - U \\ 3U_3 - U_1 - U_2 + \alpha e^{-\frac{\beta}{U_T}U} \cdot e^{\frac{\beta}{U_T}U_3} - \alpha \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad \text{for } \mathbf{u} = \begin{pmatrix} U_1 \\ U_2 \\ U_3 \end{pmatrix}.$$

▲

(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 `Uout` (at node 1 in Fig. 56) for a *sorted* vector of input voltages `Uin` (at node 4) and for a thermal voltage $U_T = 0.5$. The parameters `alpha`, `beta` pass the (non-dimensional) diode parameters.

Use Newton's method to solve $\mathbf{F}(\mathbf{u}) = \mathbf{0}$ with a relative tolerance of $\tau = 10^{-6}$.

SOLUTION of (9-7.b):

An iteration of the multivariate Newton method is $\mathbf{u}^{(k+1)} = \mathbf{u}^{(k)} - \left(D\mathbf{F}(\mathbf{u}^{(k)})\right)^{-1}\mathbf{F}(\mathbf{u}^{(k)})$. The components of the Jacobian $D\mathbf{F}$ are defined as $D\mathbf{F}(\mathbf{u})_{ij} := \frac{\partial F_i(\mathbf{u})}{\partial u_j}$. For the function obtained in a) we get $D\mathbf{F}(\mathbf{u}) = \mathbf{J} = \begin{pmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 + \frac{\alpha\beta}{U_T} e^{\beta\frac{U_3 - U}{U_T}} \end{pmatrix}$. For the implementation in EIGEN, see file `nonlinear_circuit.cpp`.

▲

(9-7.c) ⚁ We are interested in the nonlinear effects introduced by the diode. Implement a C++ fuunction

```
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 of (9-7.c):

The nonlinear effects can be observed by calculating the differences between the solutions U_{out}: see file `nonlinear_circuit.cpp`.
The relationship $U_{\text{out}} = U_{\text{out}}(U_{\text{in}})$ is obviously not affine-linear.

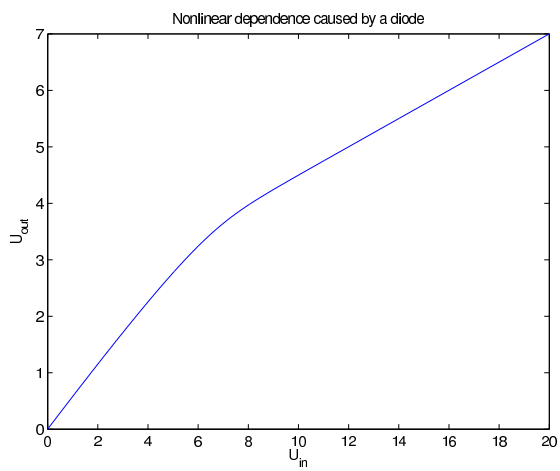


Fig. 57

▲

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

$$z^3 = 1 \tag{9.8.1}$$

has three solutions:

$$z_1 = 1 \quad , \quad z_2 = -\frac{1}{2} + \frac{1}{2}\sqrt{3}\imath \quad , \quad z_3 = -\frac{1}{2} - \frac{1}{2}\sqrt{3}\imath \, , \tag{9.8.2}$$

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 $(x,y) \mapsto z = x + \imath y$. 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 of (9-8.a):

We have

$$z^3 - 1 = (x + \imath y)^3 - 1 = x^3 + 3\imath x^2 y - 3xy^2 - \imath y^3 - 1 = x^3 - 3xy^2 - 1 + \imath(3x^2 y - y^3) = 0 \, .$$

Thus, Eq. (9.8.1) is equivalent to the 2×2 non-linear system of equations

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix} x^3 - 3xy^2 - 1 \\ 3x^2 y - y^3 \end{bmatrix} = \mathbf{0} \, , \quad \mathbf{x} := \begin{bmatrix} x \\ y \end{bmatrix} \, . \tag{9.8.3}$$

▲

(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 of (9-8.b):

The Newton iteration in $\mathbb{R}^n$ reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(D\mathbf{F}(\mathbf{x}^{(k)})\right)^{-1} \mathbf{F}(\mathbf{x}^{(k)}) \, . \tag{9.8.4}$$

see [Lecture → Eq. (8.4.1.1)], where $\mathrm{D}\,\mathbf{F}$ is the Jacobian

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \begin{bmatrix} 3x^2 - 3y^2 & -6xy \\ 6xy & 3x^2 - 3y^2 \end{bmatrix} \, . \tag{9.8.5}$$

▲

(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 $\mathrm{D}F(\mathbf{x})$ for for the function $F : \mathbb{R}^2 \to \mathbb{R}^2$ found in Sub-problem (9-8.a).

SOLUTION of (9-8.c):

C++ code 9.8.6: Evaluation of F

```
Vector2d F(const Vector2d &z) {
  Vector2d Fz;
  // TO DO: Implement the function F : R^2 -> R^2 such that F(z)=0 is
  // equivalent to z^3 = 1 (interpreting z as a complex number).
  // START
  Fz << z(0) * z(0) * z(0) - 3.0 * z(0) * z(1) * z(1) - 1.0,
      3.0 * z(0) * z(0) * z(1) - z(1) * z(1) * z(1);
  // END
  return Fz;
}
```

C++ code 9.8.7: Evaluation of Jacobian $\mathrm{D}F$

```
Matrix2d DF(const Vector2d &z) {
  Matrix2d DFz;
  // TO DO: Implement the Jacobian of F at z.
  // START
  double a = 3.0 * (z(0) * z(0) - z(1) * z(1));
  double b = 6.0 * z(0) * z(1);
  DFz << a, -b, b, a;
  // END
  return DFz;
}
```

▲

(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 is creates and saves a plot in `julia.png`.

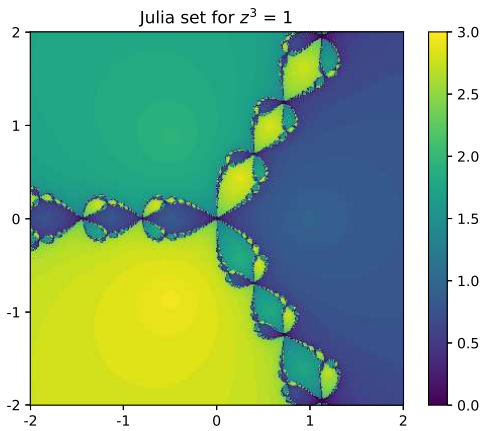


Fig. 58

◁ The Julia set for $z^3 - 1 = 0$ on a mesh containing 780×780 points, `N_it=20`.

This is how your final plot should look like.

HINT 1 for (9-8.d): You may stop the iteration once you are closer in distance to one of the three roots of unity than 10^{-4}.

The three (non connected) sets of points whose iterations converge to the different z_i are called Fatou domains: their boundaries are the Julia sets. ⌟

SOLUTION of (9-8.d):

C++ code 9.8.8: Plotting of attractors of complex roots

```
void julia(void) {
  // Exact solutions of z^3 = 1 for complex z.
  // Use 2-dimensional real vectors to represent complex numbers.
  Vector2d z1, z2, z3;
  z1 << 1, 0;
  z2 << -0.5, 0.5 * std::sqrt(3);
  z3 << -0.5, -0.5 * std::sqrt(3);

  // Tolerance and maximum number of iterations for Newton's method.
  double tol = 1e-4;
  int N_it = 20;

  // The image will have a resolution of res*res.
  int res = 780;
  // We imagine that we have a res*res grid Z on the square [-2,2] with
  // Z(0,0)=(-2,-2),     Z(0,res-1)=(-2,2),
  // Z(res-1,0)=(2,-2), Z(res-1,res-1)=(2,2), etc.
  // C(i,j) is the color assigned to the (i,j)-th point on the grid.
  MatrixXd C(res, res);

  // TO DO: Fill C with real numbers, such that each entry C(i,j) corresponds to
  // which of the roots (z1, z2, or z3) Newton's method converges
  // (if it converges), when using Z(i,j) as a starting point.
  // The values in C will be interpreted as colors below.
  // Additional: You may choose different shades of each color to indicate
  // how many iterations were needed to reach the tolerance.
  // Hint: To speed up runtimes, start with a low value for the parameter res.
  // START
  C.setZero();
  for (int i = 0; i < res; i++) {
    for (int j = 0; j < res; j++) {
      // Newton iteration starting at Z(i,j)=z=(x,y).
```

```
      double x = -2 + 4.0 * i / (res - 1);
      double y = -2 + 4.0 * j / (res - 1);
      Vector2d z(x, y);
      for (int k = 0; k < N_it; k++) {
        z -= DF(z).lu().solve(F(z));
        // Check if we are sufficiently close to any of the three roots:
        if ((z - z1).norm() < tol) {
          C(i, j) = 1.0 - 1.0 * k / N_it;
          break;
        }
        if ((z - z2).norm() < tol) {
          C(i, j) = 2.0 - 1.0 * k / N_it;
          break;
        }
        if ((z - z3).norm() < tol) {
          C(i, j) = 3.0 - 1.0 * k / N_it;
          break;
        }
      }
    }
  }
  // END
  // Axis labels
  std::vector<double> ticks(5);
  for (int i = 0; i < 5; ++i) {
    ticks[i] = i * (res - 1) / 4;
  }
  std::vector<std::string> labels{"-2", "-1", "0", "1", "2"};

  plt::figure();
  // Need to transpose C because plt::imshow() uses
  // first index for y axis and second index for x axis.
  // Different colormaps can be found at
  // https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html
  plt::imshow(C.transpose(), {{"cmap", "viridis"}, {"origin", "lower"}});
  plt::colorbar();
  plt::xticks(ticks, labels);
  plt::yticks(ticks, labels);
  plt::title("Julia set for z^3 = 1");
  plt::savefig("./cx_out/julia.png");
}
```

For each starting point at most `N_it` iterations are carried out. For a given starting point $\mathbf{x}^{(0)}$, as soon as the condition $|x_1^{(k)} + ix_2^{(k)} - z_\ell| < 10^{-4}$, $\ell \in \{1,2,3\}$, is satisfied, we assign a color depending on the root z_i and on the number of iterations done. Each attractor z_ℓ, $\ell \in \{1,2,3\}$ is associated to blue, green, or yellow, respectively; lighter colors correspond to points with faster convergence. Starting points for which the iteration does not converge within `N_it` iterations are dark blue.

▲

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 $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$), the following iterative method has been proposed:

$$\begin{aligned} \mathbf{y}^{(k)} &= \mathbf{x}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})\,, \\ \mathbf{x}^{(k+1)} &= \mathbf{y}^{(k)} - \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{y}^{(k)})\,, \end{aligned} \tag{9.9.1}$$

where $\mathrm{D}\,\mathbf{F}(\mathbf{x}) \in \mathbb{R}^{n,n}$ is the Jacobian matrix of $\mathbf{F}$ evaluated in the point $\mathbf{x}$.

(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 of (9-9.a):

If $\mathbf{F}(\mathbf{x}^{(k)}) = \mathbf{0}$ then $\mathbf{y}^{(k)} = \mathbf{x}^{(k)} + \mathbf{0} = \mathbf{x}^{(k)}$ and $\mathbf{x}^{(k+1)} = \mathbf{y}^{(k)} - \mathbf{0} = \mathbf{x}^{(k)}$. So, by induction, if $\mathbf{F}(\mathbf{x}^{(0)}) = \mathbf{0}$ then $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} = \mathbf{x}^{(0)}$ for every k.

Conversely, if $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)}$, then, by the recursion of the Newton method:

$$\mathbf{F}(x^{(k)}) = \mathbf{F}(x^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)})) \quad \Leftrightarrow \quad D\mathbf{F}(\mathbf{x}^{(k)})^{-1}\,\mathbf{F}(\mathbf{x}^{(k)}) = 0 \quad \Leftrightarrow \quad \mathbf{F}(\mathbf{x}^{(k)}) = 0\,.$$

▲

(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 of (9-9.b):

C++11-code 9.9.2: Implementation of `mod_newt_step_scalar`.

```
template <typename Scalar, class Function, class Jacobian>
Scalar mod_newt_step_scalar(const Scalar& x,
                            const Function& f,
```

```
                                const Jacobian& df) {
    Scalar y = x + f(x) / df(x);
    return y - f(y) / df(x);
}
```

Get it on GitLab (modnewt.cpp).

▲

(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.

HINT 1 for (9-9.c): The exact solution is $x = \tan(a) = 0.123624065869274\ldots$.

HINT 2 for (9-9.c): Remember that $\arctan'(x) = \frac{1}{1+x^2}$.

SOLUTION of (9-9.c):

We implement a generic function that performs as many steps of a generic iteration and terminates if the error is sufficiently small, or if the maximum number of iterations has been reached. To this function, we pass the stepping function, the error function, the intial guess, and termination parameters.

C++11-code 9.9.3: Implementation of `sample_nonlinear_solver`.

```
template <class StepFunction, class Vector, class ErrorFunction>
bool sample_nonlinear_solver(const StepFunction& step,
                             Vector & x,
                             const ErrorFunction& errf,
                             double eps = 1e-8, int max_itr = 100) {
    // Temporary where to store new step
    Vector x_new = x;
    double r = 1;

    for(int itr = 0; itr < max_itr; ++itr) {

        // Compute error (or residual)
        r = errf(x);

        std::cout << "[Step " << itr << "] Error: " << r << std::endl;

        // Advance to next step, x_new becomes x_{k+1}
```

```
        x_new = step(x);

        // Termination conditions
        // If tol reached, the we have convergence
        if (r < eps * norm(x)) {
            std::cout << "[CONVERGED] in " << itr << " it. due to err. err = "
                      << r << " < " << eps << "." << std::endl;
            return true;
        }
        x = x_new;
    }

    // If max it reached
    std::cout << "[NOT CONVERGED] due to MAX it. = " << max_itr
              << " reached, err = " << r << "." << std::endl;
    return false;
}
```

Get it on GitLab (modnewt.cpp).

C++11-code 9.9.4: Implementation of `mod_newt_ord`.

```
void mod_newt_ord() {
    // Setting up values, functions and jacobian
    const double a = 0.123;
    auto f = [&a] (double x) { return atan(x) - a; }; // Function which must see a
    auto df = [] (double x) { return 1. / (x*x+1.); }; // Its derivative

    const double x_ex = tan(a); // Exact solution

    // Store solution and error at each time step
    std::vector<double> sol;
    std::vector<double> err;

    // Compute error and push back to err, used in
    // general_nonlinear_solver as breaking condition errf(x) < eps
    auto errf = [&err, x_ex] (double & x) {
        double e =  std::abs(x - x_ex);
        err.push_back(e);
        return e;
    };

    // Perform convergence study with Modified newton for scalar
    std::cout << std::endl
              << "*** Modified Newton method (scalar) ***"
              << std::endl << std::endl;
    std::cout << "Exact: " << x_ex << std::endl;

    // Initial guess and compute initial error
    double x_scalar = 5;
    sol.push_back(x_scalar);
    errf(x_scalar);

    // Lambda performing the next step, used to define a proper
    // function handle to be passed to general_nonlinear_solver
    auto newt_scalar_step = [&sol, &f, &df] (double x) -> double {
        double x_new = mod_newt_step_scalar(x, f, df);
        sol.push_back(x_new);
        return x_new;
```

```
    };

    // Actually perform the solution
    sample_nonlinear_solver(newt_scalar_step, x_scalar, errf);

    // Print solution (final)
    std::cout << std::endl
              << "x^*_scalar = " << x_scalar
              << std::endl;

    // Print table of solutions, errors and EOOC
    auto space = std::setw(15);
    std::cout << space << "sol."
              << space << "err."
              << space << "order"
              << std::endl;
    for(unsigned i = 0; i < sol.size(); ++i) {
        std::cout << space << sol.at(i)
                  << space << err.at(i);
        if(i >= 3) {
            std::cout << space << (log(err.at(i)) - log(err.at(i-1))) /
                (log(err.at(i-1)) - log(err.at(i-2)));
        }
        std::cout << std::endl;
    }
}
```

Get it on GitLab (modnewt.cpp).

The order of convergence is approximately 3:

solution	error	order
5	4.87638	
0.560692	4.87638	
0.155197	0.437068	
0.123644	0.0315725	1.08944
0.123624	2.00333e-05	2.80183
0.123624	5.19029e-15	2.99809

▲

(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 $\mathbf{F} : \mathbb{R}^n \to \mathbb{R}^n$ passed through the functor `f`, which complies with `std::function<Vector(const Vector &)>`; The **Vector** type can be assumed to have the capabilities of **Eigen::VectorXd**, whereas the type **Jacobian** must have an evaluation operator that returns an object compatible with **Eigen::MatrixXd**.

HINT 1 for (9-9.d): Do not forget to perform a single LU-decomposition of the Jacobian and reuse it. See [Lecture → Code 8.4.3.6] for an example. ⌟

SOLUTION of (9-9.d):

▲

In the sequel we consider the non-linear system of equations

$$\mathbf{F}(\mathbf{x}) := \mathbf{A}\mathbf{x} + \begin{bmatrix} c_1 e^{x_1} \\ \vdots \\ c_n e^{x_n} \end{bmatrix} = \mathbf{0}\,, \tag{9.9.5}$$

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 $\mathrm{D}\,\mathbf{F}(\mathbf{x})$ for $\mathbf{F}$ as given in (9.9.5).

SOLUTION of (9-9.e):

We can simply rely on componentwise partial derivatives to derive

$$\mathrm{D}\,\mathbf{F}(\mathbf{x}) = \mathbf{A} + \begin{bmatrix} c_1 e^{x_1} & & & & \\ & c_2 e^{x_1} & & & \\ & & \ddots & & \\ & & & \ddots & \\ & & & & c_n e^{x_n} \end{bmatrix} \in \mathbb{R}^{n,n}\,. \tag{9.9.6}$$

▲

(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 of (9-9.f):

▲

(9-9.g) ⚃ (20 min.) [depends on Sub-problem (9-9.f)]

Write a C++ function

```
void mod_newt_sys_test(void);
```

that tests your implementation of `mod_newt_sys()` for

$$\mathbf{A} = \begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \quad , \quad \mathbf{c} = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} ,$$

and determines the order of convergence from suitable tabulated values.

HINT 1 for (9-9.g): Determine a reference solution by calling `mod_newt_sys()` with a very low tolerance. The invoke that function with `tol = 0` and small and increasing values of `maxit`. ⌟

SOLUTION of (9-9.g):

We implement a "steping" function and reuse the previous codes.

C++11-code 9.9.7: Implementation of `mod_newt_step_system`.

```
template <typename Vector, class Function, class Jacobian>
Vector mod_newt_step_system(const Vector & x,
                            const Function& f, const Jacobian& df) {
    auto lu = df(x).lu();
    // Reusing LU decomposition
    Vector y = x + lu.solve(f(x));
    return y - lu.solve(f(y));
}
```

Get it on GitLab (modnewt.cpp).

C++11-code 9.9.8: Implementation of `mod_newt_sys`.

```
void mod_newt_sys() {

    // Function parameters
    MatrixXd A = MatrixXd::Random(4,4);
    VectorXd c = VectorXd::Random(4);

    A = (A*A.transpose()).eval(); // Make sure A is symmetric
    c = c.cwiseAbs(); // Make sure c is > 0 uniformly

    // Handler for function and jacobian in standard format.  Must see
        matrix A and vector c.
    auto F = [&A,&c] (const VectorXd & x) {
        VectorXd tmp =  A*x + c.cwiseProduct(x.array().exp().matrix()).eval();
        return tmp;
    };
    auto dF = [&A, &c] (const VectorXd & x) {
        MatrixXd C = A;
        VectorXd temp = c.cwiseProduct(x.array().exp().matrix());
        C += temp.asDiagonal();
        return C;
    };

    // Container for errors
```

```
    std::vector<double> err;
    // Define lambda for breaking condition, which also stores
    // the error of the previous step.  Must see 'err'.
    auto rerr = [&err] (VectorXd & x) {
        if(err.size() > 0)
            return err.back();
        else return 1.;

    };

    std::cout << std::endl << "*** Modified Newton method (system) ***"
              << std::endl << std::endl;
    VectorXd x_system = VectorXd::Zero(c.size()); // Initial guess

    // Refactor (i.e. wrap) the step such that it is compatible with
       general_nonlinear_solver.
    // Must see F, dF and 'err'.
    // We also store the error.
    auto newt_system_step = [&F, &dF, &err] (const VectorXd & x) -> VectorXd {
        VectorXd x_new = mod_newt_step_system(x, F, dF);
        double e = (x - x_new).norm()/ x_new.norm();
        err.push_back(e);
        return x_new;
    };

    // Actually performs computations
    sample_nonlinear_solver(newt_system_step, x_system, rerr);

    // Sanity check
    std::cout << std::endl << "x^*_system = "
              << std::endl << x_system
              << std::endl;
    std::cout << std::endl << "F(x^*_system) = "
              << std::endl << F(x_system)
              << std::endl;
}
```

Get it on GitLab (modnewt.cpp).

▲

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:

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b}\,, \tag{9.10.1}$$

as in [Lecture → Ex. 8.4.1.23]. Here, $\mathbf{A} : \mathbb{R}^n \to \mathbb{R}^{n,n}$ is a matrix-valued function:

$$\mathbf{A}(\mathbf{x}) := \begin{bmatrix} \gamma(\mathbf{x}) & 1 & & & & \\ 1 & \gamma(\mathbf{x}) & 1 & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & \gamma(\mathbf{x}) & 1 \\ & & & & 1 & \gamma(\mathbf{x}) \end{bmatrix}, \quad \gamma(\mathbf{x}) := 3 + \|\mathbf{x}\|_2 \tag{9.10.2}$$

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 of (9-10.a):

The fixed-point iteration $\mathbf{x}^{(k)} \to \mathbf{x}^{(k+1)}$, $k \in \mathbb{N}$, is defined by

$$\mathbf{x}^{(k+1)} \in \mathbb{R}^n\colon \quad \mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k+11)} = \boldsymbol{v} \quad \Leftrightarrow \quad \mathbf{x}^{(k+1)} = \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b}\,. \tag{9.10.3}$$

(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 $\mathbf{x}^{(k)}$.

HINT 1 for (9-10.b): For the sake of efficiency you have to use sparse matrix data types, see [Lecture → Section 2.7.2]. ⌟

SOLUTION of (9-10.b):

In this case we know in advance the maximal number of non-zeros per column/row. Thus we can do componentwise initialization without efficiency penalty, see [Lecture → Code 2.7.2.1].

C++11-code 9.10.4: Iterate computation.

```
Eigen::VectorXd fixed_point_step(const Eigen::VectorXd &xk,
                                 const Eigen::VectorXd &b) {
  Eigen::VectorXd x_new;
  // TO DO (9-10.b): implement one step of the fixed point iteration
  // START
  // Define matrix by fixed point x^k
  const int n = xk.size();
  double nrm = xk.norm();
  Eigen::SparseMatrix<double> A(n, n);
  A.reserve(3 * n);
  for (int i = 0; i < n; ++i) {
    if (i > 0)
      A.insert(i, i - 1) = 1;
    A.insert(i, i) = 3 + nrm;
    if (i < n - 1)
      A.insert(i, i + 1) = 1;
  }
  // solve linear system
  Eigen::SparseLU<Eigen::SparseMatrix<double>> Ax_lu(A);
  x_new = Ax_lu.solve(b);
  // END
  return x_new;
}
```

▲

(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 of (9-10.c):

C++11-code 9.10.5: Solution of fixed point.

```
Eigen::VectorXd solveQuasiLinSystem(double rtol, double atol,
                                    Eigen::VectorXd &b) {
  int n = b.size();

  Eigen::VectorXd x(n);
  // TO DO (9-10.c): apply the iterative method fixed_point_step to solve
  // the quasi-linear system
  // START
  x = b;
  Eigen::VectorXd x_new(n);
```

```
  int max_itr = 100;
  for (int itr = 0; itr < max_itr; ++itr) {

    x_new = fixed_point_step(x, b);

    double err = (x - x_new).norm();
    x = x_new;

    if (err < atol || err < rtol * x_new.norm()) {
      std::cout << "Fixed point method successful" << std::endl;
      break;
    }
  }
  // END
  return x;
}
```

▲

(9-10.d) (10 min.) Let $\mathbf{b} \in \mathbb{R}^n$ be given. Write the recursion formula for the solution of

$$\mathbf{A}(\mathbf{x})\mathbf{x} = \mathbf{b} \tag{9.10.6}$$

with the Newton method.

SOLUTION of (9-10.d):

The Newton iteration, as provided in the lecture document in [Lecture → Ex. 8.4.1.23], reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2}\right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \,. \tag{9.10.7}$$

▲

(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 of (9-10.e):

We replace the inversion with the SMW formula: for regular $\mathbf{M} \in \mathbb{R}^{n,n}$, $\mathbf{u}, \mathbf{v} \in \mathbb{R}^n$,

$$(\mathbf{M} + \mathbf{u}\mathbf{v}^\top)^{-1} = \mathbf{M}^{-1}\left(\mathbf{I} - \frac{\mathbf{u}\mathbf{v}^\top\mathbf{M}^{-1}}{1 + \mathbf{v}^\top\mathbf{M}^{-1}\mathbf{u}}\right) \text{, if } \quad 1 + \mathbf{v}^\top\mathbf{M}^{-1}\mathbf{u} \neq 0 \,,$$

and get, for instance,

$$
\begin{aligned}
\mathbf{x}^{(k+1)} &= \mathbf{x}^{(k)} - \left(\mathbf{A}(\mathbf{x}^{(k)}) + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top}{\|\mathbf{x}^{(k)}\|_2}\right)^{-1} (\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\left(\mathbf{I} - \frac{1}{\|\mathbf{x}^{(k)}\|_2}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}}{1+\frac{(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}{\|\mathbf{x}^{(k)}\|_2}}\right)(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b} + \mathbf{A}(\mathbf{x}^{(k)})^{-1}\frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}(\mathbf{A}(\mathbf{x}^{(k)})\mathbf{x}^{(k)} - \mathbf{b}) \\
&= \mathbf{A}(\mathbf{x}^{(k)})^{-1}\left(\mathbf{b} + \frac{\mathbf{x}^{(k)}(\mathbf{x}^{(k)})^\top(\mathbf{x}^{(k)} - \mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{b})}{\|\mathbf{x}^{(k)}\|_2 + (\mathbf{x}^{(k)})^\top\mathbf{A}(\mathbf{x}^{(k)})^{-1}\mathbf{x}^{(k)}}\right).
\end{aligned}
$$

Other rearrangements of this formula are possible.

▲

(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).

HINT 1 for (9-10.f): If you did not manage to solve Sub-problem (9-10.e), use directly the Newton's method formula. ⌟

SOLUTION of (9-10.f):

C++11-code 9.10.8: Newton step.

```
Eigen::VectorXd newton_step(const Eigen::VectorXd &x,
                            const Eigen::VectorXd &b) {
  Eigen::VectorXd x_new;
  // TO DO (9-10.f): implement one Newton step exploiting the
  // Sherman-Morrison-Woodbury formula.
  // START
  // Define matrix by fixed point x^k
  const int n = x.size();
  double nrm = x.norm();
  Eigen::SparseMatrix<double> A(n, n);
  A.reserve(3 * n);
  for (int i = 0; i < n; ++i) {
    if (i > 0)
      A.insert(i, i - 1) = 1;
    A.insert(i, i) = 3 + nrm;
    if (i < n - 1)
      A.insert(i, i + 1) = 1;
  }

  // Reuse LU decomposition with SMW
  Eigen::SparseLU<Eigen::SparseMatrix<double>> Ax_lu(A);

```

```
  // Solve a bunch of systems
  Eigen::VectorXd Axinv_b = Ax_lu.solve(b);
  Eigen::VectorXd Axinv_x = Ax_lu.solve(x);
  // Next step
  x_new = Axinv_b + Axinv_x * ( x.transpose() * (x - Axinv_b)) /
                        (x.norm() + x.dot(Axinv_x));
  // END
  return x_new;
}
```

▲

(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 of (9-10.g):

C++11-code 9.10.9: Solve Newton.

```
Eigen::VectorXd solveQLSystem_Newton(double rtol, double atol,
                                     Eigen::VectorXd &b) {
  int n = b.size();

  Eigen::VectorXd x(n);
  // TO DO (9-10.c): apply the iterative method newton_step to solve
  // the quasi-linear system
  // START
  x = b;
  Eigen::VectorXd x_new(n);
  int max_itr = 100;
  for (int itr = 0; itr < max_itr; ++itr) {

    x_new = newton_step(x, b);

    double err = (x - x_new).norm();
    x = x_new;

    if (err < atol || err < rtol * x_new.norm()) {
      std::cout << "Newton method successful" << std::endl;
      break;
    }
  }
  // END
  return x;
}
```

▲

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

$$A \xrightarrow{\lambda_1} B \xrightarrow{\lambda_2} C.$$

That is A decays into B with a decay rate λ_1. B itself decays into a third substance C with decay rate λ_2. The evolution of the amounts $\Phi_A(t), \Phi_B(t)$ of the substances A and B can be modelled by the following coupled system of ordinary differential equations, see [**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 of (9-11.a):

The first equation is independent of the second one and its solution is

$$\Phi_A(t) = e^{-\lambda_1 t} A_0.$$

We insert it into the second equation of system (9.11.1) and we obtain

$$\Phi_B'(t) \;=\; -\lambda_2 \Phi_B(t) + \lambda_1 \Phi_A(t) \;=\; -\lambda_2 \Phi_B(t) + \lambda_1 e^{-\lambda_1 t} A_0 \,.$$

This equation is of the form $y'(t) = cy(t) + g(t)$ and can be solved using the variation of constants formula

$$y(t) = e^{ct} y(0) + \int_0^t e^{c(t-\tau)} g(\tau)\, d\tau \,.$$

Thus, the solution of the second equation is

$$\begin{aligned} \Phi_B(t) &= e^{-\lambda_2 t} B_0 + \int_0^t e^{-\lambda_2 (t-\tau)} \lambda_1 e^{-\lambda_1 \tau} A_0 \, d\tau &&= e^{-\lambda_2 t} B_0 + \lambda_1 e^{-\lambda_2 t} A_0 \int_0^t e^{(\lambda_2 - \lambda_1)\tau} \, d\tau \\ &= e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} (e^{(\lambda_2 - \lambda_1)t} - 1) A_0 &&= e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(e^{-\lambda_1 t} - e^{-\lambda_2 t} \right) A_0. \end{aligned}$$

It satisfies the initial condition $\Phi_B(0) = B_0$, we test if it is a solution of the ODE (9.11.1):

$$\begin{aligned} \Phi_B'(t) &= -\lambda_2 e^{-\lambda_2 t} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(-\lambda_1 e^{-\lambda_1 t} + \lambda_2 e^{-\lambda_2 t} \right) A_0 \\ &= -\lambda_2 e^{-\lambda_2 t} B_0 + \lambda_2 \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_2 t} A_0 - \lambda_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} e^{-\lambda_1 t} A_0 \end{aligned}$$

$$
\begin{aligned}
&= -\lambda_2 \Phi_B(t) + \lambda_2 \frac{\lambda_1}{\lambda_2 - \lambda_1} \Phi_A(t) - \lambda_1 \frac{\lambda_1}{\lambda_2 - \lambda_1} \Phi_A(t) \\
&= -\lambda_2 \Phi_B(t) + \lambda_1 \Phi_A(t).
\end{aligned}
$$

▲

(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 λ_1 and λ_2. This leads to an overdetermined *non-linear* system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, which has to be solved in least squares sense, see [Lecture → Section 8.5]. 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 Φ_B.

Write down the function $\mathbf{F}$ for the current problem.

HINT 1 for (9-11.b): You cannot do this without having solved Sub-problem (9-11.a). ⌟

SOLUTION of (9-11.b):

The N components of the function which describes the overdetermined system are given by

$$
F_i(\mathbf{x}) \quad = \quad \Phi_B(t_i) - m_i \quad = \quad e^{-\lambda_2 t_i} B_0 + \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i} \right) A_0 - m_i, \qquad i = 1, \dots, N.
$$

For $N > 4$ we have more equations (N measurements) than unknowns (4 parameters), it is not possible in general to solve $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ exactly but only in least squares sense:

$$
\mathbf{x}^* = \operatorname*{argmin}_{\mathbf{x}} \|\mathbf{F}(\mathbf{x})\|_2^2 = \operatorname*{argmin}_{\mathbf{x}} \sum_{i=1}^{N} |F_i(\mathbf{x})|^2.
$$

▲

(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 of (9-11.c):

Remember the definition of the vector $\mathbf{x} = (A_0, B_0, \lambda_1, \lambda_2)^T$; the gradient of each component F_i, $i = 1, \dots, N$, of $\mathbf{F}$ is given by

$$
\mathbf{grad}\, F_i(\mathbf{x}) = \begin{pmatrix} \frac{\lambda_1}{\lambda_2 - \lambda_1} \left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i} \right) \\ e^{-\lambda_2 t_i} \\ \frac{\lambda_2}{(\lambda_2 - \lambda_1)^2} \left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i} \right) A_0 - \frac{\lambda_1 t_i}{\lambda_2 - \lambda_1} e^{-\lambda_1 t_i} A_0 \\ -t_i e^{-\lambda_2 t_i} B_0 - \frac{\lambda_1}{(\lambda_2 - \lambda_1)^2} \left(e^{-\lambda_1 t_i} - e^{-\lambda_2 t_i} \right) A_0 + \frac{\lambda_1 t_i}{\lambda_2 - \lambda_1} e^{-\lambda_2 t_i} A_0 \end{pmatrix} \in \mathbb{R}^4.
$$

The Jacobian $\mathbf{DF}$ is the collection of these vectors:

$$\mathbf{DF}(\mathbf{x}) = \begin{pmatrix} \mathbf{grad}\, F_1(\mathbf{x})^T \\ \vdots \\ \mathbf{grad}\, F_N(\mathbf{x})^T \end{pmatrix} \quad \in \mathbb{R}^{N,4}.$$

The Gauss-Newton iteration reads

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} - \mathbf{s}\,, \quad \text{where } \mathbf{s} \text{ is the solution of the L.S. linear system} \quad \mathbf{DF}(\mathbf{x}^{(k)})\,\mathbf{s} \overset{L.S.}{=} \mathbf{F}(\mathbf{x}^{(k)}),$$

which corresponds to the statement that the increment $\mathbf{s}$ is the solution of the minimization problem

$$\mathbf{s} = \operatorname*{argmin}_{\mathbf{a}\in\mathbb{R}^4} \|\mathbf{F}(\mathbf{x}^{(k)}) - \mathbf{DF}(\mathbf{x}^{(k)})\,\mathbf{a}\|_2.$$

▲

(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 λ_1, λ_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?

HINT 1 for (9-11.d): Choose as initial guess $\mathbf{x}^{(0)} = (1,1,1,0.1)^T$. ⌟

HINT 2 for (9-11.d): The `Eigen::Array` classes should prove useful for implementing $\mathbf{F}$ and $\mathbf{DF}$. ⌟

HINT 3 for (9-11.d): Plot your fitted Φ_B along with the measured data to spot errors in your implementation. ⌟

SOLUTION of (9-11.d):

The implementation of $\mathbf{F}$ and $\mathbf{DF}$ is straightforward. QR decomposition is used to solve each Least Squares Problem.

C++11-code 9.11.2: Solution of (9-11.d)

```
    std::vector<double> gn_update;
    Eigen::Vector4d x(1., 1., 1., 0.1), s;
    do {
        Eigen::Vector4d s = DF(x, t, m).colPivHouseholderQr().solve(F(x, t, m));
        x -= s;
        gn_update.push_back(s.lpNorm<Eigen::Infinity>());
    } while (gn_update.back() > 1e-14);
```

An examination of the changes in `gn_update` shows linear convergence.

▲

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

$$(x - m_1)^2 + (y - m_2)^2 = r^2. \tag{9.12.1}$$

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 of (9-12.a):

Expanding the squares in the circle equation and using the new unknown $c = r^2 - m_1^2 - m_2^2$, we obtain the equivalence

$$\text{eq. (9.12.1)} \quad \Leftrightarrow \quad x^2 - 2m_1x + m_1^2 + y^2 - 2m_2y + m_2^2 - r^2 = 0 \quad \Leftrightarrow \quad -2xm_1 - 2ym_2 - c = -x^2 - y^2.$$

Thus, for the data (x_i, y_i), $i = 1, \dots, N$, we have the $N \times 3$ linear system

$$\begin{pmatrix} -2x_1 & -2y_1 & -1 \\ \vdots & \vdots & \vdots \\ -2x_N & -2y_N & -1 \end{pmatrix} \begin{pmatrix} m_1 \\ m_2 \\ c \end{pmatrix} \overset{L.S.}{=} \begin{pmatrix} -x_1^2 - y_1^2 \\ \vdots \\ -x_N^2 - y_N^2 \end{pmatrix}.$$

▲

(9-12.b)

Write a C++ function

```
Vector3d circl_alg_fit(const VectorXd &x, const VectorXd & y);
```

that receives point coordinates in the vectors `x` and `y` and solves the overdetermined system in least squares sense and returns a 3-dimensional vector (m_1, m_2, r).

SOLUTION of (9-12.b):

C++11-code 9.12.2: Fitting circle using algebraic method.

```
/*!
 * \brief circl_alg_fit
 * \param[in] x
 * \param[in] y
 * \return
 */
Vector3d circl_alg_fit(const VectorXd &x,
                       const VectorXd & y) {
    assert(x.size() == y.size() && "Size mismatch!");

    unsigned int n = x.size();

    MatrixXd A(n,3);
    A << -2*x, -2*y, -VectorXd::Ones(n);
    VectorXd b = -x.array()*x.array()
            - y.array()*y.array();

// Vector3d z = lsqSVD(A, b);
// Vector3d z = lsqNRM(A, b);
    Vector3d z = lsqHHR(A, b);

    z(2) = std::sqrt(z(2) + z(0)*z(0) + z(1)*z(1));

    return z;
}
```

Get it on GitLab (circle_approx.cpp).

▲

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

$$d_i = \left|\sqrt{(m_1 - x_i)^2 + (m_2 - y_i)^2} - r\right| \qquad i = 1, \dots, N,$$

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

$$\mathbf{z}^* = \operatorname*{argmin}_{\mathbf{z}} \|\mathbf{F}(\mathbf{z})\|^2,$$

see [Lecture → Section 8.5].

(9-12.c) ⚀ Write down the concrete function $\mathbf{F} : \mathbb{R}^{n_1} \to \mathbb{R}^{n_2}$ for this non-linear least squares problem.

You can use the auxiliary values

$$R_i = \sqrt{(x_i - m_1)^2 + (y_i - m_2)^2}\,, \quad i = 1, \dots, N.$$

SOLUTION of (9-12.c):

Define three dimensional vector $\mathbf{z} = [z_1, z_2, z_3]^T := [m_1, m_2, r]^T$. Define $\mathbf{F} : \mathbb{R}^3 \to \mathbb{R}^N$ such that, for $i = 1, \dots, N$, its components read

$$F_i(\mathbf{z}) = \sqrt{(x_i - z_1)^2 + (y_i - z_2)^2} - z_3 = \sqrt{(x_i - m_1)^2 + (y_i - m_2)^2} - r = R_i - r = d_i.$$

Notice that $\|\mathbf{F}(\mathbf{z})\|^2 = \sum_{i=1}^n d_i^2$.

▲

(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 Φ and the Hessian $\mathbf{H}\Phi(\mathbf{z})$.

SOLUTION of (9-12.d):

First of all we compute a few derivatives that will be useful later on:

$$\partial_{m_1} R_i = \frac{-(x_i - m_1)}{R_i}, \quad \partial_{m_1}\frac{1}{R_i} = \frac{x_i - m_1}{R_i^3}, \qquad \partial_{m_2} R_i = \frac{-(y_i - m_2)}{R_i}, \quad \partial_{m_2}\frac{1}{R_i}, = \frac{y_i - m_2}{R_i^3}, \tag{9.12.3}$$

$$\partial^2_{m_1,m_1} R_i = \frac{1}{R_i} - \frac{(x_i - m_1)^2}{R_i^3}, \; \partial^2_{m_2,m_2} R_i = \frac{1}{R_i} - \frac{(y_i - m_2)^2}{R_i^3}, \; \partial^2_{m_1,m_2} R_i = \frac{-(x_i - m_1)(y_i - m_2)}{R_i^3}. \tag{9.12.4}$$

Now the gradients of the components of $\mathbf{F}$, assembled in the Jacobian $\mathbf{DF}$, are

$$\mathbf{grad}\, F_i(\mathbf{z}) = \begin{bmatrix} \frac{-(x_i - m_1)}{R_i}, & \frac{-(y_i - m_2)}{R_i}, & -1 \end{bmatrix}^T, \qquad \mathbf{DF}(\mathbf{z}) = \begin{bmatrix} \mathbf{grad}\, F_1(\mathbf{z})^T \\ \vdots \\ \mathbf{grad}\, F_N(\mathbf{z})^T \end{bmatrix} \in \mathbb{R}^{N,3}.$$

Using the formulas in [Lecture → Section 8.5.1], we compute $\mathbf{grad}\,\Phi(\mathbf{z})$

$$\mathbf{grad}\,\Phi(\mathbf{z}) = \mathbf{DF}(\mathbf{z})^T \cdot \mathbf{F}(\mathbf{z}) = -\sum_{i=1}^N (R_i - r) \begin{bmatrix} \frac{x_i - m_1}{R_i} \\ \frac{y_i - m_2}{R_i} \\ 1 \end{bmatrix},$$

and the Hessian

$$\begin{aligned} \mathbf{H}\Phi(\mathbf{z}) &= \underbrace{\mathbf{DF}(\mathbf{z})^T}_{\in\mathbb{R}^{3,N}} \cdot \underbrace{\mathbf{DF}(\mathbf{z})}_{\in\mathbb{R}^{N,3}} + \sum_{i=1}^N \underbrace{F_i(\mathbf{z})}_{\in\mathbb{R}} \underbrace{\mathbf{H}F_i(\mathbf{x})}_{\in\mathbb{R}^{3,3}} \\ &= \sum_{i=1}^N \begin{bmatrix} \frac{x_i - m_1}{R_i} \\ \frac{y_i - m_2}{R_i} \\ 1 \end{bmatrix} \begin{bmatrix} \frac{x_i - m_1}{R_i}, & \frac{y_i - m_2}{R_i}, & 1 \end{bmatrix} + \sum_{i=1}^N (R_i - r) \begin{bmatrix} \frac{1}{R_i} - \frac{(x_i - m_1)^2}{R_i^3} & \frac{-(x_i - m_1)(x_i - m_2)}{R_i^3} & 0 \\ \frac{-(x_i - m_1)(x_i - m_2)}{R_i^3} & \frac{1}{R_i} - \frac{(y_i - m_2)^2}{R_i^3} & 0 \\ 0 & 0 & 0 \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} N + r\sum_{i=1}^N \left(\frac{(x_i-m_1)^2}{R_i^3} - \frac{1}{R_i}\right) & r\sum_{i=1}^N \frac{(x_i-m_1)(y_i-m_2)}{R_i^3} & \sum_{i=1}^N \frac{(x_i-m_1)}{R_i} \\ r\sum_{i=1}^N \frac{(x_i-m_1)(y_i-m_2)}{R_i^3} & N + r\sum_{i=1}^N \left(\frac{(y_i-m_2)^2}{R_i^3} - \frac{1}{R_i}\right) & \sum_{i=1}^N \frac{(y_i-m_2)}{R_i} \\ \sum_{i=1}^N \frac{(x_i-m_1)}{R_i} & \sum_{i=1}^N \frac{(y_i-m_2)}{R_i} & N \end{bmatrix}.$$

▲

(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 Φ.

SOLUTION of (9-12.e):

The Gauss-Newton method reads

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} - \mathbf{s}\,, \qquad \text{where} \quad \mathbf{s} = \operatorname*{argmin}_{\mathbf{a}\in\mathbb{R}^n} \|\mathbf{F}(\mathbf{z}^{(k)}) - \mathbf{DF}(\mathbf{z}^{(k)})\,\mathbf{a}\|_2.$$

C++11-code 9.12.5: Fitting circle using geometric method.

```
Vector3d circl_geo_fit(const VectorXd &x, const VectorXd & y) {

}
```

Get it on GitLab (circle_approx.cpp).

▲

(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 of (9-12.f):

The iteration of the Newton method [Lecture → Eq. (8.5.1.4)] reads

$$\mathbf{z}^{(k+1)} = \mathbf{z}^{(k)} - \mathbf{H}\Phi(\mathbf{z}^{(k)})^{-1}\,\mathbf{grad}\,\Phi(\mathbf{z}^{(k)}).$$

C++11-code 9.12.6: Fitting circle using algebraic method.

Get it on GitLab (circle_approx.cpp).

▲

(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 of (9-12.g):

The norm of the iteration corrections, their rates and the order of convergence, for the Gauss-Newton method are:

```
0.016781921781802 0                 0
0.000376686392908 0.022445962852483 0
0.000012267807813 0.032567695686573 0.901963405665314
0.000000252164042 0.020554939048614 1.134392882682618
0.000000009356509 0.037104850164724 0.847953914409908
0.000000000220690 0.023586749344818 1.137541458413033
0.000000000007329 0.033208458089076 0.908696794653441
0.000000000000187 0.025533305995339 1.077187869363118;
```

for the Newton method are:

```
0.017081920254751 0                 0
0.000016606030814 0.000972140752693 0
0.000000000079130 0.000004765117210 1.766748966559383
0.000000000000001 0.000009145529603 0.946798348902960.
```

The convergence of Gauss-Newton method is linear with rate about 0.03. Newton method converges too fast to determine the order. On the other side, the cost of every iterate is higher (than G.-N.) since it requires the evaluation of the Hessian of Φ.

See also the solution of **??** and the convergence plot in **??**.

▲

9-12.III: constrained fit using SVD

As you may know, for $a \neq 0$ the solution set of the quadratic equation

$$a\mathbf{x}^T\mathbf{x} + \mathbf{b}^T\mathbf{x} + c = 0, \quad \mathbf{b} \in \mathbb{R}^2,\ a, c \in \mathbb{R}, \tag{9.12.7}$$

(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 of (9-12.h):

Defining $\mathbf{x} = (x_1, x_2)$, we have

$$0 \;=\; a\mathbf{x}^T\mathbf{x} + \mathbf{b}^T\mathbf{x} + c \;=\; a(x^2 + y^2) + b_1 x + b_2 y + c$$

exactly when

$$\left(x + \frac{b_1}{2a}\right)^2 + \left(y + \frac{b_2}{2a}\right)^2 = \frac{b_1^2 + b_2^2}{4a^2} - \frac{c}{a}\,.$$

From this we obtain

$$m_1 = \frac{-b_1}{2a}, \quad m_2 = \frac{-b_2}{2a} \quad \text{and} \quad r = \sqrt{\frac{b_1^2 + b_2^2}{4a^2} - \frac{c}{a}};$$

(we have to assume $c/a < (b_1^2 + b_2^2)/(4a^2)$).

▲

(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:

$$a\mathbf{x}_i^T\mathbf{x}_i + \mathbf{b}^T\mathbf{x}_i + c = 0 \quad i = 1, \dots, N, \qquad \left\|(a, b_1, b_2, c)^T\right\|_2^2 = 1\,.$$

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 of (9-12.i):

The least squares linear system to be solved is just the collection of the equations (9.12.7) corresponding to every couple of data (x_i, y_i); it reads:

$$\mathbf{A}\mathbf{v} = \begin{bmatrix} (x_1^2 + y_1^2) & x_1 & y_1 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ (x_N^2 + y_N^2) & x_N & y_N & 1 \end{bmatrix} \begin{bmatrix} a \\ b_1 \\ b_2 \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}.$$

Since it is homogeneous (the right-hand side is $\mathbf{0}$), our constrained least squares problem is

$$\mathbf{v}^* = \underset{\mathbf{v}\in\mathbb{R}^4,\ \|\mathbf{v}\|^2=1}{\operatorname{argmin}} \|\mathbf{A}\mathbf{v}\|^2,$$

that is a minimization problem on the unit 4-dimensional sphere.

The solution is analogous to the one provided by [Lecture → Eq. (3.4.4.3)]: $\mathbf{v} = \mathbf{V}_{:,4}$, that is, $\mathbf{v}$ is the last column of the third term in the SVD decomposition $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.

This is implemented in the following code listing.

▲

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 of (9-12.j):

The centers of the circles computed with linear L.S., Gauss-Newton method, Newton method and non-linear constrained minimization are

```
(3.060303565727350, 0.743607321042323)
(3.043238251582548, 0.745678313458607)
(3.043238251582542, 0.745678313458603)
(3.029250408263415, 0.731531712909173)
```

and their radii are:

```
4.109137036074779
4.105855837954330
4.105855837954332
4.137744484439090 .
```

As expected, the two procedures that solve the same formulation of the problem (Gauss-Newton and Newton) provide the same results up to machine precision.

▲

End Problem 9-12

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

$$\dot{\mathbf{Y}} = \mathbf{A}\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad \text{with} \quad \mathbf{A} \in \mathbb{R}^{n\times n}. \tag{12.1.1}$$

whose solutions are *matrix-valued functions* $\mathbf{Y} : \mathbb{R} \to \mathbb{R}^{n\times n}$.

(12-1.a) (10 min.) Show that for *skew-symmetric* $\mathbf{A}$, i.e. $\mathbf{A} = -\mathbf{A}^\top$ we have:

$$\mathbf{Y}(0) \text{ orthogonal} \quad \Longrightarrow \quad \mathbf{Y}(t) \text{ orthogonal} \quad \forall t\,.$$

HINT 1 for (12-1.a): Remember from [Lecture → Def. 6.2.1.2] what property distinguishes an orthogonal matrix. Thus you see that the assertion we want to verify boils down to showing that the bilinear expression $t \mapsto \mathbf{Y}(t)^\top \mathbf{Y}(t)$ does not vary along trajectories, that is, its time derivative must vanish.

HINT 2 for (12-1.a): The fact that $\frac{d}{dt}\mathbf{Y}(t)^\top \mathbf{Y}(t) = \mathbf{O}$ can be established by means of the product rule [Lecture → Eq. (8.4.1.10)] and using the differential equation. Sloppily speaking, the derivative of a product of time-dependent matrices can be computed in the same way as the derivative of a product of functions.

SOLUTION of (12-1.a):

Let us consider the time derivative of $t \mapsto \mathbf{Y}(t)^\top \mathbf{Y}(t)$. We find

$$\begin{aligned}\frac{d}{dt}(\mathbf{Y}^\top\mathbf{Y}) &= \dot{\mathbf{Y}}^\top\mathbf{Y} + \mathbf{Y}^\top\dot{\mathbf{Y}} = (\mathbf{A}\mathbf{Y})^\top\mathbf{Y} + \mathbf{Y}^\top\mathbf{A}\mathbf{Y} \\ &= -\mathbf{Y}^\top\mathbf{A}\mathbf{Y} + \mathbf{Y}^\top\mathbf{A}\mathbf{Y} = 0\,.\end{aligned}$$

This implies $\mathbf{Y}(t)^\top\mathbf{Y}(t)$ is constant. From this, it follows that the orthogonality is preserved, as claimed. In fact $\mathbf{I} = \mathbf{Y}(0)^\top\mathbf{Y}(0) = \mathbf{Y}(t)^\top\mathbf{Y}(t)$ for all t.

▲

(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 eeulstep(const Eigen::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)

HINT 1 for (12-1.b): Since (12.1.1) is a linear ordinary differential equations, implicit timestepping schemes give rise to linear systems of equations, which can be solved using EIGEN's built-in elimination solvers. ⌟

SOLUTION of (12-1.b):

The explicit Euler method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}(t_k, \mathbf{Y}_k).$$

For the given differential equation we therefore obtain

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{A}\mathbf{Y}_k.$$

C++11-code 12.1.2: Explicit Euler method.

```
MatrixXd eeulstep(const MatrixXd &A, const MatrixXd &Y0, double h) {
  // TO DO: Implement ONE step of explicit euler applied to Y0,
  // for the ODE Y' = A*Y
  // START
  return Y0 + h * A * Y0;
  // END
}
```

The implicit Euler method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}(t_{k+1}, \mathbf{Y}_{k+1}).$$

For the given differential equation this yields

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{A}\mathbf{Y}_{k+1} \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = (\mathbf{I} - h\mathbf{A})^{-1}\mathbf{Y}_k$$

Hence, every step of the single-step method entails solving a linear system of equations, which can be done using EIGEN's direct elimination solvers introduced in [Lecture → § 2.5.0.8].

C++11-code 12.1.3: Implicit Euler method.

```
MatrixXd ieulstep(const MatrixXd &A, const MatrixXd &Y0, double h) {
  // TO DO: Implement ONE step of implicit euler applied to Y0,
  // for the ODE Y' = A*Y
  // START
  return (MatrixXd::Identity(3, 3) - h * A).partialPivLu().solve(Y0);
  // END
}
```

Finally, the implicit mid-point method is given by

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + h\mathbf{f}\big(\tfrac{1}{2}(t_k + t_{k+1}), \tfrac{1}{2}(\mathbf{Y}_k + \mathbf{Y}_{k+1})\big),$$

hence

$$\mathbf{Y}_{k+1} = \mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}(\mathbf{Y}_k + \mathbf{Y}_{k+1}) \quad \Longrightarrow \quad \mathbf{Y}_{k+1} = \big(\mathbf{I} - \tfrac{h}{2}\mathbf{A}\big)^{-1}\big(\mathbf{Y}_k + \tfrac{h}{2}\mathbf{A}\mathbf{Y}_k\big).$$

C++11-code 12.1.4: Implicit midpoint method.

```
MatrixXd impstep(const MatrixXd &A, const MatrixXd &Y0, double h) {
  // TO DO: Implement ONE step of implicit midpoint rule applied to Y0,
  // for the ODE Y' = A*Y
  // START
  return (MatrixXd::Identity(3, 3) - h * 0.5 * A)
      .partialPivLu()
      .solve(Y0 + h * 0.5 * A * Y0);
  // END
}
```

!

The method `PartialPivLu()` can also be replaced with the generic call `lu()`.

▲

(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 ($\rightarrow$ [Lecture $\rightarrow$ 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 of (12-1.c):

C++11-code 12.1.5: Orthogonality testing code.

```
std::tuple<double, double, double> checkOrthogonality(void) {
  // TO DO (12-1.c): compute and tabulate the Frobenius norms of Y_k'*Y_k - I
  // for 20 steps of eeulstep, ieulstep and impstep.
  // Return the values corresponding to the last step.
  // START
  unsigned int n = 3;
  double h = 0.01;
  MatrixXd M(n, n);
  M << 8, 1, 6, 3, 5, 7, 9, 9, 2;

  // Build Q
  HouseholderQR<MatrixXd> qr(n, n);
  qr.compute(M);
  MatrixXd Q = qr.householderQ();

  // Build A
  MatrixXd A(n, n);
  A << 0, 1, 1, -1, 0, 1, -1, -1, 0;
  MatrixXd I = MatrixXd::Identity(n, n);

  MatrixXd Meeul = Q, Mieul = Q, Mimp = Q;

  std::vector<int> sep = {5, 15};
  std::cout << std::setw(sep[0]) << "step" << std::setw(sep[1]) << "exp. Eul"
            << std::setw(sep[1]) << "imp. Eul" << std::setw(sep[1]) << "Mid-Pt"
            << std::endl;
  // Norm of Y'Y-I for initial value
  std::cout << std::setw(sep[0]) << "-1" << std::setw(sep[1])
            << (Meeul.transpose() * Meeul - I).norm() << std::setw(sep[1])
            << (Mieul.transpose() * Mieul - I).norm() << std::setw(sep[1])
            << (Mimp.transpose() * Mimp - I).norm() << std::endl;

  std::vector<double> norms(3);
  // Norm of Y'Y-I for 20 steps
  for (unsigned int j = 0; j < 20; ++j) {
    Meeul = eeulstep(A, Meeul, h);
    Mieul = ieulstep(A, Mieul, h);
    Mimp = impstep(A, Mimp, h);

    norms[0] = (Meeul.transpose() * Meeul - I).norm();
    norms[1] = (Mieul.transpose() * Mieul - I).norm();
    norms[2] = (Mimp.transpose() * Mimp - I).norm();

    std::cout << std::setw(sep[0]) << j << std::setw(sep[1]) << norms[0]
              << std::setw(sep[1]) << norms[1] << std::setw(sep[1]) << norms[2]
              << std::endl;
  }
  return std::make_tuple(norms[0], norms[1], norms[2]);
  // END
}
```

The following table records the Frobenius norm of $\mathbf{Y}_k^\top \mathbf{Y}_k - \mathbf{I}$ for the three different single-step method investigated in this problem:

step	exp. Eul	imp. Eul	IMP
-1	3.42194e-16	3.42194e-16	3.42194e-16
0	0.000424264	0.000424137	6.71891e-16
1	0.000848655	0.000848146	1.01146e-15
2	0.00127317	0.00127203	1.56074e-15
3	0.00169782	0.00169578	2.00648e-15
4	0.00212259	0.00211941	2.50877e-15
5	0.00254749	0.00254291	3.07073e-15
6	0.00297252	0.00296629	3.60267e-15
7	0.00339768	0.00338954	3.96759e-15
8	0.00382296	0.00381266	4.43968e-15
9	0.00424837	0.00423565	5.09343e-15
10	0.00467391	0.00465852	5.46397e-15
11	0.00509958	0.00508125	5.88117e-15
12	0.00552537	0.00550387	6.35815e-15
13	0.00595129	0.00592635	6.73769e-15
14	0.00637734	0.00634871	7.36616e-15
15	0.00680352	0.00677095	7.89645e-15
16	0.00722983	0.00719305	8.40091e-15
17	0.00765626	0.00761503	8.81256e-15
18	0.00808282	0.00803688	9.42399e-15
19	0.00850951	0.00845861	1.00449e-14

Orthogonality is preserved only by the implicit midpoint rule. Explicit and implicit Euler methods do not preserve orthogonality.

▲

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 $\mathfrak{A} \in \mathbb{R}^{s,s}$ and the vector $\mathbf{b} \in \mathbb{R}^s$ from the Butcher scheme are passed to the constructor, whereas the right-hand side vector field $\mathbf{f}$, the final time $T > 0$, the number N of *equidistant timesteps*, and the intial state $\mathbf{y}_0$ are arguments of the `solve()` method, which is supposed to return the sequence $(\mathbf{y}_k)_{k=0}^N$ of states generated by the explicit Runge-Kutta method.

SOLUTION of (12-2.a):

There are many ways to implement the formulas of [Lecture → Def. 11.4.0.9]. Thanks to the restriction to explicit RK-SSM all merely involve $\mathbf{f}$-evaluations.

C++11-code 12.2.1: Solution of (12-2.a).

```
template <class State> class RKIntegrator {
public:
  /*!
   *! \brief Constructor for the RK method.
   *! Performs size checks and copies A and b into internal storage.
   *! \param[in] A Matrix containing coefficents of the Butcher tableau,
   *! must be (strictly) lower triangular (no check is done).
   *! \param[in] b Vector containing coefficients of lower
   *! part of Butcher tableau.
   */
  RKIntegrator(const MatrixXd &A, const VectorXd &b)
      // TO DO: Use an initializer list to Initialize the private
      // variables A, b, s.
      // START
      : A(A), b(b), s(b.size())
  // END
```

```
  {
    // TO DO: Use assert() to make sure A is a square matrix of
    // shape s*s with s=b.size().
    // START
    assert(A.cols() == A.rows() && "Matrix must be square.");
    assert(A.cols() == b.size() && "Incompatible matrix/vector size.");
    // END
  }

  /*!
   *! \brief Perform the solution of the ODE.
   *! Solve an autonomous ODE y' = f(y), y(0) = y0, using a
   *! RK scheme given in the Butcher tableau provided in the
   *! constructor. Performs N equidistant steps up to time
   *! T with initial data y_0.
   *! \tparam Function type for function implementing the rhs function.
   *! Must have State operator()(const State & x).
   *! \param[in] f function handle for rhs in y' = f(y), e.g. \
   *! implemented using lambda funciton.
   *! \param[in] T The final time T.
   *! \param[in] y_0 Initial data y(0) = y_0 for y' = f(y).
   *! \param[in] N Number of steps to perform. Step size is h = T/N.
   *! Steps are equidistant.
   *! \return The vector containing all steps y^n (for each n)
   *! including initial and final value.
   */
  template <class Function>
  std::vector<State> solve(Function &&f, double T, const State &y0,
                           unsigned int N) const {
    std::vector<State> res;
    // Will contain all steps, reserve memory for efficiency
    res.reserve(N + 1);
    // TO DO: Compute the solution of y'=f(y), y(0)=y0 up to time T using
    // N Runge-Kutta steps, and store the solution at step n in res[n].
    // Use the private method step() for the Runge-Kutta step.
    // START
    // Step size
    double h = T / N;
    // Will contain all steps, reserve memory for efficiency
    res.reserve(N + 1);
    // Store initial data
    res.push_back(y0);

    // Initialize some memory to store temporary values
    State ytemp1 = y0;
    State ytemp2 = y0;
    // Pointers to swap previous value
    State *yold = &ytemp1;
    State *ynew = &ytemp2;

    // Loop over all fixed steps
    for (unsigned int k = 0; k < N; ++k) {
      // Compute, save and swap next step
      step(f, h, *yold, *ynew);
      res.push_back(*ynew);
      std::swap(yold, ynew);
    }
    // END
    return res;
  }
```

```
private:
  /*!
   *! \brief Perform a single step of the RK method.
   *! Solve an autonomous ODE using an explicit Runge Kutta Method.
   *! Compute a single explicit RK step y^{n+1} = y_n + \sum...
   *! starting from value y_0 and storing next value in y_1.
   *! \tparam Function type for function implementing the rhs.
   *! Must have State operator()(State x)
   *! \param[in] f function handle for rhs f, s.t. y' = f(y)
   *! \param[in] h step size
   *! \param[in] y_0 initial state
   *! \param[out] y_1 next step y^{n+1} = y^n + ...
   */
  template <class Function>
  void step(const Function &f, double h, const State &y0, State &y1) const {
    // TO DO: Compute a single step of the explicit Runge-Kutta method
    // (defined by A, b, s) for y'=f(y) starting at State y0 and using
    // step size h. Store the result in the State y1.
    // START
    // create vector holding next value
    y1 = y0;
    // Reserve space for increments
    std::vector<State> k;
    k.reserve(s);

    // Loop over the size of RK
    for (unsigned int i = 0; i < s; ++i) {
      // Compute increments and save them to k
      State incr = y0;
      for (unsigned int j = 0; j < i; ++j) {
        incr += h * A(i, j) * k.at(j);
      }
      k.push_back(f(incr));
      y1 += h * b(i) * k.back();
    }
    // END
  }

  //! Matrix A in Butcher scheme
  const MatrixXd A;
  //! Vector b in Butcher scheme
  const VectorXd b;
  //! Size of Butcher matrix and vector A and b
  unsigned int s;
};
```

There is no real reason to introduce the method `step()`; everything can be implemented in `solve()`.

▲

(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]):

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t))y_1(t) \ , \tag{12.2.2}$$
$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2)y_2(t) \ , \tag{12.2.3}$$
$$\mathbf{y}(0) = [100, 5]^\top \ , \tag{12.2.4}$$

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]):

$$\begin{array}{c|ccc} 0 & 0 & & \\ \frac{1}{3} & \frac{1}{3} & 0 & \\ \frac{2}{3} & 0 & \frac{2}{3} & 0 \\ \hline & \frac{1}{4} & 0 & \frac{3}{4} \end{array} \tag{12.2.5}$$

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 of (12-2.b):

The main task is to prepare the inputs for the application of **RKIntegrator**. The right-hand-side function $\mathbf{f}$ can be passed as lambda function, see [Lecture → Section 0.2.3].

C++11-code 12.2.6: Solution of (12-2.b)

```
double RK3prey() {
  double conv_rate = 0;
  // Dimension of state space
  unsigned int d = 2;
  // Exact value y(10) at final time T = 10 (approximated)
  Eigen::VectorXd yex(d);
  yex << 0.319465882659820, 9.730809352326228;

  // TO DO: Use an RKIntegrator to solve the predator/prey IVP up to T=10,
  // as described in (12-2.b). Tabulate the error as the number of steps
  // increases N = 2^7, 2^8, ..., 2^14, and estimate the convergence rate.
  // HINT: You may use polyfit() to calculate the convergence rate.
  // START

  // Implementation of butcher scheme
  unsigned int s = 3;
  Eigen::MatrixXd A(s,s);
  Eigen::VectorXd b(s);
  A << 0,      0,      0,
       1./3.,  0,      0,
       0,      2./3.,  0;
  b << 1./4.,  0,      3./4.;

```

```
  // Initialize RK with Butcher scheme
  RKIntegrator<Eigen::VectorXd> RK(A,b);

  // Coefficients and handle for prey/predator model
  double alpha1 = 3.;
  double alpha2 = 2.;
  double beta1 = 0.1;
  double beta2 = 0.1;
  auto f = [&alpha1, &alpha2, &beta1, &beta2] (const Eigen::VectorXd & y) {
    auto temp = y;
    temp(0) *= alpha1 - beta1*y(1);
    temp(1) *= -alpha2 + beta2*y(0);
    return temp;
  };

  // Initial value for model
  Eigen::VectorXd y0(d);
  y0 << 100, 5;

  // Final time for model
  double T = 10.;

  // Array of number of steps (for convergence study)
  ArrayXd N(8);
  N << 128, 256, 512, 1024, 2048, 4096, 8192, 16384;
  ArrayXd Error(N.size());

  // Start convergence study
  std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::endl;
  for(unsigned int i = 0; i < N.size(); ++i) {
    auto res = RK.solve(f, T, y0, N[i]);
    double err = (res.back() - yex).norm();
    std::cout << std::setw(15) << N[i] << std::setw(15) << err << std::endl;
    Error[i] = err;
  }

  VectorXd coeffs = polyfit(N.log(),Error.log(),1);
  conv_rate = -coeffs(0);
  // END
  return conv_rate;
}
```

N	error norm	rate
128	8.9601	
256	0.60584	3.88651
512	0.0688263	3.13791
1024	0.00838567	3.03696
2048	0.00103847	3.01347
4096	0.000129287	3.00581
8192	1.61305e-05	3.00271
16384	2.01448e-06	3.00131

In the table beside the column "rate" gives a "local" estimate of the rate α obtained from the quotient of two successive error norms:

$$\mathrm{err}(\tfrac{1}{2}h) : \mathrm{err}(h) \approx 2^{-\alpha} \quad \Leftrightarrow \quad \alpha \approx \frac{\log \mathrm{err}(\tfrac{1}{2}h) - \log \mathrm{err}(h)}{\log 2} .$$

Obviously, the 3-stage RK-SSM given in (12.2.5) is of order 3.

▲

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 Ψ^h define the discrete evolution of an order p Runge-Kutta single step method for the autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$, $\mathbf{f} : D \subseteq \mathbb{R}^d \to \mathbb{R}^d$. We define a new discrete evolution operator:

$$\boxed{\tilde{\Psi}^h := \frac{1}{1-2^p}\left(\Psi^h - 2^p \cdot (\Psi^{h/2} \circ \Psi^{h/2})\right)}\,, \tag{12.3.1}$$

where $\circ$ denotes the composition of mappings.

(12-3.a) ⚀ (10 min.) If Ψ^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 of (12-3.a):

Writing the explicit/forward Euler discrete evolution as $\Psi^h(\mathbf{y}_0) := \mathbf{y}_0 + hf(\mathbf{y}_0)$ (and $p = 1$), we obtain:

$$\begin{aligned}
\tilde{\Psi}^h(\mathbf{y}_0) &= \frac{1}{1-2^p}\left(\mathbf{y}_0 + hf(\mathbf{y}_0) - 2^p \cdot (\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0) + \frac{h}{2}f(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) \\
&= \mathbf{y}_0 + \frac{1}{1-2^p}\left(h(1-2^{p-1})f(\mathbf{y}_0) - 2^{p-1}hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) \\
&= \mathbf{y}_0 + \left(hf(\mathbf{y}_0 + \frac{h}{2}f(\mathbf{y}_0))\right) .
\end{aligned}$$

This is the evolution operator of the **explicit midpoint method**, a 2-stage explicit Runge-Kutta method, see [Lecture → Eq. (11.4.0.6)].

▲

(12-3.b) ⚀ (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 $\tilde{\Psi}^h\mathbf{y}_0$ when given the underlying Ψ through the functor `Psi`. Objects of type **DiscEvlOp** must provide an evaluation operator:

```
Eigen::VectorXd operator()(double h, const Eigen::VectorXd &y);
```

providing the result of $\Psi^h(\mathbf{y})$. For instance, suitable C++ lambda functions satisfy this requirement, see [Lecture → Section 0.2.3].

SOLUTION of (12-3.b):

The formula (12.3.1) can be implemented directly:

C++ code 12.3.2: Solution of (12-3.b).

```
template <class DiscEvlOp>
Vector psitilde(DiscEvlOp &&Psi, unsigned int p, double h, const Vector &y0) {
  // TO DO (12-3.b): apply the evolution operator \tilde{\Psi}
  // with step-size h to the value y0
  // START
  return ( Psi(h,y0) - (2<<(p-1))*Psi(h/2., Psi(h/2.,y0)) ) / (1. - (2<<(p-1)));
  // END
}
```

▲

(12-3.c) ⚀ (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 Ψ passed through the functor object `Psi`. The function should return the sequence $(\mathbf{y}_k)_{k=0}^N$ produced by the single-step method when started with initial value $\mathbf{y}_0$ (given as argument `y0`).

HINT 1 for (12-3.c): You can use **std::vector**`<Vector>::push_back()` to add elements to the end of a **std::vector**. ⌟

SOLUTION of (12-3.c):

Requested is a straightforward implementation of [Lecture → Eq. (11.3.1.6)].

C++ code 12.3.3: Solution of Sub-problem (12-3.c).

```
template <class DiscEvlOp>
std::vector<Vector> odeintequi(DiscEvlOp &&Psi, double T, const Vector &y0,
    unsigned int N) {
  std::vector<Vector> Y;
  // TO DO (12-3.c): Compute y from time 0 to T using N equidistant time steps
  // return a std::vector containing all steps y_0,...,y_N
  // START
  double h = T / N;
  double t = 0.;
  Y.reserve(N+1);
  Y.push_back(y0);
  Vector y = y0;

  while( t < T ) {
```

```
    // append new value
    y = Psi(h,Y.back());
    Y.push_back(y);
    // time stepping
    t += std::min(T-t,h);
  }
  // END
  return Y;
}
```

▲

(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)

$$\dot{y} = 1 + y^2 \quad , \quad y(0) = 0 , \tag{12.3.4}$$

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 Ψ 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 uniforms steps with $N = 2^q, q = 2, \dots, 12$, also returns an estimate of the rate of algebraic convergence based on linear regression.

HINT 1 for (12-3.d): The solution of the IVP Eq. (12.3.4) is $\tan(t)$. ⌟

SOLUTION of (12-3.d):

C++ code 12.3.5: Sub-problem (12-3.d): Implementation of `testcvpExtrapolatedEuler()`

```
double testcvpExtrapolatedEuler(void) {

  double conv_rate;
  double T = 1.;
  Vector y0 = Vector::Zero(1);
  auto f = [] (const Vector &y) -> Vector { return Vector::Ones(1) + y*y; };
  // TO DO (12-3.d): tabulate the values of the error corresponding to
  // \tilde{\psi}, where \psi is the explicit Euler method.
  // return the empirical convergence rate using polyfit.
  // Hint: first define a lambda for \psi. Then use psitilde to obtain a
  // suitable input for odeintequi.

  // START
  auto Psi = [&f] (double h, const Vector & y0) -> Vector {
    return y0 + h*f(y0);
  };
  unsigned p = 1;
  // lambda corresponding to \tilde{\psi}
  auto PsiTilde = [&Psi, &f, &p] (double h, const Vector & y0) -> Vector {
    return psitilde(Psi, p, h, y0);
  };
```

```
  // exact value
  Vector y_ex1(1);
  y_ex1(0) = std::tan(T);
  // values for convergence study
  Eigen::ArrayXd err(11);
  Eigen::ArrayXd N(11);

  std::cout << "Error table for equidistant steps:" << std::endl;
  std::cout << "N" << "\t" << "Error" << std::endl;
  for(int i = 0; i < 11; ++i ) {
    N(i) = std::pow(2, i + 2);
    Vector yT = odeintequi(PsiTilde, T, y0, N(i)).back();
    err(i) = (yT - y_ex1).norm();
    std::cout << N(i) << "\t" << err(i) << std::endl;
  }
  // compute fitted rate
  Vector coeffs = polyfit(N.log(),err.log(),1);
  conv_rate = -coeffs(0);
  // END
  return conv_rate;
}
```

N	$\lvert y_N(1) - y_{ex}(1)\rvert$
4	0.0650468
8	0.0209801
16	0.00595508
32	0.00158261
64	0.000407501
128	0.000103355
256	2.60232e-05
512	6.52882e-06
1024	1.63508e-06
2048	4.09129e-07

The empiric convergence rate is 2 ($O(h^2)$), which is evidence that the single-step method $\tilde{\Psi}^h$ from (12.3.1) enjoys order 2.
Conversely, the order of the underlying forward Euler method is $p = 1$.
Hence, the construction from (12.3.1) manages to raise the order of the method by 1.

▲

(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 Ψ and $\tilde{\Psi}$, where Ψ is passed through the argument `Psi`.

Step rejection and stepsize correction and prediction as explained in [Lecture → Section 11.5] is to be employed as in [Lecture → Code 11.5.0.21]. The argument `T` supplies the final time, `y0` the initial state, `h0` an initial stepsize, `p` the order of the discrete evolution Ψ, `reltol` and `abstol` the respective tolerances, and `hmin` a minimal stepsize that will trigger premature termination.

The function should return a vector of tuples $(t_k, \mathbf{y}_k)$, $k = 0, \dots, M$, 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 $(\mathbf{y}_k)_k$ the sequence of states generated by the single-step method.

HINT 1 for (12-3.e): Just adapt [Lecture → Code 11.5.0.21] using the discrete evolutions Ψ and $\widetilde{\Psi}$ from (12.3.1) instead of the generic **Psilow** and **Psihigh**. ⌟

SOLUTION of (12-3.e):

C++ code 12.3.6: Solution of Sub-problem (12-3.e).

```
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) {
  std::vector<double> t;
  std::vector<Vector> Y;
  // TO DO (12-3.e): Compute y from time 0 to T with adaptive time step.
  // Display a warning if the tolerance cannot be met with minimum
  // step size hmin. return a pair of vectors containing the times and
  // the computed values.
  // START
  t.push_back(0.);
  Y.push_back(y0);
  Vector y = y0;

  double h = h0;
  while( t.back() < T && h > hmin ) {
    // psitilde as higher order step
    Vector y_high = psitilde(Psi, p, std::min(T-t.back(),h), y);
    Vector y_low = Psi(std::min(T-t.back(),h),y);
    // estimated error of current step
    double est = (y_high - y_low).norm();
    // effective tolerance
    double tol = std::max(reltol*y.norm(), abstol);

    h = h*std::max(0.5, std::min(2., std::pow(tol/est,1./(p + 1))));

    if( est < tol ) {
      // accept time step
      y = y_high;
      Y.push_back(y_high);
      t.push_back(t.back() + std::min(T-t.back(), h));
    }
  }
  if (h < hmin) {
    std::cerr << "Warning: Failure at t="
      << t.back()
      << ". Unable to meet integration tolerances without reducing the step size
         below the smallest value allowed ("<< hmin <<") at time t." << std::endl;
  }
  // END
  return std::make_pair(t,Y);
}
```

▲

(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: `h0` $= 1/100$, `reltol` $= 10e-4$, `abstol` $= 10e-6$, `hmin` $= 10e-5$. Finally, the function should use MATPLOTLIBCPP to plot the approximated solution and store the plot in the file `tangent.png`. Do not forget to add axis labels.

SOLUTION of (12-3.f):

C++ code 12.3.7: Implementation of `solveTangentIVP()`

```
void solveTangentIVP(void)
{
  auto f = [] (const Vector &y) -> Vector { return Vector::Ones(1) + y*y; };
  Vector y0 = Vector::Zero(1);
  // TO DO (12-3.f): run the adaptive integration algorithm and plot the
  // resulting values of y(t).
  // Hint: you might use a loop to convert a std::vector<Vector> into a
  // std::vector<double>, since each Vector has size 1

  // START
  double T = 1.5;
  unsigned p = 1;
  double h0 = 1./100.;

  auto Psi = [&f] (double h, const Vector & y0) -> Vector { return y0 + h*f(y0); };
  //run adaptive algoritm
  std::pair<std::vector<double>, std::vector<Vector>>  vec_pair
                = odeintssctrl(Psi, T, y0, h0, p, 10e-4, 10e-6, 10e-5);
  std::vector<double> t = vec_pair.first;
  std::vector<Vector> Y = vec_pair.second;
  // convert type for plot
  std::vector<double> y(Y.size());
  for (unsigned i = 0; i < Y.size(); ++i) {
    y[i] = Y[i](0);
  }

  // plotting results
  plt::figure();
  plt::plot(t, y, "+", {{"label", "approx IVP"}});
  // exact solution
  Vector x = Vector::LinSpaced(100,0.0,T);
  Vector exact = x.array().tan();
  plt::plot(x, exact, {{"label", "tangent"}}  );
  plt::legend();

  plt::title("Approximate vs exact solution");
  plt::xlabel("t");
  plt::ylabel("y");
  plt::grid("True");

  plt::savefig("./cx_out/tangent.png");
  // END
}
```

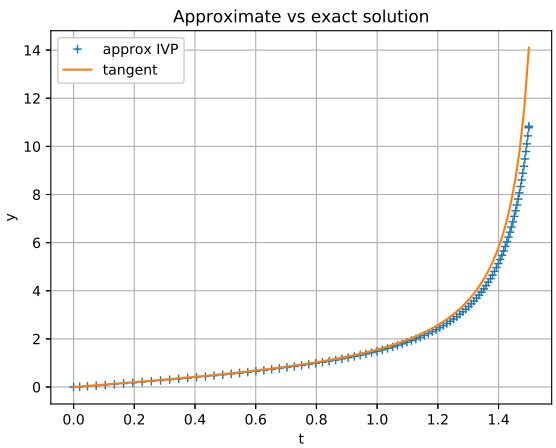


Fig. 59

◁ ◁ As in [Lecture → Ex. 11.5.0.4] the adaptive timestep control manages to detect the blow-up of the solution and reduces the timestep when approach the time $t_* = \pi/2$ from which the solution ceases to exist.

▲

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 $[0, T]$:

$$\begin{aligned} 2\ddot{u}_1 - \ddot{u}_2 &= u_1(u_2 + u_1)\,, \\ -\ddot{u}_{i-1} + 2\ddot{u}_i - \ddot{u}_{i+1} &= u_i(u_{i-1} + u_{i+1})\,, \qquad i = 2, \dots, n-1\,, \\ 2\ddot{u}_n - \ddot{u}_{n-1} &= u_n(u_n + u_{n-1})\,, \\ u_i(0) &= u_{0,i} \qquad i = 1, \dots, n\,, \\ \dot{u}_i(0) &= v_{0,i} \qquad i = 1, \dots, n\,. \end{aligned} \tag{12.4.1}$$

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 $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}),\ \mathbf{y}(0) = \mathbf{y}_0$.

HINT 1 for (12-4.a): What we face in (12.4.1) is an implicit ODE. First convert it into the form $\ddot{\mathbf{u}} = \mathbf{g}(\mathbf{u})$ for a suitable function $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^n$.

HINT 2 for (12-4.a): The function $\mathbf{g}$ introduced in the previous hint can be expressed by means of the inverse of a suitable tridiagonal matrix.

HINT 3 for (12-4.a): See [Lecture → Rem. 11.1.2.5] for a recipe on how to convert a second-order ODE into a first-order system.

SOLUTION of (12-4.a):

The second-order IVP can be rewritten as a first-order one by introducing the derivative of $\mathbf{u}$ as auxiliary vector-valued function $\mathbf{v}(t) := \dot{\mathbf{u}}(t)$

$$\begin{aligned} \dot{u}_i &= v_i \qquad i = 1, \dots, n\,, \\ 2\dot{v}_1 - \dot{v}_2 &= u_1(u_2 + u_1)\,, \\ -\dot{v}_{i-1} + 2\dot{v}_i - \dot{v}_{i+1} &= u_i(u_{i-1} + u_{i+1}) \qquad i = 2, \dots, n-1\,, \\ 2\dot{v}_n - \dot{v}_{n-1} &= u_n(u_n + u_{n-1})\,, \\ u_i(0) &= u_{0,i} \qquad i = 1, \dots, n\,, \\ v_i(0) &= v_{0,i} \qquad i = 1, \dots, n\,. \end{aligned}$$

The ODE system can then be written in the vector form

$$\dot{\mathbf{u}} = \mathbf{v}\,, \qquad \mathbf{C}\dot{\mathbf{v}} = \mathbf{r}(u_1, \dots, u_n) := \begin{bmatrix} u_1(u_2 + u_1) \\ \vdots \\ u_i(u_{i-1} + u_{i+1}) \\ \vdots \\ u_n(u_n + u_{n-1}) \end{bmatrix}\,,$$

where $\mathbf{C} \in \mathbb{R}^{n,n}$ is:

$$\mathbf{C} := \begin{bmatrix} 2 & -1 & 0 & \cdots & 0 \\ -1 & 2 & -1 & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & -1 & 2 & -1 \\ 0 & \cdots & 0 & -1 & 2 \end{bmatrix} .$$

In order to be able to use standard interfaces to RK-SSM for first-order ODEs, collect $\mathbf{u}$ and $\mathbf{v}$ in a $(2n)$-dimensional vector $\mathbf{y} := \left[\begin{smallmatrix}\mathbf{u}\\ \mathbf{v}\end{smallmatrix}\right]$. Then the final system of first-order ODEs reads

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_{n+1} \\ \vdots \\ y_{2n} \\ \mathbf{C}^{-1}\mathbf{r}(y_1,\dots,y_n) \end{bmatrix} .$$

▲

(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 $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$, $\mathbf{M} \in \mathbb{R}^{d,d}$.

HINT 1 for (12-4.b): The general formulas are given in [Lecture → Def. 11.4.0.9] and their connection with Butcher schemes is made explicit in [Lecture → Eq. (11.4.0.11)]. ⌟

SOLUTION of (12-4.b):

The task boils down to specializing the formulas of [Lecture → Def. 11.4.0.9] for $\mathbf{f}(\mathbf{y}) = \mathbf{M}\mathbf{y}$:

$$\begin{aligned} \mathbf{k}_1 &:= \mathbf{M}\mathbf{y}_n \,, \\ \mathbf{k}_2 &:= \mathbf{M}\left(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_1\right) , \\ \mathbf{k}_3 &:= \mathbf{M}\left(\mathbf{y}_n + \frac{h}{2}\mathbf{k}_2\right) , \\ \mathbf{k}_4 &:= \mathbf{M}(\mathbf{y}_n + h\mathbf{k}_3) \,, \\ \mathbf{y}_{n+1} &:= \mathbf{y}_n + \frac{h}{6}\mathbf{k}_1 + \frac{h}{3}\mathbf{k}_2 + \frac{h}{3}\mathbf{k}_3 + \frac{h}{6}\mathbf{k}_4 \,, \\ \mathbf{y}_{n+1} = \mathbf{\Psi}^h\mathbf{y}_n &:= \mathbf{y}_n + h\mathbf{M}\mathbf{y}_n + \frac{h^2}{2}\mathbf{M}^2\mathbf{y}_n + \frac{h^3}{6}\mathbf{M}^3\mathbf{y}_n + \frac{h^4}{24}\mathbf{M}^4\mathbf{y}_n \,, \end{aligned} \tag{12.4.2}$$

where h is the size of the timestep and $\mathbf{\Psi}^h$ stands for the discrete evolution operator underlying the single-step method, see [Lecture → Def. 11.3.1.5].

Remark. The solution of the ODE $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ can be written as $\mathbf{y}(t) = \exp(\mathbf{M}t)$ with the **matrix exponential**

$$\exp(\mathbf{X}) := \sum_{k=0}^{\infty} \frac{1}{k!}\mathbf{X}^k \,, \quad \mathbf{X} \in \mathbb{R}^{d,d} \,. \tag{12.4.3}$$

This means that $\mathbf{y}(t_{k+1}) = \exp(\mathbf{M}h)\mathbf{y}(t_k)$ and the evolution operator according to [Lecture → Def. 11.1.3.2] for $\dot{\mathbf{y}} = \mathbf{M}\mathbf{y}$ has the form $\mathbf{\Phi}^h\mathbf{y} = \exp(\mathbf{M}h)\mathbf{y}$. Comparing with the formula for $\mathbf{\Psi}^h$ we notice that $\mathbf{\Psi}^h$ can be obtained by truncating the power series for $\mathbf{\Phi}^h$ after the fifth term.

▲

(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`). `y0` contains the state before the step and the function has to return the state after the step in `y1`.

SOLUTION of (12-4.c):

Using the Butcher scheme from [Lecture → Ex. 11.4.0.15], the implementation is straightforward from [Lecture → Def. 11.4.0.9] and [Lecture → Eq. (11.4.0.11)].

C++ code 12.4.4: Solution of Sub-problem (12-4.c)

```
template <class Function, class State>
void rk4step(Function &&odefun, double h, const State &y0, State &y1) {
  // TO DO: (12-4.c) Implement a single step of RK4 for
  // the ODE y' = odefun(y), starting from y0 and using
  // the step size h. Save the result in y1.
  // START
  auto k1 = odefun(y0);
  auto k2 = odefun(y0 + h / 2 * k1);
  auto k3 = odefun(y0 + h / 2 * k2);
  auto k4 = odefun(y0 + h * k3);

  y1 = y0 + h / 6 * k1 + h / 3 * k2 + h / 3 * k3 + h / 6 * k4;
  // END
}
```

▲

(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

$$n = 5\,, \qquad u_i(0) = i/n\,, \qquad v_i(0) = -1\,, \qquad T = 1\,,$$

and tabulate the Euclidean norm of the error at final time $T = 1$ for numbers $N = 2, 4, 8, \ldots, 1024$ of uniform timesteps. As reference solution use the value obtained with $N = 2^{12}$ equidistant timesteps. The function should also return an estimate of the rate of algebraic convergence obtained by means of linear regression for which you can use the supplied function `polyfit()` from `polyfit.hpp`.

Note that you should use EIGEN's sparse matrix data type for any sparse matrix encountered. Refer to [Lecture → Section 2.7.2] for details.

SOLUTION of (12-4.d):

N	Error norm	approx. rate
2	0.249353	
4	0.0147421	4.08017
8	0.000783647	4.2336
16	4.19215e-05	4.22444
32	2.34881e-06	4.15769
64	1.375e-07	4.09442
128	8.29035e-09	4.05186
256	5.08553e-10	4.02696
512	3.15424e-11	4.01103
1024	2.05072e-12	3.94309

The rate estimate in the table is obtained based on the quotient of two subsequent error norms.
We observe algebraic convergence with rate 4: this is expected, since the function $\mathbf{f}$ is smooth and the classical RK4 single-step method is known to be of order 4.

Note the use of EIGEN's sparse matrix format and sparse elimination solvers as explained in [Lecture → Section 2.7.3].

Also note the use of a lambda function to pass the right-hand-side function to the generic implementation of the RK-SSM in `rk4step()`.

C++ code 12.4.5: Solution of Sub-problem (12-4.d)

```
double testcvgRK4() {
  // Parameters and initial values:
  double T = 1;
  int n = 5;
  VectorXd y0(2 * n);
  for (int i = 0; i < n; ++i) {
    y0(i) = (i + 1.) / n;
    y0(i + n) = -1;
  }

  // TO DO: (12-4.d) Implement the function f from (12-4.a) as a lambda
  // function, i.e. the right hand side of the system of first order ODEs.
  // START
  // Define the relevant tridiagonal matrix.
  Eigen::SparseMatrix<double> C(n, n);
  // Each column has at most 3 non-zero entries.
  C.reserve(VectorXi::Constant(n, 3));
  C.insert(0, 0) = 2;
  for (int i = 1; i < n; ++i) {
    C.insert(i, i) = 2;
    C.insert(i, i - 1) = -1;
    C.insert(i - 1, i) = -1;
  }
  C.makeCompressed();

  auto f = [n, C](VectorXd y) {
    // The system of ODEs is y' = f(y) with y = [u;v],
```

```
    // and f(y) = f([u;v]) = [v;C^{-1}g(u)]
    VectorXd fy(2 * n);
    fy.head(n) = y.tail(n);

    VectorXd g(n);
    g(0) = y(0) * (y(1) + y(0));
    g(n - 1) = y(n - 1) * (y(n - 1) + y(n - 2));
    for (int i = 1; i < n - 1; ++i) {
      g(i) = y(i) * (y(i - 1) + y(i + 1));
    }

    Eigen::SparseLU<Eigen::SparseMatrix<double>> Csolver;
    Csolver.compute(C);
    fy.tail(n) = Csolver.solve(g);
    return fy;
  };
  // END

  double conv_rate = 0;
  // Table header
  std::cout << std::setw(8) << "N" << std::setw(20) << "Error" << std::endl;

  // TO DO: (12-4.d) Tabulate the error at time T=1, using
  // N=2,4,...,1024 RK4 steps. Use N = 2^12 steps to calculate
  // the "exact" solution. Then, estimate the algebraic convergence
  // rate of the errors.
  // HINT: You can use polyfit() to calculate the convergence rate.
  // START

  // For reference, calculate solution using N = 2^12 steps.
  int N_exact = std::pow(2, 12);
  double h = T / N_exact; // step size
  VectorXd yT_exact = y0;
  for (int step = 0; step < N_exact; step++) {
    // yT_exact is the solution at time t=h*step
    VectorXd y_tmp;
    rk4step<std::function<VectorXd(VectorXd)>, VectorXd>(f, h, yT_exact, y_tmp);
    yT_exact = y_tmp;
  }

  // Calculate solution using N = 2,...,2^kmax steps.
  int kmax = 10;
  VectorXd Error(kmax);
  for (int k = 0; k < kmax; k++) {
    int N = std::pow(2, k + 1); // number of steps
    double h = T / N;           // step size
    VectorXd yT = y0;
    // Take N RK4 steps:
    for (int step = 0; step < N; step++) {
      // yT is the solution at time t=h*step
      VectorXd y_tmp;
      rk4step<std::function<VectorXd(VectorXd)>, VectorXd>(f, h, yT, y_tmp);
      yT = y_tmp;
    }
    Error(k) = (yT - yT_exact).norm();
    std::cout << std::setw(8) << N << std::setw(20) << Error(k) << std::endl;
  }

  // Estimate convergence rate
  // Get natural logarithm of N by log(N) = log(2)*log2(N).
```

```
  VectorXd logN = std::log(2) * VectorXd::LinSpaced(kmax, 1, kmax);
  VectorXd coeffs = polyfit(logN, Error.array().log(), 1);
  conv_rate = -coeffs(0);
  // END

  return conv_rate;
}
```

▲

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

$$\dot{\mathbf{Y}} = -(\mathbf{Y} - \mathbf{Y}^\top)\mathbf{Y} =: \mathbf{f}(\mathbf{Y}) \quad , \quad \mathbf{Y}(0) = \mathbf{Y}_0 \in \mathbb{R}^{n,n}, \tag{12.5.1}$$

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 `Y0`. Set the absolute tolerance to 10^{-10} and the relative tolerance to 10^{-8}. The output should be an approximation of $\mathbf{Y}(T) \in \mathbb{R}^{n\times n}$.

Instructions on the use of ode45, see [Lecture → 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** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

SOLUTION of (12-5.a):

The class **ode45** can take **Eigen::MatrixXd** as **StateType**. Alternatively, one can transform Matrices to Vectors (and viceversa), using the **Eigen::Map** facility, see [Lecture → Rem. 1.2.3.6].

C++11-code 12.5.2: Implementation of `matode()`.

```
MatrixXd matode(const MatrixXd &Y0, double T) {
  // TO DO (12-5.a): use the ode45 class to find an approximation
  // of the matrix IVP Y' = -(Y - Y')*Y at time T
  MatrixXd YT;
  // START
  // Define the RHS
  auto F = [] (const MatrixXd & M) {
      return -(M - M.transpose())*M;
  };
  ode45<MatrixXd> O(F);

  // Set tolerances
  O.options.atol = 10e-10;
  O.options.rtol = 10e-8;

  // Return only matrix at T, (solution is vector
  // of pairs (y(t_k),t_k) for each step k
```

```
  YT = O.solve(Y0, T).back().first;
  // END
  return YT;
}
```

▲

(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).

HINT 1 for (12-5.b): A time-dependent differentiable function is constant if and only if its derivative vanishes. ⌟

HINT 2 for (12-5.b): Use the product rule for differentiable matrix-valued functions $t \mapsto \mathbf{A}(t), \mathbf{B}(t)$

$$\frac{d}{dt}(\mathbf{A}(t)\mathbf{B}(t)) = \dot{\mathbf{A}}(t)\mathbf{B}(t) + \mathbf{A}(t)\dot{\mathbf{B}}(t)\,, \tag{12.5.3}$$

which is a special version of the general product rule [Lecture → Eq. (8.4.1.10)]. ⌟

SOLUTION of (12-5.b):

By the product rule and using the fact that $\mathbf{Y}$ is a solution of the IVP we obtain

$$\begin{aligned}\frac{d}{dt}(\mathbf{Y}^\top(t)\mathbf{Y}(t)) =& \frac{d}{dt}(\mathbf{Y}^\top(t))\mathbf{Y}(t) + \mathbf{Y}^\top(t)\frac{d}{dt}(\mathbf{Y}(t))\\ =&(-(\mathbf{Y}(t) - \mathbf{Y}^\top(t))\mathbf{Y}(t))^\top\mathbf{Y}(t) + \mathbf{Y}^\top(t)(-(\mathbf{Y}(t) - \mathbf{Y}^\top(t))\mathbf{Y}(t))\\ =& -\mathbf{Y}^\top(t)\mathbf{Y}^\top(t)\mathbf{Y}(t) + \mathbf{Y}^\top\mathbf{Y}(t)\mathbf{Y}(t)\\ & -\mathbf{Y}^\top\mathbf{Y}(t)\mathbf{Y}(t) + \mathbf{Y}^\top(t)\mathbf{Y}^\top(t)\mathbf{Y}(t) = 0.\end{aligned}$$

implying that the map is constant.

▲

(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 of (12-5.c):

Let $\mathbf{Y}_k$ be the output of `matode()`. We compute $\mathbf{Y}_0^\top\mathbf{Y}_0 - \mathbf{Y}_k^\top\mathbf{Y}_k$ and check if the (Frobenius) norm is smaller than a constant times the machine precision `eps`. Even for an orthogonal matrix, we have to take into account round-off errors!

C++11-code 12.5.4: Implementation checking invariant.

```
bool checkinvariant(const MatrixXd &M, double T) {
  // TO DO (12-5.c): check if Y'*Y is preserved at the time T by matode.
  // START
  MatrixXd N = matode(M, T);

  if( (N.transpose()*N-M.transpose()*M).norm() <
      10 * std::numeric_limits<double>::epsilon()* M.norm()) {
      return true;
  } else {
      return false;
  }
  // END
}
```

▲

(12-5.d) ⚁ (20 min.) [depends on Sub-problem (12-5.a)]

The so-called **discrete gradient method** for (12.5.1) reads

$$\mathbf{Y}_* = \mathbf{Y}_0 + \tfrac{1}{2}hf(\mathbf{Y}_0) \quad , \quad \mathbf{Y}_1 = (\mathbf{I} + \tfrac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*^\top))^{-1}(\mathbf{I} - \tfrac{1}{2}h(\mathbf{Y}_* - \mathbf{Y}_*^\top))\mathbf{Y}_0 \,. \tag{12.5.5}$$

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 of (12-5.d):

Apart from realizing that the formal inverse matrix in (12.5.5) must be implemented as the solution of a linear system of equations, the implementation is straightforward.

C++ code 12.5.6: Implementation of `cvgDiscreteGradientMethod()`

```
double cvgDiscreteGradientMethod(void) {
  // TO DO (12-5.d): compute the fitted convergence rate of the Discrete gradient
  // method. Also tabulate the values M and the errors.
  double conv_rate;
  // START
  double T = 1.0;
```

```
  // initial value
  MatrixXd Y0 = MatrixXd::Zero(5,5);
  Y0(4,0) = 1;
  for (unsigned int i = 0; i < 4; ++i) {
    Y0(i,i + 1) = 1;
  }
  // reference solution
  MatrixXd Y_ex = matode(Y0, T);

  // define the rhs
  auto F = [] (const MatrixXd & M) {
      return -(M - M.transpose())*M;
  };

  MatrixXd I = MatrixXd::Identity(5,5);
  ArrayXd MM(8);
  ArrayXd err(8);

  std::cout << "Error for equidistant steps:" << std::endl;
  std::cout << "M" << "\t" << "Error" << std::endl;
  for (unsigned int i = 0; i < 8; ++i) {
    unsigned int M = 10*std::pow(2,i);
    double h = T/M;
    MM(i) = M;
    MatrixXd Y = Y0;
    for (unsigned int j = 0; j < M; ++j ) {
      MatrixXd Ystar = Y + 0.5 * h * F(Y);

      MatrixXd Yinc = 0.5 * h * (Ystar - Ystar.transpose());
      Y = (I + Yinc).lu().solve((I - Yinc)*Y);
    }
    err(i) = (Y - Y_ex).norm();
    std::cout << M << "\t" << err(i) << std::endl;
  }

  // compute fitted rate
  VectorXd coeffs = polyfit(MM.log(),err.log(),1);
  conv_rate = -coeffs(0);
  // END
  return conv_rate;
}
```

M	$\|\mathbf{Y}_M - \mathbf{Y}(T)\|_F$
10	0.00260457
20	0.000645272
40	0.000160726
80	4.00747e-05
160	9.9619e-06
320	2.44364e-06
640	5.84909e-07
1280	1.98872e-07

◁ Doubling the number of timesteps leads to a reduction of the error by roughly a fector of four.

This hints at *algbraic convergence* in the number of timesteps with rate 2: $\mathrm{err} = \mathrm{O}(\mathrm{M}^{-2}) = \mathrm{O}(\mathrm{h}^2)$ for $M \to \infty$, where $h > 0$ is the size of the uniform timestep.

We conclude that the discrete gradient method is a single-step method *of order* 2.

▲

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, fi nal 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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0, \tag{12.6.1}$$

where $\mathbf{f}\colon \mathbb{R}^n \to \mathbb{R}^n$ and $\mathbf{y}_0 \in \mathbb{R}^n$. Using the class `RKIntegrate` 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 $\mathbf{f}$. The input `y0` passes the initial value $\mathbf{y}_0$.

For each k, the function should output the error at the final point $E_N = |\mathbf{y}_N(T) - \mathbf{y}_{2^{15}}(T)|$, $N = 2^k$, $k = 1, \dots, 12$, 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 of (12-6.a):

Let us find an expression for the order of convergence. Set $N_k = 2^k$. We readily derive

$$\frac{E_{N_{k-1}}}{E_{N_k}} \approx \frac{C2^{-(k-1)r_k}}{C2^{-kr_k}} = 2^{r_k}, \qquad r_k \approx \log\left(\frac{E_{N_{k-1}}}{E_{N_k}}\right) / \log(2).$$

A reasonable approximation of the order of convergence is given by

$$r \approx \frac{1}{\#K} \sum_{k \in K} r_k, \qquad K = \{k = 1, \dots, 15 : E_{N_k} > 5n \cdot 10^{-14}\}. \tag{12.6.2}$$

C++11-code 12.6.3: Implementation of `errors`.

```
template <class Function>
void errors(const Function &f, double T,
            const VectorXd &y0,
            const MatrixXd &A, const VectorXd &b) {

    RKIntegrator<VectorXd> rk(A, b);

    std::vector<double> error(15);
    std::vector<double> order(14);

    double sum = 0;
    int count = 0;
    bool test = 1;

    std::vector<VectorXd> y_exact = rk.solve(f, T, y0, pow(2,15));

    for(int k = 0; k < 15; k++) {
        int N = std::pow(2,k+1);
        std::vector<VectorXd> y1 = rk.solve(f,T,y0,N);

        error[k] = (y1[N] - y_exact[std::pow(2,15)]).norm();

        std::cout << std::left << std::setfill(' ')
                  << std::setw(3)  << "N = "
                  << std::setw(7) << N
                  << std::setw(8) << "Error = "
                  << std::setw(13) << error[k];

        if ( error[k] < y0.size() * 5e-14) {
            test = 0;
        }
        if ( k>0 && test ) {
            order[k-1]=std::log2( error[k-1] / error[k] );
            std::cout << std::left << std::setfill(' ')
                      << std::setw(10)
                      << "Approximated order = " << order[k-1]
                      << std::endl;
            sum += order[k-1];
            count = k;
        }
        else std::cout << std::endl;
    }
    std::cout << "Average approximated order = "
              << sum / count
              << std::endl << std::endl;
}
```

Get it on GitLab (order_not_all.cpp).

▲

(12-6.b) ⚃ (20 min.) Calculate the analytical solutions of the logistic ODE (see [Lecture → Ex. 11.1.1.1])

$$\dot{y} = (1-y)y, \quad y(0) = 1/2, \tag{12.6.4}$$

and of the initial value problem

$$\dot{y} = |1.1 - y| + 1, \quad y(0) = 1. \tag{12.6.5}$$

HINT 1 for (12-6.b): For a scalar autonomous ODE the family of solutions can be determined by direct integration

$$\dot{y} = f(y) \quad \Rightarrow \quad \int_0^t \frac{\dot{y}(\tau)}{f(y(\tau))}\,\mathrm{d}\tau = t \quad \Rightarrow \quad \int_{y(0)}^{y(t)} \frac{1}{f(\sigma)}\,\mathrm{d}\sigma = t\,. \tag{12.6.6}$$

Thus, if $t \mapsto G(t)$ is a principal of f^{-1}, that is $\dot{G} = f^{-1}$, then

$$G(y(t)) = G(y(0)) + t \quad \Rightarrow \quad y(t) = G^{-1}(G(y(0)) + t)\,, \tag{12.6.7}$$

provided that G is monotone in the possible range of values of y. ⌟

HINT 2 for (12-6.b): For the logistic ODE the solution is given in [Lecture → Eq. (11.1.1.3)]. ⌟

HINT 3 for (12-6.b): For (12.6.5) you have to distinguish cases depending on the sign of $1.1 - y(t)$. ⌟

SOLUTION of (12-6.b):

As far as (12.6.4) is concerned, the solution is $y(t) = (1+e^{-t})^{-1}$ (see [Lecture → Eq. (11.1.1.3)]).

Let us now consider (12.6.5). Because of the absolute value on the right hand side of the differential equation, we have to distinguish two cases $y(t) < 1.1$ and $y(t) > 1.1$. Since the initial condition is given by $y(0) = 1 < 1.1$, we start with the case $y(t) < 1.1$. For $y(t) < 1.1$, the differential equation is $\dot{y} = 2.1 - y$. Separation of variables

$$\int_1^{y(t)} \frac{1}{2.1-\tilde{y}}\,d\,\tilde{y} = \int_0^t d\,\tilde{t}$$

yields the solution

$$y(t) = 2.1 - 1.1e^{-t}, \quad \text{for } y(t) < 1.1.$$

For $y(t) > 1.1$, the differential equation is given by $\dot{y} = y - 0.1$ with initial condition $y(\ln(\frac{11}{10})) = 1.1$, where the initial time t^* was derived from the condition $y(t^*) = 2.1 - 1.1e^{-t^*} \overset{!}{=} 1.1$. Separation of variables yields the solution for this IVP

$$y(t) = \tfrac{10}{11}e^t + 0.1.$$

Together, the solution of the initial IVP is given by

$$y(t) = \begin{cases} 2.1 - 1.1e^{-t}, & \text{for } t \le \ln(1.1) \\ \frac{10}{11}e^t + 0.1, & \text{for } t > \ln(1.1). \end{cases}$$

▲

(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)]

$$\begin{array}{c|ccc} 0 & & & \\ 1/2 & 1/2 & & \\ 1 & -1 & 2 & \\ \hline & 1/6 & 2/3 & 1/6 \end{array}$$

- the classical RK method of order 4, see [Lecture → Ex. 11.4.0.15] 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 of (12-6.c):

Using the expression for the order of convergence given in (12.6.2) we find for (12.6.4):

- Eul: 1.06
- RK2: 2.00
- RK3: 2.84
- RK4: 4.01

This corresponds to the expected orders. However, in the case of the ODE (12.6.5) we obtain

- Eul: 1.09
- RK2: 1.93
- RK3: 1.94
- RK4: 1.99

The convergence orders of the explicit Euler and Runge–Kutta 2 methods are as expected, but we do not see any relevant improvement in the convergence orders of RK3 and RK4. This is due to the fact that the right hand side of the IVP is not continuously differentiable: the convergence theory breaks down.

▲

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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{12.7.1}$$

with smooth right hand side $\mathbf{f}\colon D \to \mathbb{R}^d$, where $D \subseteq \mathbb{R}^d$ is the state space. We take for granted that for all initial states, solutions exist for all times (global solutions, see [Lecture → Ass. 11.1.3.1]).

By its very definition given in [Lecture → Def. 11.1.3.2], the evolution operator

$$\mathbf{\Phi}\colon \mathbb{R} \times D \to D, \quad (t,\mathbf{y}) \mapsto \mathbf{\Phi}(t,\mathbf{y})$$

satisfies

$$\frac{\partial \mathbf{\Phi}}{\partial t}(t,\mathbf{y}) = \mathbf{f}(\mathbf{\Phi}(t,\mathbf{y})).$$

Next, we can differentiate this identity with respect to the state variable $\mathbf{y}$. We assume that all derivatives can be interchanged, which can be justified by rigorous arguments (which we won't do here). Thus, by the chain rule, we obtain, after swapping partial derivatives $\frac{\partial}{\partial t}$ and $\mathrm{D}_{\mathbf{y}}$,

$$\frac{\partial\, \mathrm{D}_{\mathbf{y}}\, \mathbf{\Phi}}{\partial t}(t,\mathbf{y}) = \mathrm{D}_{\mathbf{y}}\, \frac{\partial \mathbf{\Phi}}{\partial t}(t,\mathbf{y}) = \mathrm{D}_{\mathbf{y}}(\mathbf{f}(\mathbf{\Phi}(t,\mathbf{y}))) = \mathrm{D}\,\mathbf{f}(\mathbf{\Phi}(t,\mathbf{y}))\, \mathrm{D}_{\mathbf{y}}\, \mathbf{\Phi}(t,\mathbf{y}).$$

Abbreviating $\mathbf{W}(t,\mathbf{y}) := \mathrm{D}_{\mathbf{y}}\, \mathbf{\Phi}(t,\mathbf{y})$ we can rewrite this as the non-autonomous ODE

$$\boxed{\dot{\mathbf{W}} = \mathrm{D}\,\mathbf{f}(\mathbf{\Phi}(t,\mathbf{y}))\mathbf{W}} \;.. \tag{12.7.2}$$

Here, the state $\mathbf{y}$ can be regarded as a parameter. Since $\mathbf{\Phi}(0,\mathbf{y}) = \mathbf{y}$, 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

$$\frac{d}{dt}[\mathbf{y}(\cdot)\;,\; \mathbf{W}(\cdot,\mathbf{y}_0)] = [\mathbf{f}(\mathbf{y}(t))\;,\; \mathrm{D}\,\mathbf{f}(\mathbf{y}(t))\mathbf{W}(t,\mathbf{y}_0)]\;, \tag{12.7.3}$$

which is autonomous again.

Now let us apply (12.7.2)/(12.7.3). As in [Lecture → Ex. 11.1.1.5], we consider the following autonomous Lotka-Volterra differential equation of a predator-prey model

$$\begin{aligned} \dot{u} &= (2-v)u\;, \\ \dot{v} &= (u-1)v\;, \end{aligned} \tag{12.7.4}$$

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 of (12-7.a):

By construction, $\mathbf{\Phi}$ is a function with values in $\mathbb{R}^d$. Thus, $\mathrm{D}_{\mathbf{y}}\mathbf{\Phi}$ has values in $\mathbb{R}^{d,d}$, and so the state space of (12.7.2) is $\mathbb{R}^{d,d}$, a space of matrices.

▲

(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 of (12-7.b):

Writing $\mathbf{y} = [u, v]^T$, the map $\mathbf{f}$ associated to (12.7.4) is $\mathbf{f}(\mathbf{y}) = [(2-v)u, (u-1)v]^T$. Therefore

$$\mathrm{D}\,\mathbf{f}(\mathbf{y}) = \begin{bmatrix} 2-v & -u \\ v & u-1 \end{bmatrix}.$$

Thus, (12.7.4) becomes

$$\dot{\mathbf{W}} = \begin{bmatrix} 2-v(t) & -u(t) \\ v(t) & u(t)-1 \end{bmatrix} \mathbf{W}. \tag{12.7.5}$$

This is a non-autonomous ODE.

▲

(12-7.c) ⚂ (10 min.) From now on we write $\mathbf{\Phi} : \mathbb{R} \times \mathbb{R}^2_+ \to \mathbb{R}^2_+$ for the evolution operator (→ [Lecture → Def. 11.1.3.2]) associated with (12.7.4). Based on $\mathbf{\Phi}$ derive a non-trivial function $\mathbf{F} : \mathbb{R}^2_+ \to \mathbb{R}^2$, $\mathbf{F} \not\equiv \mathbf{0}$, which evaluates to zero for the input $\mathbf{y}_0$ if the period of the solution of system (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 of (12-7.c):

Let $\mathbf{F}$ be defined by

$$\mathbf{F}(\mathbf{y}) := \mathbf{\Phi}(T_P, \mathbf{y}) - \mathbf{y}.$$

If the solution of the system (12.7.4) with initial value $\mathbf{y}_0$ has period T_P, then we have $\mathbf{\Phi}(T_P, \mathbf{y}_0) = \mathbf{y}_0$, so $\mathbf{F}(\mathbf{y}_0) = 0$.

▲

(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 of (12-7.d):

By definition of $\mathbf{W}$ we immediately obtain

$$\mathrm{D}\,\mathbf{F}(\mathbf{y}) = \mathbf{W}(T_P, \mathbf{y}) - \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix.

▲

(12-7.e) ⚁ (10 min.) [depends on Sub-problem (12-7.c)]

Argue, why the solution of $\mathbf{F}(\mathbf{y}) = 0$ will, in gneneral, not be unique. When will it be unique?

HINT 1 for (12-7.e): Study [Lecture → § 11.1.2.3] again. Also take a look at [Lecture → Fig. 361]. ⌟

SOLUTION of (12-7.e):

If $\mathbf{y}_0 \in \mathbb{R}^2_+$ is a solution, then every state on the trajectory $\mathbf{y}_0$ will also be a solution. Only if the trajectory starting in $\mathbf{y}_0$ collapses to a point, that is, if $\mathbf{y}_0$ is a stationary point, $\mathbf{f}(\mathbf{y}_0) = 0$, and there is no other trajectory with the prescribed period, we can expect uniqueness.

▲

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 $\mathbf{f}$ as a functor object to the constructor

```
template <class StateType,
class RhsType = std::function<StateType(const StateType &)>>
class ode45 {
  public:
  ode45(const RhsType &rhs);
  // ......
}
```

   Template parameters are

- **StateType**: type of initial data and solution (state space), the only requirement is that the type possesses a normed vector-space structure, that is, it must implement the operations `+`, `*`, `*=`, `+=` and assignment/copy operators. Moreover a `norm()` method must be available. EIGEN's vector and matrix types, as well as fundamental types are eligible as **StateType**.
- **RhsType**: type of rhs function (automatically deduced).

The argument `rhs` must be of a functor type that provides an evaluation operator

```
StateType operator()(const StateType & vec);
```

It can also be a lambda function.

2. (optional) Set the integration options: set data members of the data structure **ode45.options** to configure the solver:

```
O.options.<option_you_want_to_set> = <value>;
```

Examples:

- `rtol`: relative tolerance for error control (default is `10e-6`)
- `atol`: absolute tolerance for error control (default is `10e-8`)

e.g.:

```
O.options.rtol = 10e-5;
```

3. Solve stage: invoke the single-step method through calling the method

```
template <class NormFunc = decltype(_norm<StateType>)>
std::vector<std::pair<StateType, double>>
solve(const StateType &y0, double T,
const NormFunc &norm = _norm<StateType>);
```

The type **NormType** should provide a norm for vectors of type **StateType**. However, this type can be deduced automatically and the argument `norm` is optional. The other arguments are

- `y0`: initial value of type **StateType** ($\mathbf{y}_0 =$ `y0`)
- `T`: final time of integration
- `norm`: (optional) norm function to call for objects of **StateType**, for the computation of the error

Return value The function returns the solution of the IVP, as a `std::vector` of `std::pair` $(y(t), t)$ for every snapshot.

For more explanations and details, please consult the in-class documentation provided in the comments.

(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 $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and $\mathbf{W}(T, [u_0, v_0]^T)$. The first component of the output pair should contain $\mathbf{\Phi}(T, [u_0, v_0]^T)$ and the second component the matrix $\mathbf{W}(T, [u_0, v_0]^T)$.

Set relative and absolute tolerances of **ode45** to 10^{-14} and 10^{-12}, respectively.

HINT 1 for (12-7.f): As in (12.7.3), both ODEs (for $\mathbf{\Phi}$ and $\mathbf{W}$) must be combined into a single autonomous differential equation on the state space $D \times \mathbb{R}^{d\times d}$. ⌟

SOLUTION of (12-7.f):

Writing $\mathbf{w} = [u, v, W_{11}, W_{21}, W_{12}, W_{22}]^T$, b y (12.7.4) and (12.7.5) we have that

$$\begin{aligned} \dot{w}_1 &= (2-w_2)w_1 & \dot{w}_3 &= (2-w_2)w_3 - w_1 w_4 & \dot{w}_5 &= (2-w_2)w_5 - w_1 w_6 \\ \dot{w}_2 &= (w_1-1)w_2 & \dot{w}_4 &= w_2 w_3 + (w_1-1)w_4 & \dot{w}_6 &= w_2 w_5 + (w_1-1)w_6 \end{aligned}$$

with initial conditions

$$w_1(0) = u_0, \quad w_2(0) = v_0, \quad w_3(0) = 1, \quad w_4(0) = 0, \quad w_5(0) = 0, \quad w_6(0) = 1,$$

since $\mathbf{W}(0,\mathbf{y}) = \mathrm{D}_{\mathbf{y}}\,\mathbf{\Phi}(0,\mathbf{y}) = \mathrm{D}_{\mathbf{y}}\,\mathbf{y} = \mathbf{I}$.

The class **ode45** has to be instantiated with the right lambda function describing the right-hand-side function of the compound ODE.

C++11-code 12.7.6: Example code.

```
std::pair<Vector2d, Matrix2d> PhiAndW(double u0,
                                      double v0,
                                      double T) {
  std::pair<Vector2d, Matrix2d> PaW;
  // TO DO: (12-7.f)  Calculate the time evolution of Phi(t,y0) and W(t,y0)
  // up to the final time T, using the inital value y0=[u0,v0].
  // Save the values of Phi and W at time T in PaW.first and
  // PaW.second respectively.
  // Hint: First write the ODE
  // [ y'(t), W'(t,y0) ] = [ f(y(t)), Df(y(t))W(t,y0) ]
  // in vectorized form, i.e. find a function g:R^6 -> R^6
  // such that the ODE w'=g(w) is equivalent.
  // Then use the ode45 class for the function g.
  // START
  auto f = [] (const VectorXd & w) {
    Eigen::VectorXd temp(6);
    temp(0) = (2. - w(1))*w(0);
    temp(1) = (w(0) - 1.)*w(1);
    temp(2) = (2. - w(1))*w(2) - w(0)*w(3);
    temp(3) = w(1)*w(2) + (w(0) - 1.)*w(3);
    temp(4) = (2. - w(1))*w(4) - w(0)*w(5);
    temp(5) = w(1)*w(4) + (w(0) - 1.)*w(5);
    return temp;
  };

  VectorXd w0(6);
  w0 << u0, v0, 1., 0, 0, 1.;

  // Construct ode solver with r.h.s
  ode45<VectorXd> O(f);
  // Set options
  O.options.rtol = 1e-14;
  O.options.atol = 1e-12;
  // Solve ODE
  auto sol = O.solve(w0, T);
  // Extract needed component
  VectorXd wT = sol.back().first;
```

```
  PaW.first  << wT(0), wT(1);
  PaW.second << wT(2), wT(4), wT(3), wT(5);
  // END
  return PaW;
}
```

▲

(12-7.g) (25 min.) [depends on Sub-problem (12-7.f) and Sub-problem (12-7.d)]

Relyiong 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 $\mathbf{F}(\mathbf{y}) = 0$ with $\mathbf{F}$. As your initial approximation, use $[3,2]^T$. Terminate the Newton iteration as soon as $|\mathbf{F}(\mathbf{y})| \leq 10^{-5}$. Validate your implementation by comparing the obtained initial data $\mathbf{y}$ with $\mathbf{\Phi}(100,\mathbf{y})$.

Remark. The residual based termination criterion recommended above [Lecture → § 8.1.2.4] 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].

HINT 1 for (12-7.g): The correct solutions are $u(0) \approx 3.110$ and $v(0) = 2.081$. ⌟

SOLUTION of (12-7.g):

Remember that the Newton iteration for $\mathbf{F}(\mathbf{y}) = 0$ reads:

$$\mathbf{y}^{(k+1)} = \mathbf{y}^{(k)} + \mathrm{D}\,\mathbf{F}(\mathbf{x}^{(k)})^{-1}\mathbf{F}(\mathbf{x}^{(k)}) \,. \tag{12.7.7}$$

The evaluation of both $\mathrm{D}\,\mathbf{F}$ and $\mathbf{F}$ for a given initial state is possible through the function `PhiAndW()`.

C++11-code 12.7.8: Example code.

```
Vector2d findInitCond(void){
  Vector2d y;
  // TO DO: (12-7.g) Use Newton's method with initial guess (3,2) to find a
  // zero of the function F from (12-7.c) for the period T_P = 5.
  // Store the resulting vector in y.
  // Hint: F and its Jacobian matrix can be computed from the result of PhiAndW().
  // START
  y << 3, 2; // Initial guess
  double T = 5; // Period we require

  // Compute Phi and W from initial guess
  std::pair<Vector2d, Matrix2d> PaW = PhiAndW(y(0), y(1), T);
  Vector2d F = PaW.first - y; // Value of F at initial guess
  Matrix2d DF; // Declare Jacobian of F

  // Until we are happy with our approximation
  while (F.norm() > 1e-5) { // Residual based termination
    // Calculate Jacobian
    DF = PaW.second - MatrixXd::Identity(2,2);
    // Use Newton iteration
```

```
    y = y - DF.lu().solve(F);
    // Test current guess
    PaW = PhiAndW(y(0), y(1), T);
    // Get value of F at the current guess
    F = PaW.first - y;
  }
  // END

  // TO DO: (12-7.g) Calculate the evolution of the Lotka-Volterra ODE up to
  // time 100, using the initial value y, found by the Newton iterations above.
  // If the evolution is indeed 5-periodic, then the solution at time 100
  // should have the value y. Print a warning message if this is not the case.
  // START
  PaW = PhiAndW(y(0), y(1), 100);
  if( (y - PaW.first).norm() > 1e-5) {
    std::cout << "Warning: Solution not periodic, y(100) != y(0)" << std::endl;
  }
  // END

  return y;
}
```

▲

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 have more options and this problem examines a class of single-step methods that also rely on $\mathrm{D}\,\mathbf{f}$.

The new class of methods is obtained by the following reasoning: if the right hand side $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$ of an autonomous initial value problem

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \,, \qquad \mathbf{y}(0) = \mathbf{y}_0 \,, \tag{12.8.1}$$

with solution $\mathbf{y} : \mathbb{R} \to \mathbb{R}^n$ is smooth, the solution $\mathbf{y}(t)$ will also be regular and it is possible to expand it into a Taylor sum at $t = 0$, see [Lecture → Thm. 8.2.2.12],

$$\mathbf{y}(t) = \sum_{n=0}^{m} \frac{\mathbf{y}^{(n)}(0)}{n!} t^n + R_m(t) \,, \tag{12.8.2}$$

with remainder term $R_m(t) = O(t^{m+1})$ for $t \to 0$.

A single step method for the numerical integration of (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 $\mathbf{Df}$.

HINT 1 for (12-8.a): Apply the chain rule, see [Lecture → § 8.4.1.8], taking into account the ODE (12.8.1) in order to substitute derivatives. ⌟

SOLUTION of (12-8.a):

For the first time derivative of $\mathbf{y}$, we just use the differential equation: $\frac{d\mathbf{y}}{dt}(t) = \mathbf{y}'(t) = \mathbf{f}(\mathbf{y}(t))$. For the second derivative, we use the previous equation, apply the chain rule and then once again insert the ODE:

$$\frac{d^2\mathbf{y}}{dt^2}(t) = \frac{d}{dt}\mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}'(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)).$$

Here $\mathbf{Df}(\mathbf{y}(t))$ is the Jacobian of $\mathbf{f}$ evaluated at $\mathbf{y}(t)$.

▲

(12-8.b) ⚁ (15 min.) [depends on Sub-problem (12-8.a)]

Verify the formula

$$\frac{d^3\mathbf{y}}{dt^3}(0) = \mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{f}(\mathbf{y}_0),\mathbf{f}(\mathbf{y}_0)) + \mathbf{D}\mathbf{f}(\mathbf{y}_0)^2\mathbf{f}(\mathbf{y}_0)\ . \tag{12.8.3}$$

HINT 1 for (12-8.b): To gain confidence, it is advisable to consider the scalar case $d = 1$ first, where $f : \mathbb{R} \to \mathbb{R}$ is a real valued function. ⌟

HINT 2 for (12-8.b): This time we have to apply both the product rule [Lecture → Eq. (8.4.1.10)] and the chain rule [Lecture → Eq. (8.4.1.9)] to the expression derived in the previous sub-problem. ⌟

HINT 3 for (12-8.b): Refresh yourself on the abstract concept of a derivative discussed in [Lecture → § 8.4.1.5].

Relevant for the case $d > 1$ is the fact that the first derivative of $\mathbf{f}$ is a linear mapping $\mathbf{D}\mathbf{f}(\mathbf{y}_0) : \mathbb{R}^n \to \mathbb{R}^n$. This linear mapping is applied by multiplying the argument with the Jacobian of $\mathbf{f}$.

Similarly, the second derivative is a *bilinear* mapping $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0) : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$. The i-th component of $\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{v},\mathbf{v})$ is given by

$$\mathbf{D}^2\mathbf{f}(\mathbf{y}_0)(\mathbf{v},\mathbf{v})_i = \mathbf{v}^\top \mathbf{H}\mathbf{f}_i(\mathbf{y}_0)\mathbf{v},$$

where $\mathbf{H}\mathbf{f}_i(\mathbf{y}_0)$ is the Hessian of the i-th component of $\mathbf{f}$ evaluated at $\mathbf{y}_0$. ⌟

SOLUTION of (12-8.b):

We follow the hint and first have a look at the scalar case. Here, the Jacobian reduces to $f'(y(t))$. Thus, given the previous sub-problem, we have to calculate

$$\frac{d^3y}{dt^3}(t) = \frac{d}{dt}\left[f'(y(t))f(y(t))\right]\ .$$

Product rule and chain rule give

$$\frac{d}{dt}\left[f'(y(t))f(y(t))\right] = f''(y(t))y'(t)f(y(t)) + f'(y(t))f'(y(t))y'(t)\ .$$

Inserting the ODE $y'(t) = f(y(t))$ once again yields

$$\frac{d^3y}{dt^3}(t) = f''(y(t))f(y(t))^2 + f'(y(t))^2 f(y(t))\ .$$

This is already equivalent to (12.8.3). The first term is quadratic in $f(y(t))$ and involves the second derivative of f, whereas the second term involves the first derivative of f in quadratic form.

To understand the formula for higher dimensions, we verify it component-wise. For each component $y_i(t)$ we have a function $f_i : \mathbb{R}^n \to \mathbb{R}$.

For the first derivative of $y_i(t)$, this is straightforward: $y_i'(t) = f_i(\mathbf{y}(t))$. Thus, the second derivative is:

$$\begin{aligned} y_i''(t) &= \frac{d}{dt}[f_i(\mathbf{y}(t))] = \sum_{j=1}^n \partial_{y_j} f_i(\mathbf{y}(t))\ y_j'(t) \\ &= [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{y}'(t) = [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{f}(\mathbf{y}(t)) \end{aligned}$$

Building up all components gives us what we have already obtained in (12-8.a): $\mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t))$, with the Jacobian $\mathbf{Df}(\mathbf{y}(t))$ which contains the gradients of the components of f row-wise.

Now, we apply product rule to $y_i''(t)$ to obtain

$$y_i'''(t) = \left\{ \frac{d}{dt}[\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \right\} \cdot \mathbf{y}'(t) + [\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \cdot \mathbf{y}''(t) \,.$$

The second term of the sum again builds up to

$$\mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{y}''(t) = \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{Df}(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t)) = \mathbf{Df}(\mathbf{y}(t))^2 \cdot \mathbf{f}(\mathbf{y}(t)) \,.$$

For the first term, we write the scalar product of the two vectors as a sum and then interchange the order of derivatives. This is possible as long as functions are sufficiently differentiable:

$$\left\{ \frac{d}{dt}[\mathbf{grad}\, f_i(\mathbf{y}(t))]^\top \right\} \cdot \mathbf{y}'(t) = \sum_{j=1}^{n} \left[\partial_{y_j} \frac{d}{dt} f_i(\mathbf{y}(t)) \right] f_j(\mathbf{y}(t)) \,.$$

Now we apply the chain rule:

$$\begin{aligned}
\sum_{j=1}^{n} \left[\partial_{y_j} \frac{d}{dt} f_i(\mathbf{y}(t)) \right] f_j(\mathbf{y}(t)) &= \sum_{j=1}^{n} \left[\partial_{y_j} \sum_{l=1}^{n} \partial_{y_l} f_i(\mathbf{y}(t)) y_l'(t) \right] f_j(\mathbf{y}(t)) \\
&= \sum_{j,l=1}^{n} \left[\partial_{y_j} \partial_{y_l} f_i(\mathbf{y}(t)) \right] f_l(\mathbf{y}(t)) f_j(\mathbf{y}(t)) \\
&= \mathbf{f}(\mathbf{y}(t))^\top \cdot \mathbf{H} f_i(\mathbf{y}(t)) \cdot \mathbf{f}(\mathbf{y}(t))
\end{aligned}$$

This is the desired result.

▲

(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]:

$$\dot{y}_1(t) = (\alpha_1 - \beta_1 y_2(t)) y_1(t) \,, \tag{12.8.4}$$
$$\dot{y}_2(t) = (\beta_2 y_1(t) - \alpha_2) y_2(t) \,, \tag{12.8.5}$$
$$\mathbf{y}(0) = [100, 5]^\top \,. \tag{12.8.6}$$

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 $[0, T]$

HINT 1 for (12-8.c): Find a suitable way to pass the data for the derivatives of the r.h.s. function $\mathbf{f}$ to the `solve` function. You may modify the signature of `solve`. ⌟

SOLUTION of (12-8.c):

For our particular example, we have:

$$\mathbf{y} = \begin{bmatrix} u \\ v \end{bmatrix}, \qquad \mathbf{f}(\mathbf{y}) = \begin{bmatrix} (\alpha_1 - \beta_1 y_2) y_1 \\ -(\alpha_2 - \beta_2 y_1) y_2 \end{bmatrix},$$

$$\mathbf{Df}(\mathbf{y}) = \begin{bmatrix} \alpha_1 - \beta_1 y_2 & -\beta_1 y_1 \\ \beta_2 y_2 & -(\alpha_2 - \beta_2 y_1) \end{bmatrix},$$

$$\mathbf{H}f_1(\mathbf{y}) = \begin{bmatrix} 0 & -\beta_1 \\ -\beta_1 & 0 \end{bmatrix}, \qquad \mathbf{H}f_2(\mathbf{y}) = \begin{bmatrix} 0 & \beta_2 \\ \beta_2 & 0 \end{bmatrix}.$$

C++11-code 12.8.7: Solution.

```
template <class State>
class TaylorIntegrator {
public:
    /*!
     *! \brief Perform the solution of the ODE
     *! Solve an autonomous ODE y' = f(y), y(0) = y0,
     *! using a Taylor expansion method constructor. Performs N
     *! equidistant steps upto time T with initial data y_0
     *! \tparam Function type for function implementing the
     *! rhs function (and its derivatives).
     *! \param[in] 'odefun' function handle for rhs f and its derivatives
     *! \param[in] T final time T
     *! \param[in] y_0 initial data y(0) = y0 for y' = f(y)
     *! \param[in] N number of steps to perform. Step size is h = T/N.
     *! Steps are equidistant.
     *! \return vector containing all steps y^n (for each n) including
     *! initial and final value
     */
    template <class Function>
    std::vector<State> solve(const Function &odefun, double T,
                             const State & y0, unsigned int N) const {
        std::vector<State> res;
        // Iniz step size
        double h = T / N;

        // Will contain all steps, reserve memory for efficiency
        res.reserve(N);

        // Store initial data
        res.push_back(y0);

        // Initialize some memory to store temporary values
        State ytemp1 = y0;
        State ytemp2 = y0;
        // Pointers to swap previous value
        State * yold = &ytemp1;
        State * ynew = &ytemp2;

        // Loop over all fixed steps
        for(unsigned int k = 0; k < N; ++k) {
            // Compute, save and swap next step
            step(odefun, h, *yold, *ynew);
            res.push_back(*ynew);
            std::swap(yold, ynew);
```

```
        }
        return res;
    }

private:
    /*!
     *! \brief Perform a single step of the Taylor expansion for
     *! the solution of the autonomous ODE
     *! Compute a single explicit step y^{n+1} = y_n + \sum...
     *! starting from value y_0 and storing next value in y_1
     *! \tparam Function type for function implementing the rhs
     *! and its derivatives.
     *! \param[in] 'odefun' function handle for rhs f and the derivatives
     *! \param[in] h step size
     *! \param[in] y_0 initial state
     *! \param[out] y_1 next step y^{n+1} = y^n + ...
     */
    template <class Function>
    void step(const Function &odefun, double h,
              const State & y0, State & y1) const {
        // Compute values for Taylor expansion,
        // including Jacobian and Hessian matrix
        auto fy0 =  odefun.f(y0);
        auto dfy0fy0 = odefun.df(y0, fy0);
        auto df2y0fy0 = odefun.df(y0, dfy0fy0);
        auto d2fy0fy0 = odefun.d2f(y0, fy0);

        // Plug values into Taylor expansion for next step
        y1 = y0 + fy0*h + dfy0fy0*h*h/2. + (df2y0fy0 + d2fy0fy0)*h*h*h/6.;
    }
};
```

Get it on GitLab (taylorintegrator.hpp).

▲

(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 $h \to 0$. This rate should be determined by linear regression.

SOLUTION of (12-8.d):

We see cubic algebraic convergence: $O(h^3)$.

C++11-code 12.8.8: Solution of (12-8.d)

```
    // Dimension of state space
    unsigned int d = 2;
```

```
    // Final time for model
    double T = 10.;

    // Initial value for model
    Eigen::VectorXd y0(d);
    y0 << 100, 5;

    // Array of number of steps (for convergence study)
    std::vector<unsigned int> N = {128, 256, 512, 1024, 2048, 4096, 8192, 16384};

    // Exact value y(10) at final time T = 10 (approximated)
    Eigen::VectorXd yex(d);
    yex << 0.319465882659820, 9.730809352326228;

    // Structure for the evaluation of f, df and d2f (Hessian bilinear form)
    struct odefun {
        std::function<Eigen::VectorXd(const Eigen::VectorXd &)> f = [this] (const Eigen::VectorXd & y) {
            Eigen::VectorXd temp = y;
            temp(0) *= (alpha1 - beta1*y(1));
            temp(1) *= (beta2*y(0) - alpha2);
            return temp;
        };
        std::function<Eigen::VectorXd(const Eigen::VectorXd &, const Eigen::VectorXd &)> df = [this] (const Eigen::VectorXd & y, const Eigen::VectorXd & z) {
            Eigen::MatrixXd Df(2,2);
            Df << alpha1-beta1*y(1), -beta1*y(0),  beta2*y(1), -alpha2+beta2*y(0);
            Eigen::VectorXd temp = Df*z;
            return temp;
        };
        std::function<Eigen::VectorXd(const Eigen::VectorXd &, const Eigen::VectorXd &)> d2f = [this] (const Eigen::VectorXd & y, const Eigen::VectorXd & z) {
            Eigen::MatrixXd H1(2,2), H2(2,2);
            H1 << 0, -beta1, -beta1, 0;
            H2 << 0, beta2, beta2, 0;
            Eigen::VectorXd temp(2);
            temp(0) = z.transpose()*H1*z;
            temp(1) = z.transpose()*H2*z;
            return temp;
        };
        const double alpha1 = 3;
        const double alpha2 = 2;
        const double beta1 = 0.1;
        const double beta2 = 0.1;
    } F;

    // Constructor
    TaylorIntegrator<Eigen::VectorXd> tint;

    // Start convergence study
    std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::setw(15) << "rate" << std::endl;
    double errold;
    for(unsigned int i = 0; i < N.size(); ++i) {
        auto res = tint.solve(F, T, y0, N[i]);
        double err = (res.back() - yex).norm();
        std::cout << std::setw(15) << N[i] << std::setw(15) << err;
```

```
        if(i > 0) {
            std::cout << std::setw(15) << log2(errold / err);
        }
        errold = err;
        std::cout << std::endl;
    }
```

Get it on GitLab (taylorprey.cpp).

```
N      error        rate
128    6.49399
256    0.533867     3.60455
512    0.0607558    3.13539
1024   0.00732692   3.05174
2048   0.000900452  3.02449
4096   0.000111607  3.01222
8192   1.38916e-05  3.00614
16384  1.73275e-06  3.00308
```

▲

(12-8.e) (5 min.) What is the disadvantage of the Taylor's method compared with a Runge-Kutta method?

SOLUTION of (12-8.e):

The error of the studied Runge-Kutta method and Taylor's method are practically identical. The obvious disadvantage of Taylor's method compared to Runge-Kutta methods is that the former involves rather complicated higher derivatives of f. If we want higher orders, those formulas get even more complicated, whereas explicit Runge-Kutta methods work with only a few evaluations of f itself, yielding results which are comparable.

Moreover, the Taylor expansion method *cannot be applied to an ODE with* $\mathbf{f}$ *given in procedural form*.

▲

End Problem 12-8 , 50 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 of (13-1.a):

The solution is given in [Lecture → Rem. 12.3.3.4].

▲

(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 $F(\mathfrak{g}) = \mathbf{0}$ with a suitable $F : \mathbb{R}^{sd} \to \mathbb{R}^{sd}$. 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 of (13-1.b):

We following the developments of [Lecture → Rem. 12.3.3.7] and rewrite the stage equations as

$$F(\mathfrak{g}) = \mathbf{0} \quad \text{with} \quad F(\mathfrak{g}) := \mathfrak{g} - h(\mathfrak{A} \otimes \mathbf{I}) \begin{bmatrix} \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_1) \\ \vdots \\ \mathbf{f}(\mathbf{y}_0 + \mathbf{g}_s) \end{bmatrix} . \tag{13.1.1}$$

Then the corresponding Newton iteration reads:

$$\mathfrak{g}^{(k+1)} = \mathfrak{g}^{(k)} - DF(\mathfrak{g}^{(k)})^{-1} F(\mathfrak{g}^{(k)}) , \tag{13.1.2}$$

with Jacobian

$$DF(\mathfrak{g}^{(k)}) = \mathbf{I} - h \begin{bmatrix} a_{11} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_1) & \dots & a_{1s} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_1) \\ \vdots & & \vdots \\ a_{s1} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_s) & \dots & a_{ss} \dfrac{\partial \mathbf{f}}{\partial \mathbf{y}}(\mathbf{y}_0 + \mathbf{g}_s) \end{bmatrix} . \tag{13.1.3}$$

▲

(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 $\mathfrak{A} \in \mathbb{R}^{s,s}$ 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 $\mathrm{D}\,\mathbf{f}$ are passed through the functor objects `f` and `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 of (13-1.c):

We implement the method discussed in Sub-problem (13-1.b). The main task is to wrap the function F from (13.1.1) and its Jacobian (13.1.3) into suitable lambda functions and pass those to `dampnewton()`.

C++11-code 13.1.4: Implementation of implicit_RKIntegrator

```
/*!
  *! \brief Implements a Runge-Kutta implicit solver for a
  *! given Butcher tableau for autonomous ODEs.
  */
class implicit_RKIntegrator {
public:
    /*!
      *! \brief Constructor for the implicit RK method.
      *! Performs size checks and copies A and b into internal storage.
      *! \param[in] A Matrix containing coefficents of Butcher tableau.
      *! \param[in] b Vector containing coefficients of
      *! lower part of Butcher tableau.
    */
    implicit_RKIntegrator(const MatrixXd & A, const VectorXd & b)
        : A(A), b(b), s(b.size()) {
        assert( A.cols() == A.rows() && "Matrix must be square.");
        assert( A.cols() == b.size() && "Incompatible matrix/vector size.");
    }

    /*!
      *! \brief Perform the solution of the ODE.
      *! Solve an autonomous ODE y' = f(y), y(0) = y0, using an
      *! implicit RK scheme given in the Butcher tableau provided in the
      *! constructor. Performs N equidistant steps upto time T
      *! with initial data y0.
      *! \tparam Function type for function implementing the rhs function.
      *! Must have VectorXd operator() (VectorXd x)
      *! \tparam Function2 type for function implementing the Jacobian of f.
      *! Must have MatrixXd operator() (VectorXd x)
      *! \param[in] f function handle for rhs in y' = f(y), e.g.
      *! implemented using lambda funciton
      *! \param[in] Jf function handle for Jf, e.g.
      *! implemented using lambda funciton
      *! \param[in] T final time T
      *! \param[in] y0 initial data y(0) = y0 for y' = f(y)
      *! \param[in] N number of steps to perform.
      *! Step size is h = T / N. Steps are equidistant.
      *! \return vector containing all steps y^n (for each n)
```

```
      *! including initial and final value
      */
    template <class Function, class Jacobian>
    std::vector<VectorXd> solve(const Function &f, const Jacobian &Jf,
                                double T, const VectorXd & y0,
                                unsigned int N) const {
        // Iniz step size
        double h = T / N;

        // Will contain all steps, reserve memory for efficiency
        std::vector<VectorXd> res;
        res.reserve(N+1);

        // Store initial data
        res.push_back(y0);

        // Initialize some memory to store temporary values
        VectorXd ytemp1 = y0;
        VectorXd ytemp2 = y0;
        // Pointers to swap previous value
        VectorXd * yold = &ytemp1;
        VectorXd * ynew = &ytemp2;

        // Loop over all fixed steps
        for(unsigned int k = 0; k < N; ++k) {
            // Compute, save and swap next step
            step(f, Jf, h, *yold, *ynew);
            res.push_back(*ynew);
            std::swap(yold, ynew);
        }
        return res;
    }

private:
     /*!
      *! \brief Perform a single step of the RK method for the
      *! solution of the autonomous ODE
      *! Compute a single explicit RK step y^{n+1} = y_n + \sum ...
      *! starting from value y0 and storing next value in y1
      *! \tparam Function type for function implementing the rhs.
      *! Must have VectorXd operator()(VectorXd x)
      *! \tparam Jacobian type for function implementing the Jacobian of f.
      *! Must have MatrixXd operator()(VectorXd x)
      *! \param[in] f function handle for ths f, s.t. y' = f(y)
      *! \param[in] Jf function handle for Jf, e.g. implemented using lambda
         funciton
      *! \param[in] h step size
      *! \param[in] y0 initial VectorXd
      *! \param[out] y1 next step y^{n+1} = y^n + ...
      */
    template <class Function, class Jacobian>
    void step(const Function &f, const Jacobian &Jf,
              double h,
              const VectorXd & y0, VectorXd & y1) const {

        int d = y0.size();
        MatrixXd eye = MatrixXd::Identity(d,d);

        // Handle for the function F describing the
        // equation satisfied by the stages g
```

```
        auto F = [&y0, h, d, this, &f, &eye] (VectorXd gv) {
            VectorXd Fv = gv;
            for (int j = 0; j < s; j++) {
                Fv = Fv - h*kron(A.col(j), eye)*f(y0+gv.segment(j*d,d));
            }
            return Fv;
        };

        // Handle for the Jacobian of F.
        auto JF = [&y0, h, d, &Jf, this, &eye] (VectorXd gv) {
            MatrixXd DF(s*d,s*d);
            for (int j = 0; j < s; j++) {
                DF.block(0,j*d,s*d,d) = kron(A.col(j),
                    eye)*Jf(y0+gv.segment(j*d,d));
            }
            DF = MatrixXd::Identity(s*d,s*d) - h*DF;
            return DF;
        };

        // Obtain stages with damped Newton method
        VectorXd gv = VectorXd::Zero(s*d);
        dampnewton(F, JF, gv);

        // Calculate y1
        MatrixXd K(d,s);
        for (int j = 0; j < s; j++) K.col(j) = f(y0+gv.segment(j*d,d));
        y1 = y0 + h*K*b;
    }

    //<! Matrix A in Butcher scheme
    const MatrixXd A;
    //<! Vector b in Butcher scheme
    const VectorXd b;
    //<! Size of Butcher matrix and vector A and b
    unsigned int s;
};
/* SAM_LISTING_END_1 */
```

▲

(13-1.d) (15 min.) Examine the following code

C++11-code 13.1.5: Initialization of implicit_RKIntegrator object

```
// Definitiion of coefficients in Butcher scheme
unsigned int s = 2;
MatrixXd A(s, s);
VectorXd b(s);
// What method is this?
A << 5. / 12., -1. / 12., 3. / 4., 1. / 4.;
b << 3. / 4., 1. / 4.;
// Initialize implicit RK with Butcher scheme
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]?

HINT 1 for (13-1.d): Scan the particular implicit Runge-Kutta single step methods presented in [Lecture → Section 12.3].

▲

SOLUTION of (13-1.d):

The Butcher scheme used corresponds to the Radau RK-SSM of order 3 (see [Lecture → Ex. 12.3.4.18] for the complete scheme). Thus, the method is A-stable and L-stable.

(13-1.e) (15 min.) The file `main.cpp` uses you implementation **implicit_RKintegrator** of general implicit RK-SSMs to solve an initial value problem for the Lotka-Volterra ordinary differential equation

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \begin{bmatrix} y_1(\alpha_1 - \beta y_2) \\ y_2(-\alpha_2 + \beta y_2) \end{bmatrix} . \tag{13.1.6}$$

with the particular paramters $\alpha_1 = 3$, $\alpha_2 = 2$, $\beta = 0.1$. The right-hand-side function and its Jacobian are implemented as suitable lambda functions. Initial state is $\mathbf{y}_0 = [100\ 5]^\top$ and final time $T = 10$. The norm of the error at final time is tabulated.

Run the code. What information can be gleaned from the table?

SOLUTION of (13-1.e):

C++11-code 13.1.7: Use of implicit_RKIntegrator for solving an IVP for the Lotka-Volterra ODE

```
// Coefficients and handle for prey/predator model
double alpha1 = 3.;
double alpha2 = 2.;
double beta1 = 0.1;
double beta2 = 0.1;
auto f = [&alpha1, &alpha2, &beta1, &beta2](const VectorXd &y) {
  auto temp = y;
  temp(0) *= alpha1 - beta1 * y(1);
  temp(1) *= -alpha2 + beta2 * y(0);
  return temp;
};

auto Jf = [&alpha1, &alpha2, &beta1, &beta2](const VectorXd &y) {
  MatrixXd temp(2, 2);
  temp << alpha1 - beta1 * y(1), -beta1 * y(0), beta2 * y(1),
      -alpha2 + beta2 * y(0);
  return temp;
};

// Dimension of state space
unsigned int d = 2;

// Final time for model
double T = 10.;

```

```
// Initial value for model
VectorXd y0(d);
y0 << 100, 5;

// Array of number of steps (for convergence study)
std::vector<unsigned int> N = {128,  256,  512,   1024,  2048,
                               4096, 8192, 16384, 32768, 65536};

// Exact value y(10) at final time T = 10 (approximated)
VectorXd yex(d);
yex << 0.319465882659820, 9.730809352326228;

// Start convergence study
std::cout << std::setw(15) << "N" << std::setw(15) << "error" << std::setw(15)
          << "rate" << std::endl;
double errold;
for (unsigned int i = 0; i < N.size(); ++i) {
  auto res = RK.solve(f, Jf, T, y0, N[i]);
  double err = (res.back() - yex).norm();
  std::cout << std::setw(15) << N[i] << std::setw(15) << err;
  if (i > 0) {
    std::cout << std::setw(15) << log2(errold / err);
  }
  errold = err;
  std::cout << std::endl;
}
```

N	error norm
128	1.64554
256	0.184465
512	0.0237424
1024	0.00326135
2048	0.000503882
4096	5.79285e-05
8192	5.10432e-06
16384	6.37473e-07
32768	7.96485e-08
65536	9.95420e-09

◁ Norm of error at final time.

We observe an asymptotic (in the last few steps) decrease of the error norm by a factor of ≈ 8 whenever the number of (equidistant) timesteps is doubled. Thus we observe *algebraic convergence* with rate 3 in the timestep size h:

$$\text{errornorm} = O(h^3) \ .$$

We expect this because the 2-stage Radau single-step method is of order 3, see [Lecture → Ex. 12.3.4.18].

▲

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

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) := \mathbf{a} \times \mathbf{y} + c\mathbf{y} \times (\mathbf{a} \times \mathbf{y}), \quad \mathbf{y}(0) = \mathbf{y}_0 = [1,1,1]^\top , \tag{13.2.1}$$

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

$$\mathbf{x} \times \mathbf{y} = \begin{bmatrix} x_2y_3 - x_3y_2 \\ x_3y_1 - x_1y_3 \\ x_1y_2 - x_2y_1 \end{bmatrix} .$$

It satisfies $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$. In Eigen, it is available as `x.cross(y)` EIGEN documentation, **#include** `<Eigen/Geometry>` required.

(13-2.a) (10 min.) Show that $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}_0\|_2$ for every solution $\mathbf{y}$ of the ODE.

HINT 1 for (13-2.a): Examine the time derivative $\frac{d}{dt}\|\mathbf{y}(t)\|_2^2$ and use the product rule.

SOLUTION of (13-2.a):

Since $\mathbf{x} \times \mathbf{y} \perp \mathbf{x}$ for all $\mathbf{x}, \mathbf{y}$, we have

$$(\mathbf{a} \times \mathbf{y} + c(\mathbf{y} \times (\mathbf{a} \times \mathbf{y}))) \cdot \mathbf{y} = 0 \quad \Rightarrow \quad D_t\|\mathbf{y}(t)\|_2^2 = 2\dot{\mathbf{y}}(t) \cdot \mathbf{y}(t) = 0 \quad \Rightarrow \quad \|\mathbf{y}(t)\|_2^2 = \text{const.}\ \forall t .$$

Thus $\|\mathbf{y}(t)\|_2 = \|\mathbf{y}(0)\|_2 = \|\mathbf{y}_0\|_2$.

▲

(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 of (13-2.b):

The right-hand side function $\mathbf{f}$ in (13.2.1) is a quadratic polynomial in the components of $\mathbf{y}$, hence smooth and defined on all of BR^3. Based on the product rule [Lecture → Eq. (8.4.1.10)] and appealing to the bilinearity of the cross product, we find

$$\mathrm{D}\,\mathbf{f}(\mathbf{y})\mathbf{h} = \mathbf{a} \times \mathbf{h} + c\mathbf{h} \times (\mathbf{a} \times \mathbf{y}) + c\mathbf{y} \times (\mathbf{a} \times \mathbf{h}) .$$

The Jacobian $\mathrm{D}\,\mathbf{f}(\mathbf{y})$ is the matrix representation of this mapping $\mathbf{h} \mapsto \mathrm{D}\,\mathbf{f}(\mathbf{y})$. Using the definition of cross product, a simple but tedious calculation shows that

$$\mathrm{D}\,\mathbf{f}(\mathbf{y}) = \begin{bmatrix} -ca_2y_2 - ca_3y_3 & -a_3 + 2ca_1y_2 - ca_2y_1 & a_2 - ca_3y_1 + 2ca_1y_3 \\ a_3 - ca_1y_2 + 2ca_2y_1 & -ca_3y_3 - ca_1y_1 & -a_1 + 2ca_2y_3 - ca_3y_2 \\ -a_2 + 2ca_3y_1 - ca_1y_3 & a_1 - ca_2y_3 + 2ca_3y_2 & -ca_1y_1 - ca_2y_2 \end{bmatrix} .$$

Thus for $\mathbf{y} = \mathbf{a}$ we obtain

$$\begin{aligned} \mathrm{D}\,\mathbf{f}(\mathbf{a}) &= \begin{bmatrix} -c(a_2^2+a_3^2) & -a_3+ca_1a_2 & a_2+ca_3a_1 \\ a_3+ca_1a_2 & -c(a_1^2+a_3^2) & -a_1+ca_2a_3 \\ -a_2+ca_1a_3 & a_1+ca_2a_3 & -c(a_1^2+a_2^2) \end{bmatrix} \\ &= \begin{bmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{bmatrix} + c\mathbf{a}\mathbf{a}^\top - c\|\mathbf{a}\|^2\mathbf{I}\,. \end{aligned}$$

The next step is an exercise in advanced linear algebra: We see that $\mathrm{D}\,\mathbf{f}(\mathbf{a})\mathbf{a} = \mathbf{0}$ and on the orthogonal complement $\mathbf{a}^\perp$ the matrix acts like a rotation (scaled with $\|\mathbf{a}\|$) minus $c\|\mathbf{a}\|\mathbf{I}$. This reveals that the spectrum is given by

$$\sigma(\mathrm{D}\,\mathbf{f}(\mathbf{a})) = \{0, -c\|\mathbf{a}\|_2^2 \pm i\|\mathbf{a}\|_2\} = \{0, -c \pm i\}.$$

Applying the criteria [Lecture → Eq. (12.2.0.13)] and [Lecture → Eq. (12.2.0.14)] we find that at the state $\mathbf{y} = \mathbf{a}$ the ODE (13.2.1) is stiff for large $c > 0$ (see [Lecture → § 12.2.0.12]).

▲

(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.

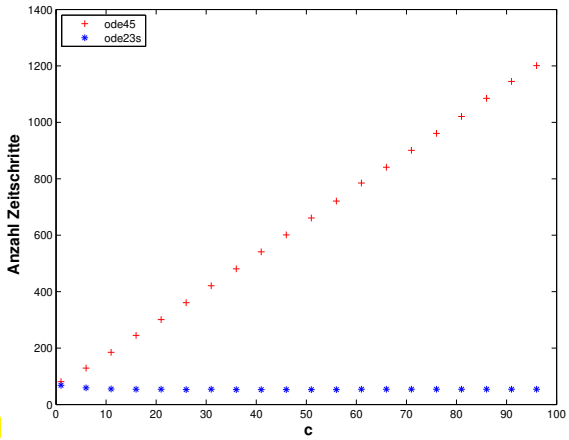


Fig. 60

SOLUTION of (13-2.c):

In the plot, we see that, for the integrator **ode45**, the number of steps rises with c. On the other hand, the A-stable method uses roughly the same amount of steps, irregardless of the value of c chosen.

As **ode45** is an explicit solver, it suffers from stability-induced step-size constraints for large $c > 0$. The A-stable integrator need not use smaller timesteps to satisfy the tolerance for big c.

This demonstrates [Lecture → Notion 12.2.0.7]: We see that the problem becomes the stiffer the larger the parameter c. This is expected from the analysis of Sub-problem (13-2.b).

▲

(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 of (13-2.d):

Recall the recursion formula for the implicit mid-point method applied to an autonomous ODE $\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y})$,

$$\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{f}\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) .$$

Its specialization for (13.2.1) reads

$$\frac{\mathbf{y}_{k+1} - \mathbf{y}_k}{h} = \mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) + c\left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right) \times \left(\mathbf{a} \times \left(\frac{\mathbf{y}_k + \mathbf{y}_{k+1}}{2}\right)\right) .$$

▲

(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 {
blic:
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;
ivate:
.......
```

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 $\mathrm{D}\mathbf{f}$ 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 of (13-2.e):

We observe that norm of the approximate states generated by the single step method *is constant*, $\|\mathbf{y}_k\|_2 = \|\mathbf{y}_0\|$ for all k, and equal to $1.73205 = \left\|[1,1,1]^\top\right\|_2$.

Our implementation relies on an auxiliary function that calls **implicit_RKIntegrator** with parameters characteristic of the implicit midpoint method. The Butcher scheme for the implicit midpoint method can be found in [Lecture → Ex. 12.3.4.2].

C++11-code 13.2.2: Example code.

```
template <class Function, class Jacobian>
std::vector<VectorXd> solve_imp_mid(Function &&f, Jacobian &&Jf, double T,
                                    const VectorXd &y0, unsigned int N) {
  std::vector<VectorXd> res(N + 1);
  // TO DO (13-1.e): Construct the implicit mid-point method with the class
  // implicit_RKIntegrator and execute the .solve() method.
  // Return the vector containing all steps including initial and final value.
  // START
  // Initialize implicit RK with Butcher scheme
  unsigned int s = 1;
  // Initialize coefficients for the implicit midpoint method
  MatrixXd A(s, s);
  VectorXd b(s);
  A << 1. / 2.;
  b << 1.;
  implicit_RKIntegrator RK(A, b);
  res = RK.solve(f, Jf, T, y0, N);
  // END
  return res;
}
```

The right-hand side function for (13.2.1) and its Jacobian as computed in Sub-problem (13-2.b) are realized as lambda functions.

C++11-code 13.2.3: Example code.

```
void tab_crossprod(void) {
  // TO DO (13-1.e): solve the cross-product ODE with the implicit RK method
  // defined in solve_imp_mid. Tabulate the norms of the results at all steps.
  // START
  double T = 10.;
  int N = 128;
  // set data
  double c = 1.;
  Vector3d y0;
  y0 << 1., 1., 1.;
  Vector3d a;
  a << 1., 0., 0.;
  // define rhs
  auto f = [&a, &c](const Vector3d &y) -> Vector3d {
    return a.cross(y) + c * y.cross(a.cross(y));
  };
  // define Jacobian of rhs
  auto Jf = [&a, &c](const Vector3d &y) {
    Matrix3d temp;
    temp << -c * (a(1) * y(1) + a(2) * y(2)),
        c * (2 * a(0) * y(1) - a(1) * y(0)) - a(2),
        a(1) + c * (2 * a(0) * y(2) - a(2) * y(0)),
        a(2) - c * (a(0) * y(1) - 2 * a(1) * y(0)),
        -c * (a(0) * y(0) + a(2) * y(2)),
        c * (2 * a(1) * y(2) - a(2) * y(1)) - a(0),
        -a(1) - c * (a(0) * y(2) - 2 * a(2) * y(0)),
        a(0) - c * (a(1) * y(2) - 2 * a(2) * y(1)),
        -c * (a(0) * y(0) + a(1) * y(1));
    return temp;
  };

```

```
std::vector<VectorXd> res_imp = solve_imp_mid(f, Jf, T, y0, N);

std::cout << "1. Implicit midpoint method" << std::endl;
std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
          << std::endl;

for (int i = 0; i < N + 1; ++i) {
  std::cout << std::setw(10) << T * i / N << std::setw(15)
            << res_imp[i].norm() << std::endl;
}
// END
```

▲

(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 $\mathbf{f} : \mathbb{R}^d \to \mathbb{R}^d$.

SOLUTION of (13-2.f):

The linearization of increment equations is also discussed in [Lecture → Ex. 12.4.0.1] and [Lecture → Eq. (12.4.0.2)].

The linear implicit mid-point method is obtained by expanding the right-hand side of the increment equation the implicit mid point method in its Taylor series in h around $h = 0$:

$$\mathbf{k}_1 = \mathbf{f}\left(\mathbf{y}_k + \frac{h}{2}\mathbf{k}_1\right) = \mathbf{f}(\mathbf{y}_k) + \frac{h}{2}D\mathbf{f}(\mathbf{y}_k)\mathbf{k}_1 + O(h^2)$$

and only taking the linear terms. Since the non-linear method is given by $\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1$, the linearization reads

$$\boxed{\mathbf{y}_{k+1} = \mathbf{y}_k + h\mathbf{k}_1^{\text{lin.}} \quad , \qquad \mathbf{k}_1^{\text{lin.}} := \left(I - \tfrac{h}{2}D\mathbf{f}(\mathbf{y}_k)\right)^{-1}\mathbf{f}(\mathbf{y}_k)} \ .$$

A step of the method just involves the solution of a single $d \times d$ linear system of equations.

▲

(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 of (13-2.g):

See file `cross.hpp`. The sequence of the Euclidean norms of the states is not exactly constant: it starts at 1.73205 and converges to 1.72853. This is due to the approximation introduced with the linearization.

C++11-code 13.2.4:

```
template <class Function, class Jacobian>
std::vector<VectorXd> solve_lin_mid(Function &&f, Jacobian &&Jf, double T,
                                    const VectorXd &y0, unsigned int N) {
  std::vector<VectorXd> res(N + 1);
  // TO DO (13-1.g): Implement the linear implicit mid-point method for
  // an autonomous ODE y' = f(y), y(0) = y0. Return the vector containing
  // all steps including initial and final value.
  // START
  // Initial step size
  double h = T / N;
  int d = y0.size();
  // Store initial data
  res.push_back(y0);

  // Initialize some memory to store temporary values
  VectorXd ytemp1 = y0;
  VectorXd ytemp2 = y0;
  // Pointers to swap previous value
  VectorXd *yold = &ytemp1;
  VectorXd *ynew = &ytemp2;
  MatrixXd eye = MatrixXd::Identity(3, 3);

  // Loop over all fixed steps
  for (unsigned int k = 0; k < N; ++k) {
    // Compute, save and swap next step
    *ynew = *yold + h * (eye - h / 2. * Jf(*yold)).lu().solve(f(*yold));
    res.push_back(*ynew);
    std::swap(yold, ynew);
  }
  // END
  return res;
}
```

C++11-code 13.2.5:

```
  // TO DO (13-1.g): solve the cross-product ODE with the implicit RK method
  // defined in solve_lin_mid. Tabulate the norms of the results at all steps.
  // START
  std::vector<VectorXd> res_lin = solve_lin_mid(f, Jf, T, y0, N);
```

```
  std::cout << "\n2. Linear implicit midpoint method" << std::endl;
  std::cout << std::setw(10) << "t" << std::setw(15) << "norm(y(t))"
            << std::endl;
  for (int i = 0; i < N + 1; ++i) {
    std::cout << std::setw(10) << T * i / N << std::setw(15)
              << res_lin[i].norm() << std::endl;
  }
  // END
```

▲

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:

$$\begin{array}{c|c} \mathbf{c} & \mathfrak{A} \\ \hline & \mathbf{b}^T \end{array} = \begin{array}{c|cccc} c_1 & \gamma & & \cdots & 0 \\ c_2 & a_{21} & \ddots & & \vdots \\ \vdots & \vdots & & & \vdots \\ \vdots & \vdots & & \ddots & \vdots \\ \vdots & \vdots & & & \ddots \\ c_s & a_{s1} & \cdots & a_{s,s-1} & \gamma \\ \hline & b_1 & \cdots & b_{s-1} & b_s \end{array}, \tag{13.3.1}$$

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

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{13.3.2}$$

should be solved numerically using a family of 2-stage SDIRK method described by the Butcher scheme

$$\begin{array}{c|cc} \gamma & \gamma & 0 \\ 1-\gamma & 1-2\gamma & \gamma \\ \hline & 1/2 & 1/2 \end{array}. \tag{13.3.3}$$

(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?

HINT 1 for (13-3.a): Recall that in every step of an implicit RK-SSM we have to solve a non-linear system of equations for the increments, see [Lecture → Rem. 12.3.3.7].

SOLUTION of (13-3.a):

For SDIRK methods the Butcher matrix is lower triangular. In this case the increment equations/stage equations can be solved separately and sequentially, computing $\mathbf{k}_1 \to \mathbf{k}_2 \to dots \to \mathbf{k}_s$ in that order.

So instead of solving a big $sd \times sd$ non-linear system of equations for a generic implicit Runge-Kutta single-step method, we need only solve s smaller $d \times d$ non-linear systems.

▲

(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 $\dot{\boldsymbol{y}} = \mathbf{f}(t, \boldsymbol{y})$.

SOLUTION of (13-3.b):

The increments $\mathbf{k}_i$ are given by:

$$\begin{aligned}\mathbf{k}_1 &= f(t_0 + h\gamma, \boldsymbol{y}_0 + h\gamma\mathbf{k}_1),\\ \mathbf{k}_2 &= f(t_0 + h(1-\gamma), \boldsymbol{y}_0 + h(1-2\gamma)\mathbf{k}_1 + h\gamma\mathbf{k}_2)\end{aligned}$$

▲

(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 of (13-3.c):

The stability function $S(z)$ of a method is derived by applying the method to the scalar, linear test equation

$$\dot{y}(t) = \lambda y(t) \ .$$

The solution can be written as

$$y_{k+1} = S(z) y_k \ ,$$

where $S(z)$ is the stability function and $z := h\lambda$.

In the case of the SDIRK method, we get:

$$\begin{aligned}k_1 &= \lambda(y_k + h\gamma k_1)\\ k_2 &= \lambda(y_k + h(1-2\gamma)k_1 + h\gamma k_2) \ .\end{aligned}$$

Therefore:

$$\begin{aligned}k_1 &= \frac{\lambda}{1 - h\lambda\gamma} y_k,\\ k_2 &= \frac{\lambda}{1 - h\lambda\gamma}(y_k + h(1-2\gamma)k_1) \ .\end{aligned}$$

Furthermore:

$$y_{k+1} = y_k + \frac{h}{2}(k_1 + k_2) \ .$$

With $z := h\lambda$ and by plugging in k_1 and k_2 we arrive at:

$$y_{k+1} = \underbrace{\left(1 + \frac{z}{2(1-\gamma z)}\left(2 + \frac{z(1-2\gamma)}{1-\gamma z}\right)\right)}_{=:S(z)} y_k \ .$$

Hence:

$$S(z) = \frac{2(1-\gamma z)^2 + 2z(1-\gamma z)^2 + z^2(1-2\gamma)}{2(1-\gamma z)^2}$$

and after collecting the powers of z in the numerator we get

$$S(z) = \frac{1 + z(1-2\gamma) + z^2(\gamma^2 - 2\gamma + \frac{1}{2})}{(1-\gamma z)^2} .$$

▲

(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 of (13-3.d):

Given the previous sub-problem, for $\gamma = 1$ the stability function is:

$$S_1(z) := \frac{1 - z - \frac{z^2}{2}}{(1-z)^2} \tag{13.3.5}$$

Verification of the A-stability of (13.3.5)**:**
By definition [Lecture → § 12.3.4.4], we need to show that $|S_1(z)| \leq 1$ for all $z \in \mathbb{C}^- := \{z \in \mathbb{C} \mid \operatorname{Re} z < 0\}$. In order to do this, we consider the stability function on the imaginary axis:

$$\begin{aligned}
|S_1(iy)|^2 &= \frac{|1 - iy - (iy)^2/2|^2}{|1-iy|^4} \\
&= \frac{|1 - iy + y/2|^2}{|1-iy|^4} \\
&= \frac{(1 + y^2/2)^2 + y^2}{(1+y^2)^2} \\
&= \frac{1 + 2y^2 + y^4/4}{1 + 2y^2 + y^4} \leq 1, \quad y \in \mathbb{R}
\end{aligned}$$

Since the only pole ($z = 1$) of the rational function $S_1(z)$ lies in the positive half plane of $\mathbb{C}$, the function S_1 is holomorphic in the left half plane. Furthermore, S_1 is bounded by 1 on the boundary of this half

plane (i.e. on the imaginary axis). So, by the maximum principle for holomorphic functions (hint), S_1 is bounded on the entire left half plane by 1. This implies in particular that S_1 is A-stable.

Verification of the L-stability of (13.3.5)**:**
S_1 is not L-stable (*cf.* definition [Lecture → § 12.3.4.11]), because:

$$\lim_{\operatorname{Re} z \to -\infty} |S_1(z)| = \lim_{\operatorname{Re} z \to -\infty} \left| \frac{1 - z - z^2/2}{1 - 2z + z^2} \right| = \frac{1}{2} \neq 0$$

▲

(13-3.e) ⚀ (5 min.) Formulate (13.3.2) as an initial value problem for a linear first order system for the function $\boldsymbol{z}(t) = (y(t), \dot{y}(t))^\top$.

SOLUTION of (13-3.e):

Define $z_1 = y$, $z_2 = \dot{y}$, then the initial value problem

$$\ddot{y} + \dot{y} + y = 0, \qquad y(0) = 1, \quad \dot{y}(0) = 0 \tag{13.3.6}$$

is equivalently to the first order system

$$\begin{aligned} \dot{z}_1 &= z_2 \\ \dot{z}_2 &= -z_1 - z_2 \end{aligned} \tag{13.3.7}$$

with initial values $z_1(0) = 1$ and $z_2(0) = 0$.

▲

(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 `z0` and returning the state after a single step of size `h`.

SOLUTION of (13-3.f):

Let $\mathbf{A} := \begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix}$. Then:

$$\begin{aligned} k_1 &= \mathbf{A}\boldsymbol{y}_0 + h\gamma\mathbf{A}k_1 \\ k_2 &= \mathbf{A}\boldsymbol{y}_0 + h(1-2\gamma)\mathbf{A}k1 + h\mathbf{A}k_2 \,, \end{aligned}$$

so

$$\begin{aligned} k_1 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}\mathbf{A}\boldsymbol{y}_0 \\ k_2 &= (\mathbf{I} - h\gamma\mathbf{A})^{-1}(\mathbf{A}\boldsymbol{y}_0 + h(1-2\gamma)\mathbf{A}k_1) \,. \end{aligned}$$

C++11-code 13.3.8: Solution of (13-3.f)

```
//! \brief One step of autonomous IVP y'' + y' + y = 0, [y(0), y'(0)] = z0 using
//! SDIRK method Use SDIRK method for first order ode z' = f(z). Steps of size
//! h. \tparam Eigen::VectorXd type of solution space y and initial data y0
//! \param[in] z0 initial data z(0) \param[in] h size of the step \param[in]
//! gamma parameter \return next step z1
Eigen::Vector2d sdirkStep(const Eigen::Vector2d &z0, double h, double gamma) {
  // Matrix A for evaluation of f
  Eigen::Matrix2d A;
  A << 0., 1., -1., -1.;

  // Reuse factorization
  auto A_lu = (Eigen::Matrix2d::Identity() - h * gamma * A).partialPivLu();
  Eigen::Vector2d az = A * z0;

  // Increments
  Eigen::Vector2d k1 = A_lu.solve(az);
  Eigen::Vector2d k2 = A_lu.solve(az + h * (1 - 2 * gamma) * A * k1);

  // Next step
  return z0 + h * 0.5 * (k1 + k2);
}
```

▲

(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 $\boldsymbol{y}_0 = [1,0]^\top$ as initial value, `T=10` as end time and `N=20,40,80,...,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 of (13-3.g):

C++11-code 13.3.9: Timestepping for (13.3.2) **with SDIRK implicit RK-SSM**

```
//! \brief Solve autonomous IVP y'' + y' + y = 0, [y(0), y'(0)] = z0 using SDIRK
//! method Use SDIRK method for first order ode z' = f(z), with N equidistant
//! steps \tparam Eigen::VectorXd type of solution space z = [y,y']! and initial
//! data z0 = [y(0), y'(0)] \param[in] z0 initial data z(0) \param[in] N number
//! of equidistant steps \param[in] T final time of simulation \param[in] gamma
//! parameter \return vector containing each step of z_k (y and y')
std::vector<Eigen::Vector2d>
sdirkSolve(const Eigen::Vector2d &z0, unsigned int N, double T, double gamma) {
  // Solution vector
  std::vector<Eigen::VectorXd> res(N + 1);
  // Equidistant step size
  const double h = T / N;
  // Push initial data
```

```
  res.at(0) = z0;
  // Main loop
  for (unsigned int i = 1; i <= N; ++i) {
    res.at(i) = sdirkStep(res.at(i - 1), h, gamma);
  }
  return res;
}
```

C++11-code 13.3.10: Testing convergence of SDIRK method for an IVP for (13.3.2)

```
  // Initial data z0 = [y(0), y'(0)]
  Eigen::Vector2d z0;
  z0 << 1, 0;
  // Final time
  const double T = 10;
  // Parameter
  const double gamma = (3. + std::sqrt(3.)) / 6.;
  // Mesh sizes
  std::vector<int> N = {20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240};
  // Exact solution (only y(t)) given z0 = [y(0), y'(0)] and t
  auto yex = [&z0](double t) {
    return 1. / 3. * std::exp(-t / 2.) *
           (3. * z0(0) * std::cos(std::sqrt(3.) * t / 2.) +
            std::sqrt(3.) * z0(0) * std::sin(std::sqrt(3.) * t / 2.) +
            2. * std::sqrt(3.) * z0(1) * std::sin(std::sqrt(3.) * t / 2.));
  };

  // Store old error for rate computation
  double errold = 0;
  std::cout << std::setw(15) << "n" << std::setw(15) << "maxerr"
            << std::setw(15) << "rate" << std::endl;
  // Loop over all meshes
  for (unsigned int i = 0; i < N.size(); ++i) {
    int n = N.at(i);
    // Get solution
    auto sol = sdirkSolve(z0, n, T, gamma);
    // Compute error
    double err = std::abs(sol.back()(0) - yex(T));

    // Print table
    std::cout << std::setw(15) << n << std::setw(15) << err;
    if (i > 0)
      std::cout << std::setw(15) << std::log2(errold / err);
    std::cout << std::endl;

    // Store old error
    errold = err;
  }
```

N	maximal error
20	0.000283567
40	5.05272e-05
80	7.73715e-06
160	1.07013e-06
320	1.40565e-07
640	1.80048e-08
1280	2.27798e-09
2560	2.86467e-10
5120	3.59161e-11
10240	4.49619e-12

◁ Norm of maximal error on temporal mesh.

We observe an asymptotic (in the last steps) decrease of the error norm by a factor of ≈ 8 whenever the number of (equidistant) timesteps is doubled. Thus we observe *algebraic convergence* with rate 3 in the timestep size h:

$$\text{errornorm} = O(h^3) \ .$$

This hints that the SDIRK single-step method has order 3 .

The numerically estimated order of convergence is 3.

▲

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:

$$\dot{\mathbf{y}} = \mathbf{f}(\mathbf{y}) \tag{13.4.1}$$

and discretize it with a *semi-implicit* Runge-Kutta SSM, a so-called **Rosenbrock method**,

$$\begin{aligned} \mathbf{W}\mathbf{k}_1 &= \mathbf{f}(\mathbf{y}_0)\ , \\ \mathbf{W}\mathbf{k}_2 &= \mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\mathbf{k}_1) - ah\mathbf{J}\mathbf{k}_1\ , \\ \mathbf{y}_1 &= \mathbf{y}_0 + h\mathbf{k}_2\ , \end{aligned} \tag{13.4.2}$$

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 of (13-4.a):

For a scalar ODE, the Jacobian is just the derivative w.r.t. y, whence $J = Df(y) = f'(y) = (\lambda y)' = \lambda$. The quantity $\mathbf{W}$ is, therefore, a scalar quantity as well:

$$W = 1 - ahJ = 1 - ah\lambda$$

If we plug everything together and assume h is small enough:

$$\begin{aligned} y_1 &= y_0 + h\frac{\mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\mathbf{k}_1) - ah\mathbf{J}\mathbf{k}_1}{W} \\ &= y_0 + h\frac{\mathbf{f}(\mathbf{y}_0 + \frac{1}{2}h\frac{\mathbf{f}(\mathbf{y}_0)}{W}) - ah\mathbf{J}\frac{\mathbf{f}(\mathbf{y}_0)}{W})}{W} \\ &= y_0 + h\frac{\lambda(\mathbf{y}_0 + \frac{1}{2}h\frac{\lambda\mathbf{y}_0}{1-ah\lambda}) - \frac{ah\lambda^2 y_0}{1-ah\lambda}}{1-ah\lambda} \\ &= \frac{(1-ah\lambda)^2 + h\lambda(1-ah\lambda) + \frac{1}{2}h^2\lambda^2 - ah^2\lambda^2}{(1-ah\lambda)^2}y_0 \\ &= \frac{(1+a^2z^2-2az) + z(1-az) + \frac{1}{2}z^2 - az^2}{(1-az)^2}y_0 \\ &= \frac{1 + (1-2a)z + (\frac{1}{2} - 2a + a^2)z^2}{(1-az)^2}y_0 \end{aligned}$$

Since $\frac{1}{2} - 2a + a^2 = 0$, it follows:

$$S(z) = \frac{1 + (1-2a)z}{(1-az)^2}$$

▲

(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 $|z| \to 0$? Deduce the maximal possible order of the method Eq. (13.4.2).

HINT 1 for (13-4.b): The idea behind this sub-problem is elucidated in [Lecture → Rem. 12.1.0.21]. Apply [Lecture → Lemma 12.1.0.23]. ⌟

SOLUTION of (13-4.b):

We compute:

$$\begin{aligned} S(0) &= 1, \\ S'(0) &= 1, \\ S''(0) &= 4a - 2a^2, \\ S'''(0) &= 18a^2 - 12a^3 \ , \end{aligned}$$

therefore

$$S(z) = 1 + z + (2a - a^2)z^2 + (3a^2 - 2a^3)z^3 + O(z^4) \ .$$

We can also compute the expansion of $\exp$:

$$\exp(z) = 1 + z + \frac{1}{2}z^2 + \frac{1}{6}z^3 + O(z^4) \ .$$

Since $2a - a^2 = \frac{1}{2}$ but $3a^2 - 2a^3 = \frac{1}{2}(\sqrt{2} - 1) \neq \frac{1}{6}$, we have:

$$|S(z) - \exp(z)| = O(|z|^3) \ .$$

Using [Lecture → Lemma 12.1.0.23], we deduce that the maximal order q of the scheme is $q = 2$.

▲

(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) $\texttt{y0} = \mathbf{y}(0)$, 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 of (13-4.c):

The implementation relies on EIGEN's direct solvers for dense linear systems.

C++11-code 13.4.3: Solution of Sub-problem (13-4.c)

```
template <class Func, class DFunc, class StateType>
std::vector<StateType> solveRosenbrock(const Func & f, const DFunc & df,
                                       const StateType & y0, unsigned int N,
                                           double T) {
    const double h = T/N;
    const double a = 1. / (std::sqrt(2) + 2.);

    // Will contain all time steps
    std::vector<StateType> res(N+1);
    // Push initial data
    res.at(0) = y0;

    // Some temporary
    StateType k1, k2;
    Eigen::Matrix2d J, W;

    // Main loop: *up to N (performs N steps)*
    for(unsigned int i = 1; i <= N; ++i) {
        StateType & yprev = res.at(i-1);

        // Jacobian computation
        J = df(yprev);
        W = Eigen::Matrix2d::Identity() - a*h*J;

        // Reuse factorization for each step
        auto W_lu = W.partialPivLu();

        // Increments
        k1 = W_lu.solve(f(yprev));
        k2 = W_lu.solve(f(yprev+0.5*h*k1) - a*h*J*k1);

        // Push new step
        res.at(i) = yprev + h*k2;
    }

    return res;
}
```

▲

(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]:

$$\mathbf{f}(\mathbf{y}) := \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{y} + \lambda(1 - \|\mathbf{y}\|^2)\mathbf{y} \,, \tag{13.4.4}$$

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 $\mathbf{y}_{\mathrm{ref}}$ with timestep size $h = 2^{-12}$. Monitor the maximal error on the temporal mesh

$$\max_j \|\mathbf{y}_j - \mathbf{y}_{\mathrm{ref}}(t_j)\|_2 \,,$$

and use it to estimate a rate of algebraic convergence by means of linear regression.

To do this write a C++ function

```
double cvgRosenbrock(void);
```

that tabulates the errors and returns the estimated rate of convergence.

SOLUTION of (13-4.d):

The Jacobian Df is:

$$D\mathbf{f}(\mathbf{y}) = \begin{bmatrix} \lambda(1 - \|\mathbf{y}\|^2) - 2\lambda y_0^2 & -1 - 2\lambda y_1 y_0 \\ 1 - 2\lambda y_1 y_0 & \lambda(1 - \|\mathbf{y}\|^2) - 2\lambda y_1^2 \end{bmatrix}$$

For the implementation, *cf.* `rosenbrock.cpp`.

▲

In complex analysis you might have hear 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

$$\mathbb{C}^- := \{z \in \mathbb{C} \mid Re(z) < 0\} \,.$$

Let $f : D \subset \mathbb{C} \to \mathbb{C}$ be non-constant, defined on $\overline{\mathbb{C}^-}$, and analytic in $\mathbb{C}^-$. Furthermore, assume that $w := \lim_{|z|\to\infty} f(z)$ exists and $w \in \mathbb{C}$, then:

$$\forall z \in \mathbb{C}^- : \quad |f(z)| < \sup_{\tau\in\mathbb{R}} |f(i\tau)| \,.$$

(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]).

HINT 1 for (13-4.e): To investigate the A-stability, calculate the complex norm of $S(z)$ on the imaginary axis $\operatorname{Re} z = 0$ and apply the ⌟

SOLUTION of (13-4.e):

We start by proving that the method is A-stable [Lecture → § 12.3.4.4], meaning that $S(z) < 1, \forall z \in \mathbb{C}, \mathrm{Re}(z) < 0$. First of all, we can compute the complex norm of the stability function at $z = iy, y \in \mathbb{R}$ as:

$$|S(iy)|^2 = \frac{|1 + (1-2a)iy|^2}{|(1-aiy)^2|^2} = \frac{1 + (1-2a)^2y^2}{(1+a^2y^2)^2} = \frac{1 + (1-4a+4a^2)y^2}{(1+2a^2y^2+a^4y^4)} \,.$$

Notice that $1 - 4a + 4a^2 = 2a^2$, therefore:

$$|S(iy)|^2 = \frac{1 + 2a^2y^2}{1 + 2a^2y^2 + a^4y^4} < 1 \,.$$

The norm of the function S is bounded on the imaginary axis. Observe that $|S(z)| \to 0, |z| \to \infty$, which, follows from the fact that the degree of the denominator of S is bigger than the polynomial degree of the numerator. Notice that $1 - 2a = 1 - 2\frac{1}{\sqrt{(2)}+2} = \sqrt{2} - 2$.

The only pole of the function is at $z = 1/a$, therefore the function is holomorphic on the left complex plane.

Applying the theorem of the hint on concludes that the absolute value of the function is bounded by 1 on the left complex plane.

The L-stability follows immediately with A-stability and the fact that the absolute value of the function converges to zero as $|z| \to \infty$.

▲

End Problem 13-4 , 80 min.